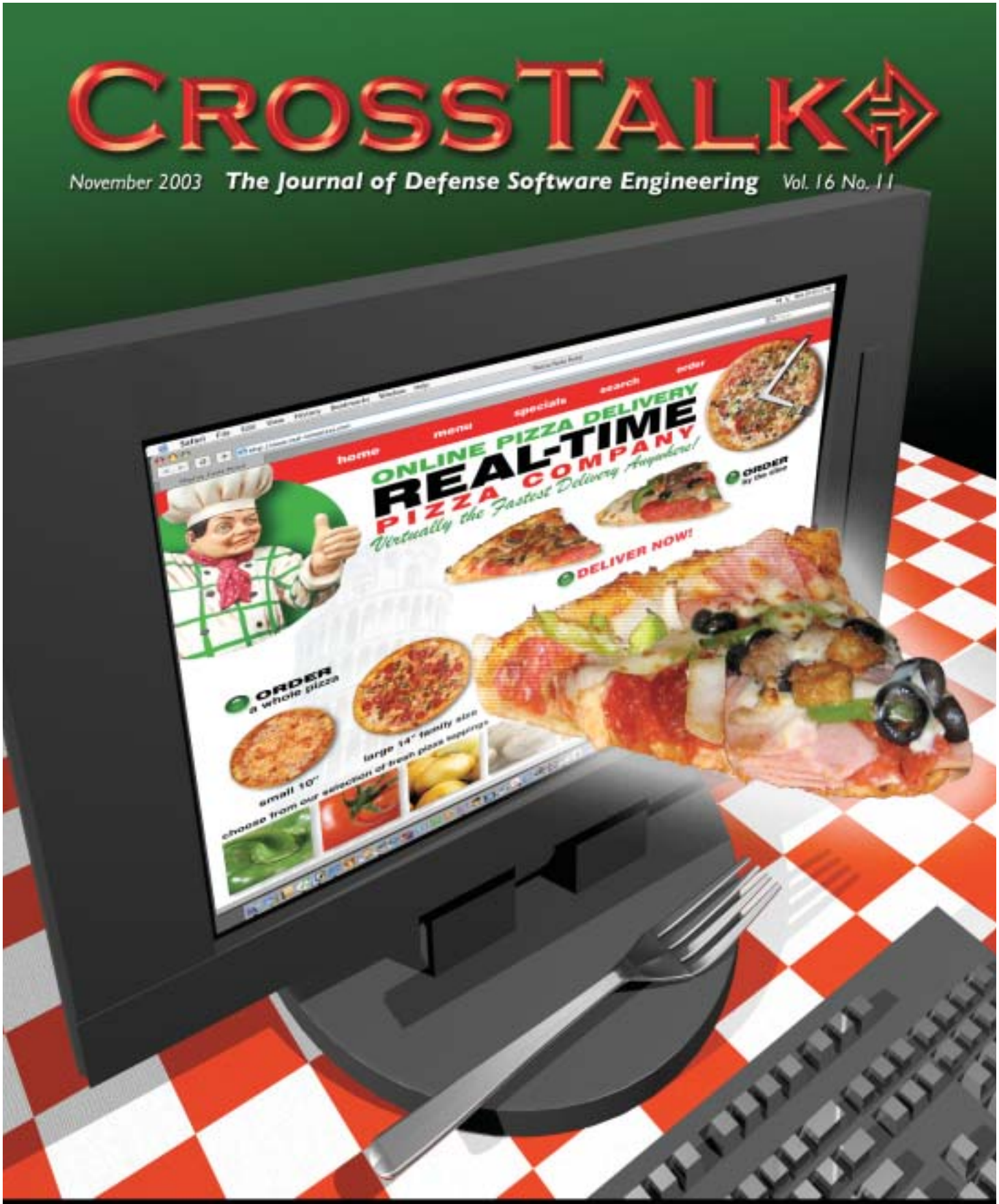
CrossTalk

November 2003 The Journal of Defense Software Engineering Vol. 16 No. 11



DEVELOPMENT *of* REAL-TIME SOFTWARE

Development of Real-Time Software

ON THE COVER
Cover Design by Kent Bingham.

Software Engineering Technology

Online Article

Departments

CrossTalk

SPONSOR	*Lt. Col. Glenn A. Palmer*
PUBLISHER	*Tracy Stauder*
ASSOCIATE PUBLISHER	*Elizabeth Starrett*
MANAGING EDITOR	*Pamela S. Bowers*
ASSOCIATE EDITOR	*Chelene Fortier-Lozancich*
ARTICLE COORDINATOR	*Nicole Kentta*
CREATIVE SERVICES COORDINATOR	*Janna Kay Jensen*
PHONE	(801) 586-0095
FAX	(801) 777-8069
E-MAIL	crosstalk.staff@hill.af.mil
CROSSTALK ONLINE	www.stsc.hill.af.mil/crosstalk
CRSIP ONLINE	www.crsip.hill.af.mil

Subscriptions: Send correspondence concerning subscriptions and changes of address to the following address. You may e-mail or use the form on p. 21.

Ogden ALC/MASE
6022 Fir Ave.
Bldg. 1238
Hill AFB, UT 84056-5820

Article Submissions: We welcome articles of interest to the defense software community. Articles must be approved by the CROSSTALK editorial board prior to publication. Please follow the Author Guidelines, available at <www.stsc.hill.af.mil/crosstalk/xtlkguid.pdf>. CROSSTALK does not pay for submissions. 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: www.stsc.hill.af.mil
Call (801) 777-7026, e-mail: randy.schreifels@hill.af.mil

Back Issues Available: The STSC sometimes has extra copies of back issues of CROSSTALK available free of charge.

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 to improve the quality of their software products, efficiency in producing them, and their ability to accurately predict the cost and schedule of their delivery.

Real-Time Software Development Requires Rigid Constraints

Development of real-time software differs from the development of non-real-time software because execution time must be considered. This brings the operating system and the underlying hardware processor architecture (memory, processor speed, bandwidth, etc.) into play. Also, in addition to new classes of defects related to stringent timing requirements, defect management in general becomes more critical because real-time applications are often life- or mission-critical.

Most real-time systems require a proprietary real-time operating system to run on, and therefore the software is typically not transferable to other platforms. Because embedded systems frequently are airborne (or spaceborne) platforms, the memory used for the software has weight and power constraints. Bandwidth issues include the amount of memory available or the number of memory cycles available. Defects can cost more than an inconvenience or money; depending on the system, defects can cost lives. For example, if the real-time software is used in a military aircraft, all data on an approaching missile is critical and time-sensitive. Only the most current data will suffice – and older data (even if only a few seconds old) is useless. However, if the real-time software is used in weather radar, and the current screen update is not available, then the last update will probably be sufficient if it is recent enough.

With these additional real-time software requirements come additional testing and maintenance requirements. The software is difficult to sustain since in many cases, changes to the software require that all of the code be retested.

Our first article is an overview that discusses many topics that must be considered while developing real-time software. Dennis Ludwig's article, *An Introduction to Real-Time Programming*, uses terms that most software developers can understand as they try to learn about the different world of real-time system development.

In *The Ravenscar Profile for Real-Time and High Integrity Systems*, Brian Dobbing and Alan Burns discuss this model for building safe and reliable real-time systems. The discussion includes the motivation behind the creation of the Ravenscar Profile, a definition of its specification, the ways in which it may be used with verification tools to produce evidence of dependability, and even a short example. Although the Ravenscar Profile is specified in Ada terms, it is based on a language-independent set of building blocks that are suitable for constructing typical real-time systems.

As discussed earlier, real-time software is often also safety-critical. Andy German shares his experiences from developing safety-critical, real-time systems in *Software Static Code Analysis Lessons Learned*. This article also presents a unique perspective for many readers since it comes from the United Kingdom (UK) and shares UK standards.

Defense Order (DO)-178B certification is a standard for the development of aviation software; however, much is expected under this type of grueling development and verification process. In *Decision Point: Will Using a COTS Component Help or Hinder Your DO-178B Certification Effort?* author Timothy J. Budden describes how the demands of DO-178B certification can be achieved with commercial off-the-shelf (COTS) modules if the vendor is a willing partner. I am hoping this perspective on the ability to use COTS software will help many readers.

In this month's supporting articles, Dr. John A. Hamilton Jr., Col. Kevin J. Greaney, and Gordon Evans discuss a process to evaluate potential vulnerabilities in shared simulation software in *Defining a Process for Simulation Software Vulnerability Assessments*. In addition, Dan W. Christenson and Lynn Silver discuss issues for tying software and hardware together in *Developing a Stable Architecture to Interface Aircraft to Commercial PCs*. Finally in our online article, Walt Lipke writes about evaluating whether or not a project will be on time and within budget in *The Probability of Success*.

Development of real-time software is often more tricky than development of other software, and the resulting software's performance is often more critical than other software. If you are currently developing real-time software, I hope you find new and useful information in this month's issue of CrossTalk. If you are in a position where you need to learn about real-time software development, I hope these articles will start you down the right path.

Elizabeth Starrett
Associate Publisher

An Introduction to Real-Time Programming

Dennis Ludwig
Aeronautical Systems Center

Real-time programming requires that you consider things that are hidden from high-level application programmers. Some of those considerations are choice of hardware, operating system, and programming language. Although these same choices have to be made by every programmer, the real-time programmer makes the choice from a different set of options.

Real-time programmers must have a intimate relationship with computer hardware, and if there is one, the operating system. Thus, another name for real-time programming is low-level programming, and at this low level, the silicon world looks a lot different from high-level application programming. This article will familiarize the reader with some of the terms and considerations of real-time programming like the selection of hardware, operating systems, and high-level language selection.

Definitions of *real time* vary [1]. Savitzky [2] provides two definitions, but here, a real-time system is defined as one in which the timing of the result is as important as the logical correctness. These systems can be classified as hard or soft. In a hard real-time system, critical computations have deadlines, and if the deadlines are not met, the system has failed. In a soft real-time system, missing a deadline may not be a failure [3]. Soft deadlines are based on average performance. To be considered hard, the computation times must be deterministic. A system is deterministic if it has the ability to respond to an event in a predictable period of time [4].

Another system classification is embedded or not embedded. The most common definition of an embedded system is one that is part of a larger system. The author prefers to define an embedded system as one that does not interact with a human. Inputs come from a sensor and outputs go to a controller, as opposed to the familiar keyboard input and video display output. Most real-time systems are embedded and consist of machine communicating with machine.

Some real-time systems are *synchronous*, but most are *asynchronous*. A synchronous system has a clock that keeps track of time and provides timing signals. An asynchronous system can accept inputs from the outside world at any time; there is no common timing signal to warn or to synchronize input. The terms synchronous and asynchronous are also applied to message passing, where they have different meanings. In message passing, synchronous means the sender waits for the message to be received; asynchronous means the sender can proceed immediately after sending a message [5].

Many real-time systems must do simultaneous tasks, and making these tasks coordinate available resources is one aspect of real-time programming. Available resources might be physical such as access to hard storage, a printer, an input/output (I/O) port, or they might be logical such as a non-shareable code segment. The following are the types of issues that real-time programmers handle: "How long will each task take to complete?" "How soon can another task be scheduled and start to run?" "Which task is most important?" "What happens if one task takes too long?" "Do the tasks have to communicate, and if so, how?"

> *"Many real-time systems must do simultaneous tasks, and making these tasks coordinate available resources is one aspect of real-time programming."*

In designing real-time systems, choices have to be made about hardware as well as software. Software decisions include operating system considerations as well as language and algorithm choices.

Hardware

Inexpensive hardware choices for controlling a real-time project include microcontrollers like the Motorola 68HC11, Intel 8051, or PIC 16F84. See [6] for an overview of more choices. A personal computer system could be used if the operating system allows access to peripheral ports. The basic requirements are input, some processing capability, and an output.

A microcontroller is optimized for data acquisition and control purposes. It contains a central processing unit (CPU), random access memory, read only memory, serial and parallel I/O ports, an analog to digital converter, and a timer circuit. All of these systems communicate via a data bus. Microcontroller folks do not refer to the CPU as a microprocessor, just as the CPU [7]. However, a microcontroller can be defined as a microprocessor with special hardware support [8].

There are hundreds of microprocessors classified as either eight, 16, or 32 bit. The Lego Mindstorms robot uses a Hitachi eight-bit H8/3292 microprocessor [9]. Another classification is reduced instruction set computing (RISC) or complex instruction set computing (CISC). RISC processors use simple instruction sets, few memory references, lots of registers, and pipelined instruction sets, but that does not make them better for all tasks [10].

Real-time processors have one or more interrupt request lines (IRQ) to connect to peripherals. When a peripheral wants CPU attention, it asserts the IRQ. This is considered an asynchronous event. If the CPU services the request, the current task is preempted, the program counter and other registers are saved on the stack, and a jump is made to the location of an interrupt service routine (ISR). The ISR is the software associated with the device causing the interrupt. When the ISR is finished, the state of the processor is restored, and execution is continued. The program counter and register states for a task are called the context of the program.

Computer program execution is a sequence of synchronous events con-

trolled by a program counter and a system clock. A software error, like a divide by zero, may cause an exception or system trap. Traps are not the same as interrupts, but they are usually handled the same way. Interrupts and asynchronous events are externally caused while traps are considered synchronous even though they are unexpected. Traps usually result from software errors.

Some items to consider when choosing hardware are the amount of random access memory needed, whether floating point assistance is required and provided, and the granularity of the system time base. The number of processors used is another decision that will also affect the software needed. In a multiprocessor system, several CPUs operate simultaneously and share the processing workload.

One method that is used to distinguish microprocessors is the millions of instructions per second (MIPS) performance (or Meaningless Indicator of Performance for Salesmen, according to [10]). One form of MIPS ratings, called relative MIPS, measures how many instructions a VAX 11/780 could have executed in the same amount of time a given computer can run a benchmark program. Different computer architectures and other factors make this rating less useful, even misleading, but it is still used [11].

Clock speed is also useless as a measure of performance because processors vary in the number of clock cycles required for memory access and other instruction executions [12].

Besides a complicated choice of hardware issues, the real-time programmer has different software issues to consider. Data structures, control structures, and operating systems look different from a low-level perspective.

Data Structures

Real-time programmers have to deal with some data structures that are normally hidden from high-level programmers. The task control block is where the CPU stores the state of the last run task so it can be restored.

The semaphore, invented by Edsger Dijkstra[1], is used to coordinate processes and shared resources. There are two types of semaphores: binary and counting. A binary semaphore is used to provide mutual exclusion. A counting semaphore is used when a resource can be used by more than one task at a time [13]. The basic counting type is an integer variable that is accessed only through two basic operations, wait and signal; however, an initialize operation is also usually provided. Modifications to the integer value of the semaphore must be executed without interruption.

A macro is a label that replaces a block of instructions that is used more than once, but only coded once. It differs from a subroutine in that the assembler inserts the code where the call is made rather than having a jump-to-it command. It works by text substitution and is usually faster than a subroutine but takes up more memory.

A pipe is a stream of data used to connect tasks, or to provide task communication. A buffer, like a first-in-first-out buffer, can implement it. This eliminates the need to use a file to store temporary results. A pipe self-regulates its flow so that it uses less disk space than a temporary file [14].

A script is a file of characters used for input or instructions to a program. The programmer can use it to simulate an interactive user or other I/O device. A script file could be a list of commands for a command interpreter such as a batch file [15].

"Remember that an interrupt request is a request. The processor may have some critical processing to finish before it responds and services the request."

A communications port consists of a queue to hold messages and two semaphores. One semaphore controls producers, or the process that generates messages, and the other controls consumers, which are the processes that use the messages.

Control Structures

Two basic software control structures are the polling loop and event-driven systems. In a polling loop, the program examines each input in turn to see if an event has occurred. The program structure is a loop, and the inputs to be examined are predetermined. If an event occurs, the polling is stopped, some action is taken, and the polling continues. Controlling refrigerator temperature could be done with a simple polling loop. The temperature would be read as input and the compressor turned on or off based on the reading. If the temperature is within controlled limits, no action is taken.

There are three kinds of event-driven systems: foreground/background, multitasking, and multiprocessor [16]. In an event-driven system, the program loops (sometimes called a spin loop) until an interrupt occurs, at which time the loop stops and services the interrupt, and then continues. Interrupt latency is the interval of time measured from the instant an interrupt is asserted until the corresponding ISR begins to execute. Remember that an interrupt request is a request. The processor may have some critical processing to finish before it responds and services the request.

Context switching time is the time the operating system takes to store the state of the processor or the contents of the registers before it begins to process another task. Because the context switching time and interrupt latency may not be constant times, making the system predictable can be a challenge for the real-time programmer.

Microcontrollers come with a monitor program that allows programmers to develop and execute software. Do not confuse this monitor with the screen monitor. The word monitor is also used for a shared data structure that contains a semaphore [2]. A monitor for a microcontroller is a program that combines a debugger, some device drivers, and a bootstrap loader program. If provided, it is usually part of the read-only memory. The bootstrap program initializes the system by setting the registers to known states, and then it calls in or loads the rest of the required software routines. A monitor may include an assembler, which is a program that translates source code into object code, and can also produce a listing file.

