CrossTALK

The Journal of Defense Software Engineering

Providing a Common Platform for Building Interoperable Systems
DII COE

Sponsored by the Embedded Computer Resources Support Improvement Program (ESIP)

September 1999
Volume 12 Number 9

Published by the Software Technology Support Center

STSC

The Strategic Battlefield

Reuel Alder
Software Technology Support Center

Chess is a game of position, material, and time. The four squares in the center of the board are the most important positions to control. However, if you cannot see where your opponent's pieces are located, your attack might proceed in traditional military style with a series of outflanking maneuvers. In the battle of Gettysburg, Gen. Robert E. Lee lost contact with his cavalry, the eyes of the army. He did not know the Union army's position or strength and was unaware of his near success in the first two attacks because he lacked communication.

Battles, like chess, are best fought with a continuous flow of accurate information.

The military understands the need for strategic battles, but building systems in the Department of Defense that effectively communicate has always been a challenge. Each service has its own way of doing business and its own set of contractors developing new and improved weapon systems. Too often the result is independently developed weapon systems that have unique methods of communication. Without an integrating strategy, interoperability is but a dream.

Computers add speed, but without an integrating strategy we merely increase the rate at which we fail to communicate. Our experience in developing computers would indicate there are infinite ways to transmit a message without really communicating. The Defense Information Infrastructure (DII) Common Operating Environment (COE) is an essential component to effective computer communications in the military.

In this issue Pamela Engert and Julie Surer (page 4) provide an introduction to building interoperable systems with DII COE as a foundation. Beginning on page 6, Lt. Col. M.J. Robillard, Dr. H. Rebecca Callison, and John Maurer outline the need and proposed approach for extending the DII COE to real-time systems. The DII COE may be viewed as "an architecture, a collection of reusable software elements, a software infrastructure, and a set of guideline and standards" used to achieve interoperability. This approach to software development can provide substantial savings in production as well as enhanced interoperability, provided the technical architecture is of good quality and continuously updated.

DII COE is an important systematic approach to the age-old problem of accurate communications in battle. The battlefield — whether military or industry — is a place where what you don't know can and will hurt you. ◆

Letter to the Editor

A Comment on Reaching Capability Maturity Model^{TM} (CMM) Level 2

I'd like to comment on the article in the June issue that describes how a government organization became "certified" at Level 2. (By the way, how does an organization get "certified"?)

The author asserts, and I think it's true, that getting management on board is the first and biggest step when an organization decides to start climbing the ladder of process improvement. The direction from the board governing the Department of Agriculture organization — go in 10 months from never assessed to Level 2, or lose funding — misses the point by a mile.

The resulting organizational behavior often then becomes "gaming the number." That type of behavior results in the kind of deadline-motivated personal heroics the author mentions. It also, in this case, seems to have modified Software Quality Assurance's mission from one of participating with developers and management to ensure process/product quality, to one of monitoring "to meet Level 2 goals."

Most disturbing to me in this article was the complete absence of the word "finding," which is the heart and soul of the CMM-based appraisal for internal process improvement (CBA-IPI) method. The IPI is about improvement along the continuum that leads to reliable software that satisfies its requirements, costs, and schedules that are congruent with plans, and continuous process improvement — not about achieving a level. Focusing on levels harmfully directs attention from process improvement and process maturity.

It is encouraging to see more federal government organizations beginning the walk to quality. It would be even more encouraging to see them do it for all the right reasons.

Name withheld by request

On the cover: Salt Lake photographer Tom Anastasion used building tools and architectural concepts to illustrate September's theme of Defense Information Infrastructure (DII) Common Operating Environment (COE).

CrossTalk Would Like to Ask You a Question

What was the best and/or worst software technology innovation of the 20th century?

Respond in writing at:
fax: 801-777-8069
e-mail: custserv@software.hill.af.mil
mail: *CrossTalk*
OO-ALC/TISE
7278 Fourth Street
Hill AFB, UT 84056-5205

We will print your responses in our special December issue on the Evolution of Software Technology.

Coming Events

SIGAda '99 Annual International Conference

Dates: Oct. 17-21, 1999
Location: Redondo Beach, Calif.
Topic: The Engineering of Industrial Strength Real-Time and Distributed Systems Using Ada and Related Technologies
Sponsor: ACM SIGAda
Contact: Hal Hart (TRW), conference chairman
E-mail: Hal.Hart@ac.org
Internet: http://www.acm.org/sigada/conf/sigada99

International Function Point Users Group Annual Conference and Workshops

Dates: Oct. 18 - 22, 1999 — workshops Oct. 18-19; conference Oct. 20 - 22
Location: New Orleans, La.
Theme: New Millennium New Metrics
Sponsors: International Function Point Users Group (IFPUG), Outsourcing Center
Contact: IFPUG, 5008-28 Pine Creek Drive, Westerville, Ohio 43081-4899
Voice: 614-895-7130
Fax: 614-865-3466
E-mail: ifpug@ifpug.org
Internet: http://www.ifpug.org

DCI's Data Warehouse World Exposition

Dates: Oct. 25-26, 1999
Location: Boston, Mass.
Focus: DCI's Data Warehouse World is the premier data warehouse event in the industry. Our prestigious training staff and advisory board features virtually all of the expert data warehouse practitioners and thought leaders who are shaping the industry today.
Contact: http://www.dci.com/datawhse/

Software Testing Analysis & Review (STAR) '99 West

Dates: Nov. 1-5, 1999
Theme: Improving Software Testing and Quality Engineering Practices Worldwide
Location: San Jose, Calif.
Sponsor: Software Quality Engineering
Topics: Specific ways to improve testing efforts and results. Field-proven techniques for testing client/server, object-oriented, GUI, and Internet applications. How to use test engineering to consistently achieve greater software quality. The best Internet/Web testing tools and how to use them effectively. How to lower development costs and boost productivity with test engineering.
Voice: 1-800-423-8378 or 904-278-0707
Fax: 904-278-4380
E-mail: sqeinfo@sqe.com

Third International Software Quality Week Europe '99 (QWE '99)

Dates: Nov. 8-12, 1999
Location: Brussels, Belgium
Sponsor: Software Research Institute
Topic: The conference theme, "Lessons Learned," reflects the tremendous accomplishments of the past few years, and aims to see what can be learned from such efforts as the Y2K, Euro Conversion, the push for e-Commerce, and the widespread use of mature software quality processes.
Contact: Rita Bral
E-mail: bral@soft.com

Introduction to the Defense Information Infrastructure (DII) Common Operating Environment (COE)

Pamela Engert and Julie Surer
The MITRE Corp.

The Defense Information Infrastructure (DII) Common Operating Environment (COE) provides a foundation for building interoperable command and control systems using reusable software components. The DII COE is comprised of many concepts. It is, in one sense, a set of reusable software components. In addition, the DII COE is also a system architecture that allows the components to be reused in command and control systems, as well as the standards and guidelines that define how the components can be constructed. This article briefly defines the DII COE architecture and describes the defined compliance criteria.

THE DII COE PROVIDES A foundation for building interoperable systems through the use of reusable software components (building blocks). The DII COE can be characterized as a number of things, depending upon one's point of view. It is an architecture, a collection of reusable software elements, a software infrastructure, and a set of guidelines and standards. More importantly, however, is that it provides a common platform (or foundation) for building interoperable systems. Therefore, one could think of the DII COE as one component of a system architecture, as it is an implementation of the Joint Technical Architecture (JTA). One could also think of the DII COE as an approach to software development — how to go about building interoperable systems on a common platform. Finally, it is important to realize that the DII COE is not a system, but a set of building blocks from which a system can be built. Global Command and Control System, Global Combat Support System, and service unique programs (like Air Force Theater Battle Management Corp. Systems) are building their systems on top of the DII COE foundation.

Definition of DII COE

DII COE is a software infrastructure, a collection of reusable software components, a set of Application Program Interfaces (APIs), and a series of specifications and standards for developing interoperable systems.

The DII COE taxonomy defines two layers of reusable software components: infrastructure services, which include the DII COE kernel services, and the underlying commercial-off-the-shelf (COTS) operating systems. Infrastructure services address the movement of data through the network and includes distributed computing and web services. The kernel provides low-level services, including a desktop environment, runtime tools, and basic system and security administration.

Common support applications provide services that address common command and control functionality, for example, mapping and message processing.

Standard APIs provide the interfaces between mission applications and reusable software components of the DII COE. Mission applications are developed on top of the DII COE and provide mission domain specific functionality.

DII COE Compliance

DII COE compliance measures the degree to which a software component, including mission applications, can plug-and-play (the degree of interoperability) in the COE. The goal of meeting DII COE compliance is to ensure seamless software component integration and system operation. A software component is assessed in four categories for COE compliance: runtime environment, style guide, architectural compatibility, and software quality. Although all four categories of compliance are considered in an assessment, the primary focus is on the runtime environment.

Runtime Environment

The runtime environment category assesses how well the proposed software segment or system (collection of segments) functions within the COE environment and the extent that the software reuses COE components. The evaluation determines if the proposed software segment can be added to the system without adversely affecting system interoperability. Segments are evaluated against the checklist in the DII COE integration and runtime specification.

Style Guide

This category assesses the user interface of a segment for consistency and conformance to the checklist in the user interface specification for the DII.

Architectural Compatibility

This category determines if proposed software is architecturally sound and compatible with the COE. Unlike the runtime environment and style guide categories, the architecture compatibility category has not been defined to date.

Software Quality

This category assesses software for portability and integration into the COE. The level of life cycle maintenance support associated with the proposed COE component is estimated. The assessment of software quality compliance level is achieved by using complexity and quality metrics collections, portability analysis, and COE API compliance analysis. COTS analysis tools are used for automated, nonintrusive compliance checking.

Runtime compliance is expressed in terms of eight levels of compliance defined in the DII COE integration and runtime specification. Level 5 compliance is considered “minimal DII” and indicates that segmented applications can share the same DII COE kernel without interfering with one another, that the segments can be installed using standard tools, and that the segments conform to the DII user interface specification. Level 8, or "full DII," implies 100 percent compliance with all DII COE runtime and user interface criteria.

DII COE Compliance Mandates

Several DII COE mandates exist for Department of Defense systems. The Office of the Under Secretary of Defense has issued a directive that all command, control, communications, computers, and intelligence systems (C4I) be JTA compliant and mandates DII COE compliance Level 5 for legacy systems and Level 6 for new systems. The goal is to reach Level 7. The JTA mandates DII COE Level 5 compliance with a goal of Level 8 for all C4I systems.

Summary

The DII COE will be a key contributor in achieving the C4I vision of providing warriors with technically advanced, interoperable command and control systems. The DII COE provides a foundation for building interoperable systems through the use of reusable software components. ◆

About the Authors

Pamela Engert joined the MITRE Corp. in Bedford, Mass., in 1986 and is now a lead engineer. She supports the System Engineering Process Office and the Acquisition Development Office. She has a bachelor of science degree in mathematics and computer science and a master's degree in engineering management.

The MITRE Corp.
202 Burlington Road
Bedford Mass. 01730-1421
Phone: 781-271-3138
Fax: 781-271-2101
E-mail: pengert@mitre.org

Julie Surer is a principal engineer at the MITRE Corp. in Bedford, Mass., where she supports the ESC/DIE (Electronic System Center) Chief Architect's Office. Surer is the Air Force DII COE chief engineer. She is also the corporate representative responsible for DII COE technical activities in the Air Force. She is also the corporate representative to The Open Group, an international standards consortium. She has a bachelor's degree from the University of Florida in engineering science and a master's degree from the University of South Florida in electrical engineering.

The MITRE Corp.
202 Burlington Road
Bedford Mass. 01730-1421
Phone: 781-377-6809
Fax: 781-377-7779
E-mail: jsurer@mitre.org

DII COE Web Information Resources

For extensive, up-to-the-minute information on DII COE, visit DISA's DII COE site at
http://spider.dii.osfl.disa.mil/dii

DISA home page
http://www.disa.mil

For current information on DII COE in the Air Force, visit
http://www.esc-dii.hanscom.af.mil/Chief_Architect/Ca_home.htm.

For current information on DII COE mandates, visit
http://www.escdii.hanscom.af.mil/Chief_Architect/dii-coe/faq/faq.html

Note: The AF ESC/DII Web pages are restricted to clients with a .mil primary domain.

HQ AFCA DII COE home page
http://www.afca.scott.af.mil/

Extending the DII COE for Real-Time

Lt. Col. Lucie M.J. Robillard
ESC/AWW
Dr. H. Rebecca Callison
The Boeing Company
John Maurer
The MITRE Corp.

The Defense Information Infrastructure Common Operating Environment (DII COE) provides an environment in which common reusable infrastructure and applications across information systems help achieve goals for interoperability. The Department of Defense (DoD) has a vision for extending these ideas for reuse and commonality to improve the effectiveness of systems performing real-time command and control (C2) missions. This article outlines the need and proposed approaches for extending the DII COE with real-time capabilities.

THE DII COE ORIGINATED with a simple observation about C2 systems: certain functions (mapping, track management, and communication interfaces) are fundamental to virtually every C2 system. Yet, these functions are built repeatedly in incompatible ways even when the requirements are the same or vary only slightly between systems. If these common functions could be extracted, implemented as a set of extensible building blocks, and made readily available to system designers, development schedules could be accelerated and substantial savings achieved through software reuse. Moreover, interoperability would be significantly improved if common software was used across systems for common functions. Realizing these benefits is DII COE's goal as stated in [1].

Several DoD systems, notably the Global Command and Control System, Global Command Support System, and Theater Battle Management Core System, utilize the DII COE, with additional systems planning anticipated for the DII COE. They are being connected into a global grid that will include, in addition to C2, sensor systems and weapons platforms. With sensors and weapons intrinsically operating in a real-time arena, and with time-sensitive targets becoming increasingly important, the application of the DII COE concepts to real-time C2 becomes more compelling.

Background

In 1996, at the Air Force Electronic Systems Command (ESC) Hanscom Air Force Base, Integrated Command and Control System (IC2S) planners began to explore the application and viability of DII COE concepts. Since critical IC2S missions were expected to respond to stringent real-time requirements that could not be satisfied by the DII COE, the ESC Commander, Lt. Gen. Ronald T. Kadish, directed that all C2 programs develop a set of requirements for real-time extensions to existing DII COE capabilities. In the spring of 1997, Air Force, Army, and Navy representatives met to discuss the high correlation of real-time requirements across the services. In July 1997, the Air Force, Army, Navy, and Marine Corps jointly petitioned the Defense Information Systems Agency (DISA) to charter a DII COE real-time technical working group (TWG) aimed at developing common requirements and recommendations for potential products to provide real-time capabilities to the DII COE. DISA approved the services' request, and the real-time TWG began meeting in August 1997.

Initial studies, conducted at ESC, highlighted numerous, relevant characteristics of real-time systems, subsequently suggesting that a piece-part approach to assembling real-time components would not be effective. In late 1997, the Air Force designated the Airborne Warning and Control System (AWACS) Program Office as executive agent for the DII COE real-time extensions. The DII COE real-time integrated product team (DII COE RT IPT) embodies that executive authority. Because their missions are so closely related, the real-time TWG and IPT are in continuous coordination, conduct joint meetings, and share data. Both the TWG and IPT enjoy the benefits from the active support and participation of Army, Air Force, Navy, and intelligence community representatives.

The concepts described here are the product of these two groups, working in collaboration with DISA.

Understanding Real-Time

Real-time process is where the computation's validity depends on logical correctness and time-sensitive completion. In a real-time system, the time that an activity[1] takes to complete and deliver results is as important to correctness as, for example, the computation's precision or accuracy. What is important is not how fast the system responds but that it responds predictably at appropriate times. For example, a protocol for synchronizing clocks across a communication network (distributed time service) is required to be accurate, not fast.

