Superweapon Software Woes

Weeding out errors in Defense Department computer programs that operate everything from missiles to tanks is a frustrating business

By IVARS PETERSON

There were only two little mistakes in the computer program that guided a new missile through its firing test last December. But the mistakes were serious enough to cause the computer-controlled missile to drift from its plotted trajectory, and a human controller had to save the missile from crashing. The result was an expensive delay in a major Department of Defense program.

Donald R. Greenlee, a specialist in the Defense Test and Evaluation office, uses the incident to illustrate that "in the digital world, it only takes one bit to bring on catastrophe." One of the errors was simply a reversal in sign, a negative where a positive should have been.

To Greenlee, the incident is also a success story. "The test was conducted, and it did what was intended — to reveal a problem," Greenlee says. Laboratory simulations allowed design engineers to duplicate the failure noted in the flight test. This narrowed down the search, and a line-by-line check of the relevant parts of the computer program eventually isolated the mistakes. The computer program was fixed, and subsequent firing tests were successful.

Software errors, mistakes in computer programs, occur frequently. According to Edward Miller of Software Research Associates in San Francisco, during software development every 1,000 lines of computer program typically contains 10 to 80 "defects," all of which must be caught somehow. Some programs are millions of lines long.

This problem is a special concern at the Defense Department because almost every system in the current and planned military inventory relies on computers and microprocessors extensively. For example, computers control the targeting and flight of missiles, coordinate and control sophisticated systems installed in high-performance aircraft, and integrate the complex activities of battlefield command. "Consequently, software has become the dominant factor in military systems," says Edith W. Martin, deputy undersecretary of defense for advanced technology. One reason for DOD's increasing reliance on software is that software changes are easier and less costly to make than altering the comparable function built into electronic circuits, she says. New threats can be met by making appropriate changes in computer programs.

Software must be reliable, particularly in life-threatening situations, and that means weeding out any errors. However, testing and evaluating computer programs efficiently and effectively is very difficult. Many senior program managers at DOD feel much more comfortable with hardware, such as jet engines or artillery shells, than with computer software. "We know very well how to test an artillery shell. We've been doing it for centuries," says Greenlee. "We know how to condition the thing environmentally — heat it up and freeze it, drop it, and then test fire it. But computer software? You can't even see it, let alone beat it with a hammer or attempt to destroy it. It requires a different intellectual approach."

U.S. Air Force

DOD's concerns were reflected in a recent software test and evaluation conference held in Washington, D.C. Sponsored by the National Security Industrial Association, a collection of defense-related companies, the meeting brought together representatives of government, industry and universities to review and discuss experiences and advances in locating software errors. Rear Admiral Isham Linder, Defense Test and Evaluation director, in his keynote address pinpointed two key questions: "How much testing is enough, and how should it be conducted?"

The most obvious method for testing a computer program is to track exhaustively every possible logical branch in the program or to try every possible calculation, but in large programs this method is impractical. Even a simple program with just 10 two-choice branches provides more than 1,000 different paths. Anything more complicated could require months of computer time and might generate thousands of pages of output. Who's going to read all the results? Who knows the results are correct? In general, an exhaustive test that covers all the possibilities is impossible.

One way to get around this difficulty is to sample randomly a small portion of the possible paths in a computer program. If enough tests are performed, and all errors found are corrected, then the probability of any remaining errors should be very low. Some software developers even deliberately seed their large programs with a known number of errors. If an independent testing group finds all the deliberately introduced errors and others besides, then the probability is high that practically all the errors in the program have been found.

"That's a far cry from dropping an artillery shell off the back of a truck," says Greenlee.

Another approach is to construct test cases that pinpoint specific types of errors known to occur during the course of software development. However, developing a comprehensive set of suitable test cases can take as much effort as writing the computer program itself. More theoretical methods, like mathematical "proofs of correctness," have turned out to be too complicated to apply in real situations or restricted to a few special cases.

Part of the problem is that computers are usually brought in when the mental task is too hard for the human user to do in sufficient time. "Therefore, you're trying to test something which, by definition, is mentally challenging," says Greenlee. "You don't sit down and check out a complicated piece of software on the back of an envelope."

"Testing is hard work," Miller told the conference participants, and no one disagreed. Victor R. Basili of the University of Maryland concluded his presentation, "It is almost frightening how many open

questions there are in a field where we have been working so long."

Within DOD, software development "ranges from a reasonably effective, disciplined approach in a few systems to near chaos in others," says Martin. A U.S. Navy study, for example, reveals 13 different mathematical systems in use for steering an airplane from one place to another. In the U.S. Air Force, as much as 90 percent of computer program lines are coded in a primitive, difficult-to-decipher computer language. In many situations, programmers find it easier to start over rather than try to modify existing software. The U.S. Army, in a 1978 survey of about 100 battlefield systems, found 34 different versions of essentially the same computer, each operated by a different computer language. This diversity creates headaches for those responsible for testing and upgrading the systems, and produces problems on the battlefield when one computer has to communicate with another.

Any standardization program to improve "interoperability" and make testing easier faces immense obstacles. For instance, Captain David Boslaugh of the Naval Material Command pointed out that the Navy has about 450 different systems and subsystems with "embedded computers." The number of computers in use is doubling every two years. About 50 million unique lines of software, in a variety of computer languages, are currently operating active systems. To redo these lines would take years, considerable expertise and at least $85 billion, said Boslaugh.