A linker combines one or more object code files to produce a hex file. Two standard formats for the hex file are Intel hex and Motorola S record files. These are American Standard Code for Information Interchange (ASCII) files so they can be transported through serial ports. A loader converts the hex file into an executable form called a binary file [17].

The foreground/background system is basically a polling loop with interrupts enabled. The loop runs in the background. Only critical processing is done inside the interrupt.

Multitasking is a technique to allocate

CPU processing time among several tasks. While an executing task is using the physical processor resources, other tasks have their resources stored in memory. These resources include the program counter, stack memory area, and stack pointer. These systems are classified as preemptive or nonpreemptive depending on whether they can preempt an existing task or not. In a preemptive system, each task is given a time slice.

Multiprocessor systems have more than one processor. For more information on design considerations for multiprocessor systems, see [18]. Multitasking and multiprocessor systems usually require an operating system to provide task synchronization and inter-task communication.

Operating Systems

Some operating systems are dedicated to a particular controller board. Some are designed exclusively for real time but not a specific board, and others are general-purpose programs that have been enhanced to provide real-time services.

Other names used for software routines that control processing are the executive, monitor, task manager, or kernel. These terms are sometimes used interchangeably. A program that sits quietly in the background until it is called to perform its task is called a daemon.

Some operating systems are available with the source code, but many are not. If a bug appears in the code, and source code is not available, then the programmer has to work closely with the vendor to resolve the problem. A freeware real-time multitasking kernel with source code can be found at Embedded Systems Programming <www.embedded.com> or at <www. ucos-ii.com>.

Information on some real-time operating systems (RTOS) can be found at <www.rtlinux.org>, <www.aero.polimi.it/~rtai>, <www.qnx.com>, <www.windriver.com>, or <http://seg.iit.nrc.ca/projects/harmony>. Other operating systems could be used (like MSDOS) for very simple real-time tasks even though they are not optimized for real time as long they provide access to the system I/O ports. Usually a RTOS must support multithreading, provide timing features, be predictable, and run with low overhead.

Operating systems are complex programs that interface hardware with user programs. Some modules that make up an operating system are the scheduler, dispatcher, context switch, memory manager, inter-process communication module, real-time clock manager, interrupt manager, and file system manager.

The scheduler is sometimes called the dispatcher [19]. The purpose of the scheduler is to select a process from among those ready to run, schedule time for it on the CPU, and maintain a list of ready processes.

The dispatcher dispatches jobs to the CPU, using the list created by the scheduler. Most real-time operating systems use a priority-based preemptive scheduler to keep the system in order. Priority-based means that some type of priority scheme will be used to determine how the schedule is made. Preemptive means that a task can be stopped, or another task can be preempted. In a nonpreemptive system, a task must run to completion or until it suspends itself.

"Predictability is extremely important in real-time programming, and to get it, you need to keep track of time. Response time is the time it takes the computer to recognize and respond to an external event."

A task gives up processor control when it terminates, when it voluntarily suspends, when its time slice is up, or when a higher priority task becomes available and the scheduler preempts the running task to let the higher priority one run. Preemptive ability reduces priority inversion, which is having a higher priority task wait on a lower priority task. Priority inversion cannot be prevented, but it can be reduced. The scheduler also preempts a task when its time slice is up in order to keep one process from completely controlling the CPU and blocking other tasks from running. The scheduler is the part of the operating system that decides who gets to do what and when.

If several tasks are allowed to have the same priority, they are executed in the order they become ready; this is called round-robin scheduling. In a static priority system, the priorities do not change during run time. Changing the priority of a task during run time is supported by some systems, and the algorithms for assigning dynamic priorities are different from the ones used for static priorities. One dynamic scheduling policy is the earliest-deadline-first algorithm. A static priority policy can be analyzed so the system reaction is more predictable.

If higher priority tasks keep a lower priority task from running, the condition is called starvation. The number of tasks should be kept to a minimum and careful consideration given to priority choices. The selection should be made based on what the task does during run time.

In a deadlock, two tasks are waiting for resources that are held by each other. Neither task has all the resources needed to complete, and will not be able to get them all because the other task is holding resources and waiting to get more. Tasks should be required to get all needed resources before proceeding, and they must get the resources in the same order.

For an application that will be recording, reporting, and storing data simultaneously, each task is a separate, scheduled instruction stream. In some systems, the instruction streams are called processes, in other systems they are called tasks, and sometimes they are called threads. Since some tasks are more important than others are, some sort of prioritization is employed. If a piece of code needs to be executed without interruptions or being preempted, a data structure called a semaphore is used.

Real-Time Languages

A lot of real-time programming is done in assembly language. C is popular, as well as C++ and Forth. Although Forth is an interpreted language, it is efficient because of its stack-oriented design. Java is also being used, or rather a form of Java is being used.

A language with automatic garbage collection is not a good choice for real time because it hinders determinism, but there is a working group making a real-time version of Java, the Real-Time Specification for Java.

Ada was designed for real time and is the most powerful of those mentioned. Annex D of the Ada language specification is devoted to real-time issues, and any compiler that implements annex D will also implement annex C, which is the Systems Programming Annex. The strong type checking can be turned off to increase speed by using a pragma, while

representation clauses allow mapping to the hardware. In Ada, a pragma is a directive to the compiler.

The Time Element

Predictability is extremely important in real-time programming, and to get it, you need to keep track of time. Response time is the time it takes the computer to recognize and respond to an external event. Survival time is the time during which the data will be valid. Throughput is the number of events that the system can handle in a given time period [20].

As an example, consider a red traffic light with a queue of cars waiting to go through. When the light turns green, it takes time for the first driver to comprehend that it is time to go. There is some reaction time for them to move the foot from the brake to the accelerator. The second car undergoes the same time delay as the driver recognizes that the first car is moving, and he or she can now begin to accelerate. Survival time is the time the light remains green. Throughput is the number of cars that get to go through the light.

Time is a factor in reading, storing, or recording data. For the system to store data after it is sensed, a disk may be used. When the read/write heads move to the proper cylinder or track, there is some seek time involved (about 25 milliseconds) and some settling time. The electromechanical movement has to settle before the read begins. The proper head has to be activated. Rotational delay is the time spent waiting for the proper record to rotate under the head. The data transfer rate is the speed at which the data is transferred from the head to the storage medium and is determined by the rotational speed and density of the recording medium. Because these times will be different for each operation, the average times must be calculated and the worst-case times known for proper predictability to be made.

Other time factors considered by a real-time programmer are bus latency and context switching time. Bus latency is the delay incurred when the CPU needs to acquire the bus to transfer a command or data. Switching the CPU from executing one process to executing another requires saving the state, or context, of the old process and loading the context of the new process. The task that does this is called a context switch, and it takes time to execute. First, a process has to be selected from those that are ready. This is performed by the scheduler part of the operating system, and the selection process has more time to be considered and accounted.

To conceptualize how processes have to work together but still compete for resources, most courses on real time use Edsger Dijkstra's dining philosophers problem [21]. There are a group of philosophers who spend their time either thinking or eating. They sit at a round table with a bowl of rice in the middle and one chopstick on either side of them. In order to eat, they have to acquire the chopstick on the left and on the right of them, and return the sticks when finished. This is a classic synchronization problem used to demonstrate allocating resources between competing processes without getting into a deadlock or starvation mode. An Ada implementation for a solution can be found in [22] Chapter 11.

"If several tasks are allowed to have the same priority, they are executed in the order they become ready; this is called round-robin scheduling."

Tools

Some software tools used by real-time programmers include simulators, debuggers, and analysis algorithms. An instruction level simulator now supports most processors. The debugger is usually provided as part of the monitor package, and the simulator will probably have a debugger associated with it. A calculator that has hex, octal, and binary capability is very useful.

Another tool is a dump routine (the Digital Command Language dump, not the Linux dump) that allows one to dump the binary contents of a file. On Windows systems, this can be done with the debug command. To find out more about it, open a command prompt window and enter: *C:>debug/?*. For Linux, a hex editor like KHexEdit can be used.

When comparing binary files or porting from one computer to another, consideration has to be given to the way bytes are ordered within a word. In Big Endian addressing, the address of a data element is the address of the most significant byte, while in Little Endian addressing, the address of the data element is the least significant byte [23]. Everyone agrees that there are eight bits to a byte and four bits to a nibble, but the definition of a word seems to vary. A word is a grouping of bits moved and processed as a unit in computing structure [24]. With that definition, a 16-bit machine has a 16-bit word, a 32-bit machine has a 32-bit word, and on an eight-bit machine, a word and a byte are the same thing.

If all of the task periods are known in advance, a set of algorithms called Rate Monotonic Analysis can be used to predict timing and throughput requirements. Unfortunately, it is not always possible to know the task periods in advance.

Useful hardware tools are a digitized oscilloscope with memory, a logic analyzer, and a counter-timer. These tools can be used to study timing execution of a routine by altering it to set a bit on a port that can be monitored, and then compensating for the time used by the added code. In-circuit emulators can produce timing information, if one is available for the processor.

Conclusion

Real-time programming involves keeping track of time, coordinating tasks, and within limits, making events predictable. This requires an understanding of hardware timing, operating system concepts, and programming skills. Programming skills involve assembly programming as well as a high-level language. As this article has shown, there is more involved in real-time programming than in application programming for a desktop computer running a popular operating system.◆

References

1. Jensen, Douglas E. "Eliminating the Hard/Soft Real-Time Dichotomy." Embedded Systems Programming Oct. 1994: 28.
2. Savitzky, Steven R. Real-Time Microprocessor Systems. New York: Van Nostrand Reinhold, 1985.
3. Obenland, Kevin M. "POSIX in Real Time." Embedded Systems Programming Apr. 2001: 137 <www.embedded.com/2001/0104>.
4. Wood, Mike, and Tom Barrett. "A Real-Time Primer." Embedded Systems Programming Feb. 1990.
5. Savitzky 75.
6. The EE Compendium <http://eecleversoul.com>.
7. Driscoll, Frederick F., et. al. Data Acquisition and Process Control With the M68HC11 Microcontroller. Mac-

COMING EVENTS

November 12-14
2003 Federal Chief Technology Officer Summit
Washington, DC
www.vanheyst.com/CTOSummit/home.htm

December 7-11
Association for Computing Machinery SIGAda Annual International Conference
San Diego, CA
www.acm.org/sigada/conf/sigada2003

December 8-10
Inside ID Identification Solutions Conference
Washington, DC
www.insideid.com/conference.asp

December 9-10
Institute for Defense and Government Advancement SoldierTech2003
Washington, DC
www.idga.org

December 11
Real-Time and Embedded Computing Conference
Seattle, WA
www.rtecc.com/seattle

January 20-22, 2004
Institute for Defense and Government Advancement Network Centric Warfare
Arlington, VA
www.idga.org

January 26-28
Third Annual Conference on the Acquisition of Software-Intensive Systems
Arlington, VA
www.sei.cmu.edu/products/events/acquisition

March 30-31
3rd Annual Southeastern Software Engineering Conference
Huntsville, AL
www.ndia-tvc.org/SESEC

April 19-22
2004 Software Technology Conference

STC

Salt Lake City, UT
www.stc-online.org

Millan Publishers Ltd., 1994: 25.
8. Herzog, James H. Design and Organization of Computer Structures. Franklin Beedle & Assoc., 1996: 576 <www.ee.furg.br/~silviacb/Arq1.html>.
9. Sato, Jin. Jin Sato's Lego Mindstorms: The Master's Technique. Trans. Arnie Rusoff. San Francisco: No Starch Press, 2002: 55.
10. Turley, Jim. "Ten Lies About Microprocessors." Embedded Systems Programming Jul. 2003.
11. Hennessy, John L., David A. Patterson, and David Goldberg. Computer Architecture: A Quantitative Approach. 2nd ed. Burlington, MA: Morgan Kaufmann, 1996: 57.
12. Savitzky 18.
13. Labrosse, Jean J. "Understanding Semaphores." Embedded Systems Programming Oct. 1992.
14. Moritsugu, Steve, et al. Practical UNIX: Contents at a Glance. Que Corporation, 2000: 910.
15. Savitzky 121.
16. Savitzky 9.
17. Spasov, Peter. Microcontroller Technology: The 68HC11. 2nd ed. Englewood Cliffs, NJ: Prentice Hall College Div., 1996: 154.
18. Thompson, Linda M. "Designing With Multiple Processors." Embedded Systems Programming May 1991.
19. Wood, Mike, and Tom Barrett. "A Real-Time Primer." Embedded Systems Programming Feb. 1990: 23.
20. Savitzky 5.
21. Silberschatz, Galvin. Operating System Concepts. Addison Wesley Longman, 1998.
22. Department of Defense Ada Joint Program Office. Ada 95 Quality and Style: Guidelines for Professional Programmers. Herndon, VA: Software Productivity Consortium, Oct. 1995.
23. Hennessy, et. al 74.
24. Herzog 579.

Note

1. Edsger Wybe Dijkstra (1930-2002) is best known for his battle to eliminate the GOTO statement from programming. He also developed an efficient shortest path algorithm and he designed and coded the first Algol 60 compiler. Many of his papers can be found at <www.cs.utestas.edu/users/EWD>.

Additional Reading

1. Clements, Alan. Microprocessor Systems Design. PWS Publishers, 1987.
2. Comer, Douglas. Operating System Design. Englewood Cliffs, NJ: Prentice Hall, 1984.
3. Jones, Steve. "Managing Real-Time Complexity." Embedded Systems Programming Apr. 1992.
4. Sasaki, Stan. "Evaluating Timing Performance." Embedded Systems Programming Oct. 1992.
5. Spasov, Peter. Microcontroller Technology: The 68HC11. 2nd ed. Englewood Cliffs, NJ: Prentice Hall College Div., 1996.
6. VanZandt, Lonnie. "Scheduling Sporadic Events." Embedded Systems Programming Dec. 2002 <www.embedded.com/2002/0212>.
7. White Papers. "Why BlueCat Linux and Real-Time LynxOS?" <www.lynuxworks.com/products/whitepapers.php3>.
8. E. Douglas Jensen's Real-Time for the Real World <www.real-time.org>. (This site has a fun clock to play with as well as much information about real-time computing.)
9. Software Engineering for Real-Time Systems Laboratory <www.enee.umd.edu/serts/bib/index.shtml>.
10. University of North Carolina. "Research in Real-Time Systems at UNC" <www.cs.unc.edu/Research/dirt/real-time.html>.
11. Jim Turley's Silicon Insider <www.jimturley.com>.
12. In-StatMDR. "Microprocessor Report." <www.MDRonline.com>.

About the Author

Dennis Ludwig is a computer engineer at the Simulation and Analysis Facility, Aeronautical Systems Center at Wright-Patterson Air Force Base in Ohio. He has worked with software for more than 20 years. He has a Bachelor of Science in electrical engineering from Louisiana Tech University, a Master of Science Administration from Georgia College, and a Master of Engineering from Mercer University.

ASC/HPEI
2180 8th St.
B145, R 225
WPAFB, OH 45433-7204
Phone: (937) 255-7887
DSN: (785) 255-7887
E-mail: dennis.ludwig@wpafb.af.mil

The Ravenscar Profile for Real-Time and High Integrity Systems

Brian Dobbing
Praxis Critical Systems

Alan Burns
University of York

The Ravenscar Profile offers a unique opportunity to developers of real-time and high integrity systems. For the first time in the history of our industry, there is direct support for constructing deterministic, concurrent software within an international standard programming language. The Ravenscar Profile is founded on state-of-the-art, deterministic concurrency constructs defined in ISO standard Ada95. This results in a set of building blocks that are basic enough for constructing most types of real-time software, while also being sophisticated enough to minimize the risk of error associated with using low-level primitives such as not releasing a lock on all paths. These building blocks are also amenable to the many forms of analyses that can be applied during development to assure the correctness of complex real-time programs, including scheduling and response time analysis, data and information flow analysis, exception freedom, and formal analysis using theorem provers and model checkers. As a result, nonfunctional requirements such as timing and ordering constraints and resource utilization can be established early in the life cycle with consequent reductions in cost, delays, and risk of failure.

The Ravenscar Profile is now established as a state-of-the-art model for building safe and reliable real-time systems. The profile was originally defined in 1997 at a workshop of international real-time experts and is named after the village of Ravenscar in northern England where the workshop was held. It is specified as a subset of concurrency features in Ada95 that exhibit determinism in key areas such as timing, memory usage, and function behavior. The original definition has been slightly refined in light of the application experience, and the final definition is being incorporated into the next ISO standard revision of the Ada language [1].

Although the Ravenscar Profile is specified in Ada terms, it is based on a language-independent set of building blocks that are suitable for constructing typical real-time systems, and as input to analysis tools that provide evidence that the concurrency requirements of the system have been met.

