

PROGRAMMED FOR DISASTER

Software Errors That Imperil Lives

by JONATHAN JACKY

ON MARCH 21, 1986, oilfield worker Ray Cox visited a clinic in Tyler, Texas, to receive radiation on his back, from which a cancerous tumor had been removed. On the basis of previous visits, he knew the procedure should be painless, but that day he felt a jolt of searing heat. Outside the shielded treatment room, the technician was puzzled by his complaint: the computer terminal used to operate the radiation machine bore only the cryptic message "Malfunction 54," indicating either an overdose or an underdose. Clinic staff were unable to find anything wrong with the machine, so they sent Cox home and continued treating other patients. But Cox's condition worsened. Spitting blood, he checked in to a hospital emergency room, where doctors suspected he had received an electric shock; in fact, he had received a lethal overdose.

Less than a month later, Malfunction 54 occurred again at the Tyler clinic, this time striking Verdon Kidd, a sixty-six-year-old bus driver. Kidd died in May of 1986, reportedly the first fatality ever caused by an overdose during radiation treatment. Meanwhile, Cox became paralyzed and lapsed into a coma, and four months later he, too, died of the injury.

As news of the Tyler incidents spread, similar mishaps in other clinics came to light: one patient in Canada and another in Georgia had received mutilating injuries in 1985; an overdose occurred in Washington State in January of 1987. In each case, the radiation had come from the Therac-25, a computer-controlled machine manufactured by Atomic Energy of Canada. After the fifth such accident was reported, the U.S. Food and Drug Administration advised—but did not order—clinics to discontinue routine use of the Therac-25 until safety features could be installed in the equipment.

Ironically, by the time the accidents occurred, radiation therapy had become a routine, safe, and frequently effective procedure, used on nearly four hundred and fifty thousand new patients a year in more than eleven hundred U.S. clinics. Much of that success was due to linear accelerators, which began to replace cobalt treatment units in the 1960s. (Linear accelerators are electric machines, capable of producing radiation beams whose



energy can be adjusted. The earlier equipment used a lump of cobalt-60 as the radiation source; the units could not be adjusted, and the intensity of the radioactivity diminished over time.)

The million-dollar Therac-25, introduced in 1983, was among the first of a generation of computer-controlled linear accelerators. With the earlier accelerators, electricity and mechanical forces were used to transmit the operator's directions directly to the wheels, levers, and cables controlling the radiation beam. But with computers, it was necessary to transmit only information, not force, so the operator's commands could be processed by software—lists of coded instructions that tell the computer what to do. Thus, complex electromechanical control systems were replaced by minicomputers, and the traditional operator's control panels, festooned with switches, but-



Cham Hendon, Air Crash, 1983

tons, and lamps, were exchanged for computer terminals. With these changes, manufacturers hoped to capitalize on the speed and versatility of computers to make radiation machines faster and more convenient.

The proper operation of an electromechanical radiation machine largely depended on the soundness of the control mechanism. When it malfunctioned, the problem could be traced to relay switches that had failed, tubes that had burned out, hydraulic fluid that had leaked away. These failures were caused by manufacturing defects or wear and could be prevented by inspecting the product and replacing faulty parts.

Computers also can wear out, but most of their problems are not so easy to understand; they are design weaknesses, caused by flaws in the logic, not the mechanics, of the control mechanisms, so there are no material defects

to track down. True, faults may exist in the hardware—the chips themselves—but, more frequently, they crop up in the software.

It was a software error, involving the operation of a switch, that killed Cox and Kidd. Linear accelerators can produce two kinds of radiation: electrons, which are used to treat superficial tumors, such as skin cancers, and X rays, which are more effective against such deeply embedded tumors as those of the cervix and the prostate. An electron beam can be produced by the accelerator directly, or an X-ray beam can be created by placing a tungsten target in the path of the electron beam, so that, as the target absorbs electrons, X rays emerge from the other side. Because this process of producing X rays is inefficient, the intensity of the electron beam must be increased tremendously when the target is in place.

To guard against the grave danger that the electron beam might attain its higher intensity without the target in place, and be driven directly into a patient, accelerators were equipped with protective circuits, called interlocks. In the Therac-25, however, both target position and beam intensity were computer-controlled. When the operator switched the machine from X ray to electron mode, the computer was counted upon to set the beam to low intensity before the target was withdrawn.

