Established 1845 SCIENTIFIC AMERICAN December 1985 Volume 253 Number 6

The Development of Software for Ballistic-Missile Defense

What some call a "Star Wars" defense would depend on computers to control an unprecedentedly complex array of weapon systems. Developing reliable software for such a defense may be impossible

by Herbert Lin

"The ultimate goal of the Strategic Defense Initiative (SDI) is to eliminate the threat posed by nuclear ballistic missiles," according to the interim charter of the Strategic Defense Initiative Organization. To achieve the goal of comprehensive defense, the organization is empowered by the Department of Defense to manage research programs that examine the feasibility of developing technology for a ballistic-missile defense (BMD) to protect cities and military assets. Such a defense would destroy or incapacitate nuclear-warhead delivery systems on the way to their targets. A wide variety of defensive weapons might be employed. They include lasers, particle beams, electromagnetic railguns and nonexplosive "kinetic kill" vehicles.

The Strategic Defense Initiative Organization recognizes that the computer technology that would control individual weapons and coordinate their operation is of comparable importance to the development of the critical new interception technologies. The command-and-control system for a comprehensive ballistic-missile defense must be capable of flawlessly receiving and acting on information pertaining to thousands of missile launches, tens of thousands of warheads and hundreds of thousands of decoys. It must do all this within the half hour it would take for an intercontinental ballistic missile (ICBM) to travel from a launch site in the Soviet Union to a target in the U.S. Because the system must be highly automated, there would be virtually no time for human intervention to correct unexpected failures. For the most part the execution of a computer program would replace human decision making once the BMD system was engaged.

Before the U.S. makes a serious effort to develop software for such a system, three questions should be considered. What is the nature of a BMD system? What are the obstacles to BMD software development? Can these obstacles be circumvented?

The comprehensive defense system most often discussed by Strategic Defense Initiative Organization officials (and the system's critics) consists of four tiers. That is, the ballistic-missile defense would attack hostile missiles in each of the four phases of their flight. The phases are the boost phase, during which a multistage launch vehicle carries the payload through the atmosphere; the postboost phase, during which nuclear warheads in reentry vehicles and "penetration aids" such as decoys are sequentially released above the earth's atmosphere by a maneuverable "bus"; the midcourse phase, during which the reentry vehicles and decoys traverse the greater part of their trajectory, and the terminal phase, during which the warheads in their reentry vehicles penetrate the atmosphere and detonate at their assigned targets.

A four-tiered ballistic-missile defense gives the interceptors several opportunities to destroy the offensive weapons. Within each tier a defensive system must successfully detect and track targets before it can destroy them [see "Space-based Ballistic-Missile Defense," by Hans A. Bethe, Richard L. Garwin, Kurt Gottfried and Henry W. Kendall; SCIENTIFIC AMERICAN, October, 1984]. Computers and appropriate software are needed to coordinate operation of the defense and evaluate its effectiveness. This coordination process is called battle management. Although experts are still undecided as to how a battle-management system should be organized, a hypothetical structure might include "local" computers and software that are responsible for battle management within each defensive tier. Each tier's system would then be connected with the other tiers through a global battle-management system.

The software guiding battle management within a given defensive tier would control the local sensors and weapons. These sensors would locate and track potential targets and distinguish actual targets from decoys. This part of the software might create a "track file" that contains all the known information about each target. The software could then allocate defensive resources in a specific tier by coordinating the track-file information with the available weapons and the programmed rules of engagement, rules that determine under what circumstances targets are to be attacked. The global battle-management system

would assess the extent and nature of an attack in progress and specify the rules of engagement for each tier. In order to prepare a local battle-management system to engage warheads that have leaked through preceding tiers, the global system might pass on track-file and sensor information obtained in the boost-phase tier to each succeeding defensive tier.