Hard real-time applies to activities that must be deterministic; critical activities have deadlines. When this processing fails to meet a deadline, the system has failed. For example, a missile-warning radar fails if the radar processor completes its computation, but is unable to deliver target reports before an incoming missile passes through a designated intercept envelope. The design emphasis when building systems with hard deadlines is to guarantee that all deadlines will be met.

Soft real-time is nondeterministic to the extent that an occasional missed deadline can be tolerated as acceptable degraded performance, not a system failure. The value of completing a soft real-time activity decreases after its deadline has passed, but the rate at which the value decreases differs between activities. The operational procedures for dealing with missed deadlines also vary. For example, systems may:

- choose to complete a late action anyway
- abandon an ongoing computation in favor of beginning the next cycle
- attempt a less complex computation instead, and/or
- begin to shed low priority, non-critical functions in an effort to correct the overload problem in future cycles.

The RT TWG recognizes a requirement for real-time extensions to the DII COE to support systems with both hard and soft real-time requirements.

Three fundamental properties are often cited as keys to building systems in which required events occur on time, every time: priority, pre-emption, and predictability. Tasks are assigned priorities according to a real-time scheduling algorithm under which theoretical scheduling guarantees can be made[2]. A pre-emptive real-time scheduler then grants system resources to the highest priority task that is ready to run, even if it must interrupt — or even starve — lower priority tasks. This real-time scheduler will often use some form of priority inheritance to limit the length of time that low priority tasks can hold shared resources and block higher priority tasks waiting for resource access. These techniques differ from priority assignments and scheduling algorithms used in general purpose computing where fairness to all users and good average response times are the objectives.

Achieving predictability depends on the component parts' design. Components must be designed as independently schedulable entities (tasks or processes) whose precise execution schedule may be determined dynamically at runtime by the real-time scheduler. They cannot be hard-coded to a particular execution schedule. To limit priority inversions, each component should minimize the time it holds any shared resource and/or disables pre-emption by a higher priority task. Components must use techniques that can be provably correct and analyzed for sharing access to resources such as peripherals, networks, and particularly shared data.

Each component also needs to be constructed for inherently predictable timing behavior. In practical terms, this restriction means that real-time applications must avoid the use of programming features with unpredictable timing. The list of unpredictable constructs includes programming approaches such as the use of dynamic allocation of memory from a heap, garbage collection, and dynamic paging of virtual memory[3].

Other Characteristics of the Real-Time Domain

While predictable timing is the defining characteristic of real-time computing, there are other characteristics typical of these systems as well.

Concurrency

To respond effectively to events that occur asynchronously in the environment with which the system interacts, real-time systems are often constructed as collections of concurrently executing tasks and processes. In contrast with the concurrent processing inherent in general purpose computing, where processes compete for resources without interacting in other ways, the tasks and processes of real-time systems cooperate closely to achieve mission objectives.

Reliability and Availability

Since military real-time systems perform activities critical to the success of military missions, they typically have rigorous reliability and availability requirements.

Operation in Harsh Environments

Real-time systems often must operate in extreme environments that are far less accommodating than a typical office or computer facility. These systems are often installed on vehicle platforms, e.g., aircraft, tank, or missile. Environmental conditions can be expected to exert significant space, weight, and power consumption constraints. Also, the hardware often must be designed to withstand environmental stresses such as extremes of temperature, shock, vibration, corrosive atmospheres, poorly conditioned electrical power, and severe electromagnetic fields. These considerations significantly restrict the choice of equipment that can be packaged with the real-time system. As such, the impact on the DII COE configuration, operating in a real-time environment, cannot be overstated.

Vision for a Real-Time Common Operating Environment

The vision of extending DII COE for real-time systems, as depicted in Figure 1, begins with the layered architecture in place for DII COE today.

The DII COE Kernel provides the basic interfaces and functions to be used by standards-based infrastructure components and DII COE-compliant applications to achieve portability between systems. The planned DII COE Configurable RT Kernel[4] extends basic DII COE concepts in two ways:

- to build the foundation of predictable execution on which real-time systems depend. The RT Kernel is hosted only on real-time operating systems (RTOSs) that provide real-time scheduling capabilities, reasonably predictable operating system performance, and the services required for timely execution of real-time tasks and processes.
- because many real-time systems operate with limited computing resources, the RT Kernel is configurable. RT Kernel services are selectable, rather than mandatory, and only those services compatible with the capabilities of the RTOS are provided for each platform. For example, the RT Kernel services for a small RTOS like VxWorks®, which supports only single-process, single-user configurations, would not include services that manage concurrent access by multiple users. The integrator of a DII COE-compliant system tailors the RT Kernel by selecting only those services required for the specific

VxWorks is a registered trademark of Wind River Systems.

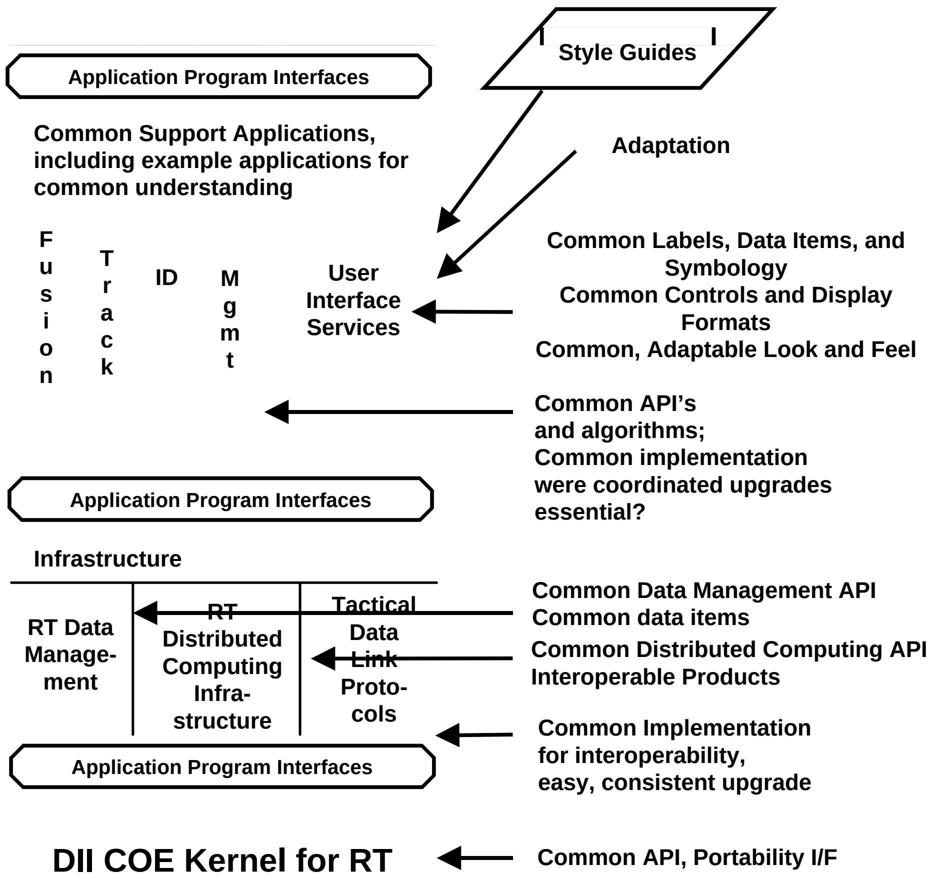
Style Guides
Application Program Interfaces
Common Support Applications, including example applications for common understanding
Adaptation
Fusion
Track
ID
Mgmt
User Interface Services
Common Labels, Data Items, and Symbology
Common Controls and Display Formats
Common, Adaptable Look and Feel
Common API's and algorithms;
Common implementation were coordinated upgrades essential?
Application Program Interfaces
Infrastructure
RT Data Manage-ment
RT Distributed Computing Infra-structure
Tactical Data Link Proto-cols
Common Data Management API
Common data items
Common Distributed Computing API
Interoperable Products
Common Implementation for interoperability, easy, consistent upgrade
Application Program Interfaces
DII COE Kernel for RT
Common API, Portability I/F

Figure 1. *Vision architecture of DII COE for real-time.*

computing configurations of the target system.

Since real-time applications often need a very efficient operating system with small memory footprint for performance reasons, the design philosophy of the DII COE RT Kernel allows a system integrator to tailor the RTOS to meet system needs[5]. Portable Operating System Interface (POSIX®) APIs for operating system services, including APIs for threads and real-time extensions specified in [3], form part of the RT Kernel API. Each DII COE RTOS will be rated for its ability to provide key functional units associated with real-time profiles in the POSIX.13 standard [4]. The dependencies of DII COE RT segments on other DII COE segments and services and on RTOS units of functionality will be documented during the segmentation of a RT software component for the DII COE. A designer of a real-time system can match system requirements to choices of DII COE applications, infrastructure, kernel services, and RTOS.

POSIX is a registered trademark of the Institute of Electrical and Electronics Engineers.

A real-time infrastructure lies above the RT Kernel to provide services for information handling. To support predictable end-to-end execution of system real-time activities, the RT infrastructure must be aware of priority and timing constraints. In the near term, the vision architecture includes a Common Object Request Broker Architecture (CORBA)-based distributed computing infrastructure for real-time. Common implementations of military communications protocols, now available in the DII COE, also must be ported or adapted as necessary for the real-time mission.

It also is envisioned that real-time data management, multi-level trust, and real-time management services will emerge as infrastructure capabilities. Therefore, the infrastructure must be real-time and standards-based to promote interoperability between computing configurations and reuse of applications across systems.

Interoperability and flexibility are key issues at the level of common support applications. Real-time exchange of theater situational awareness is a keystone of emerging concepts for network-centric warfare. The DII COE vision for real-time embraces a common interpretation of information communicated. Beyond the infrastructure components required for basic communication between computers and systems, DII COE is expected to include other applications that contribute to a common understanding of and response to the operational situation: track management, correlation, combat identification, and fusion of sensor data across systems.

Where feasible, components of today's DII COE will migrate to real-time platforms. In other cases, the real-time environment will provide standard access mechanisms through which real-time components may "reach back" to access nonreal-time functions executing on nonreal-time platforms. In other circumstances, new capabilities may be added for the real-time domain.

The Domain of DII COE for Real-Time

The range of real-time systems runs from small, tightly coupled embedded controllers to large-scale data-intensive tracking and control systems. It is reasonable to ask which of these systems should use DII COE for real-time. The overriding requirement for C2 interoperability in real-time drives the initial definition of the domain of DII COE for real-time. Anytime a real-time system needs to interoperate with other computing systems for effective execution of C2, it becomes a candidate for DII COE compliance assessment. When the system's interactions involve time-critical calculations or the exchange of time-critical data (activities that have real-time constraints), DII COE for real-time is a concern.

DII COE compliance in real-time systems, however, does not imply that every computer of a real-time system must achieve compliance in the same

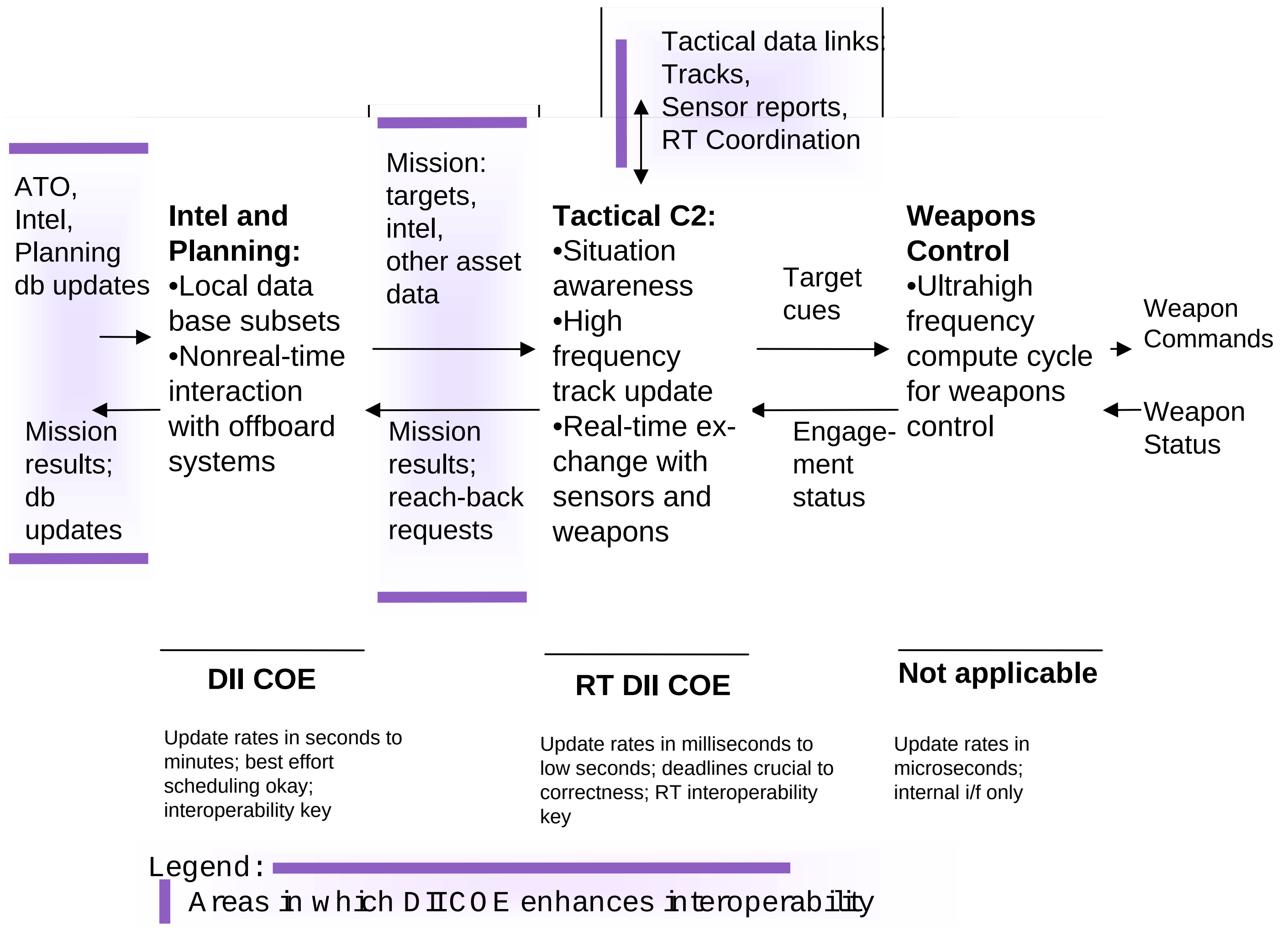
Tactical data links: Tracks, Sensor reports, RT Coordination
ATO, Intel, Planning db updates
Intel and Planning:
•Local data base subsets
•Nonreal-time interaction with offboard systems
Mission: targets, intel, other asset data
Tactical C2:
•Situation awareness
•High frequency track update
•Real-time exchange with sensors and weapons
Target cues
Weapons Control
•Ultrahigh frequency compute cycle for weapons control
Weapon Commands
Weapon Status
Mission results; db updates
Mission results; reach-back requests
Engagement status
DII COE
Update rates in seconds to minutes; best effort scheduling okay; interoperability key
RT DII COE
Update rates in milliseconds to low seconds; deadlines crucial to correctness; RT interoperability key
Not applicable
Update rates in microseconds; internal i/f only
Legend:
Areas in which DII COE enhances interoperability

Figure 2. *Breakdown for typical weapons C2 system: DII COE (non-RT), DII COE RT, and exempt.*

way. DII COE-compliant real-time systems will often be comprised of a distributed computing system in which some computers execute the existing nonreal-time DII COE, others are DII COE RT platforms, and exceptions from DII COE compliance are made for others.