Usually, it worked that way. (At Tyler, more than five hundred patients had been treated without mishap in the two years preceding the accidents.) But if, because of a software error, the operator first selected X rays and then switched to electrons, by hitting the up-arrow key and typing over the previous instruction, the target was withdrawn while the full-intensity beam remained on. In the cases of Cox and Kidd, the Therac-25 delivered about twenty-five thousand rads of electrons, more than a hundred times the prescribed dose.

The problem in the software itself was compounded by a weakness in the user interface (the system the computer employs to inform the operator of what it is doing), which encouraged operators to run the machine in a hazardous fashion. According to a therapist at the site of the Georgia accident, the Therac-25 typically issued as many as forty error messages a day, most of which indicated that beam intensity was slightly less than it should be. Such messages could be canceled by pressing the terminal's *P* key, and operators quickly learned to respond this way to virtually all error messages—each one difficult to interpret, because the problem was referred to by number instead of being described in words. At Tyler, the only indication of trouble the operator saw was the code "Malfunction 54." She repeatedly pushed *P*, turning on the beam again and again. Ray Cox was burned twice before he managed to move out of the way.

The fault cannot be placed simply on the programmers who wrote software for the Therac-25; even the best programmers make lots of mistakes, because software writing is a painstaking task. The underlying problem was that Atomic Energy of Canada failed as an organization to guard against such errors. One of the earlier models, the Therac-20, contained electric circuits that prevented the beam from being turned on in such instances of malfunction. This mechanism was omitted from the Therac-25, and it is evident that no sufficient review of the safety implications of this omission ever was made.

Fortunately, only eleven Therac-25s were in use when the hazards became known. But the incidents raised concerns about therapy machines soon to be introduced by other manufacturers, and about other kinds of computer-operated medical equipment, as well. The FDA anticipates that by 1990, virtually all devices produced by the eleven-billion-dollar-a-year medical-electronics industry will contain computers. And the Therac accidents were just the worst examples of a trend the agency has been tracking for several years: computer-related problems in medical devices are on the increase.

Twice as many manufacturer recalls of computer-controlled medical equipment occurred in 1984 as in 1982 or any year before, and most were due to software errors. One blood analyzer displayed incorrect values because addition, rather than subtraction, had been programmed

into a calibration formula. A system for monitoring the blood pressure, heart rate, and other vital signs of several patients at once mixed up its data, so the name of one patient was attached to the readings for another. And in one ventilator, a software error allowed concentrations of oxygen and other vital gases to drop without warning. In many of these applications, failure of the computer-controlled system could be deadly.

Such problems are not restricted to medicine. Because of their low cost and versatility, computers are replacing mechanical operations in all kinds of products. The Airbus Industries A320 airliner attracted great press attention when it was introduced last year, because it was the most extensively "fly-by-wire" commercial airliner ever built. (Computers, rather than cables and hydraulics, connect the pilot's controls to the airplane's engines, rudder, and ailerons.) In new cars, computers manage fuel injection and spark timing, and, in some cases, the suspension and antilock braking mechanisms, as well. (General Motors is experimenting with a "drive-by-wire" automobile, in which there is no physical connection, except the computer, between the steering wheel and the tires.) On railroads, computers operate the track switches that are supposed to prevent trains from colliding. Computers are used widely to direct assembly line equipment in factories and generators in power plants; even some emergency shutdown systems in nuclear reactors are computer-controlled. And in weapons systems, computers warn of imminent attack, identify and track targets, aim guns and steer missiles, arm and detonate explosives.

A file of problem reports maintained by Peter Neumann, at SRI International (a computer-industry think tank, in Menlo Park, California), under cosponsorship of the Association for Computing Machinery, lists more than four hundred incidents in a range of industries in which software problems caused or threatened serious injury or significant financial loss. Unless steps are taken to ensure that computer-operated machinery is every bit as safe as the mechanical equipment of old, such alarming software failures are bound to multiply.

WHEN MANUFACTURERS began installing computers in medical equipment and other machinery, they introduced a problem never encountered in strictly mechanical devices: programming errors, popularly known as bugs, which are the natural result of the way software is produced. The process begins when the designers draw up specifications for the software, which requires that they anticipate every task the computer is supposed to be able to direct its machine to perform, as well as every response it should make to situations it might encounter. (For a radiation machine, say, designers would specify that the computer be able to turn on the machine, adjust the beam to various intensities, and keep X ray-strength radiation from operating in the absence of the shield.) Then the design is divided into modules, which are work assignments for individual programmers.

