Chapter 9
Software

CONTENTS

Tables

Chapter 9
Software

INTRODUCTION

The performance of a ballistic missile defense (BMD) system would strongly depend on the performance of its computers. Chapter 8 describes the pervasiveness of computers in the operation of a BMD system, and as well as in its development, testing, and maintenance.[1] Sequences of instructions called *software* would direct the actions of the computers, both in peacetime and in battle. As shown in table 8-1, software is responsible both for the actions of individual components of the system (e.g., a radar), and for coordinating the actions of the system as a whole. As coordinator, software maybe thought of as the glue that binds the system together. As the system manager, software assesses the situation based on data gathered by sensors and reports from system components, determines battle strategy and tactics, and allocates resources to tasks (e.g., the weapons to be fired at targets.)

The role of software as battle manager is crucial to the success of a BMD system. If software in a particular *component* failed-even if the failure occurred in all components of the same type simultaneously-other components of different types might compensate. But if the *battle management* software failed catastrophically, there would be no way to compensate. Furthermore, the battle management software may expected to compensate for systemic failures, both because of its role as manager and because software is perceived to be more flexible than hardware. Consequently, the battle management software would have to be the most dependable kind. Thus it is the focus of most of the SD I software debate.

*Table 8-1 illustrates many of the ways in which computers would be used in a deployed BMD system.

Note: Complete definitions of acronyms and initialisms are listed in Appendix B of this report.

The BMD Software Debate

The envisaged BMD system would be complex and large, would have to satisfy unique requirements, and would have to work the first time it is used in battle. Many computer scientists, and software engineers in particular, have declared themselves unwilling to try to build trustworthy software for such a system. They claim that past experience combined with the nature of software and the software development process makes the SDI task infeasible. David Parnas has summarized their major arguments.[2] Other computer scientists, however, have stated that their belief that the software needed for a Strategic Defense Initiative (SDI) BMD could be built with today's software engineering technology. Frederick Brooks, for example, has said:

> I see no reason why we could not build the kind of software system that SDI requires with the software engineering technology that we have today.[3]

Those willing to proceed believe that an appropriate system architecture and heavy use of simulations would make the task tractable. Their arguments are summarized in a study prepared for the Strategic Defense Initiative Organization (SDIO) by a group known as The Eastport Group.[4] The critical role played by the software in BMD makes it important to understand both positions.

[1] David L. Pamas, "Software Aspects of Strategic Defense Systems," ***American Scientist, 73:432-40,*** September-October 1985.

[3] From a statement by Dr. Frederick P. Brooks at the Hearings before the Subcommittee On Strategic and Theater Nuclear Forces of the Committee On Armed Services, United States Senate, S. Hrg. 99-933, p. 54.

[4] Eastport Study Group, "A Report to the Director, Strategic Defense Initiative Organization," 1985.

The Role of Software in BMD

Software for BMD would be expected to:

- be the agent of system evolution, permitting changes in system operation through reprogramming of existing computers;
- perform the most complex tasks in the system, such as battle management;
- be responsible for recovery from failures, whether they are hardware or software failures; and
- respond to threats, both anticipated and unanticipated, against the system.

A BMD system would not be trustworthy and reliable unless both hardware and software were trustworthy and reliable. Because of rapid progress in hardware technology in recent years, and because of differences in their natures, hardware reliability is not as hotly-debated an issue as software reliability. As Brooks puts it in his discussion of current software engineering technology:

> Not only are there no silver bullets now in view, the very nature of software makes it unlikely that there will be any-no inventions that will do for software productivity, reliability, and simplicity what electronics, transistors, and huge-scale integration did for computer hardware. We cannot expect ever to see twofold gains every two years.
>
> First, one must observe that the anomaly is not that software progress is so slow, but that computer hardware progress is so fast. No other technology since civilization began has seen six orders of magnitude in performance-price gain in 30 years.[5]

Software Complexity

The software engineer called upon to produce large, complex software systems is partly a victim of his medium. Software is inherently flexible. There are no obvious physical constraints on its design (e.g., power, weight, or number of parts) so software engineers undertake tasks of complexity that no hardware engineer would. Brooks summarizes the situation as follows:

> Software entities are more complex for their size than perhaps any other human construct because no two parts are alike . . . In this respect, software systems differ profoundly from computers, buildings, or automobiles, where repeated elements abound.
>
> Digital computers are themselves more complex than most things people build: They have very large numbers of states. This makes conceiving, describing, and testing them hard. Software systems have orders-of-magnitude more states than computers doe

Software Issues

Of course, complex systems are successfully built and used. However, given the current state of the art in software engineering, complex systems are not trusted to be reasonably free of catastrophic failures *before* a period of extensive use. During that period, errors causing such failures may be found and corrected. A central issue in the debate over BMD software is whether it can be produced so that it can be trusted to work properly the first time it is used, despite the probable presence of errors that might cause catastrophic failures. A critical point in the debate over this issue is how one would judge whether or not the software was trustworthy. If evaluations of trustworthiness were to rely on the results of simulations of battles, then a second critical point is how closely and accurately actual BMD battles could be simulated.

A second central issue in the software debate is whether a BMD system imposes unique requirements on software. Critical points surrounding this issue are:

- whether there are existing similar systems that could serve as models for the development of BMD software;
- whether requirements would be sufficiently well understood in advance of use so that trustworthy software could be designed;

[4] Frederick P. Brooks, Jr., "No Silver Bullet, Essence and Accidents of Software Engineering," ***IEEE Computer*** vol. 20, No. 4, April 1987, p. 10.

[6] Ibid.

- whether all potential threats against a BMD system could be anticipated, and, if not; and
- whether the software could be designed to handle unanticipated threats during the course of a battle.

Adding fuel to the debate over whether software could meet BMD requirements is the slow progress in software technology in recent years when compared to hardware technology.

An obstacle to settling this issue is the current uncertainty over the purposes of a BMD system. Software requirements would depend on the threat and countermeasures to be faced, the expected strategies of both the offense and the defense; and the technology to be used in the system, e.g., kinetic-energy v. directed-energy weapons. A system intended to defend the population would have different requirements than one intended to defend only critical military targets. A system to be deployed in phases would oblige the software developers to know the changes in requirements and architecture to be expected between each phase before they designed the software for the initial phase.

Among the developers of large, complex systems who attended OTA's workshop on SDI software, there was unanimous agreement that software development should not be started until there was a clear statement of the requirements of the system.[7] All system requirements would not have to be known in detail before software development could be started. But if the requirements for a system component could not be written, neither could the specifications for the software that was part of that component.

Catastrophic Failure

Both critics and supporters of the feasibility of building software to meet SDI requirements agree that large, complex software systems, such as an SDI BMD system would need, would contain errors. They disagree on whether the software could be produced so that it would not fail catastrophically. Several different meanings of catastrophic failure have been used. It is sometimes related to whether or not a BMD system would deter the Soviets from launching ICBMs at the United States:

> Ballistic missile defense must . . . be credible enough in its projected wartime performance during peacetime operations and testing to ensure that it would never be attacked.[8]

It can also be taken to mean that

> The system has failed catastrophically if the U.S. bases its defense on the assumption that the system will function effectively in battle and then a major flaw is discovered so that we are defenseless.[9]

This chapter assumes a technical definition: a catastrophic failure is a decline in system performance to 10 percent or less of expected performance. A BMD system designed to destroy 10,000 warheads would be considered to have failed catastrophically if it stopped only 1,000 of the 10,000. The figure 10 percent is an arbitrary one; it has been adopted as illustrative of a worst-case failure.