Exceptions to DII COE compliance will be made for computing nodes that perform specialized tasks such as signal processing, which typically use specialized hardware and operating systems to achieve their goal. Likewise, DII COE-compliance requirements will not be applied to computer processing units embedded in hardware line replaceable units like network interface cards, controller cards for other computing peripherals, and other controllers that are tightly integrated with commercial or military hardware devices.

Guidelines for assessing how DII COE compliance requirements should be applied in a system are available in [5]. Using this procedure, each computer in a system is uniquely classified as DII COE (non-RT), DII COE RT, or exempt. Figure 2 depicts the allocation that might result in a typical weapons C2 system with external interfaces to other C2 and weapons systems.

Real-Time Capabilities in DII COE 5.0

In DISA Release 5.0, scheduled for October 2000, the following real-time capabilities will be available for incorporation into military C2 platforms:

1. a configurable DII COE RT Kernel for Lynx Operating System (LynxOS™) and Sun Solaris™;
2. a CORBA product with extensions for real-time; and
3. support tools to aid users in developing DII COE RT segments and building customized DII COE-compliant configurations for real-time.

Configurable RT Kernel

As noted earlier, the DII COE Configurable RT Kernel has two parts: an RTOS with POSIX application program interfaces and selectable DII COE Kernel services for real-time.

LynxOS was chosen for the reference implementation for real-time. It provides determinism for hard real-time execution, supports the full range of units of functionality defined in the POSIX.13 standard for real-time profiles, and has the sponsorship of system program offices for several current weapon system development programs. LynxOS provides a solid foundation to support real-time software applications in the DII COE.

LynxOS is a trademark of Lynx Real-time Systems Inc. Sun and Solaris are trademarks of Sun Microsystems Inc.

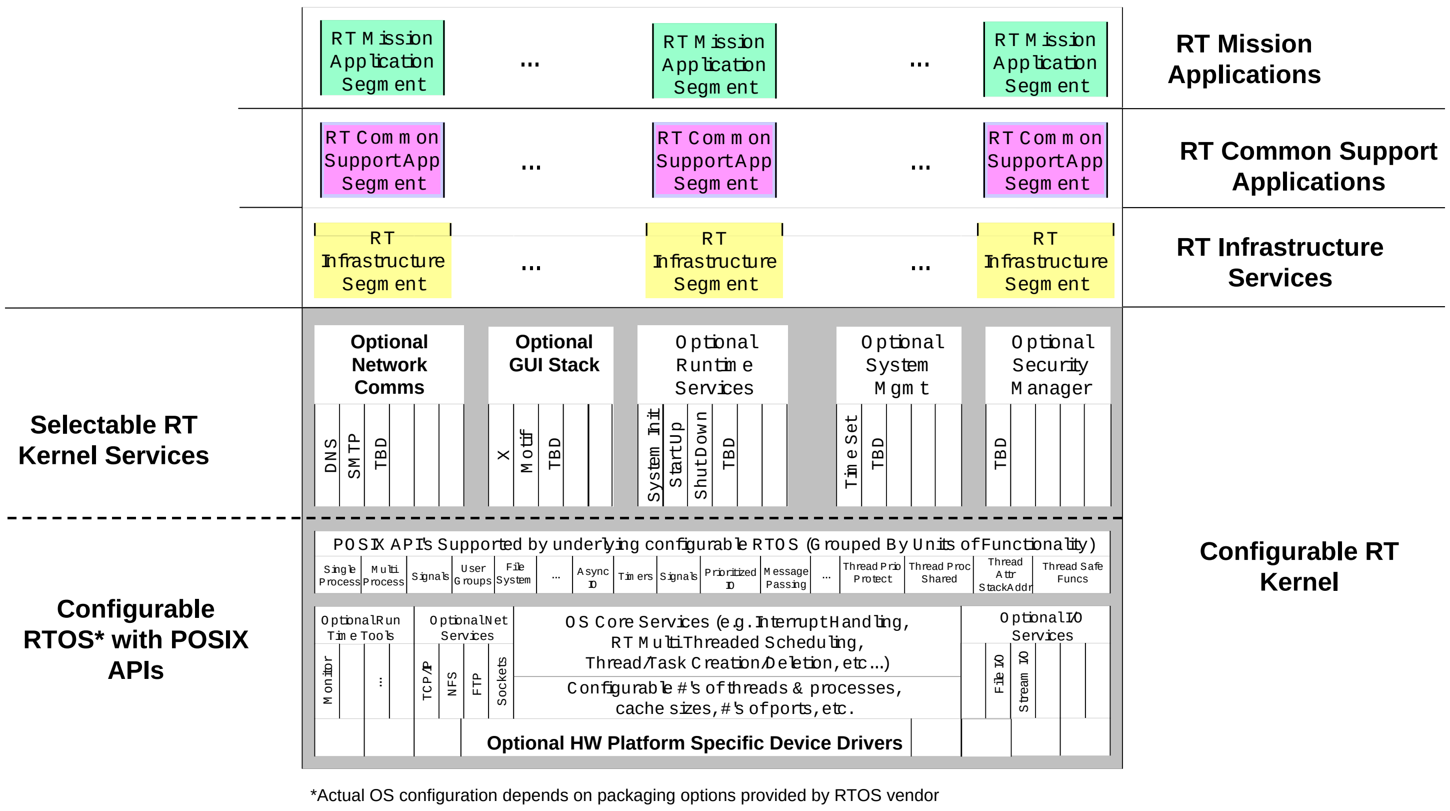
RT Mission Application Segment
...
RT Mission Application Segment
...
RT Mission Application Segment
RT Mission Applications
RT Common Support App Segment
...
RT Common Support App Segment
...
RT Common Support App Segment
RT Common Support Applications
RT Infrastructure Segment
...
RT Infrastructure Segment
...
RT Infrastructure Segment
RT Infrastructure Services
Selectable RT Kernel Services
Optional Network Comms
DNS
SMTP
TBD
Optional GUI Stack
X
Motif
TBD
Optional Runtime Services
System Init
StartUp
ShutDown
TBD
Optional System Mgmt
Time Set
TBD
Optional Security Manager
TBD
Configurable RTOS* with POSIX APIs
Configurable RT Kernel
POSIX API's Supported by underlying configurable RTOS (Grouped By Units of Functionality)
Single Process
Multi Process
Signals
User Groups
File System
...
Async IO
Timers
Signals
Prioritized IO
Message Passing
...
Thread Prio Protect
Thread Proc Shared
Thread Attr StackAddr
Thread Safe Funcs
Optional Run Time Tools
Monitor
...
Optional Net Services
TCP/IP
NFS
FTP
Sockets
OS Core Services (e.g. Interrupt Handling, RT Multi Threaded Scheduling, Thread/Task Creation/Deletion, etc ...)
Configurable #'s of threads & processes, cache sizes, #'s of ports, etc.
Optional I/O Services
File I/O
Stream I/O
Optional HW Platform Specific Device Drivers
*Actual OS configuration depends on packaging options provided by RTOS vendor

Figure 3. *Reference architecture for configurable RT Kernel.*

The RT Kernel services to be provided in the initial release 5.0 of DII COE for real-time are 1) commercial off-the-shelf products for X, Motif, and Domain Name Server, and 2) government off-the-shelf services for system startup and shutdown, setting system time, and starting and stopping DII COE processes. These services are documented in [6]. Figure 3 depicts a representative tailoring of the RT Kernel through selection of kernel services and operating system capabilities.

CORBA Infrastructure for Real-Time

CORBA is an international standard [7] for distributed computing that is governed by the object management group (OMG). The CORBA standard provides for flexible interconnection of objects in a client/server model for distributed computing. Four of CORBA's key objectives are support for location independence, operating system independence, hardware independence, and language independence in the design of software components. Figure 4 shows the role that an object request broker (ORB), appropriately extended for real-time, plays in integrating independently developed components into a flexible real-time architecture.

Additions to the CORBA standard to enable real-time computing with end-to-end predictability are documented in the Real-Time CORBA Joint Revised Submission [8], which the OMG is considering adopting. These extensions allow for associating real-time priorities with tasks and requests, passing priority information between communicating components, and the expressing and monitoring of timing constraints for requests. The proposed RT CORBA specification also defines a scheduling service that will provide a consistent real-time scheduling model across a CORBA-based system.

HARDPack by Lockheed-Martin Federal Systems (LMFS) is the leading candidate as the initial RT ORB. HARDPack is a commercial ORB that supports Ada, C, and C++ and includes extensions for real-time performance. HARDPack is cognizant of real-time request priorities and provides the capability to associate deadlines with requests. It extends the CORBA standard with reliable and unreliable broadcast and multicast capabilities, features commonly used for efficient communications in real-time C2 systems. HARDPack also implements the Encapsulated Scheduler to assist in implementing real-time scheduling.

HARDPack is used on at least two Air Force C2 programs: AWACS and Region/Sector Air Operations Center (R/SAOC). Because the RT CORBA standard has not yet been formalized, the real-time extensions to HARDPack are proprietary. LMFS actively participates in standardization and is committed to align its product with the commercial standard within a year of its adoption.

Including CORBA in the DII COE infrastructure for real-time enables the construction of components that can be used in a variety of configurations in different systems. However, it is imperative that components are properly designed to take advantage of this flexibility. When it is reasonable to expect that a given application component will not always execute on the same central processing unit with another component with which it interacts, the component(s) should be designed in such a way that introducing a network between the components can be tolerated[6]. In general, this consideration will drive designs toward relatively large-grained components with coarse-grained

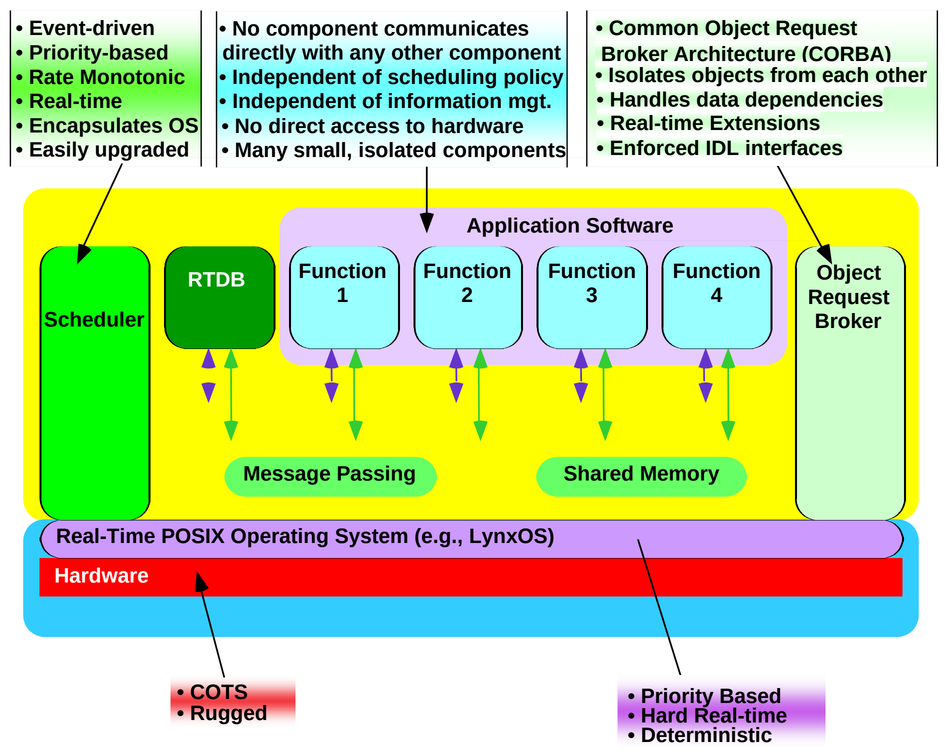
• Event-driven
• Priority-based
• Rate Monotonic
• Real-time
• Encapsulates OS
• Easily upgraded
• No component communicates directly with any other component
• Independent of scheduling policy
• Independent of information mgt.
• No direct access to hardware
• Many small, isolated components
• Common Object Request Broker Architecture (CORBA)
• Isolates objects from each other
• Handles data dependencies
• Real-time Extensions
• Enforced IDL interfaces
Application Software
Scheduler
RTDB
Function 1
Function 2
Function 3
Function 4
Object Request Broker
Message Passing
Shared Memory
Real-Time POSIX Operating System (e.g., LynxOS)
Hardware
• COTS
• Rugged
• Priority Based
• Hard Real-time
• Deterministic

Figure 4. *RT CORBA infrastructure.*

interactions rather than designs involving fine-grained interactions between distributed objects.

Build-Time Integration and Supporting Tools

With a goal of constructing a real-time system with predictable performance and reliable behavior, the integration of real-time segments will take place in the integration laboratory using build-time tools rather than run-time plug-and-play installation. The functions of configuration, link, load, and test in the laboratory will need to be performed using new tools with the DII COE inventory. These Build-Time Tools assist the systems engineer to accomplish several functions in series:

1. select the complete list of DII COE real-time segments for the target environment
2. analyze the inter-segment dependencies
3. choose the selectable RT Kernel Services required by the segments
4. analyze the inter-kernel dependencies
5. and select the required POSIX units of functionality to be provided by the RTOS. The product of the tools is a list of components that must be configured to provide the required functional capabilities. The RT Kernel can then be configured using commercial development tools. The specification for the Build-Time Tools appears in [9].

The Future of DII COE for Real-Time

This effort is on the ground floor to provide a solid foundation on which to build future real-time capabilities in the DII COE. A great deal of work remains to enable widespread reuse of DII COE RT application software; most critical is the establishment of an architecture at the application program interface level. In addition, further requirements analysis and real-time product nominations will be done in the following DII COE functional areas: management services, multi-level trust, mapping, alerts, correlation, message processing, data management, track management, combat identification, and communications. System program offices are being sought out which already utilize software products built with an open architecture and POSIX conformance approach that meet the real-time requirements for C2 interoperability. The goal is to improve C2 interoperability for weapon platforms as the DoD reaches toward the goal of Joint Vision 2010. ◆

About the Authors

Lt. Col. Lucie Robillard is the Air Force Executive Agent for Real-Time DII COE at Hanscom AFB, Mass. She is chairwoman for the DII COE real-time integrated product team (IPT) that has the charter to make real-time extension to DII COE a reality for all services. She is a Level 3 certified acquisition professional. She has joint assignment experience. A majority of her assignments have dealt with software acquisition and engineering. She has a bachelor's degree in electrical engineering from the University of Vermont and a master's degree in systems management from University of Southern California.

ESC/AWW
3 Eglin St.
Hanscom AFB, Mass. 01730
Voice: 781-377-2679
Fax: 781-377-1069
E-mail: robillardl@hanscom.af.mil
Internet: http://spider.osfl.disa.mil/dii/aog_twg/twg/DISAWEB.HTML

Dr. H. Rebecca Callison leads the Boeing team supporting Lt. Col. Robillard and the DII COE real-time IPT. She has 25 years of experience in the design and implementation of real-time systems, principally in the area of defense systems. She has a bachelor's degree from the University of South Carolina, a master's degree in computer science/systems analysis/design from the University of Pennsylvania, and a doctorate degree from the University of Washington. She has served on the faculty of Oregon State University. She has research interests in the areas of software architectures for real-time systems and concurrency control for real-time.

The Boeing Co.
20403 68th Avenue S.
Kent, Wash. 98032
Voice: 253-657-3952
Fax: 253-657-0505
E-mail: rebecca.callison@boeing.com

John Maurer leads MITRE's real-time and performance engineering section. Maurer also chairs the DII COE real-time technical working group. He has a bachelor's degree in mechanical engineering from MIT and 24 years experience implementing software-intensive DoD systems. His work experience includes real-time system development for airborne surveillance systems and Army vehicle systems.