Each programmer, acting independently, creates the software for his module, by entering into a video terminal a list of statements, in programming language, instructing the computer to behave as the specifications direct. Because computer programming is a highly creative endeavor—no less creative than composing a novel, for example

—there is no single correct approach. Just as the novelist relies on inventiveness to get a story convincingly across to the reader, the programmer must rely on ingenuity to impart complete and unambiguous instructions to the computer. That is why no two programmers will write the same program in exactly the same way, and why there is so much room for introducing errors or for coming up with instructions not comprehensive enough to work properly.

During the final stage of software production, the individual modules are woven into the finished program. Regrettably, it is not always possible to predict how modules will interact. As Marvin Minsky, dean of American artificial intelligence researchers, put it:

When a program grows in power by an evolution of partially understood patches and fixes, the programmer begins to lose track of internal details, loses his ability to predict what will happen, begins to hope instead of know, and watches the results as though the program were an individual whose range of behavior is uncertain.

Consequently, producing quality software is as much a function of how well the entire program is designed, and how well the steps are coordinated, as of how well the individual lines of code are written.

The only way to tell whether a program has been properly designed is to test it. But even the most rigorous of trials can overlook flaws, because the number of situations a program must contend with is limitless. To check whether a computerized calculator adds numbers correctly, for instance, one could add hundreds of random

numbers and see whether the program consistently produces the correct result, but since the supply of numbers is virtually infinite, it is hopeless to try testing every possible sum, and the few that might trigger an error, owing to some software flaw, are likely to be missed. So errors routinely remain in software when it reaches the market, to be discovered and corrected over time.

Programmers accept that they are likely to introduce, inadvertently, about fifty errors in every thousand lines of code. Most of these are weeded out during testing; market versions typically contain only two to three errors in a thousand lines. But this means that even a good program—with, say, fifty thousand lines of code—may contain more than a hundred errors. Usually, the damage can be repaired, though at some cost in time and annoyance. The state sends you a twenty-million-dollar tax bill? Clear it up with a telephone call—or several calls. The telephone switching computer cuts off your connection? Hang up and dial again. A word processor deletes your letter? Type it over, and this time make a backup copy. Experienced computer users develop a defensive style, a whole repertoire of tactics to keep software errors from getting the better of them. Only human adaptability and ingenuity make it possible to base a computerized society on imperfect products.

But some products require better defenses against errors; when a computer controls a linear accelerator or a jetliner, the results of an error cannot always be overcome or ignored. If the patient dies or the plane crashes, the computation cannot be redone.



Cham Hendon, Train Crash, 1986

IT IS POSSIBLE to improve computer products for which safe operation is critical, by adapting, for software development, the principles of safety engineering now used to minimize hazards in non-computer-operated assembly line equipment, medical devices, and the like. The first step is to recognize that safety must be designed into a product, not added on as an afterthought. That requirement alone might lead to a design quite different from one that would be used if cost and performance were the only considerations.

One way of keeping hazards to a minimum is through a process known as software engineering, by which each stage in the programming of a computer is described in writing, and the descriptions are reviewed by outside programmers. Typically, the documents produced include such things as a specification, describing in exhaustive detail what the product is supposed to do; a design guide, telling how the program is organized; a test plan, spelling out a series of trials that are supposed to show that the program works as promised; and a test report, presenting results and explaining how any problems were resolved. Requiring programmers to obtain approval of each document before proceeding to the next step enforces an orderly development process, and, ultimately, helps ensure quality. It is analogous to a standard practice of civil engineering whereby detailed designs are subjected to thorough analysis and review before anyone starts pouring concrete.

Such a system runs counter to the stereotype of the eccentric genius programmer. It requires that programmers spend half their time on planning and design, much of the rest of it on testing, and only fifteen to twenty percent actually writing code. Software engineering is also time-consuming, and therefore costly, though studies have shown that fixing an error after a product has reached the customer may cost as much as a hundred times more than catching it early in development.

Software engineering is practiced on a large scale by the U.S. Defense Department and its contractors, who must produce sixteen documents for each computer program. Even that system is not foolproof. Some software developed to the department's standards still reaches the field with serious errors: One computer-controlled wing-mounted launcher improperly retained its grip after its missile was ignited, creating what was described as the world's largest pinwheel, when the aircraft went violently out of control. And a jet crashed when its flight-control program was confronted with an unanticipated mechanical problem. But despite its imperfections, the military's documentation process has been mainly beneficial. Something as complex as a fly-by-wire aircraft, for example, could never be relied upon to operate safely if it were produced by traditional techniques, and, for the most part, the military's planes have flown safely.