Traditional methods of implementing concurrency in a predictable way have focused on approaches such as using cyclic executives that repeatedly execute a set of functions in a fixed order in preset time frames. However, such approaches have become inadequate as system complexity has increased and the burden of maintaining correct static timelines during system upgrade becomes prohibitive. This has led to wider acceptance of concurrent programming as the preferred approach.

Yet the quality of evidence for concurrency properties traditionally has been rather low. This is because guarantees such as sufficiency of scheduling to meet deadlines, accuracy in timing behavior, correct execution-time ordering of events, and correct levels of protection for access to shared data are difficult to establish for all possible operational scenarios by testing alone. In addition, the tools that may be used to provide this kind of evidence require specialized inputs that define deterministic timing behavior, often based on using a specific kind of model or restricted source language, and supported by a real-time operating system with totally deterministic timing characteristics.

> *"The advent of implementations of the Ravenscar Profile ... heralds the availability of the most rigorous environment for developing high-integrity concurrent programs."*

The advent of implementations of the Ravenscar Profile, including some with supporting evidence certifiable to standards such as Radio Technical Commission for Aeronautics, Inc. (RTCA) Defense Order (DO)-178B [2] level A, heralds the availability of the most rigorous environment for developing high-integrity concurrent programs. This offers a unique opportunity to developers of real-time and high integrity systems to be able to demonstrate early in the life cycle that nonfunctional requirements (such as failure modes, timing and ordering constraints, and predictable resource usage) are satisfied rather than discovering deficiencies during the integration phase when corrections are often very difficult and costly to implement.

In this article, we present the following: the motivation behind the creation of the Ravenscar Profile, a brief definition of its specification, the ways in which it may be used with verification tools to produce evidence of dependability, and a short concluding example.

Motivation

The major drivers that influenced the definition of the Ravenscar Profile are as follows:

- Inclusion of reliable and predictable building blocks for real-time systems.
- Elimination of non-deterministic and highly complex concurrency constructs.
- Support for a variety of application-level analytical verification models and techniques.
- Practical generation of formal evidence of safety and reliability certification for the implementation.

These drivers are highly complementary. The overall goal is twofold. First is the ability to develop application software that includes concurrency and interrupt-related activity in such a way that is suitable for analysis by sophisticated verification tools and techniques. Second is the ability to show early in the life cycle that the software implementation meets high integrity and safety-critical requirements.

The verification tools can provide evidence to the highest levels of assurance that the software meets the related requirements while also being free from run-time error. Examples of the kinds of verification tools and techniques that may be used with the profile include the following:

- Scheduling analyzers and response-time analyzers to show that all hard deadlines and data freshness require-

ments are met.
- Model checkers to show that the required system states exist and can be reached, and that no undesired states can occur.
- Static analyzers and formal proof tools to show that the code has been correctly constructed to meet its design specification and is free from run-time exceptions.

The motivation behind the execution environment to support implementations of the profile is to satisfy the following real-time and high integrity constraints: the footprint is small; the scheduling, synchronization, and timing characteristics are deterministic; the timing accuracy is at the resolution of the underlying system clock; and the run-time support library is simple enough to generate evidence of predictability, reliability, and safety.

Definition

The Ravenscar Profile is formally defined in terms of Ada95 constructs, and this definition has been accepted for inclusion in the revision to the ISO standard definition of the Ada language that is scheduled for 2005 release. The full definition is contained in a guide on using the Ravenscar Profile for high integrity systems [3]. The main components of a Ravenscar program are as follows:

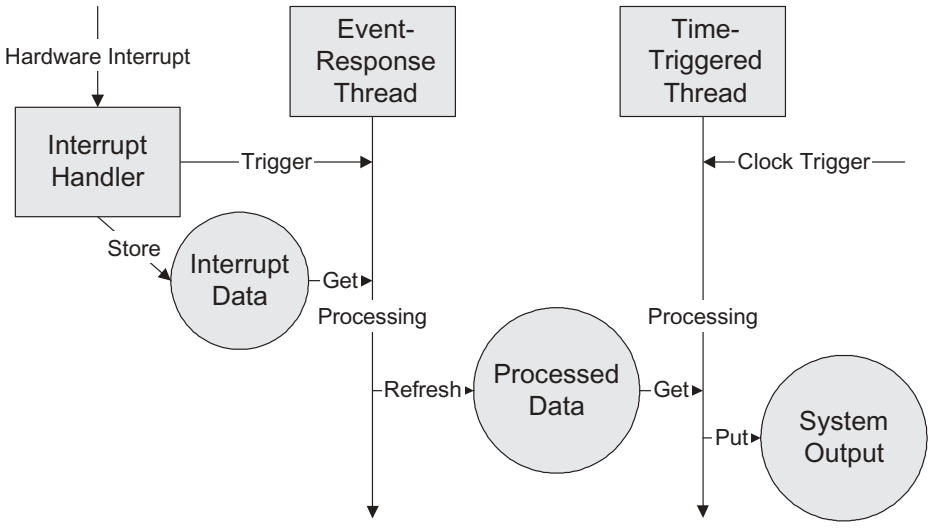
1. A fixed set of threads that may be cyclic (time-triggered) or aperiodic (event-triggered), including a thread parameterization mechanism.
2. A fixed set of protected objects that provide mutually exclusive access to shared data, including a protected object parameterization mechanism.
3. A fixed set of synchronization objects that provide suspend/resume capability for threads, including the communication of protected data as part of resumption.
4. A fixed set of interrupt handlers that may store data under mutually exclusive protection.
5. A synchronous delay facility based on absolute time values that are accurate to the resolution of the underlying system clock.
6. A deterministic fixed-priority preemptive thread scheduling policy.
7. A policy to enforce mutual exclusion that prevents deadlocks and minimizes the worst-case time that a thread is blocked due to contention.

Figure 1 illustrates the combination of some of these components. In this small example, a hardware interrupt is delivered when fresh external data is available to the system. The interrupt handler stores the data and triggers a response thread to process it. The processed data is stored in a shared data store, and a cyclic thread periodically obtains the latest version of this data to further process it to drive a system output.

Verification

Each thread of control is independently verified for conformance to its specification. This includes a demonstration of meeting its functional, performance, and resource utilization requirements, for example, by performing requirements-based testing or by using static analysis methods. Then the program as a whole is verified against all of its concurrency requirements, which include the following:
- Synchronization and communication interactions.
- Freshness of shared data.
- Execution order dependencies.
- Timing constraints such as meeting deadlines.

Figure 1: *Examples of Ravenscar Profile Building Blocks in Combination*



In each of these cases, sophisticated tools and techniques exist to automate the verification process to show that concurrency requirements have been met and to produce supporting evidence for a regulatory authority if necessary. The tool-based approach can be used early in the development life cycle and also simplifies the process of re-verification (and perhaps re-certification) after the system has undergone modification during maintenance or a midlife update.

In the rest of this section we look at three currently supported techniques for concurrency verification:
1. Static analysis.
2. Scheduling analysis.
3. Formal analysis.

Static Analysis

Existing static analysis tools and techniques can be used to achieve high levels of proof of correctness and absence of run-time errors in sequential programs, for example, see [4, 5]. The SPARK language recently has been extended to support the Ravenscar Profile as its concurrency model in such a way as to preserve the same level of integrity assurance as is possible for sequential programs [6]. This is a major advance in the extent of achievable confidence that concurrent programs are provably correct and cannot result in run-time exceptions being raised.

At the thread level, relating to an individual task or interrupt handler, the analysis is largely unaffected by the addition of concurrency constructs. In particular, the static analysis does not consider the temporal aspects – for example, the thread-level data and information flow analysis assumes that the thread will be activated after suspension at some stage.

The main change to existing sequential flow analysis is that references to shareable, protected objects must be considered volatile at all times because the value read may be generated by another program thread at any time. In particular, if a thread writes a shareable, protected object and later reads it, there can be no assumption that the value written will still be there when the read is performed. This volatility is already supported for sequential programs that access external data such as via an input/output port. Having modeled the volatility of shared data in this way, the existing benefits of proof of correctness and absence of run-time errors can be realized for each thread.

At the program level, the major extension to static analysis to support concurrency is to be able to describe the intended data and information flow across

thread boundaries, and then to verify that the actual program achieves it. The check is realized by the composition of each thread-level flow analysis with those of the thread interactions via shareable, protected objects. The intended program-wide flow relation can then be compared automatically with the computed actual flow, and any discrepancies reported as errors.

A valuable side effect of this analysis is in the assurance that the constructed program is complete. If a thread were inadvertently omitted from the program build, the computed flow analysis would not take into account the data interactions caused by that thread – hence the computed flow analysis would report an error when compared with the intended flows.

Scheduling Analysis

Recent research in scheduling theory has found that accurate analysis of real-time behavior is possible given a careful choice of the scheduling/dispatching method together with suitable restrictions on the interactions allowed between threads, for example, see [7] chapter 13. An example of a scheduling method is *preemptive fixed priority scheduling*. Example analysis schemes are *Rate Monotonic Analysis* (RMA) and *Response Time Analysis* (RTA). Preemptive fixed priority scheduling is generally used with a deterministic mutual exclusion policy such as *Priority Ceiling Protocol* (PCP) to avoid unbounded priority inversion and deadlocks. This provides a model suitable for the scheduling analysis of concurrent real-time systems that is also scaleable to programs for distributed systems.

This model supports *cyclic* and *aperiodic* threads that communicate and synchronize in a controlled way, and that each may have timing deadlines. These deadlines may be the following:

- **Hard.** When the failure to meet the deadline is an unacceptable failure of the system.
- **Firm.** When occasional missed deadlines can be tolerated but there is no value in completing the action when the deadline is missed.
- **Soft.** When occasional missed deadlines can be tolerated and there is value in completing the action when the deadline is missed.

The Ravenscar Profile requires using the standard preemptive fixed priority thread scheduling policy known as *First-In-First-Out* (FIFO)_Within_Priorities, and using PCP.

Extensive and mature tool support exists for both RMA and RTA, and for the static simulation of concurrent real-time programs. The primary aim of analyzing the real-time behavior of a system is to determine whether it can be scheduled in such a way that it is guaranteed to meet its timing constraints. Whether the timing constraints are appropriate for meeting the requirements of the system is not an issue for scheduling analysis. Such verification requires a more formal model of the program and the application of techniques such as model checking (see below).

Formal Analysis

The formal analysis of concurrent programs has been a fruitful research topic for a number of years. Current standard techniques allow many important properties of a concurrent program to be statically checked, for example the following:

- **Dependability.** The set of threads should not enter any undesirable state (for example deadlock, livelock).
- **Liveness.** All desirable states of the set of threads must be reached eventually (that is, useful progress should always be made).

In a real-time concurrent system, liveness becomes *bounded liveness* since desirable states must be reached by known deadlines.

Standard programming languages do not have their semantics defined in a formal mathematical way. Hence it is necessary to link a model of the program to the program itself. This link cannot be formal, but can be precise. Using standard patterns as found in the Ravenscar Profile helps this linkage. The formal model could be derived from the code or, more likely in an engineering process, the model is derived from requirements, and the code is obtained via a series of refinements from the model.

Verification is via either a proof-of-theoretic approach or model checking. An algebraic description can be proved to be deadlock free, for example, by using a theorem prover. Alternatively, a state transition description can be exercised by an exhaustive search of the set of states the program can enter. This *checking of the model* can deduce that all desirable states, and no undesirable states, can be reached.

For real-time systems, it is possible to add time parameters to the concurrency model and to then validate temporal aspects of the program. A common formalism for this type of state transition system is called a timed automaton. Tool support for model checking sets of timed automata is well advanced. One of the very useful features of model checking tools is that they all produce a well-defined counter example for any failed check.

There is already experience using model checking to validate Ravenscar programs. It is possible to add worst-case and best-case execution times for state transitions and to then check that deadlines are never missed. Alternatively, model checking can be used to validate the top-level description of the timing constraints, leaving scheduling analysis to check deadline satisfaction once execution times from the implementation are known. Typical of the verification that can be achieved with this approach is to check some end-to-end deadline through a number of threads, assuming that each thread itself meets its timing requirements.

Implementations

There are several mature implementations of the Ravenscar Profile in Ada95. These include Ada run-time systems that execute directly on the target board, and those that rely on services provided by a commercial off-the-shelf (COTS) real-time operating system (RTOS). Some of these implementations are part of systems that have achieved formal safety certification to the highest integrity level, for example Radio Technical Commission for Aeronautics, Inc. (RTCA)/ Defense Order (DO)-178B [2] Level A. In addition, there is growing support for the profile's building blocks within COTS RTOSs in a high-level language-neutral manner such that C programs can use them, for example.

Example

We can apply concurrency verification techniques to the simple example in Figure 1. By assigning deadlines to both threads, model checking would be able to verify that data from the interrupt was sufficiently fresh when used to influence the system output. Moreover once the execution times were known for the two threads (and the interrupt handler), it would be possible to use scheduling analysis to confirm that the deadlines for each thread would indeed be met in all possible executions of the program. Finally, static analysis could show correct system-wide data and information flow, for example, that the occurrence of the interrupt directly affects the system output, as well as absence of run-time errors.

Figure 2 (see page 12) shows some Ada code fragments for the expression of the example in Figure 1 using Ravenscar Profile constructs.

Conclusion

The Ravenscar Profile offers a unique

```
protected Interrupt_Data is
   procedure Handler; -- The interrupt handler code
   pragma Attach_Handler (Handler, Interrupt);
   entry Get (New_Data : out Raw_Data); -- Retrieves the interrupt data
private
   The_Interrupt_Data : Raw_Data;
end Interrupt_Data;

protected Processed_Data is
   procedure Refresh (New_Data : Data); -- Updates the processed data
   procedure Get (Latest_Data : out Data); -- Gets the latest data
private
   The_Processed_Data : Data;
end Processed_Data;

task body Event_Response is
begin
   loop
      Interrupt_Data.Get (New_Data); -- Waits until new interrupt data is available
      Process (New_Data, Output_Data); -- Processes it
      Processed_Data.Refresh (Output_Data); -- Writes the new processed data
   end loop;
end Event_Response;

task body Time_Triggered is
begin
   loop
      delay until Next_Period; -- Waits until start of next cycle
      Processed_Data.Get (Latest_Data); -- Gets the latest processed data
      Process (Latest_Data, New_Output); -- Processes it further
      System_Output.Write (New_Output); -- Writes the new system output
   end loop;
end Time_Triggered;
```

Figure 2: *Ada Code Fragments of Ravenscar Profile Constructs*

opportunity to developers of real-time and high integrity systems to establish high levels of confidence in the verification of concurrency properties and requirements early in the development life cycle. The profile defines a set of building blocks for constructing deterministic, concurrent software. The benefits of using the Ravenscar Profile include portability via international standardization, plus support from a wide range of sophisticated analysis tools. In addition, there exist implementations of the profile to the highest levels of safety certification. As a result, there is the opportunity to minimize the risk of deploying complex concurrent systems containing errors that are hard to find by testing methods alone, both during initial production and during long-term maintenance.◆

References

1. *Ada95 Reference Manual*. Cambridge, MA: Intermetrics, 1995. International Standard ISO/IEC 8652:1995(E) <www.adahome.com/rm95> and <www.adapower.com/rm95/index.html>.
2. RTCA-EUROCAE. Software Considerations in Airborne Systems and Equipment Certification. DO-178B/ED-12B. Dec. 1992 <www.rtca.org>.
3. Burns, Alan, Brian Dobbing, and Tuillo Vardanega. "Guide for the Use of the Ada Ravenscar Profile in High Integrity Systems." York, United Kingdom: University of York, Jan. 2003. Technical Report YCS 348 <www.cs.york.ac.uk/ftpdir/reports/YCS-2003-348.pdf>.
4. Barnes, John. High Integrity Software – The SPARK Approach to Safety and Security. Addison-Wesley, 25 Apr. 2003.
5. Chapman, Rod, and Peter Amey. Industrial Strength Exception Freedom. Proc. of ACM SIGAda, Houston, TX, 2002 <www.sparkada.com>.
6. Amey, Peter, and Brian Dobbing. High Integrity Ravenscar. Proc. of Reliable Software Technologies – Ada-Europe 2003, Toulouse, France, June 2003. <www.sparkada.com/downloads/high_ integrity_ravenscar. pdf>.
7. Wellings, Andrew J., and Alan Burns. Real-Time Systems and Programming Languages. 3rd ed. Addison-Wesley, 5 Apr. 2001.

About the Authors

Brian Dobbing is a principal consultant at Praxis Critical Systems. He was a key member of the International Real-Time Ada Workshop that defined the first version of the Ravenscar Profile, and has been heavily involved in the evolution of the profile ever since. He was the chief architect of the first implementation of the profile, the Aonix product ObjectAda/Raven, that achieved formal safety certification to RTCA DO-178B Level A. He is a member of ISO Working Group 9 (Ada) and of the ISO Annex H Rapporteur Group (high integrity systems).