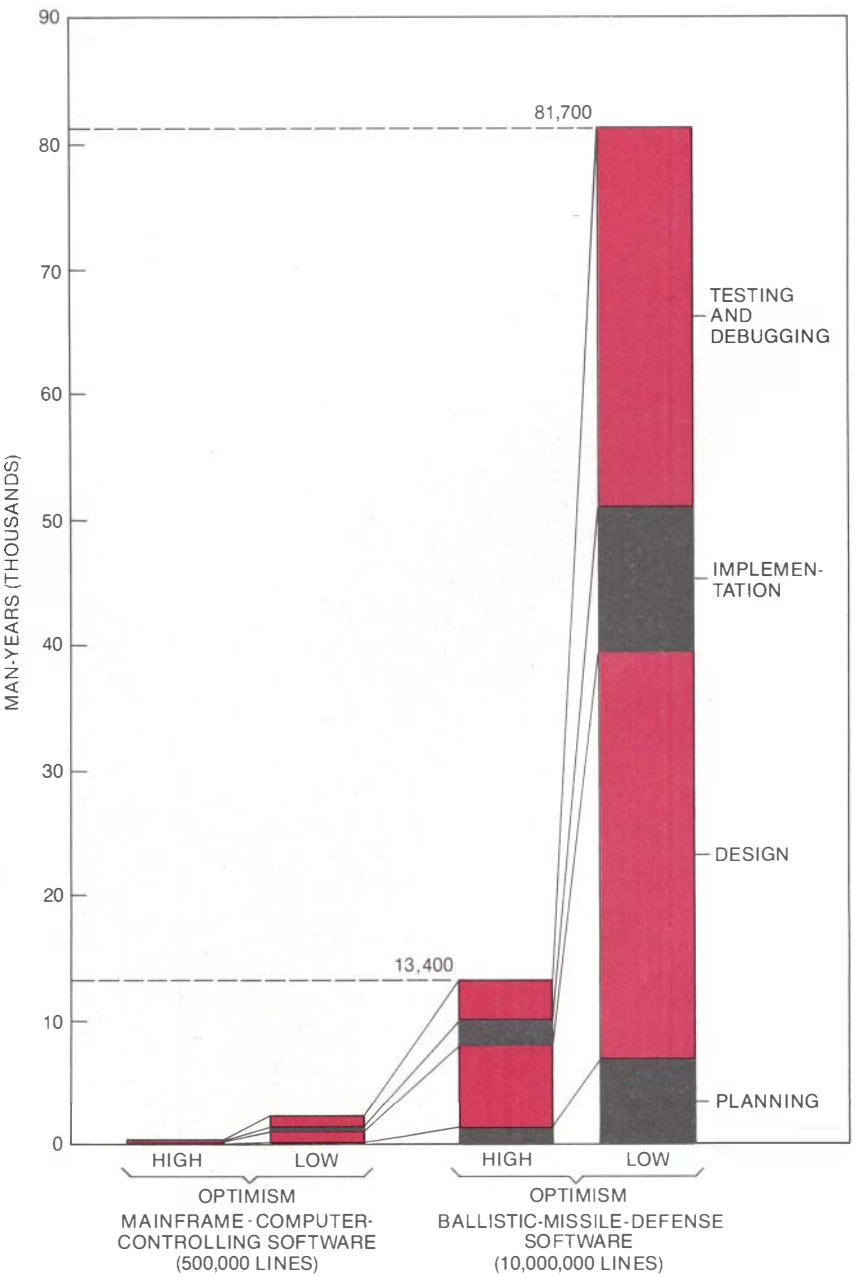
A ballistic-missile defense depends heavily on the software controlling it; defective software might lead to failure. Hence software development is a critical factor in attaining the objectives for which the system is designed.

Software development is an intellectual process that can be divided into distinct conceptual phases. These include planning, design, implementation, testing and debugging. In actual software-development projects the phases are not always carried out sequentially; for example, plans can change after a project has begun. Consequently developers may have to redesign a piece of software.

The first crucial area of software development is planning: the developer needs to determine what functions the software is to perform and to envision the different situations to which the software must respond. Specification, or deciding what action should be taken and at what time, becomes more difficult as a task grows in complexity. A simple example from everyday life makes the point. If the task is to count the number of people in a small audience, it is intuitively obvious how to proceed. If the audience consists of tens of thousands of people in a football stadium, the counting task requires much greater elaboration. For example, what defines the boundary of the stadium? How does one define a person? (Should the fetus in a pregnant woman be counted as a person?) Clearly, factors that appear infrequently and are therefore irrelevant in the first case may complicate the specification of a task as it grows in size.

The precise specification of what a ballistic-missile defense must do is a complicated task. For instance, the statement "Shoot down all Soviet missiles" is sufficient if the world contains only Soviet missiles and every Soviet missile should be shot down under all circumstances. But the world is not so simple. How can Soviet missiles be distinguished from non-Soviet missiles? What if a Soviet missile is headed for a target in East Germany? While these questions only scratch the surface of specification, they broach a fundamental problem developers will encounter, namely that it is often difficult to decide whether some particular aspect of a program is desirable or undesirable.

Moreover, developers will need to accurately predict every contingency and then decide how the software should respond to each. For instance, what procedures should the software follow if the computers in one defensive tier's battle-management system fail? How might a ballistic-missile defense differentiate a Soviet space-shuttle launch from that of a Soviet ICBM? Because of the nearly infinite number of possibilities, developers will be hard pressed to foresee all possible circumstances. Simply writing out all situations a BMD system might have to face and the appropriate actions to be taken in each case might plausibly require tens of thousands of pages. In comparison the U.S. Tax Code, which is the