Generic Software Issues

Much of the debate concerning BMD software is about software problems common to all complex, critical software systems.[10] A good example is whether software can be designed to recover from failures automatically. BMD proponents argue that producing trustworthy BMD software would not call for general solutions to such problems. They feel that the specificity of the application permits special-case solutions that would work well enough for BMD. Opponents argue that BMD software would demand better solutions for such problems as failure-recovery than any system

[7] Attendees at the workshop, held Jan. 8, 1987 in Washington, DC, included software developers who participated in the development of SAFEGUARD, Site Defense, telephone switching systems, digital communications networks, Ada compilers, and operating systems.

[8] Charles A. Zraket, "Uncertainties in Building a Strategic Defense," *Science* 235:1600-1606, March 1987.

[9] David L. Parnas, personal communication, 1987.

[10] As described, for example, in David L. Parnas, "Software Aspects of Strategic Defense Systems," op. cit., footnote 2.

previously built. They say that approaches proposed for SDI have been tried in the past and have not been shown to be effective.

This chapter is primarily concerned with arguments over the generic issues. First, there are as yet no clear statements of BMD software requirements, whether for battle management or particular BMD system components, let alone proposed software designs or proposed solutions for BMD for any of the generic problems. Application-specific analysis must await those requirements, designs, and solutions.

Second, there seems to be agreement that BMD software would be more complex than any previously built. The first conclusion of volume V of the Fletcher report was:

> Specifying, generating, testing, and maintaining the software for a battle management system will be a task that far exceeds in complexity and difficulty any that has yet been accomplished in the production of civil or military software systems.[11]

Third, tasks for BMD software differ in important ways from the tasks performed in today's weapons systems and command, control, and communications systems. It is true that many BMD software tasks would resemble those for current systems: e.g., target tracking, weapons release and guidance, situation assessment, and communications control in real time. The differences from current systems are that a BMD system would:

- permit less opportunity for human intervention,
- have to handle more objects in its battle space,
- have to manage a larger battle space,
- use different weapons and sensor technology,
- contain vastly more elements,
- have more serious consequences of failure,
- have to operate in a nuclear environment,
- be under active attack by the enemy, and
- be useless if it failed catastrophically during its first battle.

Accordingly, the debate over generic software issues is an appropriate one for BMD software.

The purpose of this chapter is to examine the key issues in the debate over the feasibility of meeting BMD software requirements. This chapter:

1. discusses why there is such a debate and includes a definition of key terms, such as "catastrophic failure" and "trustworthiness";
2. analyzes properties often claimed to be important for BMD software-e. g., trustworthiness, reliability, correctness, low error incidence, fault tolerance, security, and safety, (including a discussion of the meaning of "reliability" as applied to software and why there is no single, simple measure of software dependability);
3. identifies the major factors that affect software dependability; and
4. characterize the demands placed on BMD software and the BMD software development process in terms of the factors affecting dependability.

The remainder of this chapter begins with a brief discussion of Department of Defense (DoD) software experience, the nature of software, traditional reliability measures, and the pitfalls inherent in applying such measures to software. Following sections deal with properties such as trustworthiness, correctness, fault tolerance, security, and safety, and with the factors that lead people to have confidence that systems have such properties. (The available technology for incorporating these properties into software is analyzed in app. A.) The chapter then presents an analysis of Strategic Defense Initiative BMD requirements from the viewpoint of those factors. The chapter concludes with: a discussion of why BMD software development is a difficult job—perhaps uniquely so; why we are unlikely to have more than a subjective judgment of how trustwor-

[11] James C. Fletcher, Study Chairman and Brockway McMillan, Panel Chairman, *Report of the Study on Eliminating the Threat Posed by Nuclear Ballistic Missiles, Volume V: Battle Management, Communications, and Data Processing* (Washington, DC: Department of Defense, Defensive Technologies Study Team, October 1983).

thy the software is, once produced: and a summary of the key software issues.

The Software Crisis

Since the mid-1970s DoD officials have increasingly recognized the difficulties in producing command, control, and information processing software for weapon systems.[12] As Ronald Enfield says:

> In the 1970s, the world's largest customer for computers–the U.S. Department of Defense—changed its focus from hardware to software as a major obstacle to progress in developing advanced weapons. Reliable software is also a crucial component of complex systems such as nuclear power plants, automatic tellers, and many other technologies that touch our lives in critical ways. Yet, as the software for these systems has grown increasingly complicated, it has become more prone to error.[13]

The complex of problems associated with trying to produce software that operated properly, on time, within budget, and maintainably over its lifetime was dubbed "the software crisis. " DoD has found that the software crisis is sometimes forcing the military to wait for software to be debugged before it can use new systems. Progress in alleviating this crisis has been slow, and the same problems would apply to producing software for BMD. Both the Fletcher and Eastport Group reports agreed that software development for BMD would be a difficult, if not the most difficult, problem in BMD development. The Eastport Group noted that:

> Software technology is developing against inflexible limits in the complexity and reliability that can be achieved.[14]

To understand why DoD and other developers of large, complex software systems have been experiencing a software crisis, it is first necessary to understand the nature of software and the demands made on it.

The Nature of Software

Digital computers are among our most flexible tools because the tasks they do can be changed by changing the sequences of instructions that direct them. Such instruction sequences are called programs, or *software* and are stored in the computer's memory. Flexibility is attained by loading different programs into the memory at different times.[15] Each make and model of computer has a unique set of instructions in which it must be programmed, generically known as *machine instructions* or *machine language.*

To simplify their job, programmers have developed languages that are easier to use than machine language. These languages, such as FORTRAN, COBOL, and Ada, are known as high *level languages,* and require the programmer to know less about how a particular computer works than do machine languages. The language in which a program is written is known as the *source language* for the program, and the text of the program is called the *source program* or *source code.*[16] A program whose source language is a high level language must be translated into machine language before being loaded into the computer's memory for execution. Some lines of text in a source program may be translated into many machine instructions, some into just a few.

There are several measures of program size. One measure is the number of lines in the text of the source program, also known as *lines of source code* (LOC), or number of machine language instructions. Size is greatly variable: a simple program to add a list of numbers may require 10 or fewer instructions, while a word

[12] An early analysis of the problem can be found in Donald W. Kosy, "Air Force Command and Control Information Processing Requirements in the 1980s: Trends in Software Technology, " Rand Report R-1012-PR, June 1974.

[13] Ronald L. Enfield, "The Limits of Software Reliability, " *Technology Review*, April 1987.

[14] Eastport Study Group Report, op. cit., footnote 4.

[15] To protect them from change, and to enhance their performance, some programs are loaded into memories that are either unchangeable or that must be removed from the computer to be changed. However, most of the memory in nearly all computer systems is of a type that is reloadable while the computer is running.

[16] Instructions and data are encoded into a computer's memory as numbers, and programs are sometimes known as codes.

processing program may take 10,000 LOC (10 KLOC). The Navy's AEGIS ship combat software consists of approximately 2 million instructions.

Size alone is not a good measure of program difficulty. Large programs can be simple, small ones very complex. The size of a program is influenced by the language, computer, programmer's expertise, and other factors. A more important question is, "How complex is the problem to be solved by the program and the algorithms used to solve it?"[17] Compounding the problem is the lack of a standard method for measuring complexity.

Failures and Errors in Computer Programs

Since a computer can only execute the instructions that are stored in its memory, those instructions must be adequate for all situations that may arise during their execution.[18] Incorrect performance by a computer program during its operation is known as a failure. Failures in computer programs result from:

- the occurrence of situations unforeseen by the computer programmer(s) who wrote the instructions,
- a misunderstanding by the programmer(s) of the problem to be solved (including misunderstandings among a group of programmers), or
- a mistake in expressing the solution to the problem as a computer program.