Praxis Critical Systems Ltd
20 Manvers St.
BATH BA1 1PX
U.K.
Phone: +44 1225 823762
E-mail: brian.dobbing@praxis-cs.co.uk

Alan Burns is head of the Computer Science Department, personal chair, and professor at the University of York, which he joined in January 1990. He has worked for many years on a number of different aspects of real-time systems engineering. His research activities have covered all major aspects of real-time and safety critical systems. Burns recently retired as chair of the Institute of Electrical and Electronics Engineers' Technical Committee on Real Time and has chaired the Real-Time Systems Symposium. He has authored/co-authored more than 350 papers and reports and eight books. His teaching activities include courses in Operating Systems, Scheduling, and Real-Time Systems.

Department of Computer Science
University of York
Heslington
YO105DD
York, U.K.
Phone: +44 1904 432779
E-mail: alan.burns@cs.york.ac.uk

Software Static Code Analysis Lessons Learned©

Andy German
QinetiQ Ltd.

The United Kingdom Ministry of Defense has been in the forefront of the use of software static code analysis methodologies, including some of the tools and their application. This article[1] discusses what is meant by static analysis, reviews some of the tools, and considers some of the lessons learned from the practical application of software static code analysis when used to evaluate military avionics software.

Most software errors are relatively harmless, albeit annoying, such as when a word processor crashes. However, errors in some types of software can have serious consequences such as the failure of an aircraft's flight control software, which could be catastrophic. Software that controls a system whose failure could endanger human life or the aircraft is termed *safety-critical software*. Its integrity is of great concern to developers, users, the public, and the certification/regulatory authority.

Recent large-scale assessments of avionics software have produced some interesting results that show how important language selection is when producing safe and reliable avionics. This article presents the following information:

- Covers some of the methods used to identify safety-critical software and functionality.
- Discusses some myths of static code analysis.
- Describes some static analysis techniques.
- Identifies some of the tools available.
- Provides some general results of the practical application of static code analysis.

Static Code Analysis

Safety-critical software must be shown fully predictable in operation and have the properties required of it [1]. In addition to dynamic testing, such code is also subject to static testing: This is the rigorous examination of software (without running it dynamically) to establish the properties that will always hold true under any operating condition. It is an examination of the code, the architectural design, and the accompanying documentation, which together provides a picture of the completeness, or otherwise, of the software system [2].

There are various techniques that come under the umbrella term *static code analysis*, and these can be characterized by their nature and depth [3]. Nature refers to the broad objectives of the analysis and could be concerned with specific properties such as portability. Depth means the analytical depth of the technique.

Identification of Safety-Critical Software

The United Kingdom (UK) Ministry of Defense (MoD) adopted the safety argument approach in 1992, as retrospective evaluation of avionics systems had become complicated. The MoD still operates the *lessons learned/best practice* approach that is used as part of the safety argument evidence. The system design standards are used to trap system safety design requirements; these are Defense Standard 00-970 [4] and Defense Standard 00-971 [5] for aircraft. The safety argument approach is now used for the complete aircraft and has major advantages; it does not limit the possible design solution by being over-prescriptive, and it can cope with rapidly changing technology.

The current preferred method for safety-critical code functionality identification (including system robustness) is to use a top-down analysis starting with the defined safety targets for tolerable catastrophic mishap rates, including aircraft loss. Recent aircraft projects have shown that *bottom-up* hazard identification produces somewhere between 700 and 1,500 hazards. The bottom-up approach does not prioritize the hazards or show their relationship to the system as a whole; their true categorization is unknown. These large numbers of hazards are difficult to manage, and so a top-down evaluation is used to refine the argument.

The top-down approach normally results in approximately 100 of the most significant hazards being identified from approximately 10 top-level accidents/events. The Hazard and Operability (HAZOP) [6] approach to system and software functionality assessment has demonstrated itself to be an invaluable tool, particularly for defining system robustness requirements. This approach allows the system designer and regulatory authorities to show, through reasoned argument, that the following occur:

- Hazards are identified.
- Safety functionality is understood.
- Robustness requirements are identified.
- Hazards are mitigated to a tolerable level.

The first and best step in hazard mitigation is to avoid using safety-critical software wherever possible.

Why Use Static Code Analysis?

For UK defense projects, Defense Standard 00-55 [7] is normally recommended. This standard details two basic approaches to safety critical software:

- The use of formal methods (correct by design).
- The static analysis of the code (conformance with the design).

These are coupled with the following:

- Selection of a suitable high-level language (including its subset definition where appropriate).
- Defensive programming.
- Independence in verification and validation activities.
- Comprehensive documentation and configuration management.
- Testing and test coverage.
- Compiler validation.

The formal methods approach has not been widely adopted for the following reasons:

- Some of the most recent aircraft entering service started development back in the late 1970s when formal methods tool and support was severely limited. This is also prior to Defense Standard 00-55 initial issue requiring the use of formal methods.
- More recently, it is because of the short lead times and hence the extensive use of commercial off-the-shelf components for which the Civil Airworthiness Authorities do not mandate the use of formal methods. The

Civil Airworthiness Authorities suggest that the use of formal methods be considered [8].

The application of static code analysis techniques in retrospect is not ideal; the process is best suited and cheapest when applied during software development. Following the UK *as low as reasonably practical* approach [9] to risk, the retrospective evaluation of safety-critical code is the only reasonable method available at present to reduce safety-critical anomalies to a minimum – after all other mitigations have been considered.

Static Analysis Myths

But We Test It

All software contains errors, and computer programs rarely work the first time [10]. Usually, several rounds of rewriting, testing, and modifying are required before a *working* solution is produced [11]. Testing usually involves running and evaluating the software across its expected range of operation. This process is limited by the tester's ability to predict this range of operation, or rather, the range of inputs that the program will receive. This is how it is possible for large *well tested* software packages to still fail periodically: The user has done something not anticipated by the tester. It would be nice to test every state of a program, but such exhaustive testing is impractical as it would take far too much time and expense.

An argument often used against static analysis is that "our software has been extensively tested." This argument does not stand up. Radio Technical Commission for Aeronautics, Inc. (RTCA) Defense Order (DO)-178B [8] Level A requires extensive modified condition/decision coverage testing while RTCA DO-178B Level B does not require this level of testing. When Level A was compared to Level B, no significant difference in anomaly rates identified by static analysis was found. Unhappily, the *hack-it-and-bash-it* methodology is still prevalent among many software developers.

Static Analysis Means It Is Safe

The phrase "static analysis means it's safe" is heard quite often. Static analysis only allows us to argue that the code is as follows:

- As compliant with the software requirements as present evaluation methods and technology allows.
- That *coding errors* have been minimized.

Static analysis does not prove that the requirements the code was developed from were correct or show that the compiled code is correct.

It Costs Too Much

Based on project experience, an average 10 percent of a military aircraft's software – or approximately 500,000 software lines of code – are found safety critical. The average cost of retrospective independent analysis for an aircraft is less than $13 million, and on average less than 0.4 percent of the total development cost for an aircraft. These costs can be further reduced if the semantic analysis element is directed. It has been shown that the most costly element of static analysis is the semantic element when comparing costs of the activity to total percentage of anomalies found. One important area for future research is the justifiable targeting of techniques.

If, however, the software is designed and analyzed as part of the development process, then the cost savings are likely when compared to normal industry costs. There are also considerable through-life cost savings and system reliability benefits.

Dissimilar Systems

Although Defense Standard 00-55 [7] allows the use of dissimilar systems to be combined to create a safety-critical system, this becomes a very difficult approach to argue as being safe. The following issues need to be addressed:

- The comparison software or *liveware* (pilots) becomes safety critical (70 percent of aircraft accidents are due to aircrew error).
- How do you prove dissimilarity?
- Reliability goes down, as the lower integrity systems are likely to disagree and fail more often.
- The warning system becomes more critical.
- The cost of ownership goes up (supporting multiple equipment, increase aircraft weight, etc.).
- Designers tend to make the same mistakes.

Main Static Analysis Techniques and Methods

Control Flow Analysis (Including Cyclomatic Complexity)

Control flow analysis can be conducted using tools or done manually at various levels of abstraction (module, node, etc.) and is done for the following reasons:

- Ensure the code is executed in the right sequence.
- Ensure the code is well structured.
- Locate any syntactically unreachable code.
- Highlight the parts of the code (e.g., loops) where termination needs to be considered.

This may result in diagrammatic and graphical representations of the code being produced.

Data Flow Analysis

The objective of data flow analysis is to show that no execution paths in the software exist that would access a variable not set to a value. Tools use the results of control flow analysis in conjunction with read/write access to variables. It can be a complex activity, as global variables can be accessed from anywhere. This analysis can also detect other code anomalies such as multiple writes without intervening reads.

Information Flow Analysis

Information flow analysis identifies how execution of a unit of code creates dependencies between the inputs to and outputs from that code. These dependencies can then be verified against the dependencies in the specification. This analysis is often particularly appropriate for a critical output that can be traced all the way back to the inputs of the hardware/software interface.

Information flow analysis may be augmented in some tools by using *annotations*. These are stylized comments that document certain assumptions about functions, variables, parameters, and types. They enable an analysis to proceed more efficiently because they give it more information relevant to a particular block of code.

Path Function Analysis (Also Called Semantic Analysis or Symbolic Execution)

Path function analysis is used to verify properties of a program by algebraic manipulation of the source text, without requiring a formal specification. It involves checking the semantics of each path through a program section or procedure. Sophisticated tools give expressions for the precise mathematical relationship between inputs and outputs from a particular program section: They effectively give the *transfer function* for that program section [12]. They step through the code, assigning expressions instead of values to each variable. Thus the sequential logic is converted into a set of parallel assignments in which output values are expressed in terms of input values – this format is easier to interpret. The tools produce an output for each path consisting of the conditions that cause the path to be executed, and the result of executing that path.

Semantic analysis reveals exactly what the code does in all circumstances for the

whole range of input variables for each program section. However, there is still the need for substantial human involvement in comparing the tool's output with the specification. Compliance analysis (formal code verification) provides a reduction in human requirements and greater automation.

Formal Verification (Also Called Compliance Analysis)

This is the process of proving, in an automated process and as far as is possible, that the program code is correct with respect to the formal specification of its requirements. All possible program executions are explored, which is not feasible by dynamic testing alone. Depending on the power of the tool being used, and its simplification ability, the involvement of analysts may be large or small.

Verification conditions can enhance compliance analysis. They consist of conditions that should be valid at the start and end of a block of code (pre- and post-conditions) and are stated at the start of that block. In a way, it is like a different form in which the programmer can explain the purpose of a block of code. The analysis might start with the post-condition and work backward to the start of the block. If, on reaching the start, the pre-condition is generated, then the block of code is provably sound.

Compliance analysis effectively performs a proof of the code against a low-level mathematical specification. In this respect, it is by far the most rigorous of the static analysis techniques. However, its value depends on the availability of a specification expressed in a suitable form. Furthermore, this rigor is at the expense of cost; productivity is around five lines of code per man-day.

Independent Evaluation

Since the 1970s, independent code inspections to reduce code error have been found to be efficient and cost effective. Experience from aircraft static code analysis carried out to date shows that code walkthrough finds about 60 percent of all the anomalies found.

Other Techniques

Syntax Checks

Syntax checks aim to find language rules violations such as using a variable of the wrong type or before it is declared. The compilers of some languages such as Ada and Pascal will carry out syntax checks automatically, whereas languages like C and assembler need additional tools.

To allow the use of analysis tools, reduce the number of likely coding violations and improve code readability. It is normal to define a rule set when designing safety-critical code to allow the tools to carry out the analysis more readily and to remove some of the more problematic features. It has been found that the size of the rule sets is dependent on the language, such as the following:

- C has some 220 rules suggested [13].
- Ada 83 has approximately 80 rules.
- Southampton Program Analysis and Development Environment (SPADE) Ada Kernel (SPARK Ada) has an extensively defined rule set; sometimes a reduction in the rules can be agreed on.

Range Checking

Range checking analysis aims to verify that the data values remain within their specified ranges, as well as maintain specified accuracy. This technique can detect the following:

> *"All software contains errors, and computer programs rarely work the first time. Usually, several rounds of rewriting, testing, and modifying are required before a working solution is produced."*

- Overflow and underflow analysis.
- Rounding errors.
- Array bounds.
- Stack usage analysis.

This is a form of shared resource analysis that establishes the maximum required size of the stack, and whether there is sufficient physical memory to support it.

Timing Analysis

Timing analysis ascertains the temporal properties of the input/output dependencies, including the worst-case execution time for the correct behavior of the overall system. It can be made difficult by language features such as manipulation of dynamic data structures, loops without static upper bounds, and by using hardware with built-in pipe and cache.

Other Memory Usage Analysis

This is required for any resource that is shared between different partitions of software. It reveals the absence of conflict between the code and other low-level components such as device drivers and resource managers.

Object Code Analysis

Object code analysis demonstrates that the object code is an accurate translation of the source code and that the compiler has introduced no errors. The analysis may be carried out by manual inspection of the machine code produced by the compiler – this can be made easier if the compiler vendor provides details of the mappings from source code to object code.

Limitations

Although the various forms of static code analysis offer many advantages to the system developer, they also impose some constraints. Using these techniques restricts language choices that may be used and the choice of the structures used within these languages. Furthermore, these analytical methods require highly skilled and experienced staff to carry out the tests and analyze the results. It is not a complete answer for the validation and verification of safety-critical software even with the use of automated tools. Other forms of testing (for example dynamic) are required to verify certain aspects, like executing critical features. Some of the restrictions of static analysis using automated tools are the following:

- Multitask applications software must be analyzed a task at a time. Another form of testing is required to check task interactions.
- Dynamic aspects of the software (for example sequences of program execution) are difficult to model with static analysis techniques.
- Most automated tools require translation to an intermediate language before they can analyze the code. Automatic translators are available for some languages, but for others one must either translate manually or write a new translator. Some language features do not have an equivalent in the intermediate language even with the automatic translators; they must be manually translated. The static analysis of the software depends on its translation model and the more skilled the analyst, the more skilled the model produced. The validation of the intermediate language model needs to be considered, as this can be a major problem.

Air Vehicle Software Analysis

The practical application of static code analysis has produced some interesting results. The range of software systems that have been subjected to analysis include the following:

- Automatic flight control.
- Engine control.
- Fuel and center-of-gravity management.
- Warning systems.
- Anti-icing systems.
- Flight management.
- Stores management.
- Air data units.
- Radio altimeters.
- Anti-skid brakes.

These systems vary in size from 3,000 lines of code to 300,000 lines of code and include languages from assembler, C, Pascal, Ada to Lucol, and SPARK Ada [14].

Effects of Previous Development Methodologies (RTCA DO-178B) [8]

It is worth reiterating that when comparing RTCA DO-178B [8] Levels A and B code, no discernible difference was found by static code analysis demonstrating that static code analysis is something you carry out in addition to testing. Even the most extensive testing does not remove the anomalies found by static code analysis. Surprising amounts of *dead* code have been found in code developed to RTCA DO-178B Levels A and B.

Effects of Language

The choice of language for a computer program is important. Not only should the functionality of the language itself be considered, but also the availability and quality of support tools and the expertise within the development team. Unfortunately, safety-critical software represents only a small subset of the global programming effort; most languages are not designed with high integrity requirements in mind. More commonly, commercial factors such as productivity and ease of use steer the development.

Some languages are better suited to the production of safety-critical software than others because they make it easier to write dependable code, and easier to demonstrate its freedom from errors. However, you must bear in mind that the language itself is a product that is susceptible to design flaws: Perfect code could still produce errors when run.

The software lines of code per anomaly in Table 1 show some of the metrics found from various programs. Table 1 shows that the poorest language for safety-critical applications is C with consistently high anomaly rates. The best language found is SPARK (Ada), which consistently achieves one anomaly per 250 software lines of code. The average number of safety-critical anomalies found is a small percentage of the overall anomalies found with about 1 percent identified as having safety implications. Automatically generated code was found to have considerably reduced syntactic and data flow errors.

Software development is often performed on a different system than that used for the final application. Therefore, the portability of the code is another factor to be considered when choosing a language. That is, how easily it will run in a different environment to the one in which it was developed.

The quality of the available compilers is also important. Modern programming languages are very complex and sophisticated and hence difficult to understand. It is therefore challenging to write high quality, dependable compilers for them [15]. Widely used compilers and development tools should be used whenever possible, so that there has been plenty of opportunity for errors to be found (and hopefully rectified). This also applies to the language used, reflecting why an attractive (but little used) language such as Modula-2 might not be chosen for a safety-critical application.