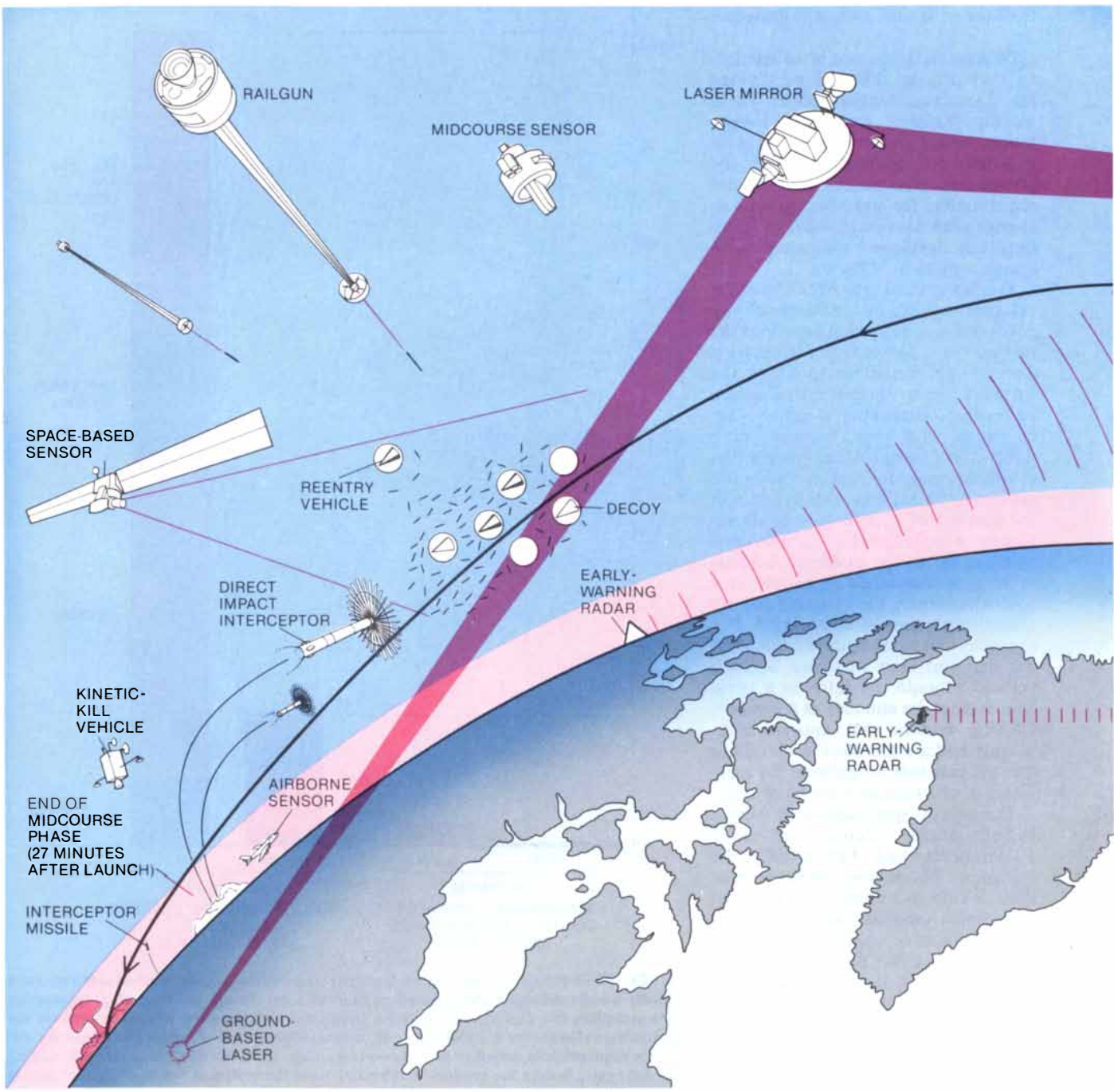
LEVEL OF EFFORT required for each of four stages in the development of software for a ballistic-missile defense (which would amount to some 10 million lines of programming code according to a Government study) is contrasted with the effort needed to produce the controlling software for a mainframe computer (500,000 lines of code). The stages are not strictly sequential; in actuality there is some overlap. Also, the final stage refers to testing and debugging before the product is delivered; once the system is "on line" further testing and debugging would be necessary. For each software-development project the bar at the left represents estimates based on highly optimistic assumptions; the bar at the right represents estimates based on less optimistic assumptions. The scaling function relating level of effort to program size is taken from *Software Engineering Economics,* by Barry W. Boehm.

legislative specification for Federal tax law, fills about 3,000 pages. The possibility that all potential scenarios would be adequately accounted for in the specification of the BMD software is at least as unlikely as the possibility that no unanticipated loopholes exist in the Federal tax laws.

Two examples may serve as further illustration. First, on June 3, 1980, the North American Aerospace Defense Command (NORAD) reported that the U.S. was under missile attack. The report was traced to a faulty computer circuit that generated incorrect signals. If the developers of the software responsible for processing these signals had taken into account the possibility that the circuit could fail, the false alert might not have occurred.

A second example, involving the sinking of the British destroyer H.M.S. *Sheffield,* came to light in the aftermath of the Falkland Islands war. According to one report, the ship's radar warning systems were programmed to identify the Exocet missile as "friendly" because the British arsenal includes the Exocet. As a result the system ignored the transmissions of a hostile Exocet's homing device and allowed the missile to reach its target, namely the *Sheffield.*