Each of these situations can cause errors in the instructions making up computer programs, errors manifested as failures when particular inputs occur.[19] The effects of errors in programs range from minor inconveniences (e.g., misspelled words in the program's output) to catastrophic failures-e. g., the cessation of all processing by the computer, wrong answers to problems like computing missile tracks, or overdoses of radiation to devices controlled by the computer.[20] [21]

Tolerating Errors

Errors in large computer programs are the rule rather than the exception. Freedom from errors cannot be guaranteed and is extremely rare. Since correcting an error requires changing the list of instructions that make up the program, the process of removing an error may, and often does, introduce anew error. For large software, the process of correcting errors is so time consuming and expensive that modifications to the software are distributed only a few times a year. As a result, lists of known errors are often published and distributed to users.[22] Where there is a high degree of human interaction with the program during its operation, the human user can usually circumvent situations where the program is known to fail-often by restricting the data input to the program or by not using features of the program known to be failure-prone.

The more critical the task(s) of the program and the smaller the degree of human intervention in the program's operation, the smaller the tolerance for errors. Accordingly, large, critical programs commonly include consistency checks whose goal is to try to detect failures, prevent them when possible, and recover from them when not. This approach is

[17] See chapter 8 for a discussion of algorithms.

[18] Some programs, known as self-modifying programs, add to or modify their own instruction sequences and then execute the resulting instructions. Nonetheless, the response of the program to input data is completely determined by the instructions that are initially stored in its memory.

[19] Errors in programs are often called bugs, although the term originally meant any cause of incorrect behavior. The origin of the term is described in John Shore, The *Sachertorte Algorithm*, (New York, NY: Penguin Books, 1986).

[20] For a sample of the variety of problems involving computers and software, see *ACM SIGSOFTSoftware Engineering Notes* 11(5):3-35, October 1986.

[21] The occurrence of a failure condition is sometimes known as an *incident*. The software may contain instructions that permit it to recover from such an incident. If the software successfully corrects the condition, it remains no more than an incident. Successful recovery from incidents requires good understanding of their causes and corrections, and requires that not too many occur at once.

[22] Manual pages describing programs used with the UNIX operating system, developed and sold by AT&T Bell Laboratories and a currently popular operating system, contain as standard sections a description of the known bugs in the programs.

discussed in more detail in a later section on fault tolerance.

Tolerating Change

As previously noted, change is both the blessing and the curse of the software engineer. Software is expected to be flexible, and his designs must accommodate change. Without its flexibility, software would be as useful. Although software does not wear out in the sense that hardware does, complex software systems apparently tolerate only a certain amount of change. The critical point occurs when changes introduce more errors than they fix, i.e., each change, on the average, introduces more errors than it removes.[23] It appears likely that increasing the rate of change decreases the time to reach the critical point. Brooks devotes a chapter to a discussion of the effects of changes in complex systems, concluding with:

> Program maintenance is an entropy-increasing process, and even its most skillful execution only delays the subsidence of the system into unfixable obsolescence.[24]

Although Brooks's discussion is more than 10 years old, it is still valid. Systems that tend to be very long-lived, e.g., 20 years old or more, undergo complete software redevelopment every few years. As an example, the Navy's Naval Tactical Data System, first built in the early 1960s, has undergone at least five major rewrites.

Traditional Reliability Measures

Reliability is one measure of system behavior. In engineering, reliability is often expressed as the average time between failures. For inexpensive consumer items, such as light bulbs, it is defined as the expected lifetime of the item, since such items are completely replaced when they fail. Complicated, expensive systems, such as automobiles, computer systems, and weapon systems, are designed to outlive any particular component by allowing repair or replacement of components when they fail. Failure of a windshield wiper blade only requires the quick, inexpensive replacement of the blade by another that meets the same specifications as the failed one.

Reliability of complicated systems is traditionally measured in mean time between failure (MTBF), or an equivalent measure such as failure rate. MTBF is measured by counting failures during operation and then dividing by the length of the observation period. For systems with no operational history, MTBF must be predicted on the basis of estimates of the MTBF of each of the system's components. Usually such an estimate is made using the assumption that component failures are random, statistically independent events. Without such an assumption, the analysis is much more difficult and often impractical for complex systems.

Reliability as measured by MTBF is useful for systems with the following characteristics:

- the time to repair the system is unimportant to the user, perhaps because a temporary replacement is available or the user has no need of the system for a while; or
- the time to repair the system is important, but can be kept very short compared to the MTBF, perhaps by keeping a stock of replacement parts on hand; and
- there are no failures so serious as to be unacceptable, e.g., failures that could result in human deaths.

Traditional Reliability Measures Applied to Software

The concept of MTBF has historically been of limited use for critical software. For applications such as BMD, repair time is extremely important. If the system, or parts of it, were to fail, the user would have either no response or a weakened response to an ICBM attack. Accordingly, the concept of MTBF alone is not sufficient to judge whether or not the system would behave as desired. Furthermore, the

[23] M. Lehman and L. Belady, "Programming System Dynamics," *ACM SIGOPS Third Symposium on Operating System Principles,* October 1971.

[24] Frederick P. Brooks, Jr., *The Mythical Man-Month: Essays on Software Engineering, (New* York, NY: Addison-Wesley, 1975).

models often used for predicting MTBF are based on assumptions that are invalid for software. Many models assume that component failures are independent and that they are random, i.e., unrelated to system inputs and states. Software components do not fail randomly: they contain errors that cause failures in the event of particular inputs and particular states. The failure of one component often causes others to fail because software components tend to be closely interrelated.

Replacing a software component by a copy of itself will cause exactly the same failure under the same conditions that caused the original to fail. Remedying a failure consists of modifying a component to remove an error in its list of instructions, not replacing a failed component with a copy. Once modified, the component can no longer be considered to be the same as the original, and previous failure data do not apply to it. Finally, a failure in one component is likely to lead to failures in others. Consequently, a stock of replacement components cannot be kept on hand in hopes of reducing repair time.

Regardless of whether MTBF were used to indicate software or hardware reliability for a BMD system, some failures would be clearly more disastrous than others. To be useful, MTBF would have to be calculated for different classes of failures.

In recent work, researchers have shown that if inputs are characterized in statistically sound ways, it is possible in testing to determine with high confidence a meaningful MTBF for a program.[25] Nonetheless, MTBF remains inadequate as the sole means of characterizing software dependability.

[25] Allen Currit, Michael Dyer, and Harlan D. Mills, "Certifying the Reliability of Software," *IEEE Transactions On Software Engineering, SE-12(1),* January 1986, pp. 3-11.

SOFTWARE DEPENDABILITY

Computer scientists and software users have devised a variety of ways to evaluate software dependability. As in deciding which automobile to buy, the buyer's concerns should determine which qualities are emphasized in the evaluation. Qualities commonly considered are:

- correctness-whether or not the software satisfies its specification;
- trustworthiness-probability that there are no errors in the software that will cause the system to fail catastrophically;
- *fault* tolerance-either failure prevention, i.e., capability of the software to prevent a failure despite the occurrence of an abnormal or undesired event—or failure recovery, i.e., capability of the software to recover from a failure when one occurs;
- availability-probability that the system will be available for use;
- security-resistance of the software to unauthorized use, theft of data, and modification of programs;
- *error* incidence—number of errors in the software, normalized to some measure of size; and
- safety—preservation of human life and property under specified operating conditions.