The MITRE Corp.
202 Burlington Road
Bedford, Mass. 01730
Voice: 781-271-2985
Fax: 781-271-4686
E-mail: johnm@mitre.org
Internet: http://spider.osfl.disa.mil/dii/aog_twg/twg/rttwg/rttwg_page.html

References

1. DISA, Defense Information Infrastructure (DII) Common Operating Environment (COE) Baseline Specification, version 3.1, April 29, 1997, DISA Joint Interoperability and Engineering Organization, Reston, Va.
2. Klein, Mark H. et al., *A Practitioner's Handbook for Real Time Analysis*, Kluwer Academic, ISBN 0-7923-9361-9.
3. Information Technology — Portable Operating System Interface (POSIX) Part 1 — System Application Program Interface (API) [C Language], ISO/IEC 9945-1:1996 (E) ANSI/IEEE Std. 1003.1.
4. Draft Standard for Information Technology — Standard Application Environment Profile — POSIX Realtime Application Support (AEP), P1003.13 Draft 9, September 1997.
5. DII COE RT TWG, "DII COE Real-time Decision Tree & Assessment Process: Deciding What's in the Domain," Jan. 20, 1999, http://spider.osfl.disa.mil/dii/aog_twg/twg/RTASSESS.html.
6. DII COE RT TWG, Software Requirements Specification for Kernel Services for the Real-Time Defense Information Infrastructure Common Operating Environment (RT DII COE), (Draft) Revision 1.0, Jan. 8, 1999.
7. Object Management Group, The Common Object Request Broker: Architecture and Specification, Revision 2.2, February 1998. (http://www.omg.org/corba/corbaiiop. html)
8. Object Management Group, Real-Time CORBA 1.0: Joint Revised Submission, Dec.10, 1998. (http://www.omg.org/techprocess/meetings/schedule/Realtime_CORBA1.0_RFP.html)
9. DII COE RT TWG, Build Time Tools Use Case Specification for Real-Time Extensions to the Defense Information Infrastructure Common Operating Environment (RT DII COE), (Draft) Revision 1.0, Jan. 15, 1999.

Notes

1. We use the term "activity" to capture the notion that a time-constrained operation of the system may include computation, communication, and other input/output and may traverse multiple computing nodes.
2. A discussion of real-time scheduling algorithms is outside the scope of this paper. The interested reader is referred to [2] for an overview of available scheduling techniques.
3. The degree to which these techniques must be avoided depends somewhat on the type of real-time system for which the component is intended and the situation in which the feature will be used. If it must perform acceptably in hard real-time systems, the ban on features with loosely bound performance must be strict. If soft real-time is the objective; the restrictions may be relaxed with caution.
4. In the rest of this paper, we will use the term "RT Kernel" as an abbreviation for "DII COE Configurable RT Kernel."
5. Commercial-off-the-shelf (COTS) tools provided by the operating system (OS) vendor are used to configure the OS, not DII COE unique software. The degree to which a specific RTOS can be configured depends on the flexibility the RTOS vendor provides.
6. As in all other situations in which a component is asked to adapt to a new computing environment, it is assumed that developers will not attempt to use the component in any way that violates the laws of physics.

CROSSTALK Wants to Hear from You

We welcome reader comments regarding *CROSSTALK* articles or matters pertaining to software engineering. Please send your comments and Letters to the Editor to crosstalk@stsc1.hill.af.mil or mail to

OO-ALC/TISE
Attn: CROSSTALK staff
7278 Fourth Street
Hill AFB, UT 84056-5205

Please limit letters to less than 250 words. Include your name, phone number, and e-mail address with any letter. We will withhold your name if you desire.

The following article can be found in its entirety on the Software Technology Support Center Web site at http://www.stsc.hill.af.mil/CrossTalk/crostalk.html. Go to the “Web Addition” section of the table of contents.

Overview of the DII COE 4.0 Kernel

Sherrie Chubin, *DISA*
Dr. Thomas I. McVittie, *JPL*
Robert B. Miller, *JPL*

On April 2, the Defense Information Systems Agency (DISA) released version 4.0 of the Defense Information Infrastructure Common Operating Environment (DII COE). Since the release is only eight months away from the millennium, fielding systems on this new version of the COE becomes prohibitive because of the amount of time it takes to complete year 2000 testing. As a result, COE 4.0 was released as a “developer’s release” or “beta release” so that the services and agencies have ample time to become familiar with the new version and to provide DISA with problem reports. By releasing COE 4.0 as a “beta release,” more developers will be able to provide input to DISA to help to build a stable COE 4.1, scheduled for release in October 1999.

With the 4.0 release, a modified kernel architecture and many functional enhancements to COE provided the software dominate improvements found in this new version. The 4.0 kernel incorporates a number of new items that improve performance and provide for greater flexibility in configuring and deploying DII COE-compliant systems. These items require changes to how developers construct future segments, and how integrators, site administrators, and security managers interact with the system. 3.x segment formats will continue to be supported in the 4.x series, however, the new 4.0 kernel does provide new capabilities that a developer can opt to take advantage of for future segment development.

This paper provides an overview of the 4.0 kernel changes specifically addressing 1) account and profile management, 2) common data store, 3) services, 4) features, and 5) bindings. The amount of detail that is presented here is intentionally kept to a minimum so that the reader becomes familiar with the changes made and why they were made, not necessarily how to write software that relies on kernel services. Detailed documentation is available with the COE 4.0 release. ◆

Performing Verification and Validation in Architecture-Based Software Engineering

Edward A. Addy
Logicon Advanced Technology

Verification and validation (V&V) now is performed during application development for many systems, especially safety-critical and mission-critical systems. The application system provides the context under which the software artifacts are validated. This article describes a framework that extends V&V from an individual application system to a product line of systems that are developed within an architecture-based software engineering environment. The product line architecture provides the context for evaluation in this approach.

Introduction

The implementation of architecture-based software engineering not only introduces new activities to the software development process, such as domain analysis and domain modeling, it also impacts other activities of software engineering including configuration management, testing, quality control, and verification and validation. Activities in each of these areas must be adapted to address the entire domain or product line rather than a specific application.

V&V methods are used to increase the level of assurance of critical software, particularly that of safety-critical and mission-critical software. Software V&V is a systems engineering discipline that evaluates software in a systems context [1]. The V&V methodology has been used in concert with various software development paradigms, but always in the context of developing a specific application system. However, an architecture-based software development process separates domain engineering from application engineering in order to develop generic reusable software components that are appropriate for use in multiple applications.

The earlier a problem is discovered in the development process, the less costly it is to correct the problem. To take advantage of this, V&V begins verification within system application development at the concept or high-level requirements phase. However, an architecture-based software development process has tasks that are performed earlier — possibly much earlier — than high-level requirements for a particular application system. In order to bring the effectiveness of V&V to bear within an architecture-based software development process, V&V must be incorporated within the domain engineering process.

On the other hand, it is not possible for all V&V activities to be transferred into domain engineering, since verification extends to the installation and operation phases of development, and validation is primarily performed using a developed system. This leads to the question of which existing and/or new V&V activities would be more effectively performed in domain engineering rather than in — or in addition to — application engineering. Related questions include how to identify the reusable components for which V&V at the domain level would be cost-effective, and how to determine the level to which V&V should be performed on the reusable components.

Differences Between V&V and Component Certification

Much work has been done in the area of component certification, which is also called evaluation, assessment, or qualification. These terms can have slightly different meanings, but refer in general to rating a reusable component against a specified set of criteria. Reuse libraries often use levels to indicate the degree to which the library has evaluated a component. The Asset Source for Software Engineering Technology (ASSET) library and the Army Reuse Center library both have four levels of certification, although the use of the term “levels” is operationally different in the two libraries [2]. Component-based libraries evaluate reusable components against criteria such as reusability, evolvability, maintainability, and portability, as well as expending various levels of effort to ensure the component meets its specification. Other schemes for component certification include the certification framework developed by the Certification of Reusable Software Components Program at Rome Laboratory [3], and the suitability testing performed by the National Product Line Asset Center on behalf of the Air Force Electronic Systems Center [4].

The common thread through all of these certification processes is the focus on the component rather than on the systems in which the component will eventually be (re)used. Michael Dunn and John Knight [5] note that with the exception of the software industry, customers purchase systems and not components. Ensuring that components are well designed and reliable with respect to their specifications is necessary, but not sufficient, to show that the final system meets the needs of the user. Component evaluation is but one part of an overall V&V effort, analogous to code evaluation in V&V of an application system.

Another distinction between V&V and component certification is the scope of the artifacts that are considered. While component certification is primarily focused on the evaluation of reusable components (usually code-level components), V&V also considers the domain model and the generic architecture, along with the connections between domain artifacts and application system artifacts. Some level of component certification should be performed for all reusable components, but V&V is not always appropriate. V&V should be conducted at the level determined by an overall risk mitigation strategy.

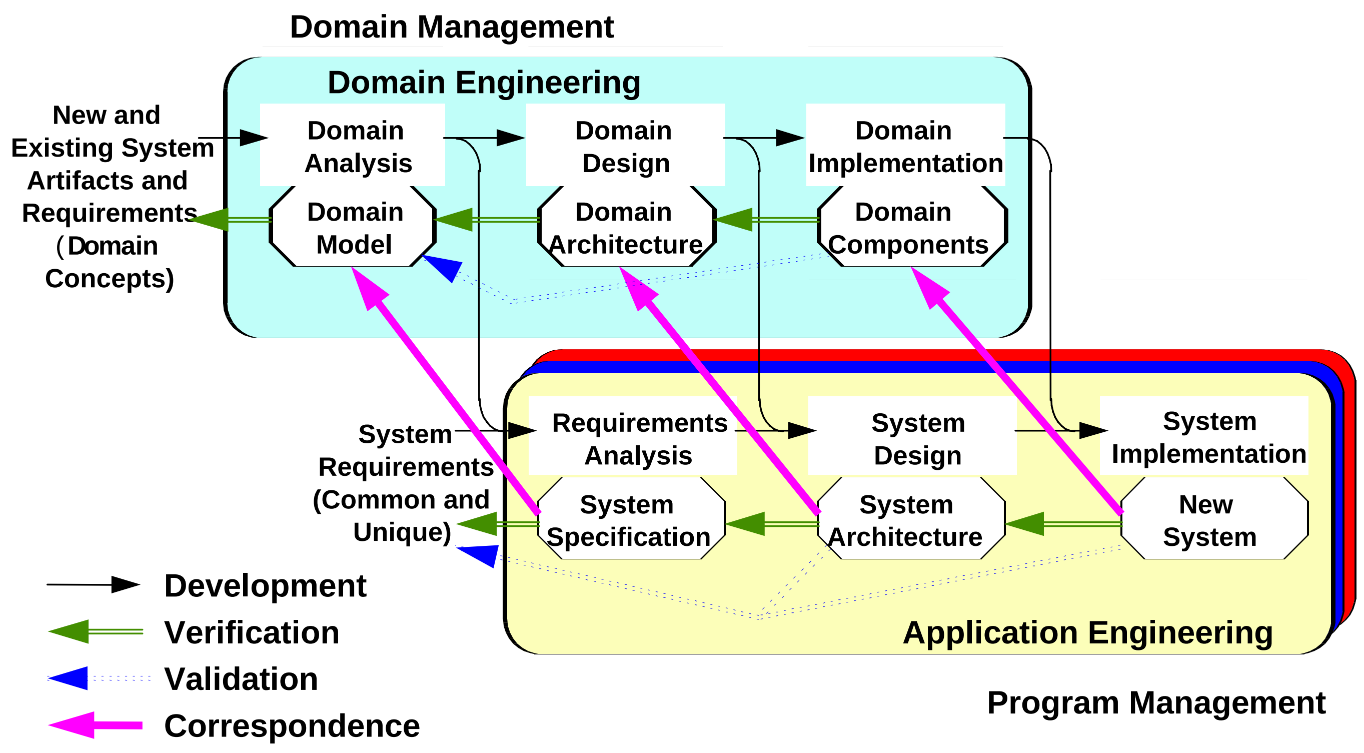

Figure 1. *Framework for V&V in architecture-based software engineering.*

Framework for Performing V&V within Architecture-based Software Engineering

A draft framework for performing V&V within architecture-based software engineering is formed by adding V&V activities to a two life cycle model involving both domain engineering and application engineering. The application-level V&V tasks described in IEEE STD 1012 [6] serve as a starting point. Domain-level tasks are added to link life cycle phases in the domain level, and transition tasks are added to link application phases with domain phases. This draft framework was refined by a working group at Reuse '96 [7], and the resultant framework is shown in Figure 1.

Domain-level V&V tasks are performed to ensure that domain products fulfill the requirements established during earlier phases of domain engineering. Transition-level tasks provide assurance that an application artifact correctly implements the corresponding domain artifact. Traditional application-level V&V tasks ensure the application products fulfill the requirements established during previous application life cycle phases. More details on the framework than allowed by the space of this article can be found in [8].

Performing V&V tasks at the domain and transition levels will not automatically eliminate any V&V tasks at the application level. However, reduction in the level of effort for some application-level tasks might be possible. The reduction in effort could occur in a case where the application artifact is used in an unmodified form from the domain component, or where the application artifact is an instantiation of the domain component through parameter resolution or through generation. Domain maintenance and evolution are handled in a manner similar to that described in the operations and maintenance phase of application-level V&V. Changes proposed to domain artifacts are assessed to determine the impact of the proposed correction or enhancement. If the assessment determines that the change will impact a critical area or function within the domain, appropriate V&V activities are repeated to assure a correct implementation.

Although not shown as a specific V&V task for any particular phase of the life cycle, criticality analysis is an integral part of V&V planning. Criticality analysis is performed in V&V of application development in order to allocate V&V resources to the most important (i.e. critical) areas of the software [9]. This assessment of criticality and the ensuing determination of the level of intensity for V&V tasks also are crucial within architecture-based software engineering. Not all domain products will be used in critical application systems, and some of those used in critical application systems may not be in a critical area of the software. Some reusable components may be used in multiple systems, but may be a part of the critical software in only one or two of the systems. V&V should be performed only on domain products that are involved in the critical software in one or more application systems, and V&V tasks should be performed at a level of intensity appropriate to the level of criticality.

Determining the domain products for which to perform V&V, and the appropriate level of intensity for the V&V tasks, is complicated by the use of the products in multiple systems, some of which may only be in early stages of planning. If a component is used in only one critical application system, it may be more cost-effective to perform V&V during application engineering for that system rather than during domain engineering. Extension of criticality analysis from application engineering to domain engineering is an important area of this framework.

V&V of Domain Artifacts

Many of the same justifications for performing V&V in a product line that includes critical systems also apply to V&V of general purpose reusable components. These general purpose components include domain artifacts for systems that are not critical, as well as reusable components that are developed for general usage rather than for a specific product line. The Component Verification, Validation, and Certification Working Group at WISR 8 found four considerations that should be used in determining the level of V&V of reusable components [10]:

- Span of application — the number of components or systems that depend on the component
- Criticality — potential impact due to a fault in the component
- Marketability — degree to which a component would be more likely to be reused by a third party
- Lifetime — length of time that a component will be used

The domain architecture serves as the context for evaluating software components in a product-line environment. However, this architecture may not exist for general use components. The working group determined that the concept of validation was different for a general use component than for a component developed for a specific system or product line. In the latter case, validation refers to ensuring that the component meets the needs of the customer. A general use component has not one customer, but many customers, who are software developers rather then end-users. Hence validation of a general use component should involve the assurance — and supporting documentation — that the component satisfies a wide range of alternative usages, rather than the specific needs of a particular end-user.