Table 1: *Software Language Anomaly Rates*

Software Language	Range	Software Lines of Code Per Anomaly	Anomalies Per Thousand Lines of Code
C	Worst	2	500
	Average	6 - 38	167 - 26
	Best (Auto Code Generated)	80	12.5
Pascal	Worst	6	167
	Average/Best	20	50
PLM	Average	50	20
Ada	Worst	20	50
	Average	40	25
	Best (Auto Code Generated)	210	4.8
Lucol	Average	80	12.5
SPARK	Average	250	4

Tools Available

There are a number of static code analysis tools available. They offer different depths of analysis, and some will only operate on a few languages. Most of them run on uncompiled source code and first translate to an intermediate language, which the analysis tool itself can read.

The time taken for the tool to analyze the code may be only a small fraction of the time taken to carry out static analysis of the code. Many tools produce reams of data that must be laboriously analyzed and processed; staff requires skill and a lot of training.

Main Tools

There are three main, well-established tools used on UK military programs.

Malvern Program Analysis Suite

Malvern Program Analysis Suite (MALPAS) was developed by Royal Signals and Radar Establishment Malvern based on research they carried out and by Southampton University in the 1970s. It is now mature and since 1986 has been supplied and supported by Advantage.

Although automatic translators exist for most languages, the main ones covered are Ada and Pascal. There is no concept of pointers in the MALPAS Intermediate Language and so to analyze C, for example, the code would first have to be purged of the use of pointers – a potentially formidable task.

SPARK

SPARK is a subset of Ada for high integrity programming first formalized by Bernard Carré and Trevor Jennings of Southampton University in 1988. It has continually evolved and nowadays it is being more widely used and is gaining general acceptance particularly as its tools now run within a *lunchtime* for an average-sized safety-critical avionics program. In addition, SPARK now supports tagged types, tasking (Ada95 Ravenscar), and *proof of exception freedom*, which have particular benefits in the context of RTCA DO-178B [8].

Control flow analysis is not needed as it is subsumed into the SPARK grammar, and thus performed *on the fly*. Data and information flow requirements have been expressed as SPARK program design rules.

Liverpool Data Research Associates Ltd. Testbed

Liverpool Data Research Associates Ltd. (LDRA) Testbed was founded in 1975 and is the oldest developer and retailer of static analysis tools. Many languages are covered: Ada, C, C++, Cobol, Coral 66, Fortran, Pascal, PL/1, PL/Mx86, and Intel and Motorola assemblers. The LDRA Testbed will perform the three flow analyses, with information flow analysis enhanced by the use of annotations.

Other Tools

Other tools include the following:

- SPADE.
- QA C Programming Research Ltd.
- Cantata and AdaTEST IPL [16].
- Alsys C-SMART – Certifiable Small Ada Run-Time [15].
- Aonix RAVENSCAR.
- PC-Lint [17].
- LCLint [18].
- PolySpace Technologies [19].
- Compaq Systems Research Center – Extended Static Checking (ESC) [20].

Conclusion

The safety argument approach should be used so that safety-critical software is minimized, safety functionality is clearly identified, and analysis required is justified.

Static code analysis is an effective software analysis technique; hence, its use is recommended in the context of safety-critical software particularly when conducted constructively as part of the software development process.

If it is conducted retrospectively, it is necessary to specify the nature and depth of any analysis carried out. Static analysis techniques should be targeted by the safety arguments. Techniques for targeted, rather than *blanket* analyses are being investigated by a number of organizations. Once developed, they may reduce the cost of analysis, while maintaining the required depth in the areas of interest.

Experience with retrospective static analysis shows that independent code walkthroughs are the most effective technique for software anomaly removal. These seem to find up to 60 percent of the errors present in the code.

The use of automatic code generation should be encouraged because this seems to result in low syntactic and data flow errors.

A safe subset of Ada must be considered when selecting a language for safety-critical systems, as this will ensure anomalies are minimized. SPARK continues to prove it is the most reliable approach to safety-critical software. However, C and its associated forms should be avoided.◆

References

1. Barnes, John. High Integrity Software – The SPARK Approach to Safety and Security. Addison-Wesley, 2003 <www.sparkada.com>.
2. Graham, Buckle. Static Analysis of Safety Critical Software. Proc. of the 16th Safety-Critical System Symposium, Springer, United Kingdom, 1998 <www.safety-club.org.uk>.
3. Wichmann, B. A., A. A. Canning, D. L. Clutterbuck, L. A. Winsbarrow, N. J. Ward, and D. W. R. Marsh. "Industrial Perspective on Static Analysis." Software Engineering Journal Mar. 1995: 69-75 <http://ieeexplore.ieee.org/xpl/tocresult.jsp?isNumber=8550>.
4. Ministry of Defense. "Design and Airworthiness Requirements for Service Aircraft." Part 1, Issue 2. Defense Standard 00-970 Dec. 1999 <www.dstan.mod.uk>.
5. Ministry of Defense. "General Specification for Aircraft Gas Turbine Engines." Issue 1. Defense Standard 00-971. 29 May 1987 <www.dstan.mod.uk>.
6. Ministry of Defense. "HAZOP Studies on Systems Containing Programmable Electronics." Part 1, Issue 2. Defense Standard 00-58. 19 May 2000 <www.dstan.mod.uk>.
7. Ministry of Defense. "Requirements for Safety Related Software in Defense Equipment." Part 1, Issue 2. Defense Standard 00-55. 1 Aug. 1997 <www.dstan.mod.uk>.
8. Radio Technical Commission for Aeronautics. "Software Considerations in Airborne Systems and Equipment Certification." RTCA DO-178B. RTCA, Inc., Dec. 1992 <www.rtca.org>.
9. Ministry of Defense. Regulation of the Airworthiness of Ministry of Defense Aircraft. 4th ed. JSP318b. Nov. 1999.
10. TA Consultancy Services Ltd. "MALPAS Training Course." TACS/9093/15. T A Consultancy, 12 Aug. 1992 <www.tagroup.co.uk/index.htm>.
11. Storey, Neil. Safety Critical Computer Systems. Addison Wesley Longman, 1995.
12. Proc. of the Advisory Group for Aerospace Research and Development Conference 545 <www.rta.nato.int/rtohistory/agard.htm>.
13. Hill, M., and L. Whiting. "Risk Reduction for C Coding." Internal QinetiQ Document. DERA Malvern, 1999.
14. Harrison, K. J. "Static Code Analysis on the C-130J Hercules Safety-Critical Software." Aerosystems International, UK, 1999 <www.damek.kth.se/RTC/SC3S/papers/Harrison. doc>.
15. Aonix <www.aonix.com>.
16. AdaTEST and Cantata <www.iplbath.com>.
17. Gimpel Software <www.gimpel.com>.
18. Networks and Mobile Systems <http://lclint.cs.virginia.edu/guide/index.html>.
19. Polyspace Technologies <www.polyspace.com>.
20. Compaq Extended Static Checking for Java <www.research.digital.com/SRC/esc/Esc.html>.

Note

1. This article is based on a paper that was presented to the Safety Critical Systems Club as "Air Vehicle Software Static Code Analysis Lessons Learned – Ninth Safety Critical Club Symposium," Bristol, United Kingdom: Springer, Feb. 2001, ISBN 1-85233-411-8.

Additional Reading

1. Ministry of Defense. "Safety Management Requirements for Defense Systems." Part 1, Issue 2. Defense Standard 00-56 Part 2 Issue 2. 13 Jan. 1996 <www.dstan.mod.uk >.
2. QAC <www.programmingresearch.com>.

About the Author

Andy German leads a group of software and safety engineers for the Test and Evaluation Sector of QinetiQ Ltd. This group provides the United Kingdom (UK) Ministry of Defense with independent certification advice for large military aircraft and Unmanned Air Vehicles. Andy has worked in the UK defense sea and air system sectors for the past 21 years and will complete a Master of Science degree in Safety Engineering from Lancaster University this year.

QinetiQ Ltd.
Safety and Signature Evaluation
Test and Evaluation Services
Room G10, Bldg. 498
MoD Boscombe Down
Salisbury, Wilts SP4 OJF
Phone: +44 (0) 1980 66 3987
Fax: +44 (0) 1980 66 3035
E-mail: agerman@qinetiq.com

Decision Point: Will Using a COTS Component Help or Hinder Your DO-178B Certification Effort?©

Timothy J. Budden
AVISTA, Incorporated

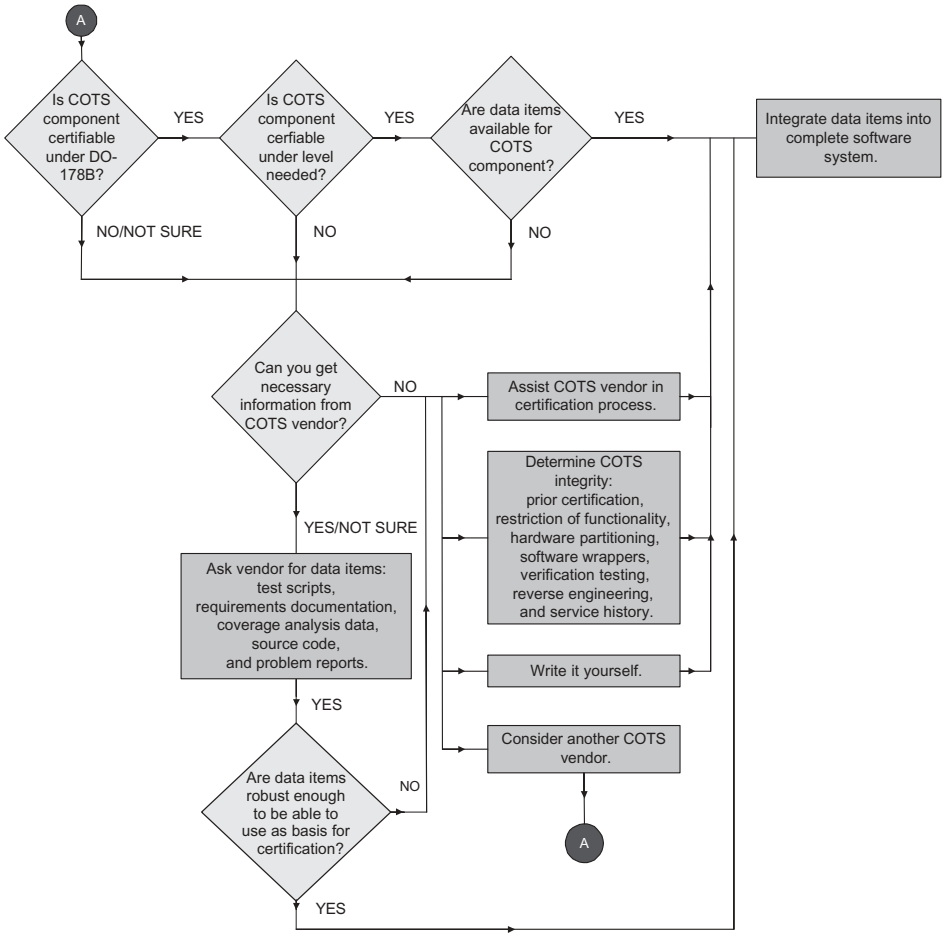
Avionics software developers today are continually challenged to cut costs and reduce time to market, without compromising the safety of their application. Many project leaders look to commercial off-the-shelf (COTS) software components as a possible means to reduce software development costs and development time. The requirements to "prove" software quality under Defense Order (DO)-178B may be difficult, but the opportunity demands consideration of COTS module integration where possible. Understand what is certifiable, how to get the right information from your vendor, and the importance of DO-178B traceability.

Nearly all embedded applications intended for avionics deployment must pass the rigorous certification guidelines developed by the Radio Technical Commission for Aeronautics, Inc. (RTCA) for use by the U.S. Federal Aviation Administration (FAA) in certifying software used in commercial aircraft. These guidelines, known as RTCA Defense Order (DO)-178B, prescribe the development and verification process for software intended for airborne systems and other equipment that must meet certain FAA criteria for airworthiness.

Generally, certification is required for airborne systems and related equipment whose failure will put human life at risk. The two regulatory bodies that primarily administer these safety-critical issues include the U.S. FAA and the Joint Aviation Authority in Europe. These agencies recognize DO-178B as an acceptable means of compliance for software approval in airborne systems.

Certification of avionics equipment is typically achieved through FAA authorization of a type certificate, parts manufacturer approval, or a technical standard order. Systems are categorized by DO-178B as Level A through Level E, based on their criticality in supporting safe aircraft flight. Level A is the most critical, as a failure of such a system could result in a catastrophic failure condition for the aircraft. Level E is the least critical, as a failure of such a system has no effect on the operational capability of the aircraft or pilot workload.

Figure 1: *Decision Tree to Determine if COTS Component Is Certifiable*



Is the COTS Component DO-178B Certifiable?

The conundrum regarding whether or not commercial off-the-shelf (COTS) modules will help or hinder the developer is better understood in the context of system certification. DO-178B certification requires applying stringent processes for all software, including COTS components that ultimately make up the end software system. This includes generating software life-cycle data items that support the entire software system, including any COTS software that may be incorporated into the application.

It is important to note that although a software component may have been previously included in other systems that were certified under DO-178B, it does not necessarily follow that the software component will be certifiable in the new system. This complicates the COTS decision. How does the software developer determine whether to incorporate a COTS component that claims to be certifiable or is believed to be certifiable?

How a software component is used is more important than its prior certification history. It is not possible for COTS vendors to receive standalone certification for particular software components they supply and to have that component *automatically* be certified when incorporated into an application. Moreover, COTS vendors who claim to be DO-178B certifiable may not be certifiable to the level (A through E) that is required. Regardless, while it is not possible to certify a COTS module in isolation, it is possible to package that COTS component in a form that facilitates certification by a systems developer.

Software Life-Cycle Data Items	Description	DO-178B Level				
		A	B	C	D	E
Planning						
Plan for Software Aspects of Certification	Used by certification authority to determine whether applicant is proposing a software life cycle commensurate with the rigor required for the level of software being developed.	XX[1]	XX	XX	X	
Software Project Development Plan	Includes objectives, standards, and software life cycles to be used in the software development process.	XX	XX	XX	X	
Software Verification Plan	Describes the verification procedures to satisfy the software verification process objectives.	XX	XX	XX	X	
Software Configuration Management Plan	Establishes methods to be used to achieve the objectives of the software configuration process throughout the software life cycle.	XX	XX	XX	X	
Software Quality Assurance Plan	Describes the methods used to achieve the objectives of the software quality assurance process.	XX	XX	XX	X	
Standards						
Software Requirements Standards	Defines the methods, rules, and tools to be used to develop the high-level requirements.	X	X	X		
Software Design Standards	Defines the methods, rules, and tools to be used to develop the software architecture and low-level requirements.	X	X	X		
Software Code Standards	Defines the methods, rules, and tools to be used to code the software.	X	X	X		
Project Development						
Software Requirements Data	Describes the high-level requirements, including derived requirements.	X	X	X	X	
Design Description	Describes the software architecture and low-level requirements that will satisfy the software high-level requirements.	X	X	X	X	
Source Code	Consists of code written in source language(s) and the compiler instructions for generating the object code from the Source Code, and link and loading data.	X	X	X	X	
Executable Object Code	Consists of a form of Source Code that is directly usable by the central processing unit of the target computer and is the software that is loaded into the hardware or system.	X	X	X	X	
Software Verification						
Software Verification Cases and Procedures	Details how the software verification process activities are implemented.	XX[2]	X	X	X	
Software Verification Results	Results that are produced by the software verification process activities.	XX	X	X	X	
Additional Data Items Spanning Entire Life Cycle						
Software Life-Cycle Environment Configuration Index	Identifies the configuration of the software life-cycle environment. The index aids reproduction of the hardware and software life-cycle environment.	X	X	X	X	
Software Configuration Index	Identifies the configuration of the software product.	X	X	X	X	
Problem Reports	Reports identify and record resolution to software product anomalous behavior, process non-compliance with software plans and standards, and deficiencies in software life-cycle data.	X	X	X	X	
Software Configuration Management						
Software Configuration Management Records	Results of the software configuration management process activities.	X	X	X	X	
Software Quality Assurance						
Software Quality Assurance Records	Results of the software quality assurance process activities.	XX[3]	XX	X	X	
Software Aspects for Certification						
Software Accomplishment Summary	Used as the primary data item for showing compliance with the Plan for Software Aspects for Certification.	X	X	X	X	

[1] The number of X's indicates the level of rigor and detail expected for that specific data item for that level of certification.
[2] Increasing in rigor and independence
[3] Independence for all four levels

Table 1: *Data Items Necessary for DO-178B Certification*

Figure 1 displays a decision tree that suggests the types of questions to ask a COTS vendor when deciding whether or not to use a COTS component as part of your avionics application. The remainder of this article focuses on the details of each stage of inquiry as reflected in Figure 1.

The ideal situation is to purchase a COTS component that provides all the necessary life-cycle data items to support DO-178B certification. However, it is by no means a common option among COTS vendors. You may need to do a bit more research to determine if you and the COTS vendor can work together to satisfy DO-178B requirements to get the necessary life-cycle data items for certification for your entire avionics application.