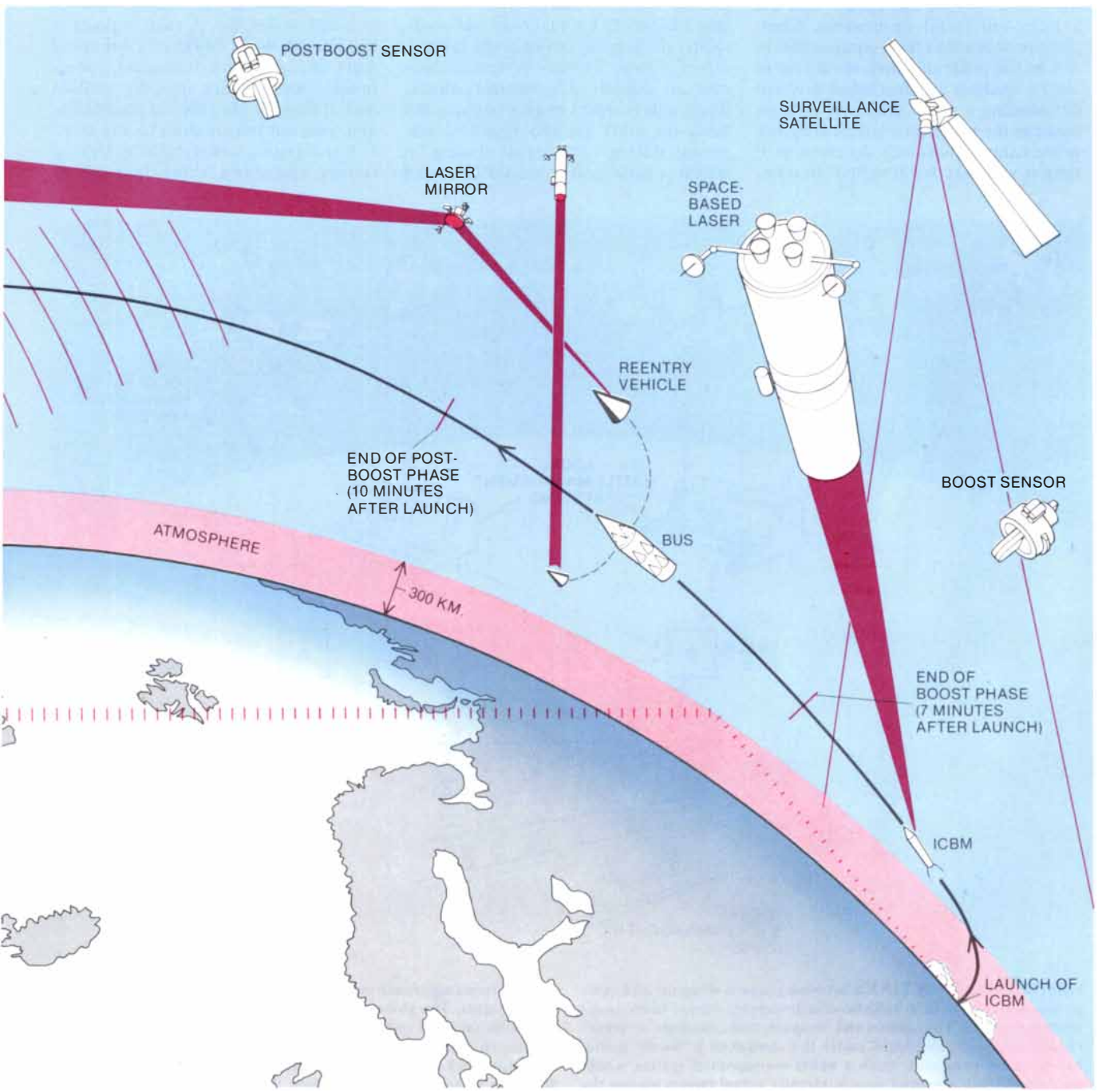
The design phase of software development is also rife with potential problems. In this phase developers ask how the specifications that emerged from the planning stage can be imple-



PROPOSED HARDWARE for a ballistic-missile-defense system includes an assortment of intercepting weaponry: high-powered lasers (based in space or on the ground), railguns and missiles. In addition satellite, airborne and ground-based sensors would be needed. Enemy ballistic missiles would be engaged in every phase of their trajectory: the boost phase, during which the missile is under pow-

mented through computers. Developers conceive the algorithms that must be incorporated into the software, the sequence certain actions are to follow, and so on. Conceptually the design phase resembles the task confronting an architect who must draw the blueprints for a house after the requirements have been determined. If the architect incorrectly assesses the ability of the surrounding land to carry off water, the basement can flood, causing damage that is expensive to repair. Similarly, design errors in software development, if not caught in the design phase, can be expensive to correct.

Important software systems have not been exempt from design errors. For example, the manned space capsule *Gemini V* missed its landing point by 100 miles because its guidance program ignored the motion of the earth around the sun. In another case five nuclear reactors were shut down temporarily because a program testing their resistance to earthquakes used an arithmetic sum of variables instead of the square root of the sum of the squares of the variables. The lesson is that neither the nature nor the frequency of planning and design errors is predictable; indeed, these errors can be eliminated only if analysts can perceive them beforehand, a task that becomes more demanding as system size or complexity increases. Detecting and correcting errors is therefore a fundamental aspect of any software-development project. How then can errors