Related Work

Although work is lacking specifically in the area of V&V as applied to architecture-based software engineering, there is related work that is applicable to some of the tasks within the framework. Component certification was discussed in a previous section, and this work is certainly applicable (although not sufficient) for V&V activity at the domain level. The analysis of architectures is the focus of attention and discussion [11, 12], but there is not as yet consensus on methods and approaches and much of this work is directed toward system architectures rather than product line architectures. One of the approaches being researched is a scenario-based analysis approach, Software Architecture Analysis Method [13]. In the area of correspondence tasks, the Centre for Requirements and Foundations at Oxford is developing a tool (TOOR) to support tracing dependencies among evolving objects [14].

Future Work

An initial, high-level framework for performing V&V in architecture-based software engineering has been developed. Once completed, this framework will allow the V&V effort to be amortized over the systems within a domain or product line. However, this framework is an outline with few details. V&V tasks that now are performed at the application level need to be adapted for the domain level, and traceability tasks need to be adapted for the transition level. New methods not used on applications but appropriate for domain models or architectures need to be considered. Since V&V should be performed as part of an overall risk mitigation strategy within the domain or product line, methods of domain criticality analysis need to be developed, with attention paid to support from emerging architecture description languages. The methods identified need to be validated by use in projects having an architecture-based software engineering approach to producing applications that require V&V. ◆

About the Author

Edward A. Addy is currently a project manager with Logicon Advanced Technology. The work on which this article is based was done while Addy was a research associate with the NASA/WVU Software Research Laboratory, a cooperative effort between West Virginia University and the NASA Ames Software IV&V Facility in Fairmont, W. Va. His research interests are in the areas of IV&V, software product lines, component-based software reuse, software safety, and risk analysis. Addy is a doctoral candidate in computer science at West Virginia University.

Logicon Advanced Technology
2003 Apalachee Parkway
Suite 211
Tallahassee, Fla. 32301
Voice: 850-219-8033
Fax: 850-219-8034
E-mail: eaddy@logicon.com
Internet: http://research.ivv.nasa.gov/~eaddy

References

1. Wallace, Dolores R. and Roger U. Fujii, "Software Verification and Validation: Its Role in Computer Assurance and Its Relationship with Software Project Management Standards." NIST Special Publication 500-165, National Institute of Standards and Technology, Gaithersburg, Md. 1989.
2. Poore, J.H., Theresa Pepin, Murali Sitaraman, and Frances L. Van Scoy, "Criteria and Implementation Procedures for Evaluating Reusable Software Engineering Assets." DTIC AD-B166803, prepared for IBM Corporation Federal Sectors Division, Gaithersburg, Md. 1992.
3. Software Productivity Solutions Inc., "Certification of Reusable Software Components, Vol. 2 - Certification Framework." Prepared for Rome Laboratory/C3CB, Griffiss AFB, N.Y. 1996.
4. Unisys, Valley Forge Engineering Center, and EWA Inc., "Component Provider's and Tool Developer's Handbook." STARS-VC-B017/001/00, prepared for Electronic Systems Center, Air Force Material Command, USAF, Hanscom AFB, Mass. 1994.
5. Dunn, Michael F. and John C. Knight, "Certification of Reusable Software Parts." 1993 Technical Report CS-93-41, University of Virginia, Charlottesville, Va.

6. IEEE STD 1012-1986 (R 1992), IEEE Standard for Software Verification and Validation Plans, Institute of Electrical and Electronics Engineers, Inc., New York, N.Y.
7. Addy, Edward A., "V&V Within Reuse-Based Software Engineering." In proceedings of the Fifth Annual Workshop on Software Reuse Education and Training, Reuse '96, http://www.asset.com/WSRD/conferences/proceedings/results/addy/addy.html. 1996.
8. Addy, Edward A., "A Framework for Performing Verification and Validation in Reuse-Based Software Engineering." *Annals of Software Engineering*, Vol. 5, 1998.
9. IEEE STD 1059-1993, IEEE Guide for Software Verification and Validation Plans, Institute of Electrical and Electronics, Inc., New York, N.Y.
10. Edwards, Stephen H. and Bruce W. Wiede, "WISR8: 8th Annual Workshop on SW Reuse." Software Engineering Notes, 22, Sept. 5, 1997, pp 17-32.
11. Tracz, Will, "Test and Analysis of Software Architectures." In proceedings, International Symposium on Software Testing and Analysis (ISSTA '96), ACM Press, New York, N.Y, pp 1-3, 1996.
12. Garlan, David, "First International Workshop on Architectures for Software Systems Workshop Summary." Software Engineering Notes, 20, July 3, 1995, pp 84-89.
13. Kazman, Rick, Gregory Abowd, Len Bass, and Paul Clements, "Scenario-Based Analysis of Software Architecture." IEEE Software, 13, Nov. 6, 1996, pp 47-55.
14. Goguen, Joseph A., "Parameterized Programming and Software Architecture." In proceedings of the Fourth International Conference on Software Reuse, IEEE Computer Society Press, Los Alamitos, Calif., pp 2-10, 1996.

A Y2K Integration Test Model

Dr. William H. Dashiell
National Imagery and Mapping Agency

An Integration Test Model provides Year 2000 (Y2K) integration test objectives keyed to specific integration test cases. This article describes a suggested basic year 2000 (Y2K) test model with lessons learned.

Background

The Y2K problem centers on the interpretation of a two-digit code representation of a year. Simplified, the Y2K problem may be seen as many computer software applications that have been programmed to interpret the two-digit year range of 00 ... 99 as meaning the years ranging from 1900 to 1999. Other software applications interpret 99 as an end-of-file or end-of-data marker. Regardless of how the problem is defined, one will interpret it as catastrophic when one's critical need is not addressed or addressed incorrectly — sometimes with irretrievable results.

Because of costs or lack of programmer resources to correct the source code (assuming that the source code is available and that an executable image may be generated) the basic quick fixes for the Y2K problems often cause two effects:

1. delaying the impact of the Y2K problem by using techniques to interpret various ranges of dates such as bridging, sliding, or fixing windows.
2. exchanging what is essentially either a single date or a well-defined collection of dates for multiple dates with unknown impacts.

Regardless of the "fix" used for the Y2K problem, formal Y2K testing to a set of Y2K objectives should be done to assess the level of risk to which the users of those applications are exposed.

Unit Testing vs. Integration (Interoperability) Testing

In more traditional software testing environments, unit testing of an application (e.g. software using services and facilities provided by an information system specific to the satisfaction of a set of user requirements) usually involves the testing of a module within a larger, whole software application entity. In the context of this article, unit testing of an application involves the whole of a single software application entity. The software application may perform specified function(s) within the computer system. Unit testing usually is performed on a hardware platform (e.g. a collection of hardware and software components that provides the services used by support and mission-specific software applications) with or without other software programs being visible, such as an operating system. Note that the hardware platform may include a client/server-based system or a Web-based system and their respective required software to enable each system to function as designed. Unit testing is not integration testing but is generally performed prior to integration testing.

Again, in more traditional software testing environments, integration testing usually involves the testing of the aggregate of the modules comprising the whole of the software application entity. In this article, integration testing is defined as an orderly testing of each of the pieces of the software applications, as defined by the user or the system specifications, in which software applications, hardware elements, or both are combined and tested to show compliance with the program design, and capabilities and requirements of the system and/or the user's needs and uses. Integration testing is aimed at exposing problems that arise when two or more applications are combined on a hardware platform. As with unit testing, the hardware platform may include a client/server-based system or a Web-based system and its respective required software to enable each system to function as designed. Typical problems identified during integration testing are improper call or return sequences, inconsistent data validation criteria, or inconsistent handling of data objects. Integration testing generally is performed following successful unit testing or "software developer" integration testing of a collection of applications.

One important objective in software testing is the validation of the application(s) under test (e.g. those applications that are subject to testing requirements). Validation testing is a process of assessing the conformance of one or more software applications to one or more standards or to a set of specifications. This process includes the administrative procedures to set up conformance assessment and to issue some formal document, such as a certificate or test report, that an agreed upon or recognized process was followed and that records which of all tests presented were passed. For tests that were failed, the formal document notes which tests failed and specifies the functionality assessed. The user of the formal report documenting failed tests may find that the functionalities represented by the tests are not needed.

Why Perform Integration Testing?

The primary purpose of testing is to satisfy a customer's needs and requirements. Unit testing primarily assesses the validation of an application by itself. However, when multiple applications share resources, the closer the testing environment is to that of the customer's environment the more likely that testing will detect anomalies. By design, integration testing encompasses multiple applications and uses either the customer's environment or a separate test environment that closely duplicates the customer's environment.

Customers
Identify the customer when designing the integration test environment and test script. For example, consider the following four categories of customer sets:

- End-users — should be the highest support priority customer set.
- Operators — should be the second priority customer because they usually are individuals who are an integral part of the production process. This customer set often operates and manages data centers.
- Maintainers — are hardware, software, and network infrastructure personnel who maintain the operational systems and provide on-the-floor support to the system user community.
- Developers — the individual product developers, such as the project managers, technology thrust managers, and capability security certification administration.

Year 2000 Integration Testing
The Y2K integration testing is designed to ensure the continuing integrity of a user's base line and to provide customers and end users with continuing integrity of that base line. While performing all integration testing, integration testers should seek to provide operational acceptance with zero open discrepancy reports (DRs). The Y2K test scripts and test reports are designed to ensure that the reported test results are accurate and repeatable.

The Y2K Integration Test Model involves its customers and end-users in the integration testing process to ensure user acceptance as well as technical base line acceptance for newly delivered capabilities.

There are five basic phases in the Y2K Integration Test Plan. While these phases were implemented in the order presented below, the awareness phase is an ongoing phase because of software changes (e.g. through application of patches, new builds, request for new functionality(ies), and results of the renovation phase) that occur throughout an application's life cycle.

Awareness
In the awareness phase, all personnel responsible for the development, testing, or who use the information technology (IT) system have been educated about the importance and impact of Y2K problems.

Assessment
This phase requires that all IT components are first unit tested by a separate unit test group. Once the unit test group successfully tests a software component, that component is transferred to the Y2K integration testers, who perform the Y2K integration test scripts on the set of software/hardware components comprising the applications under test. When applications under test are not unit tested, this Integration Test Model suggests that integration testers should perform the integration test script with the knowledge that sources of errors may not be easily traced. The assessment phase includes a strategy and plan to correct the deficiencies with full regression testing of the applications under test.

Renovation
The renovation phase documents the software/hardware changes, obsolescence of software/hardware, and upgrades to software/hardware. (Renovation is performed by other activities or organizations.) Full regression testing of renovated applications is strongly recommended.

Validation
This phase describes the test and verification process for all IT software components possibly affected by the Y2K problem. All validation testing is designed to occur in an isolated testing environment wherein regression and future software integration testing may be performed without impact on operational production systems. Full regression testing of all replaced or converted system components should be done.

Baseline
The base line phase describes the operational base line of software wherein the newly tested software is integrated. A properly constructed baseline

- supports multiple control levels;
- provides for storage and retrieval of configuration items/units;
- provides for the sharing and transfer of configuration items/units between control levels within the library;
- provides for the storage and recovery of archival versions of configuration items/units;
- ensures correct creation of products from the software base line library;
- supports generation of reports; and,
- provides for the maintenance of the library structure.

Objectives
The primary objective is to ensure full regression testing of all software components for Y2K compliance.

In carrying out the integration testing responsibility, specific goals have been derived to govern the general operational testing procedures and particularly Y2K integration testing to:

- maintain a focused commitment to and support of the migration of legacy systems into a base line;
- identify and respond quickly to changing priorities;
- partner with your software system control personnel (e.g. executive decision makers) and your user community to ensure compatible, integrated test planning, scheduling, and execution to minimize the need for partial capability acceptance and retest;
- adhere to all of your software community standards, policies, and procedures;
- provide testing that ensures the continuing integrity of your operational base line;
- involve your customers and end-users to ensure user acceptance as well as technical base line acceptance for newly delivered capabilities;

Test Obj.#	Target Date to be Tested	Description
0.0	Current day, date, and time	Tests whether software properly processes current day, date, and time. A basis to start testing.
1.0	Saturday, Aug. 21, 1999 through Sunday, Aug. 22, 1999	Tests roll-over of the Global Positioning System (GPS) 10 bit epoch. Days are correctly recognized as Saturday and Sunday, respectively. *See GPS note*.
2.0	Wednesday, Sept. 8, 1999 through Thursday, Sept. 9, 1999	The numeric value of the day (999) is equal to the null void code sometimes used in programming. Day is correctly recognized as Thursday.
3.0	Thursday, Sept. 30, 1999 through Friday, Oct. 1, 1999	Tests critical roll-over of federal fiscal year 2000 roll-over. Days are correctly recognized as Thursday and Friday, respectively.
4.0	Friday, Dec. 31, 1999 through Saturday, Jan. 1, 2000	Critical midnight crossing from 1999 into the year 2000. Days are correctly recognized as Friday and Saturday, respectively.
5.0	Monday, Jan. 3, 2000	First day back to work for most employees after year 2000 begins. Day is correctly recognized as Monday.
6.0	Sunday, Jan. 9, 2000 through Monday, Jan. 10, 2000	Tests roll over from single digit days to double digit days in year 2000. Day is correctly recognized as Monday.
7.0	Tuesday, Feb. 29, 2000 through Wednesday, March 1, 2000	Tests critical roll-over of first leap day in the first leap year after year 2000 begins. Days are correctly recognized as Tuesday and Wednesday, respectively.
8.0	Saturday, Sept. 30, 2000 through Sunday, Oct.1, 2000	Tests roll-over from single digit month to double digit month in year 2000. Days are correctly recognized as Saturday and Sunday, respectively.
9.0	Sunday, Dec. 31, 2000 through Monday, Jan. 1, 2001	Critical midnight crossing from 2000 into 2001. Tests roll over to new millennium. Days are correctly recognized as Sunday and Monday, respectively. This date is the last day of the second millenium on the Gregorian calendar. The ordinal date 00.365 was the last day of 1900 (Julian Calendar). Since 2000 is a leap year, its last day is 00.366. An incomplete algorithm for determining the length of the year might cause an ordinal- based system to transition into the new millennium a day too early.
10.0	Sunday, Feb. 29, 2004 through Monday, March 1, 2004	Tests roll over from first leap year not affected by a century or millennium transition. Days are correctly recognized as Sunday and Monday, respectively. Julian date (sometimes called Ordinal Date) function should return Nth day of year.

Table 1. *Y2K test script dates.*

- ensure test scripts and test base lines are developed that can produce accurate and repeatable results in satisfying the test requirements;
- achieve scheduled testing deadlines established by the customer;
- proceed to operational acceptance with zero open DRs.

Scope of Y2K Integration Testing

As a first step, the integration tester is urged to test for proper processing of the current date and time prior to starting the Y2K test dates. The integration tester should be a software tester with professional experience who will review each test objective and decide its applicability to the applications under test and to modify those test objectives and test procedures to more properly match the functionality of the applications under test. This professional experience allows the tester to make professional judgements and evaluations based upon the test objective and his or her testing experiences.

The integration tester must provide an audit trail. The recommended methodology is to leave each test objective and procedure as written. In the test report, the tester should document each deviation from the objectives or procedures, with a rationale for each change.

Certain dates are widely recognized as among the most important in Y2K integration testing. These dates, which form the basis for the Y2K test script, are shown in Table 1.

