

Complexity Plagues Reliable Software Development

Military systems require unique software solutions that aren't always off the shelf, but there are tools and methodologies that can help make code more robust.

Roy C. Webster
Senior Editor

Complexity is the major obstacle to developing reliable, fail-safe software for electronic and weapons systems in the defense and military sectors. The challenge to software reliability over the next 20 years will not be speed, cost or performance, but will be a matter of complexity. Creating reliable software using technology and tools currently available is extremely difficult because so much depends on the creative skills of individual programmers and developers.

The trend today is toward complex systems (Figure 1). Software is being written by large teams of programmers trying to produce increasingly complex programs in less time on shorter budgets. This results in errors that will be part of the finished product. It is commonly known that almost all software has bugs in it. The larger and more complex the code, the greater potential for bugs. Time-to-market pressures only make matters worse both in industry and in the military.

“The military looks at reliability differently depending on the application,”

said Steve Paavola, SKY Computers marketing director. “Systems that are integral to the safety of a pilot, for example, receive a great deal more attention in terms of software reliability than other lesser important systems that are not life, nor mission critical.”

For instance, modern aircraft are really flying computer platforms. A computer controls the operating parameters of the engine, fuel flow temperatures, engine rpm, inlet and exhaust temperatures, oil temperature and afterburner settings. The term “fly-by-wire” means there is no physical connection between what the pilot does and the actual control of the aircraft. All control is by computer electronics. But mission-critical software is not only found in an airplane. The same life or death scenario also is true in other military applications such as the monitoring of control parameters for nuclear reactors in submarines.

In addition, the magnitude of the reliability problem is compounded by having to integrate non-safety-critical and ancillary software on these platforms. Traditionally, the military would test millions of lines of software code line-by-line to ensure that each would deliver the expected result. Avionics applications utilize simulators to test all software before it goes “live” on an air-

The Softer Side

plane. An actual flight environment is established to simulate what the software will be required to do in actual flight to make or break the software before implementation.

Human Factor

Unfortunately, the human programmer is the weak link in developing reliable, predictable and fail-safe software. The mere fact that software debuggers exist emphasizes that the fundamental tools needed to ensure reliability and predictability do not exist. The use of a debugger only means that the design has been compromised and therefore the code's impact on the system has to be verified.

Typically, an independent verification test suite consisting of a collection of inputs and collection of expected outputs is applied on an executable software and hardware combination. The results are matched to verify that the system behaves correctly. By definition, this process is never complete and only results in software that is as reliable as the I/O scenarios invented by the human programmers. Thus, only some percentage of fault coverage can be achieved.

The verification test suite, according to Leo Mirkin, director of advanced product development at SKY Computers, is where the software problem most often occurs. He says, "What you have imagined are a lot of scenarios with inputs and outputs. If you missed a scenario, it means you have not checked the system against the combination of possible inputs. The number of scenarios approaches infinity as the complexity of the software increases." Something is

always missed because it's not possible—or extremely hard—to anticipate all the events that will occur. Thus, the system is unreliable by definition because not all the parameters have been exercised.

Unified Modeling Language

Today there is a movement to replace high-level languages with a development process that is divided into large-scale

corrections when necessary and enable efficient team collaboration for moderate and large-scale systems.

The idea is to have a combination of computer and software that behaves in the precise way it was specified or anticipated; that is, predictable. It should not deviate from the specified behavior. The next generation of quickly maturing approaches includes eliminating