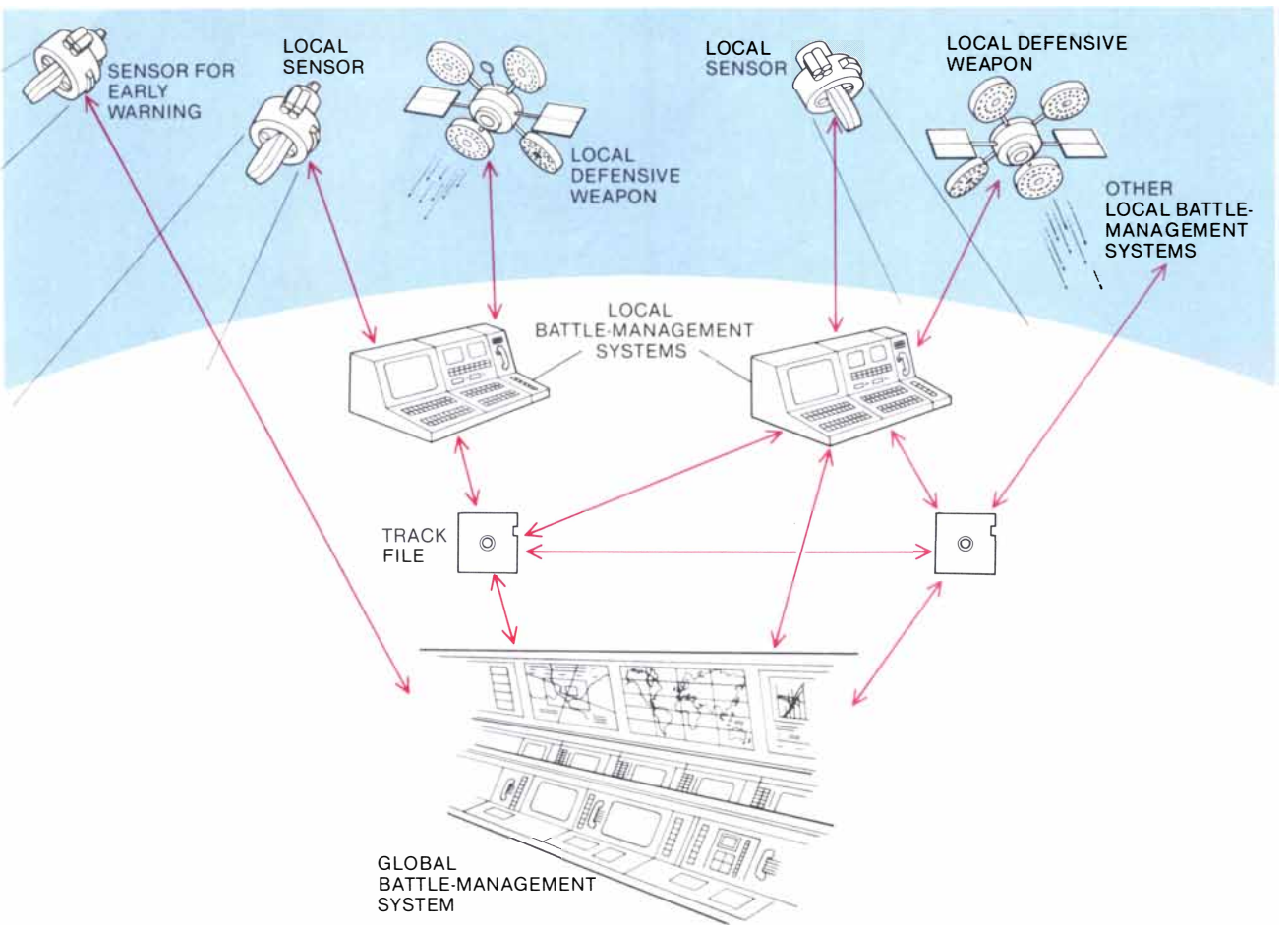
ered flight in the atmosphere; the postboost phase, during which reentry vehicles (equipped with nuclear warheads), decoys and obscuring metallic chaff are released; the midcourse phase, during which the various objects travel the major part of their course in space, and the terminal phase, during which the reentry vehicles penetrate the atmosphere to detonate over their respective targets.

be found and confidence in the operation of the system be assessed?

Two techniques are effective for assessing the trustworthiness of a software system. One technique is analytical. It requires that the program's correctness be proved by mathematically certifying that its output conforms to certain formally specified criteria when expected input data are received. Although proofs of program correctness help to ensure that a software system meets its theoretical specifications, they do not ensure that the system will fulfill its mission. Complete proofs are at least comparable in size to the programs they are trying to verify; analysts are thus faced with understanding a proof that is as complicated as the program itself. A program proof cannot guarantee the amount of time it will take for a system to complete "real-time computations" (a critical factor that will be discussed below), nor can it reveal the nature of the output when the system receives unanticipated input data. Finally, proofs of correctness cannot provide any promise that the actual program specifications themselves are correct.

A more important assessment technique is empirical testing. Clearly a ballistic-missile defense cannot be subjected to a large-scale empirical test under realistic conditions. The cost of such a test would be staggering. Of greater concern, however, is the fact that the Soviet Union could not confidently distinguish between the launching of a large number of test missiles and an actual U.S. nuclear attack. System developers must therefore fall back on more limited types of empirical testing: small-scale testing, in which a BMD system would be tested against a few missiles, and simulation testing, in which a computer would mimic large-scale threats to the targets a BMD system protects. Some experts maintain that such tests make actual large-scale ones unnecessary. Although such tests do increase confidence, they cannot ensure that the mission of a comprehensive ballistic-missile defense will be accomplished.

Problems that arise from integrating individual components into an effective system often appear only when the complete system is tested near its limits, something that cannot be done in small-scale tests. A case in point is the World Wide Military Command and Control System (WWMCCS), a communication network used by civilian and military authorities to coordinate and transmit information to and from U.S. military forces in the field. During routine operations, when the message



COMMUNICATION LINKS between sensors, weapons and computers are crucial in a ballistic-missile-defense (BMD) battle-management system. The sensors and weapons that constitute a "layer" of defense would be placed under the control of a "local" battle-management computer. Such a battle-management system would locate and track potential targets, identify actual targets among the debris and decoys, assign its weapons to specific targets at specific times and assess whether the target was successfully destroyed. All relevant information concerning the targets that have been engaged would then be placed on a "track file" and passed on to the next local battle-management system as well as a global battle-management system. The global system would be in constant communication with all local systems. For each of them it would specify the precise circumstances under which given targets could be attacked, transmit track-file information and coordinate actions to protect the BMD system itself. In addition the global battle-management computer would receive the first warning that an attack was under way. The network diagrammed here could be severely disrupted by an attack on the single vertex: the global battle-management system. Such vulnerability could be minimized through redundancy.

traffic is low, the WWMCCS has performed satisfactorily. When the message traffic is high, however, the performance of the system has suffered. In a 1977 exercise, when it was connected to the command-and-control systems of several regional commands, the WWMCCS had an average success rate for message transmission of only 38 percent.