Global Positioning System Note: Users of the Global Positioning System (GPS) should note that GPS does not have a Y2K problem. However, a clock overflow problem, called the "Z-count roll-over" does exist and is sometimes erroneously labeled as a Y2K problem. This clock roll-over occurs every 1,024 weeks; the first roll-over having occurred Aug. 21, 1999. Despite the publication of a GPS specification, some receiver manufacturers did not account for the Z-count roll-over in the satellite clock. Some affected receivers can be manually reset, or if they have flash memory or removable Programmable Read Only Memory (PROM), they can be reset to accommodate the roll-over. Those that cannot be reset must be replaced.

The following selected generic test objectives are widely recognized as the important test objectives in Y2K integration testing. It is the responsibility of the integration tester to select those objectives that are applicable to the applications under test and to develop a formal test procedure and a formal expected results for each selected test objective. The selected generic test objectives are shown in Table 2.

Sample Integration Test Script

Each Y2K test objective is developed into a specific test that the tester uses as a basis for assessing conformance to Y2K requirements. Each tester is encouraged to pursue additional testing when errors or abnormalities appear.

All testing procedures are reported in the test report with the observed test results. Below is an example of a test objective with its associated test procedure(s) and expected results.

Note for tester: When a test objective is not applicable to an applications under test, use the following statement:

Recording results: The test objectives are not applicable to the applications under test because the required functionality is not supported.

Test No. 1

Test objective (TO) No. 1: Tests roll-over of the GPS 10 bit epoch. Days are correctly recognized as Saturday and Sunday, respectively.

Test procedure, Part A for TO No. 1A: Set system date to Saturday, Aug. 21, 1999 (1999-08-21) at or about 23:00 hours. Check each commercial-off-the-shelf (COTS)/government-off-the-shelf (GOTS) application in turn for the correct date and time. Exchange the current date and time between appropriate applications and check that the date is correct within the time period.

Note to tester: Set time sufficiently prior to midnight to allow you to assess each of the applications under test in a timely manner.

Expected results: Date must be Saturday Aug. 21, 1999

Generic Test Objective	Rationale	Example Test Elements
Event triggers: processes that cause the automatic invocation of a procedure at a specified time.	Event triggers generally start the execution of a procedure when the current time is equal to or greater than the scheduled event time. Events scheduled in 1999 to occur in the year 2000 may be misinterpreted when the applications compare dates with only two digit year information.	Alarm systems should notify the recipient on time. E-mail should send a message after a specified time. Project management tools should correctly schedule milestone/dates into the next century or millennium. Automated periodic reports such as MIS systems should produce timely reports as scheduled.
Error handling: the process of detecting and responding to any discrepancy between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition.	Whether the user interactively inputs date information or whether date data is supplied via some other source, an application should possess a means to assess the legitimacy of the date data. If the input date data is not acceptable to the processing logic, then an error should be reported.	Error messages should report that input date(s) are out of range. Error messages must display dates in a format that reliably differentiates centuries.
Queries, Filters, and Data Views: These are higher-order functions that accept a predicate and a list and return those elements of the list for which the predicate is true.	These higher-order functions generally operate by taking portions of dates and comparing values to similar portions of other dates. The ability to correctly complete numerical comparisons on dates is essential to these functions.	All comparison (e.g. <, >=, >, =<) and logical operators, (e.g. and, or, not, xor) must be properly processed.
Comparing or sorting dates: sorted dates should be correctly sorted in either ascending or descending order.	All date-based comparisons or sorts must be performed correctly.	Data containing dates that are passed between applications must be correctly sorted, both ascending and descending.

Table 2. *Y2K generic test objectives.*

between 23:00 and 23:59 hours.

Recording results: Record the result for each application as “passed,” “failed,” or “n/a.”

Test procedure, Part B for TO No. 1B: Wait long enough to allow date to roll over. Check applications for date and time and again exchange the current date and time between appropriate applications and check that the date is within the correct time period.

Expected results: Date must be Sunday, Aug. 22, 1999 between 00:00 and 00:59 hours.

Recording results: Record the result for each application as “passed,” “failed,” or “n/a.”

Test No. 9

Test objective No. 9: Critical midnight crossing from 2000 into the year 2001. Tests roll-over to new millennium. Days are correctly recognized as Sunday and Monday, respectively. This date is the last day of the second millennium on the Gregorian calendar. The ordinal date 00.365 was the last day of 1900 (Julian Calendar). Since 2000 is a leap year, its last day is 00.366. An incomplete algorithm for determining the length of the year might cause an ordinal-based system to transition into the new millennium a day too early.

Test procedure, Part A for TO No. 9: Set system date to Sunday, Dec. 31, 2000 (2000-12-31) at or about 23:00 hours. Check each COTS/GOTS application, in turn, for the correct date and time. Exchange the current date and time between appropriate applications and check that the date is correct within the time period.

Note to tester: Set time sufficiently prior to midnight to allow you to assess each of the applications under test in a timely manner.

Expected results: Date must be Sunday, Dec. 31, 2000 between 23:00 and 23:59 hours.

Recording results: Record the result for each application as “passed,” “failed,” or “n/a.”

Test procedure, Part B for TO No. 9: Wait long enough to allow date to roll over. Check applications for date and time and again exchange the current date and time between appropriate applications and check that the date is within the correct time period.

Expected results: Date must be Monday, Jan. 1, 2001 between 00:00 and 00:59 hours.

Integration Test Report

The Integration Test Report should provide:

- a full description of the software/ hardware test environment
- a test number to identify the test report
- the test preparations (e.g. obtaining all software in a correctly configured format)
- the test script (or a reference to the formal test script to allow future replication)
- a full description of the testing procedures, including any additional testing resulting from observed abnormalities, or changes to the test objective and/or test procedure and the rationale for the changes
- the operator notes (e.g. background information, history, glossary, rational), as needed
- any acronyms used in the test report
- any points of contact (e.g. names, addresses, and telephone numbers)
- a recommendation (e.g. whether the software is approved for inclusion into the standard build/up-grade; or approval is denied with an explanation.)

Lessons Learned

- There are several COTS products that vendors claim are Y2K compliant. These products are Y2K compliant with a shift in the way end-users enter their data into the application; there are no technical workarounds. It is the responsibility of upper management to provide the basis for a policy directive to change the way end-users enter data. These known problems were not used when developing the suggested Y2K integration testing script.

- Y2K integration testing is not validating the results of unit testing. A tester should review the documents associated with unit testing and may use them as a basis for the integration testing. In some instances, the tester may find omissions of, or inconsistencies in, required data in the unit test reports. In these instances, the tester should work to resolve these discrepancies because inaccurate unit test reports could invalidate the integration testing efforts.
- Some applications may not coexist on the same operational system. For example, different versions of Microsoft Office will not coexist on the same testing system at the same time. Therefore, two tests must be conducted for each system. For example, integration testing should be conducted with one version of MS Office and all other applications, then with a different version of MS Office and all other applications. ◆

About the Author

William H. Dashiell is a computer scientist at the Department of Defense National Imagery and Mapping Agency. He has worked on the development of software testing by statistical methods using binomial models, coverage designs, mutation testing, and usage models. He has contributed to the development of conformance and testing protocols for federal, national, and international information technology standards. He has a bachelor's degree in business administration and education, a master's degree in education technology, and a doctorate in mathematics education from the University of Maryland. He also has a master's degree in computer science from Hood College in Maryland.

National Imagery and Mapping Agency
1200 First St. SE M/S N-61
Washington DC 20303-0001
Voice: 703-281-8836
Fax: 703-281-8957

Further Readings

1. U.S. General Accounting Office, Accounting and Information Management Division; GAO/AIMD-10.1.21. Year 2000 Computing Crisis: A Testing Guide; Exposure Draft; June 1998.
2. URL: http://www.nist.gov/y2k/datetest.htm (Test Assertions for Date and Time Functions).
3. URL: http://www.state.de.us/ois/y2000/testplan.htm (Year 2000 Conversion Directive Test Plan).
4. URL: http://www.microsoft.com/technet/topics/year2k/default.htm (MicroSoft Year 2000 Readiness Disclosure and Resource Center Web site).
5. URL: http://tecnet0.jcte.jcs.mil:9000/htdocs/teinfo/directives/soft/ds2167a.htm (DoD-STD-2167A Defense System Software Development).
6. URL: http://www.stsc.hill.af.mil/Crosstalk/crostalk.html

Call for Articles

If your experience or research has produced information that could be useful to others, *CROSSTALK* will get the word out. We welcome articles on all software-related topics, but are especially interested in several high-interest areas. Drawing from reader survey data, we will highlight your most requested article topics as themes for future *CROSSTALK* issues. In future issues, we will place a special, yet nonexclusive, focus on

Risk Management
February 2000
Article Submission Deadline: Oct. 1, 1999

Education and Training
March 2000
Article Submission Deadline: Nov. 3, 1999

Cost Estimation
April 2000
Article Submission Deadline: Dec. 4, 1999

We will accept article submissions on all software-related topics at any time; issues will not focus exclusively on the featured theme.

Please follow the *Guidelines for* CROSSTALK *Authors*, available on the Internet at http://www.stsc.hill.af.mil.

Ogden ALC/TISE
ATTN: Heather Winward
CROSSTALK Associate Editor/Features
7278 Fourth Street
Hill AFB, UT 84056-5205

Or e-mail articles to features@stsc1.hill.af.mil. For more information, call 801-775-5555 DSN 775-5555.

Who is to Blame for the Y2K and Similar Bugs?

Alka Jarvis
Cisco Systems
Dr. Vern J. Crandall
Vern J. Crandall & Associates Inc.
Cindy Snow
Intel Corp.

In the beginning, programming and software architectural design were considered an art. Programmers used their creative skills to build software. Constraints were viewed in terms of "compilers" and "operating systems," not in terms of fundamental design criteria. Quality assurance personnel and testing organizations still are considered to come at the end of the "food chain" and their job is to find any errors that might exist in the software. These people are told, "You must accept the software product the way it has been designed and programmed, and make it work." The implication is that you are at the mercy of the software developers. The real problem is not inadequate testing. It is the lack of quality controls during the crucial design and coding stages. Insufficient controls at this stage interpolate to insufficient design and architecture. The year 2000 bug is a prime example of the lack of such quality controls. It could have been avoided. Someone is to blame.

Introduction

The United State's quality guru, W. Edwards Deming, won the respect of Japanese management after World War II by preaching a philosophy of commitment to quality and continuous improvement of company-wide processes. His recognition that quality cannot be inspected in, but must be designed in, raised an awareness in American businesses. The companies that are industry leaders and retain their customer base continuously evaluate their processes, services, and delivery mechanisms and improve them [1, 2].

We think Deming was right. We also think that the pioneers in the software field were on the right track. Their publications indicated both the right software development problems to be solved and the right way to solve them. These publications include:

- D.L. Parnas [3] with his definition and justification of modularity via information hiding.
- Wayne Stevens, Larry Constantine, and Glenford Myers [4] with their definition and justification of minimizing dependencies via composite design.
- Edsgar Djikstra [5] with his definition and justification of structured programming.

This paper focuses on modular and composite design, and structured programming — and the need for quality controls to insure such. Adherence to these three principles at the design and coding phases insures a higher quality product — even before it goes to test. These software development guidelines were known in the 1960's, early enough to have avoided the year 2000 bug (Y2K).

We choose to focus on the "quality" principles that, had they been followed, would have avoided the Y2K problem. These include both architectural and coding quality principles, defined below:

Architectural Quality

Architectural quality insures that software is designed to be modular, to minimize dependencies and states, and to maximize reusability. Architectural quality reduces the number of modules created by recognizing functional similarities and designing one generic module in place of several "one-offs."

A component that is reused is designed to be state independent. An anti-virus software product, for example, gives the user the ability to configure an automatic action if a virus is found. The user will want to choose the automatic action for real-time virus scanning, for screen-saver scanning, for scheduled scans — and for servers and clients. The configure automatic actions component should be designed, coded, tested, and translated once, but reused in all of the scenarios presented above. Driven by the need for reusability, the component will be designed to work the same regardless of which scan type is being defined. It will be state independent; independent of the context from which it was triggered. A good architecture also separates data from functionality. An example is a data-driven diagnostic tree where each node contains a trigger, an expected response, and further directions based on the actual response. A second example is a data-driven installation component. We have seen a successful setup architecture that compartmentalizes the functionality (add/delete/start/stop services; copy/delete files; etc.) then lists the actual files to be managed as data. Different applications can be installed, upgraded, and uninstalled with no code changes to the setup component, requiring only changes to the data.

How to Design-In Architectural Quality

We propose developing a Graphical User Interface (GUI) prototype based on marketing requirements. It is our experience that such a prototype hastens identification of reusable objects, is a good tool for communicating the design, and can be mapped to marketing requirements (one should be able to identify in the prototype how each function in the marketing document can be triggered). The approved prototype is examined for objects that appear multiple times, such as the configure automatic actions component. Common functions are combined into one component/module. The goal is to minimize components and maximize reuse. Components are represented in the architecture. Architectural constructs include:

1. Executive modules: Modules in which high-level logic is packaged, making it possible for direct traceability back to the specification.
2. Fan-in modules: Normally, the term describes a situation where multiple modules call a single module. This concept is used for module reuse and for building libraries of reusable functions. We use the term fan-in to describe a method of removing dependencies. When dependencies have been identified across various pieces of code, the code is extracted from the modules in which it appears and fan-in is used to package that code, and dependency, in one place.
3. Massage modules: These modules are used when a fan-in module can almost be used. That is, a condition is slightly different than that expected by the fan-in module. The massage module then modifies the format so that a nonconforming condition is made to conform, and the fan-in module can be used.
4. Informational strength modules: If the dependencies within a set of modules are so different that they cannot be related by fan-in or massage modules, they can be packaged in informational strength modules. These modules have multiple interfaces and multiple entry points allowing each dependency to be considered as a separate program.
5. Transaction (event) processors: Transaction processors use the menu to identify the desired state and transfer the system to that state, defining it as an independent subsystem. Each transaction processor system has its own set of independent states.

The finalized architecture points out all modules that need coding. The GUI prototype will demonstrate how the code should look and feel. The combination allows for novel task assignment. Senior engineers should be working on proof-of-concept and new technology research. Junior engineers should be assigned to the basic coding tasks, and should represent the critical path. That way, if they get behind on the schedule, senior engineers can be pulled back to help catch up. If proof-of-concept, the most difficult to predict schedule for, takes longer than expected (within bounds) the critical path time is not affected.

We recognize that composite design stressing independence usually involves tradeoffs with performance and efficiency.

Code Quality

Code is produced for the modules identified in the architecture. Quality code reduces dependencies, is correct, is logical, and is easy to certify. Code quality is controlled by constraint to three canonical forms (sequence, iteration, and selection) defined later. Code quality is certified via manual pattern matching of each piece of code. We know of one company where two different reviewers look at code. If it does not follow the proposed constructs, or if it is not easily understood, it is looked at by a third reviewer who usually sends it back for redesign. Code quality should be verified before a module is sent to the test group.

Further, coded algorithms should be reviewed and certified for correctness. The quality controls proposed by this paper guarantee architectural quality and code quality as defined above.

Legacy Software Wisdom

As mentioned in the introduction, we consider three discoveries in software development over the last 35 years to be most significant. Each of these discoveries has been based on the work of others, and each has influenced various methodologies.

Modularity via Information Hiding

The major concept Parnas introduced was "information hiding" as the basis of modularity. This has been interpreted as the hiding of a function, leading to reusability. Reusability is maximized when dependencies are minimized. This leads to our approach.