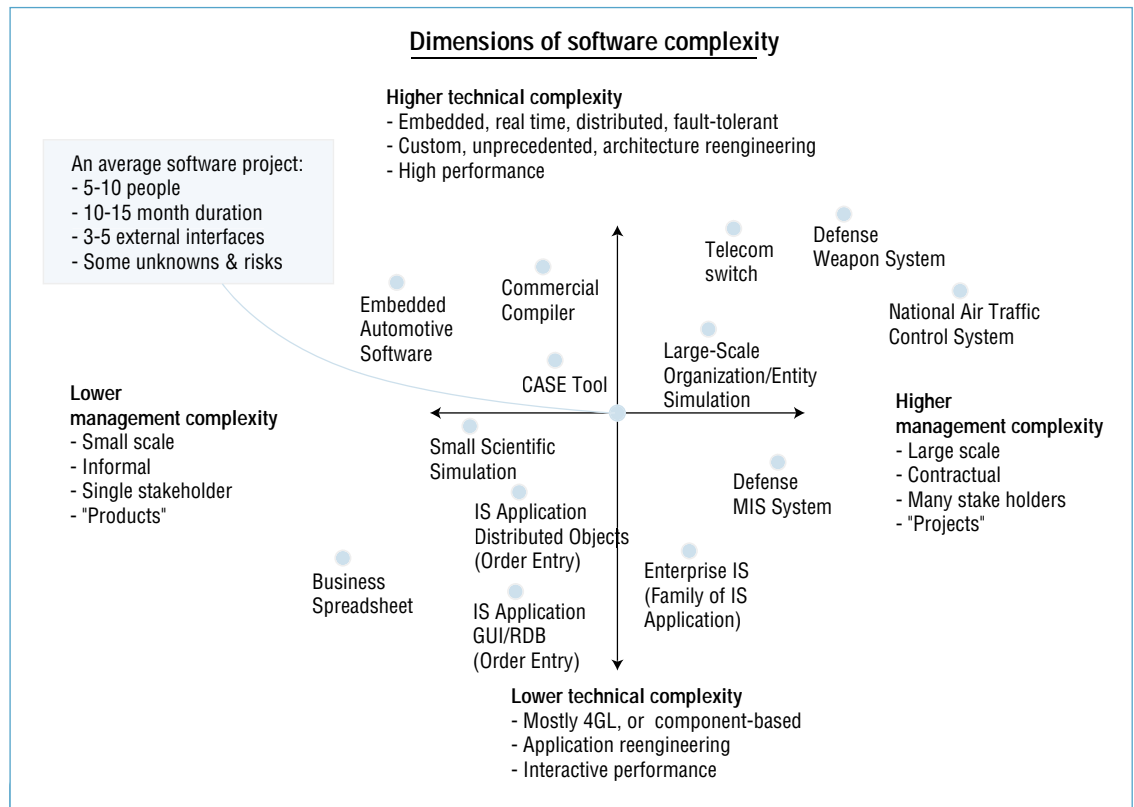


Figure 1

Defense and weapons systems and systems such as air traffic control are among some of the most demanding applications, in terms of both management and technical complexity.

activities or phases to simplify what needs to be done. Furthermore, many software architects are using a semantic framework and a notational scheme that together comprise the modeling language. The key is to develop a process that captures all the modeling elements to produce systems of consistent quality, to reliably produce systems with complex behavioral requirements and to predict completion times and development costs. It is also necessary to identify milestones during development that enable mid-course

error-prone human involvement in writing C or C++ code or any programming language. The concept is to have humans do what humans do best: specify requirements. These requirements will be implemented using a new, different type of notation called Unified Modeling Language (UML) that has been invented over several generations of very complex systems primarily in defense and military environments.

UML is a language for visualizing, specifying, constructing and documenting

the components of a software-intensive system. This language is an open standard that supports the entire software development lifecycle, supports applications areas, is based on experience and needs of the user community and is supported by many tools. It is therefore ironic that the software community has come full circle by its use of UML.

System analysts writing system specifications for programmers originally used UML. Today system analysts are replacing the programmers, and they are able to write the specifications for all parts of the entire system. This specification describes the system's behavior in a simple fashion, and it isn't necessary to understand how the system will implement it. Rather, the analyst merely specifies how an object will react once an event occurs.

And with the advent of powerful microprocessors, once the system's behavior is defined, software development can be done on workstations and personal computers that are now able to perform object-oriented programming (OOP). David Kleidermacher, director of engineering, Green Hills Software, says, "It is a common belief that if you program in an object-oriented manner using OOP and other qualities of the programming language, you will produce more reliable software. Because software is becoming more complex, programmers have to put more thought into its reliability."

Writing Reliable Code: An Operating System Can Help

Green Hills' Kleidermacher also believes a lot of the responsibility for building a reliable system falls on the operating system. One way to ensure greater software reliability from the operating system is to write it to use memory protection, a feature available in most embedded system microprocessors. However, many operating systems do not take advantage of the memory management unit (MMU), but simply turn it off when the system is booted. This dramatically reduces the reliability of the software. Taking advantage, the MMU increases reliability.

For example, it is possible to have running on the same microprocessor

software from two vendors such as a military avionics mission-critical application for flight control and a non-critical application from a second company. An operating system that doesn't use memory protection will have applications running in the same space as the kernel. Potentially, the application can crash, take the kernel with it and disable the whole system. This is clearly unacceptable in an avionics environment. Memory protection enables segregation of the system into applications that will run in their own address space.

Writing the operating system to use memory protection will force the system to segregate applications to run in their own memory address space. For instance, the Green Hills INTEGRITY RTOS is heavily used in avionics and other critical military applications. It employs multiple, protected address spaces that separate applications from each other and also from the kernel that controls the operating system. The RTOS, tightly integrated with the Green Hills AdaMULTI software development environment, is optimized for mission-critical embedded applications that require high reliability, security and testability. The RTOS features advanced memory protection capabilities, dynamic download, task- and system-level debug, a configurable real-time Event Analyzer, POSIX support and TCP/IP networking.