Get the Right Information From the COTS Vendor

Knowing what data items to get from your potential COTS vendor will depend upon your overall system approach to certification. Moreover, the level of detail necessary for certain data items varies based on the level of DO-178B software certification to which your avionics software application must comply. The process of obtaining the necessary information to support certification from the COTS vendor requires a formal business relationship between your companies. At a minimum, you should expect that the COTS vendor would work closely with your system developers to ensure acceptance of the COTS component within your avionics system.

Table 1 outlines the data items from your software life-cycle process that are expected as part of the overall certification process. These data items are as follows:

- Planning documents that include Plan for Software Aspects for Certification, Software Project Development Plan, Software Verification Plan, Software Configuration Management Plan, and Software Quality Assurance Plan.
- Standards documents that include Software Requirements Standards, Software Design Standards, and Software Code Standards.
- Project development data that include software requirements data, design description, source code, and executable object code.
- Software verification that includes Software Verification Cases and

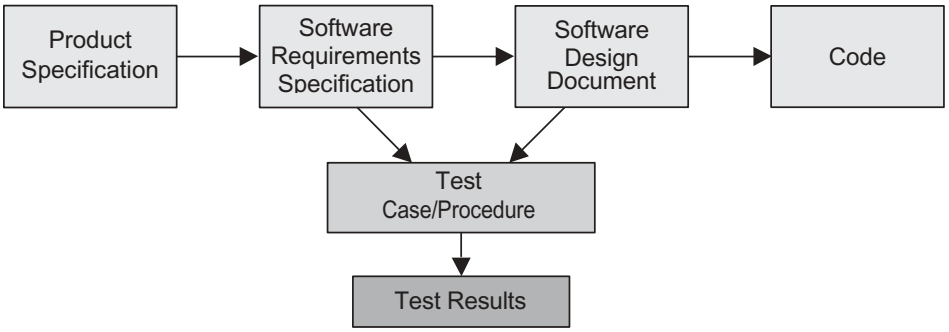


Figure 2: *Traceability Flow*

Procedures, and Software Verification Results.
- Additional life-cycle data items that span the entire life cycle, including Software Life-Cycle Environment Configuration Index, Software Configuration Index, and Problem Reports.
- Configuration management records.
- Software quality assurance records.
- Software accomplishment summary.

Detailed descriptions of these data items can be found in various official publications governing DO-178B development such as RTCA DO-178B, "Software Considerations in Airborne Systems and Equipment Certification," Dec. 1, 1992. As Table 1 suggests, different certification levels may require different degrees of detail or completeness in each data item. Understanding the certification level that you plan for your application is a necessary precursor to the dialogue with your COTS vendor regarding needed data items.

Mixing and matching COTS vendor data items with your own data items can be done in a variety of ways. For example, you may wish to incorporate details of the COTS vendor's verification process into your overall Software Verification Plan. You may then decide either to include the COTS vendor's test case/procedure data into your own Software Verification Cases and Procedures document, or have it stand alone as a cases and procedures document solely for the COTS component. The key is that the processes are documented and followed, and that the data items are captured, regardless of how they are packaged.

The Importance of Traceability and Independence

Traceability is a well-defined manner to objectively assess the rigor applied to the development and verification of the entire system. That is, satisfying the traceability requirements of DO-178B certification involves documenting how downstream life-cycle elements link to upstream life-cycle elements. For example, design elements and test case/procedure elements must be linked to originating requirement elements.

To verify your software, DO-178B requires both a Requirements Coverage Analysis (RCA) and a Structural Coverage Analysis (SCA). The RCA requires traceability and ensures that a test case/procedure exists for every software requirement. The SCA uncovers code elements that were not covered through execution of requirements-based tests. The rigor of the SCA varies with the criticality level of the software. Having top-to-bottom traceability also facilitates regression analysis activities when change inevitably occurs. The traceability flow is shown in Figure 2.

"More times than not, your leadership will be demanded in helping to bridge understanding with your proposed COTS vendor of how and what is required to support the certification effort."

Independence is also an important DO-178B topic. Independence is the separation of responsibilities that ensures the accomplishment of objective evaluation. For software verification process activities, independence is achieved when a person(s) other than the developer of the item being verified performs the verification activity; a tool(s) may be used to achieve equivalence to the human verification activity. For the software quality assurance process, independence also includes the authority to ensure corrective action.

A COTS vendor who provides data items such as requirements-based test cases/procedures may also need to prove that these tests were produced by someone other than the code developer for a given set of requirements. Independence requirements vary with the software level, but they are primarily related to verification and software quality assurance activities.

Both parties understanding these elementary concepts of traceability and independence often facilitate effective communication with your COTS vendor regarding certification. The most prevalent obstacles to incorporation of COTS modules are a lack of life-cycle data items (e.g., requirements data, design data, and test cases/procedures), traceability data, and independence. The rigor of DO-178B development is seldom adopted in commercial applications and often not understood or appreciated by embedded software developers.

What if You Can't Get the Information You Need?

In the end, the COTS vendor may be either unable or unwilling to provide the necessary data items associated with the COTS component. In this situation, you can pursue one of the following alternatives:

1. **Assist the COTS vendor in the certification process.** The COTS vendor may be interested in collaborating with you to certify your application of their product to DO-178B guidelines. This option can be a win-win situation for both parties. An example of a potentially successful business arrangement could include your company receiving the source code of the COTS module for little or no license fee in exchange for your company's assistance in working together to certify the module under DO-178B. The COTS vendor would presumably benefit from the experience gained by having a library of required life-cycle data items, as well as the promotional value of having their module(s) branded as *certifiable*.
2. **Without assistance from the vendor, determine the COTS component integrity.** One or more of the following methods may be helpful as a process to obtain the necessary information to support compliance of the COTS module with DO-178B certification:
   - Reference prior certification records in which the COTS component was approved as part of an earlier certified or qualified system application.
   - Restrict the functionality by only using a subset of functionality and certify only those functions. This

may limit the amount of information required for the certification.
- Partition the system. This method prevents failures from noncritical functions affecting critical functions (such as implementing functions on different processors or different memory partitions).
- Use software protection wrappers to limit the functionality exposed to the certification-targeted application. *Wrapper software* accompanies other software to improve compatibility or security such as the deactivation of unneeded functionality, and/or check the validity of parameters.
- Analyze the COTS vendor's data items for certifiability and use/enhance as necessary.
- Reverse engineer the COTS data items. This requires reconstructing the data, which can be a difficult task, perhaps requiring as much effort as recreating the development in-house. However, the process will produce software life-cycle data that can be reviewed and analyzed to satisfy the DO-178B objectives.
- Reference the service history of the COTS component. This can provide previous in-service experience of the component. However, the data integrity of the service history records must be validated, which thus requires information about the problem tracking process and the software configuration management process used originally by the COTS vendor.

3. **Write the functionality in-house.** This may be your best option, especially if there are no COTS vendors that have the necessary DO-178B certifiable component(s), or who are unwilling and/or unable to provide you the necessary data items. Despite the usefulness and appeal of COTS solutions, the cost and time to develop the software or systems component in-house may be considerably less than attempting to bludgeon your way into certification of software never developed with intentions of satisfying the stringent quality concerns of DO-178B.
4. **Consider another COTS vendor.** If there are other COTS vendors that have the necessary DO-178B certifiable component(s), or are more willing and/or able to give you the necessary data items, then consider these vendors as viable alternatives. The value of working with vendors who have already committed (or are willing to commit) their energies to ensuring a DO-178B quality product should be readily apparent.

Conclusion

Certification under DO-178B is one of the most grueling development and verification processes developed, but for good reasons. There can be no compromises to software and systems quality when lives are at stake both in the air and on the ground. The requirements to *prove* software quality under DO-178B may require you to think again about your plan to incorporate a COTS module in your application. However, the opportunity to speed your time to market and improve your development productivity demands that you at least consider COTS module integration where possible.

As described in this article, the demands of DO-178B certification can be achieved with COTS modules if your vendor is a willing partner who understands the value, importance, and professionalism that is expected of DO-178B development. More times than not, your leadership will be demanded in helping to bridge understanding with your proposed COTS vendor of how and what is required to support the certification effort. The business payoff is significant for all parties, and the quality solution that results is of pride to all.

More information on RTCA DO-178B is available online at <www. rtca.org>.◆

About the Author

Timothy J. Budden is senior programs manager at AVISTA, Incorporated. His experience spans a wide range of fixed- and rotary-wing aircraft systems. He has developed a working knowledge of all software life-cycle phases and RTCA DO-178B guidelines through years of successful certification experience. Prior to joining AVISTA 12 years ago, Budden was employed by McDonnell Douglas. He has a Bachelor of Science in electrical and computer engineering from the University of Notre Dame.

AVISTA, Incorporated
P.O. Box 636
1575 U.S. Highway 151 East
Platteville, WI 53818
Phone: (608) 348-8815
Fax: (608) 348-8819
E-mail: tim.budden@avistainc.com

Defining a Process for Simulation Software Vulnerability Assessments

Dr. John A. Hamilton Jr.
Auburn University

Col. Kevin J. Greaney
Missile Defense Agency

Gordon Evans
Booz Allen Hamilton

The need for simulation software vulnerability assessment is being driven by three major trends. They are increased use of modeling and simulation for training and operational planning, increased emphasis on coalition warfare and interoperability, and increased awareness of the potential security risks inherent in sharing operationally useful software. This article will describe in an unclassified manner the process developed by the U.S. Missile Defense Agency and Auburn University to evaluate potential vulnerabilities in shared simulation software.

This may seem an obvious point to CrossTalk readers, but many members of the Department of Defense modeling and simulation community are not software engineers. Therefore, the model's software implementation must be analyzed as well as the model itself.

The security model for many missile defense simulations is similar to that used in the heyday of the U.S. Army's nuclear weapons training. When virtually all U.S. Army medium and heavy artillery batteries were nuclear capable, it was necessary to conduct training for units, particularly Reserve Component units that did not have secure facilities to handle classified models. The result was an unclassified training system that only provided classified results when given actual weapons performance data. Soldiers were thus able to train on how to do the targeting calculations without handling classified information as shown in Figure 1.

Applying this model to missile defense simulations is more difficult because the calculations are much more complex, and there are many more parameters to deal with. Furthermore, these simulations are implemented in software, and that increases the complexity. Therefore, the Missile Defense Agency's Models and Simulation Directorate (MDA/SES) developed a vulnerability assessment process in partnership with Auburn University's Information Assurance Laboratory. We next describe the process and the environment that framed our process.

Assessing the Threat

U.S. missile defense programs, training, tactics, and procedures are a matter of intense interest to foreign intelligence agencies. Intelligence agencies do not face the same economic constraints, as do practitioners of economic espionage. For this reason, military-relevant software may be attacked in ways that would not be feasible for an industrial reverse-engineering application.

An unclassified analysis of one missile defense simulation Web site showed that nearly one third of the site's hits could be traced back to the People's Republic of China [1]. The largest number of recorded hits came from Beijing, and this was more than twice the number of hits from any other country, including the United States. This obviously does not include undetected intrusions.

> *"An unclassified analysis of one missile defense simulation Web site showed that nearly one third of the site's hits could be traced back to the People's Republic of China."*

Defining a Process

Information compiled into binary code is not secure. Just because it is hard to extract information from a binary file does not mean it is impossible to do so [2]. As shown in Figure 2, our process analyzes inputs, outputs to the simulation software, and the executable binaries.

Phase 1: Inputs

When analyzing the system inputs, we look at how well we can simulate a system based on open-source data. Next, we search for buffer overflows. Given the popularity of the C programming language, it is usually not hard to find a buffer overflow – either in the application or in the operating system it is running on. Whether a buffer overflow can be used to compromise sensitive information in the application remains to be seen. Theoretically, one could jump to a code segment written to start dumping out intermediate calculations. Operating systems are written in C and are vulnerable to buffer overflows, too.

Buffer overflows can be used for more than just jumping a program to an unauthorized code segment. We found that entering control characters into an entry screen would bring up a debugger providing important clues on how the program was originally compiled.

What if the simulation developers used explicit bounds checking for every input? One thing to look for is any sensitive information that can be gleaned from the bounds. An interceptor that has actual minimum and maximum ranges as bounds would be an example of a possible vulnerability.

Phase 2: Attacking the System

Simulation software runs on top of operating systems. On distributed simulation implementations, operating system vulnerabilities may be exploited to remotely compromise the simulation software. It is often instructive to study the installed files of a software distribu-

Figure 1: *U.S. Army Nuclear Weapons Training Model Circa 1980 (Unclassified)*

tion to learn more about the program structure and contents.

A good definition for reverse engineering can be found at [3]. Van Deursen defines reverse engineering as,

> The process of analyzing a subject system with two goals in mind: (1) to identify the system's components and their interrelationships, and (2) to create representations of the system in another form or at a higher level of abstraction. [3]

In this process, we look at both disassembly of code as well as decompilation.

Disassembly is reconstructing assembly language code from a binary. Eric Imsand and Adam Sachitano have disassembled missile defense simulations using *dis* on Solaris and a shareware program called Hackman on Windows platforms. Both dis and Hackman are disassembly programs. They report the following:

> The Hackman application ran easily. After launching the program, it was simply a matter of selecting the tool, either a hex editor or disassembler, and choosing the file to open. *Hackman* opened the file, and disassembled it without further user interaction. The entire disassembly process took approximately six hours running on a 400 MHz Intel Celeron processor with 128 MB of RAM. [4]

Imsand and Sachitano produced about one gigabyte of assembly code and were later able to reassemble the binary and successfully run it.

With assembly code in hand, it is possible to insert additional instructions to create a modified binary that dumps every variable value to an output device. It is also possible to search for string literals.

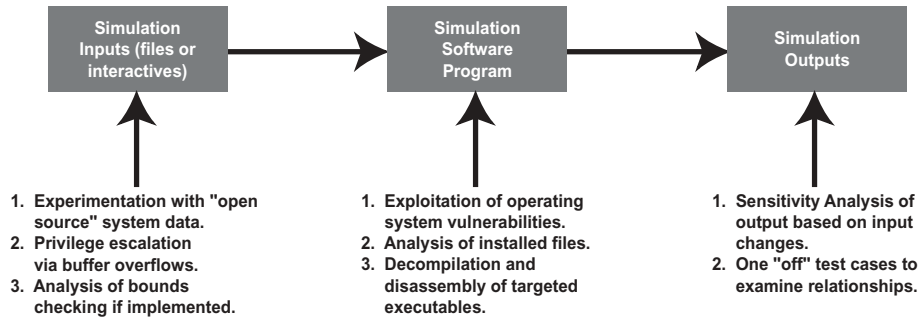


Figure 2: *Process for Simulation Software Vulnerability Analysis*

Decompilation is the generation of high-level source code from low-level input [5]. We have experienced little success in decompiling, primarily due to our reliance on freeware and shareware tools. Commercial decompilers are available and the state of the art in this area continues to improve.

Weide, Heym, and Hollingsworth discuss reverse engineering of large legacy software systems. They conclude that the reverse engineering of such systems is *intractable* in the sense that if one is given real (high-level) legacy code, the time required to show the validity of an explanation for why it exhibits a certain behavior is at least exponential compared to the size of the source code [6]. However, their same paper asserts a caveat that is repeated here: "This does not mean that the task is impossible. It means that it is prohibitively costly for large systems" [6]. We would add that what is prohibitively expensive in the commercial sector is not necessarily prohibitively expensive for a high-priority intelligence effort.

Phase 3: Outputs

We attempt to determine the internals of the programs by analyzing the outputs and their sensitivity to changes based on carefully chosen inputs. In general, we believe that missile defense simulations of any importance are too complicated to make this a useful strategy for reverse engineering the simulation. However, it is possible to gain insight into specific aspects of the simulation by constantly running it and making minor changes to the input and tracking the changes. These one *off* test cases are constructed by varying only one input parameter. If the simulation is well documented, this strategy can be used in conjunction with an analysis of the documentation to determine internal relationships between parameters.

Refining and Applying the Process for Different Levels of Assurance

Given the costs associated with vulnerability analysis, we defined three sets of tasks providing three levels of assurance: High, Medium, and Low. These categories reflect the level of effort required for the analysis. The requirements for each are enumerated in Table 1.

It is important to recognize that anything sensitive in the source code is vulnerable. It is hard, time consuming, and expensive to get at it – but it is naïve to think that a hostile intelligence agency would not make such an attempt. Next, we address each item in Table 1.