Minimizing Dependencies via Composite Design

Composite design, based on the 1974 paper and presented by Myers [6, 7, 8], addresses the development of architectures focused on minimizing dependencies. While module strength, module coupling, and other design issues have been enumerated by those describing composite design or structured analysis and design or the "Yourdon Methodology," no one after Myers attempted to explain the reasons for the various module strengths and couplings in terms of dependencies — except, perhaps, Lawrence Peters [9] and Vern Crandall [2]. The elimination of dependencies, initially considered an integral part of composite design, is a guiding factor to insure architectural quality. We think it has a far more critical impact than might be expected.

Structured Programming

Another concept we will discuss regards the work of Djikstra. While he deserves the credit for publicizing structured programming, others have had an influence. The original paper leading to this approach to programming comes from professors C. Boehm and G. Jacopini [10] from Italy. (Actually, the concepts originally appeared in a 1938 textbook on linear algebra, but we cannot find the reference.)

The late Harlan Mills of IBM probably had as much to do with popularizing structured programming as anyone. He simplified the proofs of Djikstra, who proved the second half of the structure theorem, and publicized the approach throughout IBM — and the English-speaking world. Structured programming is based on the definition of a proper program in conjunction with the structure theorem, and the correctness theorem. These theorems and definition are paraphrased below:

Definition of a Proper Program

- It contains a single entry and a single exit for the entire structure and for any structures inside.
- It contains no "dead code," i.e. it contains no code or logic structures

that cannot be reached or have no effect on the results.
- It contains no "eternal loops," i.e. all loops are finite and cannot continue to run forever.

Some interpret a proper program "single entry/single exit" to mean "no GO TO's." GO TO's, however, may be necessary to implement structured programming forms in some languages such as assembly language and Basic.

Structure Theorem as Understood by R.C. Linger, H.D. Mills, and B.I. Witt [11]

- It can be shown that any proper program is the equivalent of one composed of sequence, iteration, and selection, plus some logical functions. In other words, any proper program can be written using only these three canonical forms.

The major value of the Structure Theorem is in its proof of the reduction of the number of patterns necessary to produce any process. It also certifies that any "spaghetti program" can be "reverse engineered" to the three canonical forms, which helps greatly when searching out Y2K-type problems.

We further propose implementing the three canonical forms as suggested below:

Sequence: Sequence should flow forward. Do not use GO TO to jump back to code with a lower sequence number. Do not use C's ++ notation because it is error and typo prone. 'X++' can easily, more dependably, and more understandably be coded as 'x = x + 1.'

Iteration: Use DO WHILE loops (pretest loops) exclusively. Do not use REPEAT UNTIL or FOR loops.

Selection: Use case statements instead of nested IF THEN ELSE structures, except in the case of success-oriented nesting. Success-oriented nesting requires a nested IF THEN ELSE structure where the innermost structure is the success situation. Test for the first or highest percentage error first in the nested set.

Limiting the coding structures to three canonical forms, and further specifying the already validated constructs used to implement the three canonical forms, provides an outline or pattern that can be quickly and easily pattern matched to prove correctness. One can easily outline the flow of any module to verify that it represents a proper program. If the solution has been correctly defined, then pattern matching can quickly validate the correctness of the code. This pattern-matching process is manual, but studies have confirmed that a manual code walkthrough is as effective in locating code defects as any other method.

Correctness Theorem, Linger, Mills, and Witt [11]

It can be shown that if the formula of a program contains at most the three canonical forms (sequence, iteration, and selection), it can be proved correct by a tour of the program tree. In other words, if all steps of the program are correct in its decomposition, then the program will be correct.

If P is a proper program, then it can be equivalently written using only three canonical forms. This means that no matter how bad the "spaghetti" that appears in a program, or any process in the world — from a cooking recipe, to directions to reach a destination, to instructions for building computer programs — the logic or logic structure can be replaced with logic or logic structures involving simple sequences of instructions, iterations (loops), and selections (branches). This is a very powerful concept.

Architectural and Code Quality Minimize Inefficient Testing

Some prescribe running software systems for a year or more to "certify" the adequate correction of the Y2K problem. The software system is tested by repeatedly executing production runs, with the hope that rarely executed code or rare conditions will be triggered and point out remaining Y2K defects. We propose that the real test issue is not one of covering the code, it is one of covering the states the code generates. Most everyone who has attempted to install and execute complex software knows that the number of states the software can take on approaches infinity. As the number of states approaches infinity, traditional testing efficiency approaches zero, defined as the percentage of the software states actually tested.

Assuring that the dependencies have been eliminated or "certified" as being transparent to the millennium bug is a better approach. Events that are independent of other events cannot cause the other events to fail.

The Year 2000 Bug was Preventable

The cost of fixing the Y2K bug already is in the millions. The societal and business costs are unknown, and even the most conservative projections seem unbelievable. Surely "Divine Providence" will not let such a profound catastrophe affect mankind, especially now that the Cold War is over.

The entire issue can be wrapped up in terms of information hiding. The millennium bug is not that an algorithm was coded over a period of more than 30 years, which would become defective around the year 2000; the bug was that the knowledge of the problem was distributed through millions of lines of code. Had quality-oriented, well-known "information hiding" strategies been employed at the time they were known, the year 2000 bug would have been a 10- to 45-minute fix in even the largest programs, because all impacted code would have been located at one point, and the impact would have been restricted to only the "code actually necessary to make use of the need to restrict the year to two digits to save 'needed memory or storage space.'"

Industry leaders call attention to the lack of programmers who know the early programming environment, reasoning that such knowledge is a prerequisite to searching for Y2K defects. This should not be an issue. For years, forward-looking teachers have insisted on their students programming in "pseudo-code," the eternal principle of "design before implementation." That means that the entire software industry should contain programmers who program in English and code in any of thousands of implementation languages. The Structure Theorem of Structured Programming guarantees that any program, however large, which is a "proper program," can be "equivalently written" using only the canonical forms. What this means is that any program — be it COBOL or Basic, Assembly Language or whatever — can be "reverse engineered" into a structured equivalent, which can easily be expressed in English. This has been taught to secretaries and clerks by one of the authors for more than 20 years in companies where the available personnel is limited. You do not need to be a professional programmer to "reverse engineer code." But once these people have performed their task, professional programmers can quickly decipher the code structure and determine how it functions. There are efficient, time-proven ways to determine the logic structure around a potential millennium bug and to create a solution.

Other Preventable Bugs

The "Christmas bug" occurred when a new employee of a major corporation sent an e-mail Christmas card to several of his new colleagues. He included their aliases in the address. As the card was received by each friend, it was immediately forwarded to all on the friend's alias list, and their aliases. This proliferation recursed until the e-mail storm brought the entire mail network to a screeching halt. Either a failure mode was not comprehended in the original design, or recursion was improperly implemented.

Mistakes can happen to trained users under natural operating conditions. Have you ever hit the wrong key on the keyboard, or clicked the mouse at the wrong point as it swooped across the screen? Mistakes with huge consequences are sometimes explained away as viruses. Such accidents more appropriately represent failure modes that were not comprehended or appropriately planned for. Recursion problems are common with loops which process differently for each iteration, and with calling sequences which repeat themselves with different results for each sequence.

Another victim tells of an $8,000 check that did not get deposited to the right account. This caused his check to bounce. A new check to cover the bounced check was cashed twice, causing a $16,000 overdraft. Due to constraints on the banking software, all subsequent bounced checks had to be processed by hand. The knowledge of the overdraft was automatically available to Visa and MasterCard. They promptly cancelled. Credit reporting agencies then produced a damaging credit rating. The bank error rippled through the entire banking and credit rating system, and cost the bank $50,000 to correct. Knowledge of the first bounced check should have been contained in one spot, and then processing inactivated until an alert could be addressed. Instead, the knowledge was duplicated, without verification of the problem, to an undetermined number of places. Errors of this type appear in systems with extremely large numbers of states. Testing is not comprehensive even if it covers all code; it must cover all states the code generates. As the number of states increase, sometimes towards infinity, the software's complexity increases to the point where comprehensive testing is impossible to define or execute. State proliferation occurs when programmers do not pay attention to dependencies among input variables and functions.

New Bug Watch: The Year 2038 Bug

American National Standards Institute (ANSI) provides a standard for date/time representation called time_t. This time standard has received wide acceptance, most notably in the Unix world. The time value is represented by a 32-bit signed integer that denotes the number of elapsed seconds from Jan. 1, 1970. The maximum time-period (or epoch as it is called in date/time lingo) will rollover at 20:14:07 Jan. 18, 2038. Depending on implementations of this ANSI standard, once again computers will be faced with a date representation problem and may not be able to distinguish between 2038 and 1970. The problem is compounded by conversion functions and interpretations of the standards. Some known variations will actually run out as soon as 2036.

We propose that software development managers and engineers discuss the alternatives to time_t use (for both user interface [UI] and non-UI implementation) now. The alternatives are easily comprehended and easily implemented.

Summary

We claim that the year 2000 and similar bugs could have been avoided by adherence to the architectural and coding quality standards insured by modularity, structured programming, and composite design. At every point where these bugs occur, they were "designed in." We propose that such defects be "designed out." We think such defects can be eliminated, and at design time. If a bug must be allowed to exist, its impact — and the knowledge of its existence — should be contained in only one module.

The cost of ignoring architectural and code quality is now obvious. We encourage those responsible for solving the Y2K problem to concentrate on the real problem. We encourage them to add and execute quality controls at the design and coding stages. We hope that executives and software engineers have learned the awful cost of ignoring these simple quality principles.

About the Authors

Alka Jarvis, a senior quality consultant at Cisco Systems, in San Jose, Calif., has an international reputation in the area of software quality and last year was named Outstanding Woman Professional for Silicon Valley. She has written three books, *ISO 9000-3*, published in 1995 by Spriner-Verlag, *Inroads to Software Quality*, published in 1997 by Prentice Hall, and *Dare To Be Excellent*, published by Prentice Hall in December 1998. Jarvis is a Registration Accreditation Board-certified quality systems lead auditor (ISO 9000) and a certified quality analyst. She has 19 years of experience in software development, eight years of which have been in total quality management. Her background consists of management of large systems — development processes, quality control, specification reviews, management of the testing process for large companies, and teaching. She has been a frequent speaker on quality-assurance issues at international and domestic events and has worked in the quality management field in a variety of capacities at Cisco Systems, Bank of America, Pacific Gas & Electric, Pacific Bell, Charles Schwab, and Apple Computers Inc. She serves as an instructor for University of California at Berkeley, Extension and University of California at Santa Cruz, Extension. She also is an adjunct professor at Santa Clara University in the field of quality and software engineering.

CISCO Systems
360 Everett Ave., Suite 1A
Palo Alto, Calif. 94391
Voice: 408-325-2758
Fax: 408-527-7995
E-mail: ajarvis@cisco.com

Dr. Vern J. Crandall, president of Vern J. Crandall & Associates Inc., just completed serving as vice president of Digital Technology International of Springville, Utah. He received his doctorate degree at the University of Washington. For 25 years, he was professor of computer science at Brigham Young University, Provo, Utah. During this time, he wrote the first software engineering curriculum for BYU and also for IBM technical education. He has consulted with Hewlett-Packard, IBM, Microsoft, Corel WordPerfect, Novell, Pacific Bell, UNISYS, and approximately 45 other companies in the computer industry. He pioneered the first software testing course ever taught in a university and has given keynote addresses at major conferences in the areas of software engineering, comparative methodologies, enterprise-wide information management, getting software products to market, software testing, and software quality assurance. He also served as vice president of software development at Novell, and on the senior staff in the software engineering group at SunSoft, a division of Sun Microsystems.

Vern J. Crandall & Associates Inc.
3549 N. University Ave., Suite 200
Provo, Utah 84604-4417
Voice: 801-375-1415
Fax: 801-375-2295
E-mail: v_crandall@sprynet.com

Cindy Snow, an engineering manager at Intel Corp., in American Fork, Utah, has a background in aerospace and academia. She worked on Naval submarines and Marine radar systems at Hughes Aircraft in Fullerton. She taught in the mathematics department at Boise State University, Idaho and in both the mathematics and computer science departments at Brigham Young University for more than 10 years. Her teaching emphasis has been in the area of operating systems. Drawing on development experience with a variety of methodologies from Waterfall to Rapid Application Development (RAD), she also taught software life cycle classes at the university level. She has designed and managed production of multiple expert systems, including a medical diagnosis expert system, and a comprehensive on-line class scheduler for university students. Recently, Snow managed software product development at a consulting company using a 4GL while helping to design and implement an applicable RAD life cycle. She continues to study in the field of software product delivery to enhance the quality of, and reduce development time for, software products.

INTEL Corp.
734 E. Utah Valley Dr., Suite 300
American Fork, Utah 84003
Voice: 801-763-2274
Fax: 801-763-2895
E-mail: cindy.l.snow@intel.com

References

1. Jarvis, Alka, and Vern Crandall, *Inroads to Software Quality: A "How To" Guide with Toolkit*, Prentice-Hall, New York, 1997.
2. Crandall, Vern J, *Computer Science 427: Software Design and Implementation, Lecture Notes*, Alexander's Print Shop, Provo, Utah. 1984.
3. Parnas, D L, "On the Criteria to be Used in Decomposing Systems into Modules," *Communications of the ACM*, December 1972, pp. 1053-1058.
4. Stevens, Wayne, Larry Constantine, and Glenford Myers. "Structured Design," *IBM Systems Journal*, Vol. 13, No. 2, pp. 115-139.
5. Dijkstra, Edsgar, "Go To Statement Considered Harmful," *Communications of the ACM*, Vol. 11, No. 3, pp. 147-178.
6. Myers Glenford, *Software Reliability Principles and Practices*, John Wiley & Sons, New York. 1976.
7. Myers, Glenford, *Composite/Structured Design*, Van Nostrand-Reinhold, New York. 1978.
8. Myers, Glenford, *Reliable Software Through Composite Design*, Van Nostrand-Reinhold, New York, 1979.
9. Peters, Lawrence, *Advanced Structured Analysis and Design*, Prentice-Hall, Englewood Cliffs, N.J. 1987.
10. Böhm, C., and G. Jacopini, "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules." *Communications of the ACM*, Vol. 9, No. 5, pp. 366-371.
11. Linger, R. C., H. D. Mills, and B. I. Witt, *Structured Programming: Theory and Practice*, Addison-Wesley, Reading, Mass. 1979.

The Five Stages of Denial

Dr. Richard Bechtold
Abridge Technology

It is probably safe to say that most contractors are not especially fond of software capability evaluations (SCEs). Anyone who performs SCEs eventually verifies this. It can easily result from watching a contractor go through the five stages of denial.