Nevertheless, because of DOD's growing reliance on computers, efforts to rationalize the software development process are continuing. Many in DOD, especially in the Army, are counting on a new, committee-built computer language, Ada, which is the result of a seven-year DOD-sponsored design effort. Ada promises to help computer programmers work more quickly, with fewer errors, and to allow the development of portable computer programs capable of running on almost any computer instead of just a few models.

The Pentagon has mandated that software for most military systems be written in Ada, and the language will probably be in routine use by 1985. However, Ada's prospects for becoming a standard language outside of DOD and military applications are limited because of doubts about its ability to handle complicated scientific calculations (SN: 7/31/82, p. 72). Some critics also see Ada as a big, complex language that eats up costly computer memory space. The language offers so many options that, despite the emphasis on programming in "packages" and the use of English-like sentences for computer instructions, Ada would be difficult to learn, they contend.

The Ada approach is part of an effort to bring more discipline to software development. At one time, computer programmers were akin to magicians, cleverly stringing together chains of logical statements, using whatever tricks they could invent, that somehow got a computer to do what it was supposed to do. Such undisciplined efforts, imbued with programmers' idiosyncracies, proved difficult to test and modify when they sprang unexpected errors.

Many conference participants argued that programs had to be written with testing in mind, and that this approach had to be emphasized in the training of programmers. Basili said that a recent experiment he conducted at the University of Maryland showed that students altered their programming styles when they knew their programs were to be tested by an independent reviewer. Several students were apologetic because they had avoided trying anything "funny" and instead concentrated on meeting the specifications and making their programs as clear as possible to the reviewer. To Basili, this shift in attitude was encouraging.

Carolyn Gannon of General Research Corp. in Santa Barbara, Calif., argued that one way of helping both programmers and testers was to compile and study the kinds of errors made during software development. This record would indicate where to look for mistakes, which tests to use to find them and how to handle them. When enough data are collected, these error analyses could be used for developing new tests and for estimating the probability of hidden errors still left in complicated computer programs. Although such data would be valuable, one problem is that programmers are reluctant to admit they make mistakes, and companies don't want the public to know how many errors are made, even if the errors are corrected, Gannon said.

Testing and evaluation already take up as much as half of the budget for software development, so contractors are naturally reluctant to spend extra money on compiling error histories. At DOD, when program budget cuts are necessary, the test program itself (as the "bringer of bad news") often is an early victim. The frequent result, however, is the discovery of surprise problems late in a program or perhaps even on the battlefield. Greenlee says, "The earlier the developer finds deficiencies, the quicker, easier and cheaper it will be to fix them."

Last year at an Electronics Industries Association meeting, Brig. Gen. Robert D. Morgan described "Airland Battle 2000," the Army's evolving doctrine for fighting on future battlefields. "The new doctrine requires continuous action by many elements," he said. "There is no forward edge of the battle area or line of scrimmage. Many battles are conducted over wide areas by units which appear to act independently but, in fact, know their role and strive for a common goal." Computers and satellite communication systems tie together the array of electronics systems that will have to operate in a "chemical, nuclear and electronic warfare environment." In such complicated "systems of systems," finding software errors early becomes even more important.

Col. Edward Akerlund of the Air Force Systems Command said these coming complex networks introduce whole new areas of problems. Programmers are just beginning to learn how to put together these large systems, and the development of testing procedures lags far behind. He said that tests are needed, for example, to ensure that when part of a system fails, the rest of the system does not go down.

Concerns like this led DOD to initiate the Software Test and Evaluation Project, an effort to develop guidelines for the test and evaluation of defense systems software and to identify useful testing tools that showed promise and were worthy of further research. One of the key issues raised during the early stages of the project involved the amount of testing required. Because it is difficult (expensive and time-consuming, too) to find every error that may exist in a computer program, one need is for a formal risk assessment procedure that balances the risks of not doing a test against the number and nature of errors likely to still reside in the program. One preliminary recommendation was that testing should be done in proportion to the risks involved if a failure were to occur. Linder noted that a quantitative measure of this risk would be very helpful for high-level decision makers who have to decide whether a certain project should proceed.

This spring, DOD plans to launch another program, a $250 million, 10-year "software initiative." One aim of the STARS (software technology for adaptable, reliable systems) program is to create a software engineering institute where DOD, in cooperation with industry and universities, can evaluate and demonstrate the usefulness of new programming techniques and integrate these ideas into military systems more quickly. The institute would also train DOD personnel. As several conference participants pointed out, plenty of programming and testing tools exist, but the information is hidden in obscure journals, locked in company testing centers or scattered in bits and pieces and applicable only to particular computers and computer languages. Some collecting and sifting of this material would be valuable, they agreed.

Spending time and effort on learning how to catch mistakes reflects a recognition that no human-designed system is perfect. Software errors are as likely to come up in a business program that generates invoices as in a program that is supposed to coordinate five space shuttle computers. However, software errors in DOD computer systems, whether in missiles, satellites or at command headquarters, can have drastic consequences. Even one little mistake could be one too many. □