For critical software, correctness and trustworthiness are important indicators of dependability. Fault tolerance assumes importance when the system must continue to perform—as in the midst of a battle—even if performance degrades. Security is important when valuable data or services maybe stolen, damaged, or used in unauthorized ways. Safety is important in applications involving risk to human life or property. Error incidence is important in assessing whether or not apiece of software should stay in use.

OTA's characterization of BMD software debendability will include all of the above-listed qualities because:

- national survival may depend on the proper operation of BMD software;

- such software would have to be trusted to operate well during the entire course of a battle; and
- it would have a long lifetime.

Early versions of a BMD system may not have goals as ambitious as later, more capable versions. Nevertheless, we still would want to be confident that the software would operate well during the course of a battle, would do so without undue pause for failure recovery, would be secure, and would be safe to operate. In addition, since it would surely undergo continual modification during its lifetime, we would need to be sure that it was being maintained without repeated introduction of new errors.

Dependability needs to be attended to from the beginning of software development, for it is not easily added on later. Software designs often must be redone after system delivery when performance has been emphasized at the cost of such factors as correctness, fault tolerance, or security. The cost of redoing software may greatly exceed the original cost. Software designed for dependability may contain mechanisms for later improving its correctness, trustworthiness, fault tolerance, security, and safety later. For example, fault tolerance was strongly considered in the design of the SAFEGUARD software. During tests of the prototype system engineers realized that the wrong set of faults had been accommodated. Because the mechanism for detecting and responding to faults had been incorporated into the design, the set of faults tolerated by the system was changed in a matter of only a few weeks. This change involved perhaps 10 percent of the lines of code in the operational software.[26]

Figures of Merit

No single figure of merit can indicate dependability. Single figures of merit generally focus on some single characteristic, such as the cost to discover a password that would permit entry to a computer system. Because software engineering is a young discipline, software engineers do not yet know very well how to evaluate software quantitatively. And because information permitting numerical evaluation of software is usually considered proprietary, few data are available anyway for such analysis. Accordingly, we would not expect a useful quantitative evaluation of BMD software dependability to be available for many years. Therefore, only a brief analysis of each software property contributing to dependability follows.

Trustworthiness is probably the most important quality for BMD software. The application is critical. Software engineers are unable to produce complex software that is correct and error-free at the current state of the art. Although BMD software should still be as nearly correct, highly available, error-free, secure, and safe to use as possible, we must above all know whether or not it could be trusted.

Correctness

Software developers work from specifications, both written and verbal, that are intended to convey the desired system behavior. The specifications are frequently developed by people with little familiarity with software, e.g., a Naval officer untrained in software development who writes specifications for a ship's combat management system. "Correct" software exhibits exactly the behavior described by its specifications. To convince himself and his customer that he has done his job, the software developer must somehow demonstrate that his software is correct.

Mathematical Correctness

Because no single technique has proved completely effective to demonstrate program correctness, software developers use a variety of techniques try to demonstrate that their software adequately approximates its specifications. Computer scientists, in recognition of the problems involved, have devoted considerable research to such techniques. They have investigated formal and informal, mathematical and non-mathematical ideas. Much of the

[26] Victor Vyssotsky, personal communication, 1987.

research attention has been focused on developing "program verification' '-mathematical techniques to verify that a computer program is correct with respect to properties required of it. Some progress has been made in mathematically proving that programs are correct. It is unlikely, though, that a sudden breakthrough will occur leading to order-of-magnitude gains in productivity and greatly improved dependability. Brooks analyzed this possibility:

> Can both productivity and product reliability be radically enhanced by following the profoundly different strategy of proving designs correct before the immense effort is poured into implementing and testing them?
>
> I do not believe we will find productivity magic here. Program verification is a very powerful concept, and it will be very important for such things as secure operating system kernels. The technology does not promise, however, to save labor. Verifications are so much work that only a few substantial programs have ever been verified.
>
> Program verification does not mean error-proof programs. There is no magic here, either. Mathematical proofs can also be faulty. So whereas verification might reduce the program-testing load, it cannot eliminate it.
>
> More seriously, even perfect program verification can only establish that a program meets its specification. The hardest part of the software task is arriving at a complete and consistent specification, and much of the essence of building a program is in fact the debugging of the specification."

Although mathematical techniques for demonstrating correctness are not frequently applied, other techniques-such as design reviews, code reviews, and building software in small increments—are. The one technique always used by software developers, however, is testing.

Testing

Program developers test a program by placing it in a simulated operating environment.[28] The simulation supplies inputs to the program, and the testers examine its output for failures.[29] They report any failures to the programmers, who correct the relevant errors and resubmit the program for testing. The sequence continues until the developers agree that the program has passed the test. The final stage of testing developmental software for large and critical systems, especially military software, is acceptance testing. A previously agreed-upon test is run to show that the software meets criteria that make it acceptable to the user.

It has been shown that testing of every possible state of the program, known as exhaustive testing, is not practical even for simple programs. To illustrate this point, John Shore calculated the amount of time required to test the addition program used by 8 digit calculators to add 2 numbers. He estimated that, at the rate of one trial per second it, would take about 1.3 billion years to complete an exhaustive test.[30]

For large, complicated programs, the number of tests that can be run practically is small compared to the number of possible tests. Therefore, developers apply a technique called scenario testing. They observe the program's behavior in an operational scenario that the program would typically encounter. They may establish the scenario by simulating the operational environment, such as an aircraft flight simulator. Alternatively, they may place the software in its actual environment under controlled conditions. For example, a test pilot may put an aircraft with new avionics software through a series of pre-determined neuvers. In the former case, the simulator must first be shown to be correct before the results can be considered valid. If the simulator itself relies on software, showing the validity of the simulation may be as difficult or more difficult than showing the correctness of the program to be tested.

---

[27] Frederick P. Brooks, Jr., "NO Silver Bullet, Essence and Accidents of Software Engineering," op. cit., footnote 5, p. 16.

[28] For presentation purposes, the discussion of testing here is simplified, omitting, e.g., component testing. Appendix A contains a more complete discussion.

[29] Good testers carefully determine the inputs to be used in advance, often including some tests using random inputs, and some using nonrandom, so as to get representative coverage of the expected operational inputs.

[30] John Shore, op. cit., footnote 19, pp. 171-172.

For systems like aircraft, such tests are so expensive that only relatively few scenarios can be flown. Flight tests of the avionics software for the Navy's A-7 aircraft, including land- and carrier-based tests, cost approximately $300,000. Scenario tests for the SAFEGUARD system consisted of installing a test version of the system at Kwajalein missile range and firing one or two missiles at a time at it.

Since exhaustive testing is not practical, testing cannot be relied upon to show that a computer program completely and exactly behaves according to its specifications or even that it contains no errors. As stated by computer scientist Edsger Dijkstra:

> Program testing can be used to show the presence of bugs, but never to show their absence![31]

The deficiencies of testing as a means of showing correctness and freedom from errors have moved software engineers to seek other methods, such as mathematical. They have also sought means of measuring error incidence. In addition, they are developing methods for random testing that permit statistical inferences about failure rates.[32]

Error Incidence

Some assert that error incidence-measured, for example, by the number of errors found per thousand lines of source code-measure program correctness. Those making this assertion assume that it is possible to count errors unambiguously and that the more errors a program has the less its behavior will conform to its specifications. They then portray the debate over BMD software dependability as hinging on the question of whether or not the software would contain errors, and how many it would contain.