In lieu of providing a pedantic explanation of the denial process, the following fictional, if slightly exaggerated, telephone conversation is used to illustrate the five stages a contractor typically goes through while you attempt to schedule him or her for a SCE, and while the contractor attempts to convince you that, somehow or another, you really have dialed a wrong number…

Ring… Ring…
Contractor Point of Contact (CPOC): Hello?
SCE Team Leader (STL): Hello. This is [name] with the [government agency]. Did you receive our e-mail stating that we are intending to perform a SCE on your organization within the next six weeks?
CPOC: Yes. But I was a bit surprised by it.
STL prepares for Denial No. 1.
STL: Why were you surprised?
CPOC: We do not do software development in our organization.
Denial No. 1 confirmed.
STL: Hmmm. Curious. We were under the impression that you are doing software development, so maybe there is a mistake some place. Would you please tell me, at a very high level, how you view the type of work you are doing for us?
CPOC: Well, mostly we design database schemas, develop and debug Structured Queries Language queries, build interactive Web sites using Java, JavaScript, and CGL, integrate all that with most of the leading commercial-off-the-shelf database tools, and, of course, we do exhaustive testing and code modification.
STL: I see. And you do not consider any of this to be software development?
CPOC: Nope. No software development. None at all. Not even the littlest bit.
STL: And the other projects in your organization? Are they doing software development?
CPOC: Definitely not! The other projects are building virtual classrooms. These are completely interactive, totally customizable, entirely user-sensitive neutral network systems driven by state-of-the-art knowledge bases and enhanced by genetic algorithms.
STL: But you do not consider this to be software development?
CPOC: Oh no, absolutely not!
STL: Well, as I am sure you are aware, our contract with you requires you to be a Level 3 software development organization. You are about 30 seconds from convincing me that you do not do any software development and, hence, are clearly not a software development organization. Is this what you are saying?
CPOC: Arrrraaaggghhhh!
STL: I'm sorry, what was that?
CPOC: Acckkkkkkkkkk!
STL: Do you need to get a glass of water or something? I'll hold…
CPOC: (cough, cough) Aaaghh. Actually, I guess you could say that some of our work is software development.
Contractor successfully moves beyond Denial No. 1.
STL: Very well. What is the best date for us to perform this SCE?
CPOC: Actually, a SCE really isn't necessary.
STL prepares for Denial No. 2.
STL: Would you please elaborate?
CPOC: Sure. Even though we do software development, we are subject to highly unusual circumstances and, hence, we deserve a waiver.
Denial No. 2 confirmed.
STL: I see. What are these circumstances?
CPOC: Our work is unique. No one on the planet does anything like what we do. The software Capability Maturity Model really does not apply to us.
STL: We typically find that all projects are unique.
CPOC: But we are very, very, very unique!
STL: You certainly sound unique. This is good, because we are quite used to evaluating unique projects. So, this really won't be a problem. Unless, of course, you are so unique that you have found a way to reliably develop complex software systems without using project plans, without managing requirements or configurations, without ensuring quality, and without tracking actual progress.
CPOC: From that perspective I suppose that just maybe we are not all that unique.
Contractor successfully moves beyond Denial No. 2.
STL: Very well. So when can we schedule the SCE?
CPOC: There is another little problem we have to discuss first.
STL prepares for Denial No. 3.
STL: Yes?
CPOC: It is going to cost us a lot of money to prepare for the SCE and I'm wondering if I can just send the invoice directly to you. It should be for somewhat less than a million dollars, I think. I will have to work the numbers a bit to know for sure.
Denial No. 3 confirmed.
STL: We always advise that you do not take any special steps to prepare for the SCE. Generally, we are only interested in looking at documentation and evidence that already exists, and asking people about what they are currently doing. Very little

preparation is necessary. Unless, of course, you are telling me that you have a lot of documentation to create.
CPOC: Oh no! I'm not saying that.
STL: Is it the evidence then? Is that what you need to create?
CPOC: No, no, of course not!
STL: That is good to hear. How about if we do everything we can to minimize your costs associated with this SCE, and you do the same?
CPOC: I guess that will probably work.
Contractor successfully moves beyond Denial No. 3.
STL: So when can we schedule the SCE?
CPOC: That depends.
STL prepares for Denial No. 4.
STL: On what?
CPOC: The SCE is going to completely disrupt all our management and project personnel, so we will need to slip our critical milestones, deadlines, and delivery dates.
Denial No. 4 confirmed.
STL: By how much?
CPOC: I'm just kind of estimating, but I think two years ought to do it.
STL: But the entire on-site period only lasts a week.
CPOC: I know. But no one has a moment of spare time. Everyone is already working massive overtime. They work 167 hours per week, then take an hour for lunch. Junior people skip their lunch and try for an hour's sleep. The schedule is really quite tight.
STL: And this is how you planned it?
CPOC: Well, no, we did not plan for it to be this way.
STL: The project is not occurring according to plan? Is this because the project is subject to some type of massive unrecoverable problem that you would like to tell me about?
CPOC: Oh, no! No problem at all. It's just, uh, we've got a lot of really enthusiastic project personnel who love their work. Yes, that is it! Teamwork! High morale! They cannot stand to leave their cubicles!
STL: But you think that they will be able to find 45 minutes where they can come to an interview?
CPOC: Probably. I guess. But it is really hard to pry them away from software development. I mean nonsoftware development. I mean software nondevelopment. Anyway, did I mention how high morale is?
Contractor successfully moves beyond Denial No. 4. Barely.
STL: Yes, you did. Do you have any preferences as to when we can schedule the SCE?
CPOC: Sure, let me check the calendar.
STL prepares for Denial No. 5.
STL: How does the calendar look?
CPOC: You are not going to believe this.
STL: Trust me, we have heard it all before.
CPOC: Well, it looks like the earliest possible window where we can do this is the month immediately after my retirement.
STL: And that will be?
CPOC: Hard to say. I'm only 22.
Denial No. 5 confirmed. ◆

About the Author

Dr. Richard Bechtold is president of Abridge Technology and an independent consultant who supports industry and government in the analysis, design, development, and deployment of improved software management, engineering, acquisition, and risk-reduction processes. He has more than two decades of experience in the software industry and holds a doctorate degree from George Mason University, where he also is an adjunct professor teaching software project management and process improvement. He has written more than two dozen works relating to software project management, software process improvement, risk management, and related topics. His latest book, *Essentials of Software Project Management*, was published this summer. (Management Concepts Inc.)

42786 Oatyer Court
Ashburn, Va. 20148
Voice: 703-729-6085
Fax: 703-729-3953
E-mail: rbechtold@mindspring.com

Call for Speakers and Exhibitors for the 12th Annual Software Technology Conference (STC)

"Software and Systems — Managing Risk, Complexity, Compatibility, and Change"

April 30 - May 4, 2000 — Salt Lake City, Utah

The 12th Annual Software Technology Conference promises to be the most exciting software technology conference for the Department of Defense yet.

Presentations will be given in concurrent tracks occurring Monday, May 1 through Thursday, May 4, 2000 in areas relating to the many aspects of software and systems technology and support. The program will include tutorials, presentations, exhibits, and "birds-of-a-feather" sessions. The general sessions will provide attendees an excellent opportunity to hear leaders in the fields of management information systems, command and control, and embedded computers espouse their vision for software and systems technology.

The STC is particularly looking for speakers who can share findings and lessons learned in applying the technologies related to our theme for STC 2000: "Software and Systems — Managing Risk, Complexity, Compatibility, and Change."

Abstracts must be submitted no later than September 17, 1999 to be considered. Three ways of submitting abstracts* listed in order of preference, are:

1. Complete the web-based form at http://www.stc-online.org
2. Send an e-mail request for an electronic form to stcabstracts@ ext.usu.edu; complete the electronic form and e-mail to abstract_ submit@ext.usu.edu
3. Complete the submittal form in the brochure, include a one-page abstract on disk, and mail to the Utah State University address in that section of the brochure.

*Please note: Abstracts submitted for the intelligence track should follow the specific instructions in that section of the abstract submittal brochure.

If you have any problems submitting your abstract, please call 435-797-0046 or e-mail stcabstract@ext.usu.edu with your questions.

We will acknowledge receipt of all complete abstract submittals via e-mail with a short message to the sender.

Only one-page submittals will be reviewed. Please be sure to maximize your one-page abstract with concise, clear, and complete information. Highlight how this presentation would benefit the conference attendees. Abstracts should emphasize processes, methods, tools, and technologies, and not the marketing of specific products, books, or services. Technologies that have been used in the operational field are preferred. Abstracts of interest to the joint services are important.

Please remember to allow sufficient time for approval from the necessary authorities in order to meet the deadline, as late abstracts will not be considered.

Abstracts can be submitted in the following categories:

- Capability Maturity — Models, Assessments, Evaluations
- Collaborative Engineering
- Configuration Management
- Data Management/Sharing
- DII COE
- Distributed Computing
- Education and Training
- Electronic Commerce
- Embedded Software
- Emerging Technologies
- Information Assurance
- Intelligence
- Internet/Intranet
- Interoperability
- Knowledge Management
- Measurement
- Modeling and Simulation
- Network Centric Systems
- OO Technology and Languages
- Open Systems and Architectures
- Outsourcing and Privatization
- Process Improvement
- Project Management
- Quality Assurance
- Re-engineering
- Risk Management
- Simulation Based Acquisition
- Software Acquisition
- Software Architecture
- Software Estimation
- Software Implementation
- Software Testing
- System Requirements
- Total Ownership Cost

If you know of anyone who would be a potential speaker for the conference, please share this information with him or her. They can obtain a Speaker Abstract Submittal Package by connecting to the http://www.stc-online.org Web site.

New exhibitors will find background information on this conference, its history, and attendance statistics helpful in planning for the conference. This information may be accessed at our Web site, http://www.stc-online.org.

To assist exhibitors in selecting booth locations, an updated exhibit hall layout — including assignments and organizations registered to date — will be maintained on the internet.

Reservations for exhibit space may be made by mail or fax only. Applications will be date- and time-stamped upon receipt. Faxed registrations will be accepted 24 hours daily. Courier deliveries will be accepted from 8 a.m. to 5 p.m. MT, Monday through Friday, excluding holidays. Postal delivery occurs at approximately 10 a.m., Monday through Friday, excluding holidays.

We welcome your participation! Please call us with any questions you may have.

Dana Dovenbarger
STC 2000 Conference Manager
OO-ALC/TISEA
7278 4th Street
Hill AFB, Utah 84056-5205
Phone: 801-777-7411 DSN 777-7411
Fax: 801-775-4932 DSN 775-4932
E-mail: Dana.Dovenbarger@hill.af.mil

CROSSTALK

SPONSOR	Lt. Col. Joe Jarzombek
PUBLISHER	Reuel S. Alder
ASSOCIATE PUBLISHER	Tracy Stauder
MANAGING EDITOR	Kathy Gurchiek
ASSOCIATE EDITOR/LAYOUT	Heather L. King
ASSOCIATE EDITOR/FEATURES	Heather Winward
VOICE	801-775-5555 DSN 775-5555
FAX	801-777-8069 DSN 777-8069
E-MAIL	crosstalk@stsc1.hill.af.mil
STSC ONLINE	http://www.stsc.hill.af.mil
CROSSTALK ONLINE	http://www.stsc.hill.af.mil/ Crosstalk/crostalk.html
ESIP ONLINE	http://www.esip.hill.af.mil

Subscriptions: Send correspondence concerning subscriptions and changes of address to the following address:

Ogden ALC/TISE
7278 Fourth Street
Hill AFB, Utah 84056-5205
E-mail: custserv@software.hill.af.mil
Voice: 801-775-5555
Fax: 801-777-8069 DSN 777-8069

Editorial Matters: Correspondence concerning Letters to the Editor or other editorial matters should be sent to the same address listed above to the attention of *CROSSTALK Editor*.

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 *Guidelines for CROSSTALK Authors*, available upon request. We do not pay for submissions. Articles published in CROSSTALK remain the property of the authors and may be submitted to other publications.

Reprints and Permissions: Requests for reprints must be requested from the author or the copyright holder. Please coordinate your request with CROSSTALK.

Trademarks and Endorsements: This DoD journal is an authorized publication for members of the Department of Defense. Contents of CROSSTALK are not necessarily the official views of, or endorsed by, the government, the Department of Defense, or the Software Technology Support Center. All product names referenced in this issue are trademarks of their companies.

Coming Events: We often list conferences, seminars, symposiums, etc., that are of interest to our readers. There is no fee for this service, but we must receive the information at least 90 days before registration. Send an announcement to the CROSSTALK Editorial Department.

STSC Online Services: This can be reached on the Internet. World Wide Web access is at http://www.stsc.hill.af.mil. Call 801-777-7026 or DSN 777-7026 for assistance, or e-mail to schreifr@software.hill.af.mil.

Back Issues Available: The STSC sometimes has extra copies of back issues of CROSSTALK available free of charge. If you would like a copy of the printed edition of this or another issue of CROSSTALK, or would like to subscribe, please contact the customer service address listed above.

The Software Technology Support Center was established at Ogden Air Logistics Center (AFMC) by Headquarters U.S. Air Force to help Air Force software organizations identify, evaluate, and adopt technologies that will improve the quality of their software products, their efficiency in producing them, and their ability to accurately predict the cost and schedule of their delivery. CROSSTALK is assembled, printed, and distributed by the Defense Automated Printing Service, Hill AFB, Utah 84056. CROSSTALK is distributed without charge to individuals actively involved in the defense software development process.

One Flew Over the Cuckoo's Cubical

What attracted you to software engineering as a profession? Money, fame, toys, independence, intellectual challenges?

What did you find? Unrealistic deadlines, clueless leaders, long pointless meetings, carpal-tunnel syndrome, cut-throat colleagues, life as a social outcast?

What profession adopts Dilbert as its poster boy? Why do we passionately defend our profession as an engineering discipline and then genuflect to cowboys, hackers, Bill Gates, and a cartoon character with no life and a bent tie?

I recently participated in a brainstorming session to determine the best software innovation of the 20th century. Three out of four answers were actually hardware innovations. What profession prides itself in its innovation and then articulates that innovation in terms of another profession?

What profession wants its lasting legacy to be Y2K?

Are we crazy? Ken Kesey's novel *One Flew Over the Cuckoo's Nest* may give us a parallel. Through the Big Chief's eyes we gain an interesting perspective on conformity and the dangers of being socially pigeonholed. A cast of crazy characters, as sane as you and I, choose to exist within the confines of insanity. For a moment Jack P. McMurphy exposes them to freedom, choice, and life outside. In a shocking end McMurphy dies; Taber, Harding, Ellis, and Martini return to the comforts of conformity; and Chief flies the coop.

Software engineers follow an eerily similar path, caught in a struggle between the software we must tame and the monsters we have created to do so. We are no more insane than our neighbors; however, thanks to Dilbert, Y2K, and our inability to meet any schedule or budget, we appear to reside, rather than work, in the asylum. What is most disturbing is that like Taber, Harding, Ellis, and Martini, we seem not to care. In fact, we revel in such images. If you question that, take a stroll around a software engineering conference. It's not so much about dress as it is attitude. We want to be respected, but only on our terms. We have set a dangerous professional precedence.

In *Introduction to Psychology*, 8th Edition, Stanford researcher David Rosenhan and 10 colleagues were admitted to a psychiatric hospital by pretending to hear voices. After their admission they acted completely normal. The staff was not suspicious. The pseudo-patients were seen by the staff in the context of a mental ward and labeled schizophrenic. Anything they did was viewed as part of their illness. When they came clean and explained they were faking and not crazy, staff members diagnosed them with paranoid delusions.

Resembling Rosenhan's group, software engineers have admitted themselves into the cubical asylum. In doing so we have pigeonholed our profession into a category that does not necessarily lead to our professional hopes or dreams. Money, respect, and responsibility seem rather scarce while deadlines, clueless leaders, meetings, and tedium are in abundance. We want out, but our appeals, like Rosenhan's, fall on deaf ears. Owners, managers, and customers alike ask how long we have been having these delusions.

Gates, Steve Jobs, and Larry Ellison have exceeded their dreams. But they do not engineer software, they make money off of software engineers. Their true talent and success is in business, *not* software engineering.

We have a choice for our profession. We can bite the dust like McMurphy, settle for the comforts of conformity, or throw the water fountain out the window and create a profession that earns respect, responsibility, and rewards.

The echoes from the playground are faint but they grow stronger — *Wier blier limber lock, three geese in a flock, one flew east, one flew west, and one flew over the cuckoo's nest. O. U. T. spells out!*

Wake up Chief! It's time to fly.

— Gary Petersen, TRI-COR Industries

Got an idea for *BACKTALK*? Send an e-mail to backtalk@stsc1.hill.af.mil

CROSSTALK
Ogden ALC/TISE
7278 Fourth Street
Hill AFB, UT 84056-5205

BULK RATE
US POSTAGE PAID
Permit No. 481
Cedarburg, WI