- **Source Code.** A line-by-line verifica-

Table 1: *Assurance Levels for Simulation Software Vulnerability Analysis*

High Assurance Level	Medium Assurance Level	Low Assurance Level
Line-by-line verification of source code.	Line-by-line verification of selected source code.	Line-by-line verification of selected source code.
Professional decompilation of executables.	String search on disassembled code.	String search on disassembled code.
Complete review of published documentation.	Targeted review of published documentation.	Targeted review of published documentation.
Open source review of weapons and systems data.	Open source review of weapons and systems data.	Analysis of degree of parameterization.
Analysis of simulation runs to evaluate training, tactics, and procedures.	Analysis of simulation runs to evaluate training, tactics, and procedures.	
Analysis of degree of parameterization.	Analysis of degree of parameterization.	

tion of a simulation with a million+ lines of code is nontrivial. Worse, there is no guarantee that such a massive effort will uncover all potential security issues. However, it is the best way to detect problems. Software engineering research has long held that the best way to find any problem in software is through desk-checking the source code. In most cases, a line-by-line verification will not be warranted. If (and this is a big if) the simulation software is well structured, then it is reasonable to exclude large portions of the code and simply focus on the modules that deal with sensitive issues. It can be argued that a more focused review of *high-risk* code could potentially be more fruitful than plowing through a massive program in its entirety. Any source code provided, as part of the distribution must be reviewed. Source code analysis can give you a worst-case vulnerability assessment.

- **Decompilation/Disassembly.** Decompilation and disassembly can be used to provide an expected case analysis. We pursue this to see what a potential adversary can learn from the binaries. For high assurance requirements, we recommend using professionals to decompile the binaries. Open market decompilers (available to a university anyway) are not yet to a point where experienced software engineers can gain useful results through reasonable efforts. We have no insight into what tools are available in the world of restricted access programs, but we believe that much better tools are theoretically possible and practical. Dissemblers are readily available and useful. It is reasonable to write scripts to do string searches on massive assembly code files and prudent to do that. In all cases, the binaries should be checked to make sure that all debugging information is stripped before the binaries are released.
- **Documentation Review.** Documentation of simulations must be included in the distribution [7]. Some simulations include more than 1,000 pages of documentation. Documentation is critical to the successful utilization of a simulation. As Sargent notes:

> Documentation on model verification and validation is usually critical in convincing users of the 'correctness' of a model and its results, and should be included in the simulation model documentation. [8]

The caveat to Sargent's assertion is that the documentation must be reviewed to make sure that no sensitive information is inadvertently released. The physics of missile trajectories are not sensitive; probability of kill for a given system is very sensitive.

- **Open Source Review.** There is a great deal of published information on missile and missile defense systems, particularly older ballistic missile systems such as scuds. One way to exercise a simulation is to create models from open source material and then experiment with them.

"It is important to recognize that anything sensitive in the source code is vulnerable. It is hard, time consuming, and expensive to get at it – but it is naïve to think that a hostile intelligence agency would not make such an attempt."

- **Analysis of Simulation Runs.** Using open source inputs provides the means to develop simulation runs and analyze the outputs. The objective is to reduce the number of unknowns in the system. The more known information that can be input, the easier the analysis.
- **Analysis of Degree of Parameterization**. Essentially, we want to verify that the model is unclassified and that classified results are only produced when classified parameters are used. If there are default values, then those values need to be checked to see if any are sensitive in nature. In general, the greater the degree of parameterization, the closer the simulation approximates the model in Figure 1.

Conclusion

In most cases, we believe that a medium assurance assessment is sufficient. Before we share simulations (missile defense or others) with our coalition partners, it is essential to know what we are sharing.

This research has demonstrated a viable, scaleable means of assessing the vulnerability of complex simulation software. We believe this methodology is appropriate for use with other simulation programs. It is always difficult to prove a negative. We do not claim that our process can prove the absence of vulnerabilities or find every vulnerability in every software implementation. However, this process can provide an important means of risk mitigation. We believe the process defined here can successfully identify vulnerabilities in simulation software.◆

References

1. Mann, Steve. "Default Report Web Site Visitors." WebTrends. NetIQ Corporation, Internal Report. 17 May 2002: 41.
2. Viega, John, and Gary McGraw. Building Secure Software. Boston, MA: Addison-Wesley, 2002: 109.
3. Van Deursen, Arie. "Reverse Engineering." Jan. 2003 <www.program-transformation.org/twiki/bin/view/Transform/ReverseEngineering>.
4. Imsand, Eric, and Adam Sachitano. "Analyzing Security Vulnerabilities in National Missile Defense Simulation Software." Unpublished, Nov. 2002.
5. Breuer, Peter T., and Jonathan P. Bowen. "Decompilation: The Enumeration of Types and Grammars." ACM Transactions on Programming Languages and Systems 16. 5 (1994).
6. Weide, Bruce W., Wayne D. Heym, and Joseph E. Hollingsworth. Reverse Engineering of Legacy Code Exposed. Proc. of the 17th International Conference on Software Engineering, Seattle, WA. New York: ACM Press, 1995: 327-331.
7. Chatham, Wade. "A Vulnerability Analysis with an Emphasis on Using Documentation." Unpublished, Nov. 2002.
8. Sargent, R. "Verifying and Validating Simulation Models." Proc. of the 28th Winter Simulation Conference, Coronado, CA. New York: ACM Press, 1995: 55-64.

About the Authors

John A. "Drew" Hamilton Jr., Ph.D., is an associate professor of computer science and software engineering at Auburn University and director of Auburn University's Information Assurance Laboratory. Prior to his retirement from the U.S. Army, he served as the first director of the Joint Forces Program Office and on the staff and faculty of the U.S. Military Academy, as well as chief of the Ada Joint Program Office. He has a Bachelor of Arts in journalism from Texas Tech University, masters degrees in systems management from the University of Southern California and in computer science from Vanderbilt University, and a doctorate in computer science from Texas A&M University.

Auburn University
107 Dunstan Hall
Auburn, AL 36849
Phone: (334) 844-6360
Fax: (334) 844-6329
E-mail: hamilton@eng.auburn.edu

Col. Kevin J. Greaney, U.S. Army, is the director, Models and Simulations, Missile Defense Agency. Prior to his assignment, Greaney served as the commander of the Communication Electronics Command Software Engineering Center-Meade from September 1997 through September 2000. Greaney was selected as a Distinguished Military Graduate prior to his commission as a second lieutenant. He has a Bachelor of Arts from Northeastern University, a Master of Science from Shippensburg University, a Master of Arts from Webster University, and is currently pursuing a doctorate in software engineering from the Naval Post Graduate School.

Missile Defense Agency
Pentagon
Washington, D.C.
Phone: (703) 697-4360
Fax: (703) 695-6133
E-mail: kevin.greaney@bmdo.osd.mil

Gordon Evans is a consultant for Booz Allen Hamilton working on-site at the Missile Defense Agency (MDA). His areas of concentration have been in systems engineering, command and control, modeling and simulations, international programs, and technology transfers. He has been the lead MDA designer and investigator for its Modeling & Simulation Vulnerability Assessment program. Evans retired from the U.S. Army in 1992 as a lieutenant colonel. During his military service, he served in multiple Field Artillery and Military Intelligence assignments. Overseas assignments have been in Germany, Korea, and Vietnam.

Booz Allen Hamilton – MDA/SES
Pentagon
Washington, DC
Phone: (703) 697-4582
Fax: (703) 695-6133
E-mail: gordon.evans-contractor @ bmdo.osd.mil

Developing a Stable Architecture to Interface Aircraft to Commercial PCs

Dan W. Christenson and Lynn Silver
Ogden Air Logistics Center

This article introduces a new concept of utilizing commercial personal computer products to interface with military and commercial aircraft regardless of their product life-cycle mismatch. Existing products are described, architecture for each implementation is derived, and their strengths and weaknesses are explored. An attempt to define the root causes for the problem of implementing interface architectures in this environment is presented. In addition, a new developmental architecture is introduced that is designed to maintain the strengths of the traditional architectures and eliminate some of the weaknesses and inefficiencies. A series of hardware/software co-development projects are described to demonstrate this new architecture. The relative performance of the architecture has been evaluated and refined by multiple implementations. These are described and future implementations examined.

In response to a growing need within the U.S. Air Force (USAF) to lower operating costs, a significant amount of research has been initiated to find logical, supportable opportunities to utilize commercial products and to replace products that have historically been custom-designed. Multiple development efforts have helped to refine concepts that have proven their utility at lowering acquisition costs while other efforts have proven to have lower sustainment costs. Recent efforts have shown that by managing the architecture of the developed equipment, it is possible to lower the overall life-cycle cost, while providing long-term, technically viable, and user-friendly equipment.

The current USAF loader/verifier of choice eliminates the obsolescence of computer hardware and software used on traditional loader/verifiers by being nonproprietary in its design. This allows the USAF to upgrade to newer personal computer (PC) platforms without a lot of expense.

Background

For the past decade, the Department of Defense (DoD) has faced budget cuts that have translated into lost personnel, slashed weapon systems development budgets, curtailed maintenance budgets, and extended weapon systems lifetimes. In this era of doing more with less, one of the easiest implemented directives was to use, to the greatest extent possible, commercial off-the-shelf (COTS) products.

However, since the DoD does not command a significant part of the electronics market share, it has little ability to affect the direction of the overall marketplace and thus COTS products. This has been further underscored by the cancellation of most military standards, because, among other reasons, the standards themselves could not be updated quickly enough to allow military product developers the opportunity to use current technology before it became obsolete.

Some of the first uses of COTS included applying commercial PCs to aircraft back-shops and flight lines. Several generations of COTS PC products have been in use by the USAF.

"It has only been since the late 1970s that any significant communication standards have existed for aircraft. If a designer is to interface with many different aircraft interface types, flexible interface techniques must be developed."

Each of the PC products introduced into the DoD has faced a similar set of initial requirements that could not be updated quickly enough to utilize current technology. Each has taken a similar path to implement the requirements, and each has had similar problems at the end of the short product lifetime. These PCs are used for a variety of uses, including user interfaces for embedded computers, memory loader/verifiers for embedded computers, test equipment, test equipment controllers, technical order delivery systems, and digital communications equipment.

A recent picture that ran in many aerospace and local publications highlights the problem that the USAF is facing. The picture was a family photo of a B-52 commander, his father, and grandfather – all three had served on the same aircraft. As weapon systems lifetimes are extended, the opportunities to update the systems becomes more challenging. While every electronic system in the B-52 may have been updated, remnants of the original infrastructure remain. Much like today's railroads that have track separation distances based on the wheelbase of Roman chariots, the avionics systems of the USAF have interface specifications that have outlived their authors.

When the F-4 left the USAF inventory in the 1990s, it retained interface descriptions that reflected its Resistor Transistor Logic (RTL) roots from the 1950s. The F-16 discrete signal interface descriptions that were written in the late 1970s have specifications most easily implemented by switches and relays.

Only in the last 10 years has the AIM-9 missile provided an interface that did not reflect its original servo-type interface developed in the 1950s. For many years, both the aircraft and the weapon implemented the original archaic analog interface, using a mixture of analog and digital circuitry only because it was the easiest way to make sure the weapon and weapon systems would be interoperable. Ultimately, just as in the case of the AIM-9 missile interface, the designer of today's computer-to-aircraft interface is forced to be compliant with the existing weapon systems interfaces.

Traditionally with each new embedded computer, a new method to communicate with and to control it was introduced. This is extremely unfortunate for today's interface design engineer. When you examine the historical rationale, there were at least four significant reasons to

create a unique protocol:

1. Aircraft-unique throughput or communications needs forced unique solutions.
2. Standards that would meet the needs were not well known.
3. It was chosen because of management concerns.
4. It was chosen because of convenience.

In many cases this uniqueness extends to voltage levels, drive current, timing, and data protocol. The number of electrical interface *standards* now exceeds 100; the number of data protocol *standards* is several times that number. This means that in most cases, each interface that the designer approaches is different from the last. It has only been since the late 1970s that any significant communication standards have existed for aircraft. If a designer is to interface with many different aircraft interface types, flexible interface techniques must be developed.

As this set of issues was evaluated, it became apparent that addressing this problem was the central issue for the architecture. Because aircraft and commercial PC life cycles are so far out of synchronization, and because that gap is growing, providing a long-term, supportable method to *buffer* the two environments is the essential artifact of this architecture. A graphical representation of the concept is shown in Figure 1.

With each new interface type, the adapter between the aircraft and the PC had to be addressed as part of the design; the logical place to build a standard was at that point, named the Aircraft Adapter Group (AAG). The name was chosen because the predecessors of this architecture used AAGs to interface their standards loader/verifier to the weapon systems. Therefore, this historical name was chosen because it is somewhat analogous to the function.

Examples of PC-Based Support Equipment

Three USAF examples of aircraft support equipment based on PC technology include the Automatic Test Systems/Product Group Manager's Digital Computer System (ATS/PGM's DCS), the F-16 Enhanced Diagnostics Aid (EDNA), and the F-15 Programmable Loader/Verifier – NT version (PLV-NT). Each of these PC-based systems was introduced into the USAF inventory in the last 15 years.

Each was acquired with similar standards that have traditionally been levied on all support equipment. Virtually none of the PC products utilized for aircraft support equipment has been used without modification. The necessity to modify the interface was driven by additional requirements associated with the unique environment of use. These environmental requirements fall into three basic categories:

1. Security environment to avoid compromise of classified data when the equipment is operated in the close proximity of those who had no need-to-know.
2. A physical environment requiring that all input/output (I/O) is installed inside the PC.
3. The PC was required to be environmentally compatible with a USAF flight line.

Forcing the PC to comply with these requirements has at least three negative effects:

1. They drive up the acquisition cost because the COTS PC selected is virtually a custom product.
2. They drive up the re-host cost because the new PC has to be re-procured from the original source because of proprietary data issues.
3. Compliance drives down the overall performance because the custom computer market lags behind standard PCs by as much as two years.

As these pieces of equipment were introduced into the inventory, they were initially well received, but quickly were considered archaic when compared to traditional PC equipment. Their lag-behind technology, the inability to use current hardware and software products, and the cost to acquire and maintain the equipment made them unpopular. Within a

Definitions

- **Ethernet —** A high-speed serial data bus generally used to implement Local Area Networks. Ethernet was not designed to power peripherals; it is therefore required that a separate power cable/supply be used.
- **Firewire —** A high-speed serial data bus generally used for video/audio processing peripherals. Firewire was designed to provide a limited amount of power to peripherals. Firewire has the liability that it is not as widely accepted in the marketplace as a Universal Serial Bus (USB) is and that with a few exceptions (like Sony), it is not generally implemented in laptops.
- **IEEE 488 —** An eight-bit parallel bus common on test equipment. The IEEE 488 standard was proposed by Hewlett-Packard in the late 1970s and has undergone a couple of revisions. It allows up to 15 intelligent devices to share a single bus with a maximum data rate of one megabit per second.
- **MIL-STD-1553 —** A military standard that is slower than most modern serial busses and does not provide power to any peripherals. Because of the complexity of the protocol, expensive integrated circuits are required to implement the interface. Its redundancy and noise immunity have made it a popular interface for aircraft use.
- **Parallel Bus —** A bus consisting of multiple signal lines that simultaneously transfer data in a parallel method.
- **RS 232 —** A simple, universal low-power serial bus that can be found in many different applications from modems to PCs, where the length and quality of the cable depends on the data speed.
- **RS 422 —** A differential serial bus designed for greater distances and higher baud rates than the RS 232. Data rates of up to 100,000 bits/second and distances up to 4,000 feet can be accommodated with the RS 422.
- **SCSI —** A Small Computer System Interface designed originally to communicate between a computer and disk drives has been used when high-speed communication is necessary. Because of the number of wires required, SCSI cables are generally bulky. SCSI was never designed to power peripherals; it is therefore required that a separate power cable/supply be used.
- **Serial Bus —** A bus consisting of a limited number of signal lines (usually one or two) that transfer data in a serial (one bit at a time) fashion.
- **Universal Serial Bus —** A USB is a high-speed serial bus universally available on all PC products. It provides a minimal amount of power, allowing implementation of simple peripherals that do not require additional power supplies.

Figure 1: *Loader/Verifier Derived Architecture*

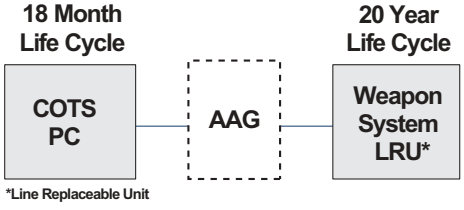


Figure 2: *F-16 EDNA and F-15 PLV-NT*

fraction of the traditional USAF product lifetime, each was considered obsolete and in need of update and/or replacement.

The USAF has not ignored this problem and as early as seven years ago, efforts were made to begin creation of support equipment standards. Efforts have also been made on the weapon systems acquisition system to drive standardization. At this point, the electrical interface *standards* fall into the following standards: IEEE 488, RS 232, RS 422, MIL-STD-1553, and *unique*. Depending on the weapon system type, about 60 percent of the interfaces are MIL-STD-1553, and 20 percent are RS 422, RS2 32, and IEEE 488. The remaining 20 percent are unique; many are simply the electrical interface between the microprocessor and its memory.

Techniques to address these interfaces have been developed. There are interface devices that implement all standard interfaces; with the development of the Field Programmable Gate Array (FPGA), interfacing to unique interface standards was greatly simplified. The unique interface timing as a minimum can be fully implemented. In the case of Transistor-Transistor Logic-based standards, the electrical portion of the interface can be addressed as well. These technologies were incorporated into F-16 EDNA and the F-15 PLV-NT, driving the acquisition costs to one-third of the traditional loader/verifier. These systems are illustrated in Figure 2.