Both critics and proponents of an attempt to build SDI software agree that any such software would contain errors. As put by the Eastport Group:

> Simply because of its inevitable large size, the software capable of performing the battle management task for strategic defense will contain errors. All systems of useful complexity contain software errors.[33]

Ware Myers notes:

> The whole history of large, complex software development indicates that errors cannot be completely eliminated.[34]

David Parnas asserts that:

> Error statistics make excellent diversions but they do not matter. A low error rate does not mean that the system will be effective. All that does matter is whether software works acceptably when first used by the customers; the sad answer is that, even in cases much simpler than SDI, it does not. What also matters is whether we can find all the "serious' errors before we put the software into use. The sad answer is that we cannot. What matters, too, is whether we could ever be confident that we had found the last serious error. Again, the sad answer is that we cannot. Software systems become trustworthy after real use, not before.[35]

Trustworthiness

Since correctness and error rates are not the real issues in the software debate, trustworthiness has become the focus. The issue is whether or not BMD software could be produced so that it would be trustworthy despite the presence of errors. In common usage, reliability and trustworthiness are often considered to be the same. In engineering usage, reliability has become associated with specific measures, such as MTBF. There have been few attempts to quantify trustworthiness, despite

[31] J. Dahl, E.W. Dijkstra, and C.A.R. Hoare, "Notes on Structured Programmingg," *structured Programming* (London: Academic Press, 1972), p. 6.

[32] See the discussion on the cleanroom method in appendix A for more details.

[33] Eastport Study Group Report, *Op.* cit., footnote 4.

[34] Ware Myers, "Can Software for the Strategic Defense Initiative Ever Be Error-Free?" *IEEE Computer* XX:61-67, November 1986.

[35] David L. Parnas, "SDI Red Herrings Miss the Boat, " (Letter to the Editor), *IEEE Computer* 20(2):6-7, February 1987.

the desirability of trustworthy systems. One possible reason may be that trust is determined qualitatively as much as quantitatively: people judge by past experience and knowledge of internal mechanisms as much as by numbers representing reliability. Another possible reason is that most systems in critical applications are safeguarded by human operators. Although the systems are trusted, the ultimate trust resides in the human operator. Nuclear power stations, subway systems, and autopilots are all examples.[36]

As noted in chapter 7, a BMD system would leave little time for human intervention: trust would have to be placed in the system, not in the human operator. Accordingly, it is important to be able to evaluate the trustworthiness of BMD software. One suggested definition is that trustworthiness is the confidence one has that the probability of a catastrophic flaw is acceptably low.[37] Trustworthiness might be described by a sentence such as "The probability of an unacceptable flaw remaining after testing is less than 1 in 1,000."[38] (This measure of trustworthiness has only recently been suggested, and no data have yet been published to support it.) Estimating trustworthiness consists of testing the software in a randomly selected subset of the set of internal states with a randomly selected subset of the possible inputs. The set of possible inputs and internal states must be known. It must be possible to recognize a catastrophic test result, i.e., the expected operating conditions must be well-understood. For BMD systems, this means understanding the expected threat and countermeasures as well as testing under conditions closely simulating a nuclear environment.

[36]Even when human operators are aware of a problem they sometimes do not or cannot react quickly enough, or with the proper procedures, to prevent disaster.

[37]David L. Parnas, "When Can We Trust Software Systems?" (Keynote Address), *Computer Assurance, Software Systems Integrity: Software Safety and Process Security Conference,* July 1986.

[38]David L. Parnas, personal Communication, 1987.

Fault Tolerance

Realizing that errors in the code and unforeseen and undesired situations are inevitable, software developers try to find ways of coping with the resulting failures. Software is considered fault-tolerant if it can either prevent or recover from such failures, whether they are derived from hardware or software errors or from unanticipated input. Techniques for fault tolerance include:

- back-up algorithms,
- voting by three or more different implementations of the same algorithm,
- error-recovery programs, and
- back-up hardware.

Program verification techniques, discussed above, attempt to prove correctness by mathematical analysis of the code. In contrast, fault-tolerant techniques attempt to cope with failures by analyzing how a program behaves during execution.

Since a BMD system would have to operate under widely varying conditions for many years, its software would have to incorporate a high degree of fault tolerance. Unfortunately, there are no accepted measures of fault tolerance, and design of fault-tolerant systems is not well understood. As an example, space shuttle flight software is designed in a way thought to be highly fault-tolerant. Four identical computers, executing identical software, vote on critical flight computations A fifth, executing a different flight program, operates in parallel, providing a backup if the other four fail. On an early attempted shuttle launch, this flight system failed because the backup program could not synchronize itself with the four primary programs. The failure, occurring just 20 minutes before the scheduled lift-off, caused the flight to be postponed for a day. It was a direct result of the attempt to make the software fault-tolerant.

The price for fault tolerance is generally paid in performance and complexity. A program incorporating considerable code for the purpose of detecting, preventing, and recovering from failures will be larger and operate more slowly

than one that does not. A successful fault-tolerant design will result in a system with higher availability than a corresponding system built without regard for fault tolerance. Producers of non-critical software may not care to pay the price. Those concerned with critical systems that must operate continuously often feel that they must.

Availability

Systems that are intended to maintain continuous operation are often evaluated by calculating their availability, i.e., the percent of time that they are available for use. Availability is easily measured by observing, for some interval, the amount of time the system is unavailable (the "down" time) and available (the "up" time) and then calculating (up time)/(up time + down time). As with other figures of merit, availability figures are useful when the conditions under which they were measured are well-known. Extrapolation outside of those conditions is risky. Since prediction of availability is equivalent to prediction of MTBF and mean time to repair (MTTR)-measures of up- and down-times, respectively—availability is at least as difficult to predict as MTBF and MTTR are individually.

Security

Computer users concerned with preserving the confidentiality of data and the effectiveness of weapon systems, such as banks or the military, consider security a necessary condition for dependability. Breaches of security that concern such users include:

- knowledge by an opponent of the algorithms implemented in a computer controlling a weapon system, allowing him to devise ways of circumventing the strategy and tactics embodied in those algorithms;
- access by unauthorized users to sensitive or classified data stored in a computer;
- denial of access by authorized users to their computers, thereby denying them the capabilities of the computer and the data stored in it; and
- substitution of an opponent software for operational software (changing even a few instructions may be potentially disastrous), allowing the opponent to divert the computer to his own uses.

Many of the preceding concerns only apply if a computer must use a potentially corruptible communications channel to receive data or instructions from another computer or from a human.[39] Any BMD system would contain such links. (The possibility that a link could be corrupted and measures for preventing such corruption are discussed in ch. 8.) Over these channels one might:

- load revised programs into the memories of the BMD computers;
- correct errors in existing programs;
- change the strategy incorporated into existing programs; or
- accommodate changes to software requirements, such as might be caused by the introduction of new technology into the BMD system.

In addition, any BMD architecture would contain communications channels for the exchange of data between battle management computers and sensors and among battle management computers.

Since the 1960s, when computers started to be used on a large scale in weapon systems, the DoD has expended considerable effort to find ways of making computer systems secure. As yet, no way has been found to meet all the security requirements for computers used in the design, development, and operation of weapon systems. As Landwehr points out in a discussion of the state of the art in developing secure software:

> At present, no technology can assure both adequate and trustworthy system performance in advance. Those techniques that have been tried have met with varying degrees of success, but it is difficult to measure their success objectively, because no good measures ex-

[39] Physical security, i.e., control of physical access to computing equipment, is a problem as well, generally unsolvable by technical means and outside the scope of this report.

ist for ranking the security of various systems.[40]

Although there is no quantifiable measure of the security of a computer system, the DoD has developed a standard for evaluating the security of computer systems.[41] The evaluation consists of matching the features provided by a system against those known to be necessary, albeit not sufficient, to provide security. For example, the second highest rating is given to those systems that let users label their data according to its security level, e.g., Confidential or Secret, then protect the labels against unauthorized modification. Furthermore, the developer must show the security model used in enforcing the protection and show that the system includes a program that checks every data reference to ensure that it follows the model. As with fault tolerance, incorporating security features into software exacts penalties in performance and complexity.