Lockheed Martin selected the Green Hills RTOS and AdaMULTI software development tools to develop Ada 95 and C/C++ software for Mission Systems in its Joint Strike Fighter (JSF) EMD aircraft. Avionics software developed by Lockheed Martin using AdaMULTI will run on airborne PowerPC processors operating under the RTOS. The AdaMULTI Integrated Development Environment (IDE) combines Ada 95/C/C++ compilers and the RTOS for developing real-time mission- and safety-critical software systems capable of meeting the security and safety standards of ISO/IEC 15408 (Common Criteria) and RTCA DO-178B.

Guaranteed resource availability is another critical factor in developing reliable systems. An example of resource

availability would have two tasks running on the system, one mission critical, the other not. Hypothetically, an error in the software causes the task to create multiple semaphores using memory from a central store. Simultaneously, all tasks on the system could be using memory from the same central store. What happens if the software is written incorrectly and the task allocates too much memory for all the semaphores? If all the memory in the central store is consumed by this one task, it adversely will impact the safety- or mission-critical tasks and they will fail.

Instead of having a central store like the typical operating system, some mission critical operating systems allocate a quota of memory to each address space that is defined statically when the system is booted. Thus, if a task is written in error, it can only exhaust its own address space but it is prohibited from adversely impacting other address spaces or the kernel. Being able to define the size of the address space and partition it from others for critical applications enhances the reliability of software.

Many operating systems don't have the capability to prevent one task from consuming too much CPU time. If one task spawns off other tasks, most schedulers will time-slice tasks of the same priority level. If too many are created because of a software error and they consume too much CPU time, the safety- or mission-critical task can get starved out.

Even with memory protection, there must be a way for the operating system to understand whether the access control is discretionary or mandatory. The system designer can define the access control desired to be enforced by the operating system. The operating system then will not allow an address space to give access to an object unless the system is designed to permit access. No task running in the address space can override the access control. Because security and fault tolerance require many of the same operating system features to be effective, an operating system that provides a high level of security will likely provide a high level of reliability for the overall system as well.

There are no tools available today to do logical checks on the condition of

The Softer Side

physical events like “Is the power supply producing power?” according to Derry Shribman, chief technical officer of Jungo, a software tools provider specializing in high-availability code and hardware drivers. This is because logical checks are very specific to the different applications. Each application has to have a different set of checks to know if the system is alive. There are some tools for assisting in transferring control of the primary computer to a redundant computer but much work must be done to integrate them. These are more like a set of utilities than a complete solution. Several vendors are currently working to try and solve these problems generically.

Reliability-Enhancing Software Futures

What lies ahead for new testing tools? The need for the customer to integrate a specific routine to test the software will always remain vs. testing hardware where generic software is used. When testing software, logic specific to the application must be available. If no ready-made tools are available to test the healthiness of the application, then an application must be written to test the healthiness of the application running.

Transfer of control is not so trivial especially in military applications that use nonstandard communications methods. If the application were TCP/IP, Ethernet or standard utilities that can assist with transfer of control from one to another, it would be straightforward. But a military application usually entails nonstandard communications using proprietary (and sometimes encrypted) wired and RF links, serial ports and other high-speed serial communications that are much more deterministic than Ethernet. These need different ways to transfer control, so too with engine and flight control.

Tools to handle engine control will need special tailoring. Development of these tools usually takes from 6 to 12 months for an existing project to be converted to high-availability specifications. This is the typical time it takes to migrate from existing code to a more robust and redundant program. Over the last two years, for example, projects in which

Jungo consulted with the Israeli Army and Israeli companies, most engine control code was destined for nonstandard operating systems and military unique nonstandard hardware.

As software becomes more complex and projects get more complicated, the trend is gravitating towards standard operating systems and tools. In the past, it was possible to code from the ground up, but now more and more ready-made tools are being used. For instance, military system designers today are approaching Jungo looking for standard generic software solutions. Trouble is there are no standard solutions available for many unique military systems.

Unfortunately, most tool solutions today are fit for noncritical applications, and the demand is for proprietary or nonstandard solutions. Hot-swap tools available today are only able to answer the first layer of high-availability military applications. And it's not a simple task to do the integration work it takes to produce a high-quality solution for high-availability applications. It is hoped that in the future there may well be software that will be able to assist at improving reliability in more of these projects, but today the reach of the software tools is quite limited. ■■

The need for the customer to integrate a specific routine to test the software will always remain vs. testing hardware where generic software is used.



Green Hills Software
Santa Barbara, CA.
(805) 965-6044
[www.ghs.com]