This was considered a great feat until it was recognized that although the new devices were designed to last 10 years, the products became obsolete in less than three years. This made the products no cheaper than their predecessors did. The lesson learned is that not only must the acquisition cost be lower, but also the product acquired must be an *add on* so that neither internal installation nor modification to the PC is required. This allows the PC, weapon systems, and the AAG to be independently modified to accommodate updates.

"At the core of this development effort is the requirement to address a problem that plagues the entire DoD: commercial product lifetimes are becoming shorter while DoD product lifetimes are becoming longer."

Requirements Development

In 1997, a team of users (Next Generation Loader/Verifier users group), program managers, and engineers were convened to begin looking at the growing problem. The support equipment requirements that forced modification to the PC were addressed.

At the core of this development effort is the requirement to address a problem that plagues the entire DoD: commercial product lifetimes are becoming shorter while DoD product lifetimes are becoming longer. As the evaluation proceeded, the basic requirement for developing a stable architecture emerged: Develop a suite of standards-based tools and products that can be functionally implemented with many technologies.

The significant driving function, as mentioned earlier, is the rapid development cycle of the commercial PC. The PC must be allowed to change to utilize current technology. For this reason and for this application, the USAF has adopted a nontraditional approach, allowing functional configuration instead of traditional physical configuration. This means no effort is to be taken with the new equipment to physically configure the PC. Any PC that complies with the functional configuration document is acceptable for use.

The need to modify the PC to accommodate security requirements was replaced with tests and procedures that accomplished the intent of the modification. These tests must be accomplished on a relatively small sample lot and are the basis for the technical data that describe additional security protocols that are necessary to protect the classified information being processed.

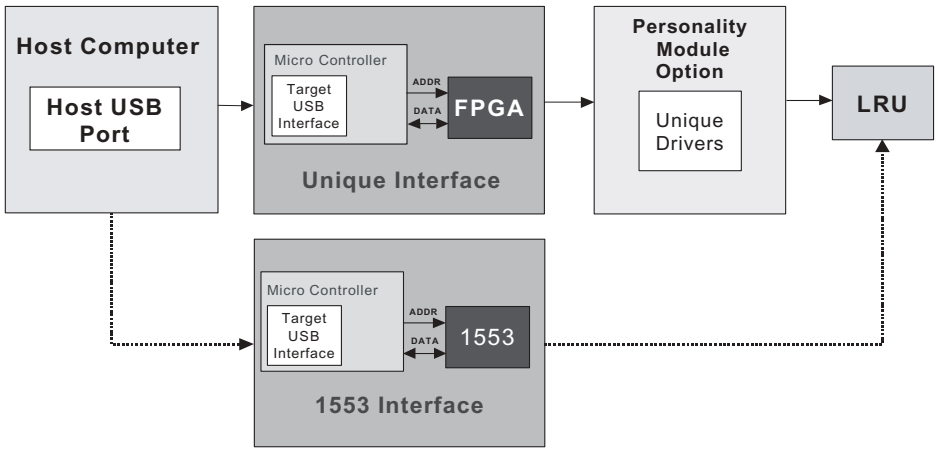
The need to have an integrated PC that addressed all the interface requirements was replaced by a lightweight PC with a small number of external interfaces that address each type of aircraft interface.

To accommodate MIL-STD-1553 remote terminal and monitor functions, without any significant local buffer in the AAG, any external interface must exceed one megabit/second (mbit/sec). Realistically the interface should be at least 10 mbit/sec to allow time for data processing and calculation of responses. This limited the commercial standards that were examined to accommodate the needs of Small Computer System Interface, Ethernet, firewire, MIL-STD-1553, and Universal Serial Bus (USB).

Since the external interface would have to be powered, an interface that could provide the needed power was desirable for two reasons. First, if the interface standard did not provide power, as in the case with MIL-STD-1553, then an external power supply would have to be used. Second, this power supply would also have to be ground-isolated to allow the system to interface with the ground reference of the aircraft. These requirements limited the selections to firewire and USB.

USB was chosen because it is part of the commercial PC standard and is back-

Figure 3: *Detailed Architecture System Diagram*



ward compatible (i.e., USB 1.0 devices will work with USB 2.0).

As implementations were examined, it became obvious that MIL-STD-1553 would be best accommodated in a separate interface, and because of the availability of commercial USB-IEEE 488, initial implementations were procured commercially. Figure 3 represents the block diagrams of the interface elements that currently implement the AAG architecture.

The strengths of prior developments include utilization of FPGAs to implement most of the weapon systems interfaces and PC-based user interfaces to lower the recurring costs. Those strengths are preserved with this architecture. Additionally, by limiting the host PC interface to one type, in this case USB, and by limiting the dependence to only two points, if the interface type becomes obsolete, the amount of re-host required to update to current technology is minimized. While USB is current in the PC environment, the host PC can be updated independently of the AAG hardware.

Possibly, the most exciting aspect of this is the blending of hardware techniques into the software arena. All of the *customization* of the hardware to accomplish the needed AAG functions is done with data that is stored on the PC. The FPGA data, the micro-controller firmware, and the 1553 configuration data are all stored in the PC as data and are effectively *executed* on the I/O elements. Any additional functions needed to implement the AAG are put in a Dynamic Link Library. A Computer Program Identification Number manages these data.

Prior Implementations

Variants of this architecture have been successfully utilized with minor variation to implement USAF subsystems in the following areas:

1. The Ogden Data Device (ODD). This device is used to interface with data transfer cartridges on F-16 and A-10 aircraft. The ODD has been utilized for mission planning purposes for more than seven years with only minor modifications and updates to the driver software.
2. The Personal Computer Memory Loader Verifier. The USAF used this device for F-4 reprogramming during Desert Shield. It has been in continuous use by allied countries for more than eight years, and has performed flawlessly.

Additional Benefits

Because the development tools can be hosted on the computer that is being utilized to host the interface, a simple, quick, and mobile development environment can be established. This allows the development environment to be taken to the integration environment during integration. This is extremely convenient when no local hot bench or integration facility is co-located with the AAG development environment. Many times the equipment to be interfaced is in remote, otherwise inaccessible locations.

The equipment is usable in multiple environments: flight line, back-shop, development, and integration facility. Because the equipment is based on prior implementations, the development costs can be lowered by reuse of development artifacts.

The ideas presented have been implemented in the Common Aircraft Portable Reprogramming Equipment (CAPRE). The CAPRE has been chosen by the USAF to be the next generation loader/verifier as shown in Figure 4.

Figure 4: *The CAPRE System*

Conclusion

The benefits of this implementation include the following:

1. Long-term supportability.
2. Simple re-host.
3. Supports shorter PC life cycle and longer weapon systems life cycle.
4. Supports hosting of the development tools on target platform.
5. Mature and stable technology.
6. Useful in multiple environments: development, back-shop, as well as flight line.

This program is being implemented under program direction of Warner Robins Air Logistics Center with technical implementation accomplished by Ogden Air Logistics Center/MASMD.◆

About the Authors

Dan W. Christenson is branch chief for the Weapon Systems Software Engineering Branch, Maintenance Directorate, Ogden Air Logistics, and supervises a 146-person organization that specializes in automatic test equipment development and maintenance. He began his career as an electronic design engineer and has continued to act as a systems design engineer for major projects, including avionics design for the F-16 and F-22. Christenson has a bachelor's degree in electrical engineering from Brigham Young University and a Master of Science in computer science from Utah State University.

Ogden Air Logistics Center
Software Engineering Division
(OO-ALC/MASM)
7278 4th St., Bldg. 100
Hill AFB, UT 84056-5205
Phone: (801) 777-9863
E-mail: dan.christenson
@hill.af.mil

Lynn Silver is Section Chief for the Automatic Test Equipment and Weapons Systems Interface Engineering Section, Maintenance Directorate, Ogden Air Logistics where he directs a 50-person organization that specializes in automatic test equipment. He has previously worked as an electronic design engineer and was part of the Quality Engineering Support Team that established process quality assurance for the Software Engineering Division and later earned Capability Maturity Model Level 5 recognition. Silver is a former publisher of CrossTalk. He has bachelor's and master's degrees in electrical engineering from the University of Utah.

Ogden Air Logistics Center
Software Engineering Division
(OO-ALC/MASM)
7278 4th St., Bldg. 100
Hill AFB, UT 84056-5205
Phone: (801) 777-3823
E-mail: lynn.silver@hill.af.mil

The Probability of Success

Walt Lipke
Oklahoma City Air Logistics Center

What are the chances of completing this project on time? What are the chances of completing this project at cost? These are extremely intriguing questions. Being able to answer them would provide project managers with very significant information. Knowing the answers, project managers would have a greater likelihood of responding appropriately to their project's status. This article discusses the concepts underlying the computation of the project performance probabilities. The statistical methods applied to the earned value indicators for cost and schedule performance are explained. The resultant of the application is a graphic intended for management presentation, which has been termed a Performance Window. This article is intended for project managers and their earned value performance analysts. The article will be more meaningful to those having an understanding of Earned Value Management. An understanding of statistics will be helpful, but is not absolutely necessary.

Approximately 15 years ago, both private and public organizations were trying desperately to improve their product quality. The U.S. government and its industries felt significant pressure from Japan's success in improved product quality. The U.S. automotive industry was in serious financial trouble. Everyone went to classes to learn about Dr. W. Edwards Deming and Statistical Process Control (SPC). Total Quality Management was in vogue.

In reaction to this concentration on quality, the country experienced a quality revolution. This, in turn, spawned more refined quality efforts such as the Software Engineering Institute's (SEI) model for software process improvement [1] and the Six Sigma program developed by Motorola [2]. Regardless of the refinement, the foundation of quality understanding was the same: Statistical Process Control, the creation of Walter Shewhart about 75 years ago [3].

In the Software Division's efforts to improve software development process efficiency and product quality, the SEI model cited previously was used. Over several years of improvement efforts, our division integrated the use of two management methodologies, Earned Value Management (EVM) and SPC. During this period of time, we developed several extensions and applications that can be used by a project manager in the following ways:

- Anomalous performance identification [4].
- Project result prediction [4].
- Project/risk planning from historical data [4, 5].
- Measurement of process improvement [4, 5].
- Management reaction to project status [6].
- Preparation of a project recovery strategy [6].

Due to space constraints, CrossTalk *was not able to publish this article in its entirety. However, it can be viewed in this month's issue on our Web site at* <www.stsc.hill.af.mil/crosstalk> *along with back issues of* CrossTalk.

WEB SITES

Technical Committee on Real-Time Systems

www.cs.bu.edu/pub/ieee-rts/Home.html

The Institute of Electrical and Electronics Engineers Computer Society's Technical Committee on Real-Time Systems addresses issues in real-time systems, including embedded systems, control systems, monitoring systems, and multimedia systems. The committee promotes and facilitates the exchange of research results and development in the areas of applications, databases, distributed and parallel systems, formal methods and timing analysis, networks, operating systems and middleware, programming languages, scheduling and resource management, security, and verification and validation.

Real-Time Application Interface

http://www.aero.polimi.it/~rtai/index.html

This is the home page of the Real-Time Linux Application Interface for Linux, which lets you write applications with strict timing constraints for your favorite operating system. Like Linux itself, this software is a community effort.

embedded.com

http://www.embedded.com

Design engineers and engineering managers use embedded.com as a resource for embedded industry news and technical design information. Editorial features include daily news feeds and product information covering the latest happenings in the embedded industry, and weekly Web columns and polls. The site features 15 years of downloadable code, an archived and indexed database of product demos, information on embedded systems conferences, and the monthly issue of *Embedded Systems Programming*.

eg3.com

http://www.eg3.com/real

eg3.com is a community resource – an edited Yahoo for design engineers, original equipment manufacturers, and programmers. Founded in 1994, eg3.com identifies the best of the Web for embedded, digital signal processing, board-level, system-on-a-chip, real-time operating system/real-time, and open source for embedded systems. Also featured is a free-text search engine and a comprehensive search spider.

Real-Time Software Engineering Code 584

www.wff.nasa.gov/~code584

NASA's Real-Time Software Engineering Branch develops ground data systems for integration and test and on-orbit operations of Earth and space science missions. Branch personnel participate in teams with flight projects, principal investigators, other Applied Engineering and Technology Directorate centers and other organizations to develop integrated hardware and software systems for real-time mission support. Branch products include assembled commercial off-the-shelf systems, custom capabilities, components, consulting, and brokering on behalf of customers.

Real Time – Military Style

In the military, we are often concerned with real-time programming. A typical scenario would be an attack aircraft approaching its target. The onboard radar-warning receiver is reporting the ground-based search radar that is painting the aircraft from several kilometers away. As the target nears, another radar pops up with the signature sweep of tracking radar. The search was successful and another stage of the game begins. The attacker becomes a target.

On the aircraft, the electronic signatures speak of launch mode, and then launch. A digital signal processor analyzes the signature, and decisions are made in real time. Is the missile radar guided? What frequency is it using? What is the power level? How many are there? From which direction is it approaching? How fast is it approaching? How is it being guided? Is the radar on board the missile, or is it ground-based and passing guidance information to the missile via a communications channel? What frequency is the communication channel? Is the missile using infrared seeking? The electronic brains put together a counter measures plan and send command signals to activate the jammers and expend the flares if they are needed.

First comes the boost phase; the missile comes to life and gets its speed up to intercept the attacker. Usually there is no guidance available at this time. The electronics and hydraulics are activated. The internal brain comes to life.

In the second phase, the missile is up to speed and being vectored to its target by the target tracking radar. Like a teenager, it quickly develops its own field of view and sets out to prove itself. Modern missiles, like modern teenagers, are smart. Onboard processors are sifting through incoming data streams looking for counter measures from its quarry so it can apply counter-counter measures. It compares its electronic signatures with the sun to make sure it isn't tracking the sun. If it is, it goes back into search mode to reacquire the target. It might compute the track path and filter out the vertical velocity, then compare it to the acceleration of gravity to make sure it is not tracking a free-falling flare. If it is, it needs to re-acquire the target.

The aircraft counter measures are pumping out electromagnetic energy and flares to create false targets and misinformation about angle or range. However, is it working? Occasionally the silicon brains have to do a look-through to see if the foe is still in pursuit. If so, they have to make another decision. Can they handle it themselves? Should they pop another flair and hope the distraction is long enough to cause a miss? Is it time to warn the pilot and tell him to get us out of here? What would R2D2 do? Unlike a human heart, the clock-ticks of the onboard processors remain steady. Real-time decisions are made as to whether the missile has taken the bait, or if it has seen through the spoof and another tactic needs to be applied.

Back and forth they go, counter measures versus counter-counter measures. The last phase of this scenario is called the endgame. Here it will be decided who has the best technology or tactics. Sometimes good tactics can counter good technology. Yet fast processors, fast algorithms, and efficient code play an important part in the final decision. Good pilot training is essential. It all happens in real time.

However, was it real or simulated? It seems like cutting-edge software is mostly developed for the military or the gaming industry. We seem to oscillate between trying to amuse ourselves or kill ourselves, and with some of the games on the market today, well, we won't go there.

For example, here are some games (well, OK, the first are the titles that we should have, followed by the real titles). Seems to fit well with the Department of Defense (DoD) mindset, right?

- "Finding Demo." ("Finding Nemo.")
- "Madder 'n Hell 2004." (Madden NFL 2004.)
- "Freaky Flyers." (No kidding – this is a real title from Midway.)

It seems to follow that advances in technology in the real world translate quickly to advances in simulation technology, which also mirrors gaming technology. In a few years, we are going to have the first generation of pilots and warfighters in the DoD that was raised on state-of-the-art warfighting simulations. I wonder if they will find the Joint Strike Fighter a letdown?

– **Dennis Ludwig**
Aeronautical Systems Center
Wright-Patterson Air Force Base

THE DOOR TO THE CMMI IS WAITING FOR YOU ...

DO YOU HAVE THE RIGHT KEYS?

If you want your organization to use common, integrated, and improved processes for both Systems and Software, we can help. The Software Technology Support Center will show your organization how to implement the process improvement methodology of the Capability Maturity Model® Integration℠ (CMMI®), which addresses productivity, performance, costs, and stakeholder satisfaction. Make sure you have the right keys. Call us.

® Capability Maturity Model and CMMI are registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

Software Technology Support Center
MASE • 6022 Fir Avenue • Building 1238 • Hill AFB, UT 84056 5820
801 775 5555 • DSN 775 5555 • FAX 801 777 8069 • www.stsc.hill.af.mil

Sponsored by the Computer Resources Support Improvement Program (CRSIP)

Published by the Software Technology Support Center (STSC)

CrossTalk / MASE
6022 Fir Ave.
Bldg. 1238
Hill AFB, UT 84056-5820

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