Safety

A software engineering journal distinguished between safety and reliability:

> Safety and reliability are often equated, especially with respect to software, but there is a growing trend to separate the two concepts. Reliability is usually defined as the probability that a system will perform its intended function for a specified period of time under a set of specified environmental conditions. Safety is the probability that conditions which can lead to an accident (hazards) do not occur whether the intended function is performed or not. Another way of saying this is that software safety involves ensuring that the software will execute within a system context without resulting in unacceptable risk.[42]

Interest in software safety has increased markedly in recent years. Formal publications specifically addressing software safety issues started appearing in the early 1980s.[43] As yet, there are no standard measures or ways of assessing software safety. Nonetheless, it is important that BMD software be safe so as to prevent accidents that are life threatening and costly. An unsafe BMD system might, for example, accidentally destroy a satellite, space station, or shuttle.

Appropriate Measures of Software Dependability

As should be clear from the preceding discussion, software dependability cannot be captured in any single measure. Correctness, trustworthiness, safety, security and fault tolerance are all components of dependability. All should be considered in the development of software for a BMD system. Attempts to quantify them in a clear-cut way require specifying too many conditions on the measure to allow useful generalization. Estimates of the dependability of BMD software would always be suspect, since in large part they would always be subjective. Until we can quantify software dependability we cannot know that we have developed dependable BMD software. The following sections discuss the factors involved in developing dependable software.

[40] Carl Landwehr, "The Best Available Technologies For Computer Security," *IEEE Computer,* July 1983, p. 93.

[41] "DoD Standard 5200.28, Department of Defense Trusted Computer System Evaluation Criteria," (Washington, DC: Department of Defense, Aug. 15, 1983).

[42] *"IEEE Transactions on Software Engineering: Special Issue On Reliability And Safety In Real-Time Process Control,* SE-12(9):877, September 1986.

[43] Nancy Leveson and Peter Harvey, "Analyzing Software Safety," *IEEE Transactions on Software Engineering, SE-9(?),* September 1983, pp. 569-579, was one of the first papers to discuss software safety.

CHARACTERISTICS OF DEPENDABLE SYSTEMS

Despite the lack of ways of quantifying confidence in software, people trust many computerized systems. Further, people are willing to undertake development of many systems with confidence that they will be reliable when finished. In this section we discuss why systerns come to be trusted and give some exampies of trusted systems. We divide methods

of gaining trust into two classes: those based on observations of the external behavior of the system, and those based on understanding how the system operates internally.

Observations of External Behavior

A system, whether containing software or not, may be considered to be a black box with connections to the outside world. One may observe the inputs that are sent to the box and the outputs it produces. The next few sections discuss methods of gaining confidence in software and systems based on black-box observations of the software.

Extensive Use and Abuse

Perhaps the most important factor inspiring confidence in software is that the software has been used extensively. A good analogy is the automobile. Confidence comes from familiarity with cars in general and frequent use of one's own car. Having seen that the engine will start when the key is turned hundreds of times gives one the feeling that it will start the next time the key is turned. Automatic teller machines, electronic calculators, word processors, and AT&T's long distance telephone network are all examples of systems controlled by software that are trusted to work properly. The trust is built on extensive experience: one has high confidence that the telephone will work the next time it is tried because it usually has in the past.

Confidence is considerably enhanced when a system continues to work even though abused. A car that starts on cold and rainy days inspires increased confidence that it will start on mild and sunny days. Observing that calls still get through under heavy calling conditions (albeit not as quickly), that dialing a non-existent number produces a meaningful response, and that calls can still be made when major trunk circuits fail boosts one's confidence that nearly all one's calls will get through under normal conditions. Conversely, system failure detracts from confidence. Having observed that issuing a particular command to a word processor sometimes results in meaningless text being inserted into a document leads one to refrain from using that command.

It is important to note that extensive confidence comes from extensive use and not from testing that incompletely simulates use. No one would consider granting a license to a pilot who had spent extensive time in a flight simulator but had never actually flown an airplane. Simulated use inspires confidence to the degree that the simulation approximates operational conditions. Real-world complications are often either too expensive or too poorly understood to simulate. In testing systems, simulators are useful for convincing ourselves that the gain from putting the system into its operating environment is worth the attendant cost and risks. They allow the jump to actual use with some confidence that disaster will not result.

Predictable Environments

Confidence in software also comes from being able to predict the behavior of the software in its operational environment. If the environment itself is predictable, the job of designing and testing the software is considerably eased. For example, engineers can predict and mathematically analyze the number of telephone calls per hour that a particular switching center will receive at any time of day. The number and type of signals that will be received on the telephone lines (e.g., the 7 digits in a local telephone number) are known because their specifications form part of the requirements for the telephone system and are determined by the designers. The software and hardware may then be designed to cope with the telephone traffic and the signal types based on the specifications.

Engineers can observe the system in operation to verify predictions before new software is placed into operation. Finally, they can observe the behavior of new software in terms of number of calls handled, number of calls rerouted, and other parameters for different traffic loads. Observing that behavior matches predictions builds confidence in the operation of the system. Nonetheless, even when the developers have extensive experience with a well-

controlled environment, they sometimes make mistakes in prediction and do not discover those mistakes until the system goes into use.

Low Cost of a Failure

Although extensive use and environmental predictability both strongly influence the amount of confidence placed in a system, they are not sufficient to induce users to continue using a system after significant failures. Large, complicated software systems inevitably experience software failures. Therefore, users don't have confidence in the software unless the risk associated with a failure is smaller than the gain from using the software. A word processor that loses documents may go unused because the cost of re-creating the document is greater than the effort saved by using the word processor.

If, however, an easy method of recovering from such losses is available, perhaps by including a feature in the word processor that automatically saves back-up copies of documents, then the cost to the user of the failure becomes acceptably low: he can recover his document when it is lost. Similarly, the cost of recovering from a disconnected phone call is small to the dialer and to the telephone company. (Although a misdialed phone call is not really a system failure, the same principle applies: users can recover quickly and easily.) The ability to recover from a failure at low cost increases confidence in and willingness to use a system.

Systems With Stable Requirements

A desire for flexibility is a prime motive for using computer systems. The behavior of a computer can be radically altered by changing its software. Radical changes may be made to a computer program throughout its entire lifetime. Because there is no apparent physical structure involved, the impact of change may not be readily appreciated by those who demand it without having to implement it. No one would ask a bridge builder to change his design from a suspension to an arch-supported bridge after the bridge was half built without expecting to pay a high price. The equivalent is often demanded of software builders with the expectation of little or no penalty in schedule, cost, or dependability.

An example is the combat system software for the first of the Navy's DD 963 class of destroyers. During the development of the software, which cost less than 1 percent of the cost of building the ship, the customers imposed major changes on the software developer. The original requirements specified that the combat system need only provide passive electronic warfare functions. One year into development the buyers added a requirement for active electronic warfare. A year later they removed the requirement for active electronic warfare. On the ship's maiden voyage its commander issued a casualty report on the software: the ship could not perform its function because of deficiencies in the software. Although the major requirements changes were probably not the only reason for the deficiencies, they were certainly a prime contributing factor.