As a way of overcoming the limitations of small-scale testing, developers often rely on simulators to create likely attack scenarios. Although simulation provides information beyond that achieved by small-scale testing alone, the technique, as applied to the testing of a ballistic-missile defense, is restricted in several ways. First, simulators may not be able to reproduce the "signatures," or parameters, of the physical phenomena associated with various events, such as simultaneous nuclear explosions, rapidly and accurately enough to test the responsiveness of the defense system. Analysts must therefore rely on assumptions about the signatures of each event they choose to simulate; that is, events are "preprocessed" before they are fed into the defense system. As a result of using preprocessed data, there can be no surprises during the simulation that might test how the system reacts to circumstances not anticipated by developers, and so the realism of the test is diminished.

Although advances in computer technology have greatly increased the computational speed of computers, and thereby facilitated the design of simulators that can model more realistic environments, increased speed will not help to simulate data on physical phenomena for which theoretical and empirical understanding is inadequate. For instance, given present knowledge, scientists would not be able to model accurately the simultaneous explosion of several closely spaced nuclear weapons under certain conditions. More important, simulators cannot satisfactorily duplicate all plausible attacks, since a determined and clever opponent ultimately chooses the parameters of an actual attack. Consequently any confidence in a BMD system that is based on simulated tests rests on the assumption that those responsible for the simulation can predict and reproduce in electronic form the range of tactics to which an enemy might resort.

The story of the Aegis air-defense system illustrates the limitations of simulation testing. The battle-management system for Aegis is designed to track hundreds of airborne objects in a 300-kilometer radius and then allocate enough weapons to destroy about 20 targets within the range of its defensive missiles [see "Smart Weapons in Naval Warfare," by Paul F. Walker; SCIENTIFIC AMERICAN, May, 1983]. Aegis has been installed on the U.S.S. *Ticonderoga,* a Navy cruiser. After the *Ticonderoga* was commissioned the weapon system underwent its first operational test. In this test it failed to shoot down six out of 16 targets because of faulty software; earlier small-scale and simulation tests had not uncovered certain system errors. In addition, because of test-range limitations, at no time were more than three targets presented to the system simultaneously. For a sizable attack approaching Aegis' design limits the results would most likely have been worse.

Errors are not unexpected in operational tests; indeed, malfunctions are inevitable during the initial shakedown exercise of a new weapon system. By the time of the next Aegis tests, they had been corrected and none recurred. Future exercises will probably uncover additional errors, which will in turn be fixed. Through this process the performance of the Aegis system will gradually improve.

Unlike the performance of Aegis, the performance of a comprehensive ballistic-missile defense against a large-scale attack will not improve with experience. Because large-scale empirical testing is impossible, the first such test for a comprehensive BMD system would be an actual large-scale attack on the U.S. A more complex battle-management system that must keep track of more targets and operate under tighter timing constraints is unlikely to have a better performance record than Aegis has. In addition a severe software failure in a full-blown battle situation would offer little or no opportunity for system designers to learn from the experience.

Nevertheless, developers of a ballistic-missile defense will attempt to improve the system's performance by testing for errors and eliminating them, by expanding software capabilities in response to newly perceived needs and by adding new hardware and concomitant software to the project. These efforts, along with bringing new workers into the project, come under what specialists call software maintenance. It can account for approximately 70 percent of the total life-cycle costs of a software-development project.

Two major maintenance issues are particularly salient in relation to a system for ballistic-missile defense. First and most important, significant errors discovered after the software has been put into operational use must be eliminated. These errors can range from an incorrect coding symbol to a fundamental design flaw. On June 19, 1985, the Strategic Defense Initiative Organization did a simple experiment. The crew of the space shuttle was to position the shuttle so that a mirror mounted on its side could reflect a laser beamed from the top of a mountain 10,023 feet above sea level. The experiment failed because the computer program controlling the shuttle's movements interpreted the information it received on the laser's location as indicating the elevation in nautical miles instead of feet. As a result the program positioned the shuttle to receive a beam from a nonexistent mountain 10,023 nautical miles above sea level. This small procedural error was of little significance to the test itself, however; a second attempt a few days later was successful. Nevertheless, the event shows that even simple errors can lead to mission failure.

Although the bug in the space-shuttle experiment was easily rectified, removing errors from a ballistic-missile defense will require considerably more effort. In particular the battle-management software must receive and process information in such a way as to keep pace with circumstances external to the computer during an attack. That is, the software must run in "real time" [*see illustration on next page*]. One problem with debugging a real-time software package is ascertaining why it may operate with one specific configuration of equipment but not with another that differs only slightly. For example, a certain missile can be currently launched at supersonic speeds by the F-4G Wild Weasel aircraft but not by the F/A-18 Hornet: a software system that works in the F-4G is not entirely compatible with the avionics of the F/A-18.