Unfortunately, the Defense Department's approach is elaborate and expensive, and thus it is practical only for huge projects costing no less than tens of millions of dollars, employing scores of programmers, and generating hundreds of thousands of lines of code. It is not economical for smaller-scale programming projects (those required to produce automated switches for railroad tracks, say, or computer-controlled medical equipment). The Institute of Electrical and Electronics Engineers

recommends the production of only six documents, and FDA officials are considering a similar approach for medical-equipment software. But some medical-equipment vendors oppose the additional documentation effort, calling it ineffective, costly, and time-consuming. Apparently, the cost of correcting hazardous errors has not yet caught up with them.

Another way of improving safety in programming involves the application of "formal methods," or techniques of mathematical logic, to software analysis. Designers create models of computer programs in specialized notations resembling equations. A model's behavior can then be analyzed before the product is actually constructed—in much the way the carrying capacity of a bridge can be calculated from an engineer's blueprints—and attempts can be made to prove that a program is error-free, employing the kind of mathematical logic used to prove the validity of theorems in geometry. The aim is to supplement trial-and-error testing, after the product is completed, with logical analysis beforehand.

Although computer scientists have been developing formal methods for more than twenty years, this approach is hardly ever put into practice. Some scientists believe the techniques are too difficult and cumbersome to be useful in evaluating any but the simplest programs (those that are only a few hundred lines long). Others caution that they promise too much, since formal proof does not guarantee that a program is perfect.

Yet formal methods need not ensure perfection to be useful, for they can uncover errors sometimes overlooked by trial-and-error testing and documentation review. Recently, the British Royal Signals and Radar Establishment, the electronics research laboratory of Britain's Ministry of Defense, applied formal methods to program fragments drawn from NATO's military software inventory. One in ten fragments was found to contain errors, many serious enough to result in the loss of the vehicle or machine controlled by the program. The British are betting heavily on formal methods to ensure software safety; new defense ministry regulations will require their use for some safety-critical products, and a government-sponsored report strongly encourages their use in civil products, as well. Nothing similar to this is planned in the United States.

IF MANUFACTURERS of computer-controlled medical devices, production-line equipment, and other potentially hazardous machinery continue to resist using safety engineering and formal methods, it may become necessary for government agencies to force their hands. We regulate all kinds of other things that pose risks to human life—buildings, bridges, airplanes, and drugs are just a few. We also require that the people who provide safety-critical services satisfy certain education requirements and pass examinations: physicians, pharmacists, lawyers, and even automobile drivers all must be government-certified.

Still, software remains largely unregulated. Aviation, nuclear power, and weapons systems purchased by the Defense Department are the few cases in which software is subject to government approval. There is no education standard required of programmers, and many computer-science curricula fail even to mention safety. Studies have



Donald Sultan, Early Morning, May 20, 1986

found that the best programmers can be more than twenty-five times as competent as the worst and that many software-design supervisors are unable to evaluate or even understand their programmers' work. Of course, efforts to regulate software or certify programmers are likely to meet resistance. "I'll fight them to the death," says Robert Ulrickson, president of Logical Services, a Santa Clara, California, company that designs computerized instruments. "I don't want to be part of an economy that's run by the government."

But, for the most part, the mere appearance of government interest in software safety has had a positive effect. In September of 1987, in the wake of the Therac-25 tragedies, the FDA announced its intent to regulate some computer software in medical devices and began to work out guidelines. At about the same time, a staff member of the U.S. House of Representatives Committee on Science, Space, and Technology began investigating soft-

ware quality-assurance practices on behalf of Congress. Seeing the writing on the wall, manufacturing and professional associations are instructing their members in software-quality assurance and acceptance testing. Employees are being sent to courses on computer-system safety, and unusually thorough attention is being devoted to checking for hazards in new machines. Some manufacturers have even contracted with outside firms to perform independent tests and evaluations. Given the low priority traditionally assigned to software safety, such efforts promise considerable improvement. ●

JONATHAN JACKY, a research assistant professor in the radiation oncology department at the University of Washington School of Medicine, in Seattle, is designing software for a computer-controlled radiation-therapy machine. His first article for THE SCIENCES, "Ada's Troubled Debut," appeared in the January/February 1987 issue.