The B-lB bomber is another example of a system where deficiencies have resulted from too much change during development. According to a report on the B-l B bomber,

> Defense officials blame many of the program's problems on the decision to begin producing the aircraft at the same time that research and development efforts were under way, forcing engineers to experiment with some systems before they were completely developed.[44]

Conversely, a system whose requirements change little during the course of development is more likely to work properly. Developers have a chance of understanding the problem to be solved: they need not continually reanalyze the problem and revise their solution. Stability of requirements is particularly important for software because of the many decisions involved in software design. Each subdivision of a program into subprograms involves decisions about the functions to be performed by the subprograms and about the interfaces be-

[44]" New Weapon Suffers From Major Defects, " Washington Post, Jan. 7, 1987, p. Al.

tween them. Writing each subprogram further involves decisions on the algorithm to be used, the way data are to be represented, the order of the actions to be performed, and the instructions to be used to represent those actions.

Decisions made early in the process are more difficult to change than those made later in the process because later decisions are often dependent on earlier ones. Furthermore, the process of change is more expensive in later phases of a project because there are more specifications and other documentation. Using data from SAFEGUARD software and software projects at IBM, GTE, and TRW, one expert has shown that, as a result of the preceding factors, error correction costs (and costs to make other software changes) increase exponentially with time. In Boehm's words:

> These factors combine to make the error typically 100 times more expensive to correct in the maintenance phase than in the requirements phase.[45]

Clearly then, for systems where requirements change little during development, not only can one have increased confidence in the software, but one can also expect it to cost less. Among the developers of large, complex systems who attended OTA's workshop on SDI software, there was unanimous agreement that BMD software development could not begin until there were a clear statement of the requirements of the system.[46]

Systems Based on Well-Understood Predecessors

As with other human engineering projects, successful software systems are generally the result of slow, evolutionary change. Where radical changes are attempted, failure rates are high and confidence in performance is low. This rule can be seen in endeavors such as bridge building[47] as well as software design.[48]

With the example and experience of a previous solution to a problem, a software developer can have the confidence that a system to solve a small variation on the problem can be correctly produced. The structure of the previous solution and the associated algorithms may be applied again with small variations. A good example is the software used by NASA to compute the orientation of unmanned spacecraft. The orientation, also known as attitude, is computed by ground-based computers while the spacecraft is in operation. Attitude is determined from the readings of sensors on board the spacecraft. The sensor readings are telemetered to earth and supplied as input to an attitue determination program for the spacecraft. The algorithms for computing orientation are well known and have been used many times. The design of the attitude determination software that incorporates the algorithms is also dependable.

The design of an attitude determination program for a new spacecraft starts with the design of an earlier program and consists of modifying the design to take into account sensor and telemetry changes. Many of the subprograms from the earlier program are reused intact, some are modified, and some new subprograms are written. A typical attitude determination program of this type is 50,000 to 125,000 lines of code in size and takes about 18 months to produce. It must be produced before the launch of the associated satellite, and must work when needed so that the satellite may be maneuvered as necessary. The de-

---

[45] Barry W. Boehm, *Software Engineering Economics* (Englewood Cliffs, NJ: Prentice-Hall, 1981), figure 4-2, p. 40.

[46] attendees at the workshop included software developers who participated in the development of SAFEGUARD, Hard Site defense, telephone switching systems, digital communication networks, Ada compilers, and operating systems.

[47] As stated by Henry Petroski in *To Engineer Is Human: The Role of Failure in Successful Design, (New* York, NY: St. Martin's Press, 1985), p. 219.

> . . . departures from traditional designs are more likely than not to hold surprises.

[48] Early compilers for the new Ada language have been so slow, unwieldy to use, and bug-ridden that they have been worthless for real software development. This situation has occurred despite the fact that compilers for older languages such as FORTRAN, for which there have been compilers since the mid-1950s, are considered routine development tasks. The main contributing factors were the many features, especially the many new features, incorporated into Ada.

veloper's confidence in his ability to meet these criteria is based on the success of the previous attitude determination programs.

The developers of the SAFEGUARD software believed they could solve the problem of defending a small area from a ballistic missile attack because similar, but somewhat simpler, problems had been solved in the past. The history of missile defense systems can be traced back to World War II anti-aircraft systems, starting with the T-10 gun director. Next came: the M-9 gun director, which ultimately attained a 90 percent success rate against the V-1 flying bombs; the Nike-Ajax missile interceptor system; then, the Nike-Hercules, improved Hercules, Nike-Zeus, Nike-X, and Sentinel ABM systems.[49] Each system typically involved some mission changes and a change of one or two components over the previous one. Although the last few of these were never used in battle, constraining judgments of success in development, the evolutionary process is clear.

Note that the evolutionary approach requires the availability of experience gained from the earlier systems. Experience may take the form of personal memories or of documentation describing earlier programs. In other words, most of the problem must be well-understood and the solution clearly described. As Parnas put it, following a series of observations on what makes software engineering hard,

> The common thread in all these observations is that, even with sound software design principles, we need broad experience with similar systems to design good, reliable software.[50]

Observations of Internal Behavior

The above approaches to gaining confidence in software are based on observing the external behavior of the software without trying to determine how it behaves internally. That is, the software is tested by observing the effects of executing computer programs rather than the mechanisms by which those effects are produced. The next few sections discuss methods based on observing the internal behavior of programs-methods that may be called "clear box" to denote that the internal mechanism used to produce behavior may now be observed.

Simple Designs

It is not practical to give mathematical proofs that software performs correctly. Given a simple design and a clear specification of requirements, it is sometimes possible to give a convincing argument that each requirement is satisfied by some component of the design. Similarly, a convincing argument can be given that a simple design is properly implemented as a program. As with reliability measures, how convincing the argument is depends on subjective judgment. Where only a weak argument can be given that the design properly implements the requirements and that the code properly implements the design, there would belittle reason to trust the software, especially in its initial period of operation. As one expert puts it,

> . . . the main principle in dealing with complicated problems is to transform them into simple ones.[51]

Put another way, each complication in a design makes it less trustworthy. Simplicity, is, of course, relative to the problem. The inherent complexity of a problem it may require complex solutions. The designer's job is to make the solution as simple as he can. As Einstein said:

> Everything should be as simple as possible, but no simpler.[52]

Disciplined Development

The software development process comprises a variety of activities. Describing software cost estimation techniques, Boehm iden-

[49] The history of missile defense systems given here was supplied in a 1987 personal communication by Victor Vyssotsky, responsible for development of SAFEGUARD software.

[50] David L. Parnas, op. cit., footnote 1.

[51] T.C. Jones, *Design Methods,* Seeds *Of Human Futures, (New* York: John Wiley & Sons, 1980).

[52] Personal communication, P. Neumann.

tifies 8 different major activities occurring during software development and 15 different cost drivers.[53] Other estimators use different factors. (One early study introduced more than 90 factors influencing the cost of software development.) Fairley lists 17 different factors that affect the quality and productivity of software.[54] There is general agreement that many factors affect software development. There is still considerable doubt over how to identify the factors that would most significantly affect anew project—particularly if there is little experience with the development environment, the personnel involved, or the application. Appendix A describes the typical software development process and some of the complicating factors.

Development of large, complicated software must be a carefully controlled process. As the size and complexity of the software increases, different factors may dominate the cost and quality of the resulting product. Based on personal observations, Horning conjectured that:

> . . . for every order of magnitude in software size (measured by almost any interesting metric) a new set of problems seems to dominate.[55]

Although it is early to expect an accurate estimate of the size of BMD software, current estimates of the size of SDI battle management software range from a factor of 2 to a factor of 30 larger than the largest existing systems (and the the accuracy of some estimates is judged to be no better than a factor of 3).[56] If Horning's statement is correct, then there is reason to suspect that currently unforeseen problems would dominate BMD software development. Solving these problems would add to the time and expense involved in producing the software, and may undermine judgments of its reliability.