A second problem of particular significance to debugging real-time software is that analysts often find it difficult to make errors recur, something that is essential if bugs are to be located. Real-time output is often determined by factors such as the arrival times of sensor inputs. These determinants of a program's behavior cannot always be reproduced with enough accuracy for the error to be found and eliminated. Indeed, finding an error requires an understanding of the precise circumstances leading to it. Because that is not always possible with large real-time systems, software errors may be identified in practice only in a probabilistic sense. Large computer programs, in fact, usually evolve through a series of incompletely understood changes. After a while the pro-

grammer can no longer predict outcomes with confidence; instead he can only hope that the desired outcome will be attained.

There is a final complication to debugging real-time software: even if an error can be located, attempts to eliminate it may not be successful. The probability of introducing an error (or more than one) while eliminating a known error ranges from 15 to 50 percent. Moreover, the majority of software-design errors that appear after software is put into service do so only following extensive operational use. Experience with large control programs (ones consisting of between 100,000 and two million lines of code) suggests that the chance of introducing a serious error in the course of correcting original errors is so large that only a small fraction of the original errors should be remedied.

In the context of a comprehensive ballistic-missile defense, one should therefore ask about the consequences of an error that would manifest itself infrequently and unpredictably. The details of the first operational launch attempt of the space shuttle provide an example. The shuttle, whose real-time operating software is about 500,000 lines of code, failed to take off because of a synchronization problem among its flight-control computers. The software error responsible for the failure, which was itself introduced when another error was fixed two years earlier, would have revealed itself, on the average, once in 67 times.

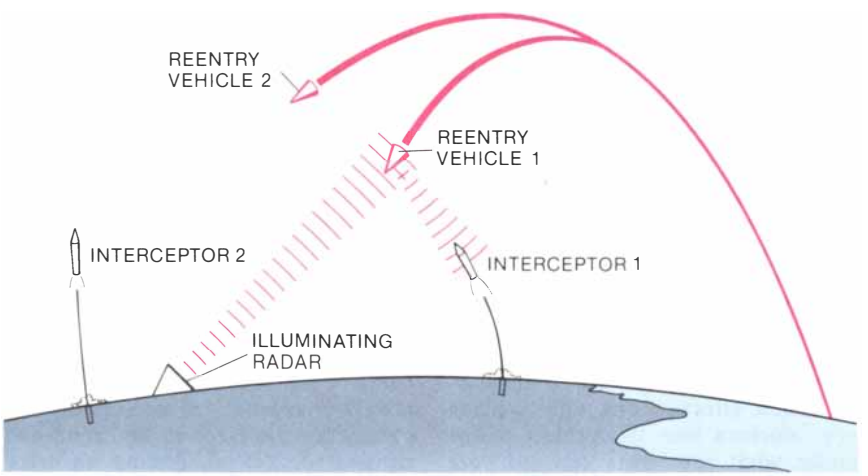
Aside from detecting and removing errors, a second aspect of software maintenance stands as a hurdle to developing a defensive system: the management of product development. The Defensive Technologies Study Team (DTST), chartered by the Department of Defense to examine the feasibility of a comprehensive ballistic-missile defense, estimates that the entire system will require a minimum of 10 million lines of programming code. In comparison, the entire software system for Aegis is an order of magnitude smaller. If the DTST estimate is low by a factor of only two, even a very optimistic software-development project will entail more than 30,000 man-years of work, or at least 3,000 programmers and analysts working for about 10 years. For this reason the project can expect a staff turnover that will reduce the institutional memory of the project. During staff transitions it is conceivable that an essential detail, such as updating a particular subprogram, could be overlooked. A tragic example of management error occurred in 1979, when an Air New Zealand airliner crashed into an Antarctic mountain; its crew had not been told that the input data to its navigational computer, which described its flight plan, had been changed.

It is also possible that Soviet agents might try to sabotage the project. They could, for instance, deliberately introduce hard-to-find flaws into the system that would become evident during a real attack but not during testing. The possibility that a program might contain some hidden "time bomb" that would cause the software to fail at a critical moment cannot be ignored. The obvious preventive step would be to impose security clearances on a "need to know" basis. Such restrictions, however, would inhibit communication among the staff working on different parts of the system and thereby increase the likelihood of serious, unintentional software errors arising.

Tightening security will not prevent the Soviet Union from thwarting the system by several other means. There is no reason to believe that Soviet officials faced with a U.S. BMD system would not try to disguise the observable characteristics of their ballistic missiles, warheads and decoys during tests or in actual use. Without reliable data on Soviet missiles, U.S. developers cannot be sure the defensive system will work. Furthermore, the Russians may develop new tactics or weapons that would force U.S. analysts to reprogram the battle-management system to meet new threats. Accumulated changes over time would probably result in many unpredictable interactions and could ultimately necessitate a total redesign of the system.

It has been suggested that the development of software for exceedingly complex systems could be facilitated by reliance on "expert systems" and automatic programming. For example, a Defense Advanced Research Projects Agency (DARPA) report stated that expert systems may be applicable to a ballistic-missile defense. Expert systems are detailed descriptions, expressed as computer rules, of the thought processes human experts use to reason, plan or make decisions in their specialties. Typically, expert systems use informal reasoning procedures or rules of thumb. For example, an expert system may take the following as a given: If a Soviet ICBM is launched, it is a threat to the U.S. It may then use the converse of the statement as a legitimate rule of inference: If there is a threat to the U.S., it is a Soviet ICBM launch. The validity of this inference tool cannot be proved or disproved using the standard tools of formal logic; the statement is not true all the time but is sensible most of the time. The reliance on informal reasoning procedures in programming expert systems leaves open the possibility that deeply embedded conceptual contradictions in the system could exist that might cause it to fail.

To date expert-system research has focused on well-defined areas such as biochemistry and internal medicine. These are areas in which human expertise is sufficiently developed to allow expert systems some success. Even so,



"REAL TIME" SOFTWARE must be executed at a rate defined by the timing of events external to the computer hardware. A computer controlling the radar illumination of an incoming reentry vehicle so that an interceptor missile can home in on the reflected signal would have to maintain the radar beam on the reentry vehicle until the vehicle was destroyed. If the radar were turned away too soon, the interceptor might "lose" its target, particularly if the reentry vehicle were designed to evade approaching interceptors. Yet the computer must re-aim the radar beam so that a second reentry vehicle could also be intercepted. The computer program has to shift the beam to the second vehicle once the first is destroyed, and it must do so quickly enough to enable a second interceptor to reach its target.

the technology is still limited. Those who recommend applying expert systems to battle management for ballistic-missile defense ignore the fact that human expertise is based on human experience. No one has expert knowledge of massive nuclear missile attacks based on experience.

Equally improbable is the idea that automatic programming, the use of computer programs to write other programs, will provide a solution. Air Force Major Simon Worden, special assistant to the director of the Strategic Defense Initiative Organization, has stated that "we're going to be developing new artificial intelligence (AI) systems to write the software. Of course, you have to debug any program. That would have to be AI too." Statements such as this one are misleading. The primary function of automatic programming is to alleviate the technical difficulties of implementing design specifications and modifying existing code. Roughly half of all software errors, however, are the result of human choices and decisions made during the planning and design phases. These human choices could not be delegated to automatic programming.

Two scenarios based on the above discussion show how the software for a comprehensive ballistic-missile defense might fail. Suppose a battle station that has successfully intercepted two missiles in operational testing using an electromagnetic railgun is faced with a large-scale Soviet missile attack. At first projectiles launched from the battle station destroy their targets. Unfortunately at the design stage developers neglected to take into account the fact that less massive objects have more recoil. As more projectiles are fired the recoil becomes stronger and skews the aiming algorithms for the railgun. As a result later projectiles are too slow and do not reach their targets in time to destroy them. Or suppose that, during the early phase of a nuclear war, U.S. and Soviet leaders agree to a cease-fire. The U.S. leaders realize that one submarine captain will not receive the cease-fire message in time but are confident that U.S. missile-defense satellites will be able to shoot down any missiles launched by mistake. Only after the submarine's missiles are launched does it become tragically clear that no developer had specified that the satellites might have to shoot down U.S. missiles. The missiles explode over the Soviet Union with catastrophic consequences.

Because they are anticipated, neither of these errors is likely to occur. The problem is not whether a system contains a particular error but rather how likely it is that the system will contain any one of millions of potential errors. The primary matter of concern is "unknown unknowns," the potential errors that remain unanticipated.

SM-2 MISSILE is part of the U.S. Navy Aegis air-defense system. The Aegis battle-management system was designed to track hundreds of airborne objects and to carry out up to 20 simultaneous intercepts. During its first operational test it failed to shoot down six of 16 targets because of faulty software. The system's malfunctions have since been repaired. A comprehensive ballistic-missile defense, which is a much more complicated and ambitious weapon system, must perform far better than the Aegis system did, and it must do so in its first try; a severe software failure during the system's first full-scale test would offer virtually no opportunity to rectify the problem. Because large-scale testing is impossible, the first empirical test of the BMD system would be an actual large-scale ballistic-missile attack.

All the problems cited in this article are related to software planned, designed, implemented and debugged by experienced software engineers; most, if not all, were corrected. The general technique for correcting these errors, namely discovering the bugs in operational use and then correcting them, is unlikely to be fruitful in the development of a BMD system. Yet a comprehensive ballistic-missile defense requires not only that software operate properly the first time, in an unpredictable environment and without large-scale empirical testing, but also that planners are positive it will do so. The Reagan Administration has stated that empirical testing is essential for maintaining the reliability of nuclear weapons and has opposed a comprehensive test ban on these very grounds. One wonders how it is possible to have confidence in a comprehensive ballistic-missile defense, which will be at least as complicated as nuclear weapons are now, without requiring it to meet comparable test standards.

Proponents of the Strategic Defense Initiative argue that even in the absence of large-scale empirical testing a ballistic-missile defense is still desirable, because the possibility that the system might work reasonably well would deter Soviet leaders from contemplating an attack. Others argue that the goals of the Strategic Defense Initiative are really more modest than the goal of a comprehensive defense against ballistic missiles, suggesting that a BMD system might be most appropriate for protecting military assets such as missile silos. Of course, for goals more circumscribed than those of comprehensive ballistic-missile defense, perfection is not a requirement and testing is much easier. Although a program that has more limited goals does have a much higher probability of success, these limited goals raise a disturbing issue. In particular they are inconsistent with President Reagan's stated vision of eliminating the threat of nuclear ballistic missiles. Consequently it is proper to judge the feasibility of the Strategic Defense Initiative on the basis of the president's challenge to the scientific community. By that standard no software-engineering technology can be anticipated that will support the goal of a comprehensive ballistic-missile defense.