The development process must be geared to controlling the effects of the dominating factors. An example is the procedures by which changes are made. Most software development can be viewed as a process of progressive change. At every phase, ideas from the previous phase are transformed into the products of the current phase. For very small projects, the changes may be kept in the mind of one person. For moderately small projects, verbal communication among the project members may suffice to keep track of changes.

For larger projects, the number of people involved and the length of time of the project require that changes be approved by small committees and that written lists of revisions be distributed to all project personnel at regular intervals. Revised products of earlier phases are also distributed to those who need them. For very large projects, formal change control boards are established and all changes to baseline designs must be approved before they are implemented. A library of approved documents and programs is maintained so that all personnel have access to the same version of all project products. The process of controlling change becomes a source of considerable overhead, but is necessary so that all project members work from the same assumptions.

Factors Distinguishing DoD Software Development

There are some similarities between DoD and commercial software. The environments where DoD uses software are also found outside of DoD. Commercial and NASA avionics systems perform many of the same functions as military avionics, and must also work in life-threatening situations. Furthermore, the software must ultimately be produced in the same form, i.e., as a computer program, often in the same or a similar language for the same or a similar computer. But the DoD development process, as described in appendix A, is often

---

"*software Engineering Economics*, op. cit., footnote 45, p. 98.

"Richard Fairley, *Software Engineering Concepts, (New* York, NY: McGraw-Hill, 1985).

[55] Jim Horning, "Computing in Support of Battle Manage-merit, "ACM *SIGSOFT Software Engineering Notes 10(5):24-27,* October 1985.

"Barry Boehm, author of *Soft ware Engineering Economics*, and deviser of the most popular analytical software cost estimation model in use today, estimated, in a personal communication, that estimates of the size of SDI battle management software with which he was familiar could easily be in error by a factor of 3.

quite different from commercial software development.

Several factors, in *combination,* distinguish DoD software from commercial software. Elements of all of these factors are found in commercial software applications, but the combination is usually not.

- *Long lifetime.* Military command and control software often has a lifetime of 20 or more years. The Naval Tactical Data System was developed in the early 1960s and is still in use.
- *Embedded. New* DoD systems must interface with other, existing DoD systems. The interfaces are not under the control of the developer, and the need for the interface was often not foreseen when the existing system was developed. Commercial software developers are generally free to develop their own interfaces, or build stand-alone systems.
- *Operating in Real Time.* Command and control systems must generally respond to events in the outside world as they are happening. A delayed response may result inhuman deaths and damage to material.
- *Life-critical.* Command and Control and weapon systems are designed to inflict death or to prevent it from occurring.
- Large. DoD systems containing hundreds of thousands of lines of code are common. The larger systems contain as many as 3 million lines of code.
- *Complex.* Command and control systems perform many different functions and must coordinate the actions of a variety of equipment based on the occurrence of external events.
- Machine-near. The programmers of command and control systems must understand details of how the computer they are using works, how the equipment that it controls works, and what the interface between the two is. Many such details are transparent to commercial programmers because of the standardization of equipment, such as printers, for which already existing software handles the necessary details. The same is not true for new weapons, sensors, and computer systems specially tailored to particular DoD applications. As an example, the computers used on board the A-7 aircraft, in both the Navy and Air Force versions, were designed for that aircraft and rarely used elsewhere. The use of non-standard equipment often means that standard programming languages cannot be used because they pro tide no instructions that can be used to control the equipment. The current DoD trend is toward standardization of computers and languages, but programmers still must deal with specialized equipment.
- *Facing Intelligent Adversaries.* DoD battle management and command, control, and communications systems must deal with intelligent adversaries who actively seek ways to defeat them.

The DoD software development process is often characterized as cumbersome and inefficient, but is a significant improvement over the situation of the early 1970s when there was no standard development process. It provides some protection, in the form of required documentation, against software that is either unmaintainable or unmaintainable by anyone except the builders. Minimal requirements for the conduct of acceptance tests also provides some protection against grossly inadequate systems. Nonetheless, the process often still produces systems that contain serious errors and are difficult to maintain.[57] [58] The complexity of BMD software development would probably require significant changes in the process, both in management and technical areas.[59] The Fletcher Study concluded that:

> Although a strong concern for the development of software prevails throughout the civil and military data-processing community,

[57] For examples of problems in such systems ss the SGT YORK Division Air Defense Gun, *see ACM SIGSOFI'Software Engineering Notes,* op. cit., footnote 20.

[58] Upgrade of the A-7E avionics software, which is small (no more than 32,000 instructions), but quite complex (to accommodate a new missile cost about $8 million).

[59] Appendix A contains a further discussion of the DoD software development process and recent technical developments that might contribute to improving it.

more emphasis needs to be placed on the specific problem of BMD:

- Expanded efforts to generate software development tools are needed.
- Further emphasis is needed on simulation as a means to assist the design of battle management systems and software.
- Specific work is needed on algorithms related to critical battle management functions.[60]

Improving the Process

Software development, a labor-intensive process, depends for its success on many different factors. Improvements tend to come from better understanding of the process. Furthermore, improvements tend to be made in small increments because of the many factors influencing the process. To produce a system successfully requires, among other things:

- availability y of appropriate languages and machines,
- employment of properly trained people,
- ood problem specification,
- stable problem specification, and
- an appropriate methodology.[61]

Current efforts in software engineering technology development concentrate on providing automated support for much of the process. Software engineering tools may contribute to small incremental improvements in the process and the product. Such tools may help progr ammers produce prototypes, write and check the consistency of specifications, keep track of test results, and manage development applications and make the job of developing and testing the software easier. As an example, some computer systems allow programs to act as if they each had their own copy of the computer's memory. This feature permits several programs to execute concurrently without risk that one will write over another's memory area. The computer detects attempts to call on memory areas beyond a program's own and can terminate the program. The computer provides the programmer with information about where in the program the failure occurred, thus helping him find the error. This memory sharing technique makes the programmer's development job easier and allows the computer to be be used for several different purposes simultaneously.

Other systems permit the programmer to define an area of the computer's memory whose contents are sent at regular intervals to an external device. This feature could be used in conjunction with a display device to ensure that the display is properly maintained without the progr ammer having to write a special program to do so. Such a feature simplifies the job of developing software for graphics applications. Also, at the cost of added hardware, it improves the performance of the computer system when used with graphic displays.

Features built into the computer may make the software development job easier, the software more dependable, and the system performance better. The penalty for this approach may be to make the computer designer's job harder and the hardware more expensive. Further, the gain in software dependability is, as in many other cases, not quantifiable. Chapter 8 contains a more detailed discussion of various computer architectures and their potential for meeting the computational needs of BMD.

Software Dependability and Computer Architecture

Variations in computer design can have a strong effect on the software dependability. Some architectures are well-suited to certain

Software Dependability and System Architecture

Just as an appropriate computer architecture may lead to improved software dependability, so may an appropriate system architecture. A BMD architecture that simplified

[60] James C. Fletcher, Study Chairman and Brockway McMillan, Panel Chairman, op. cit., footnote 11.

[61] It is only in the last few years that the job title "software engineer" has been used. There is no qualification standard for software engineers, and no standard curriculum. Few universities or colleges yet offer an undergraduate major in software engineering, and there is only one educational institution in the country that offers a master's degree in software engineering.

coordination and communication needs among system components, such as different battle managers, would simplify the software design and might lead to improved software dependability. As with computer architecture, there would be a penalty: decreased coordination usually leads either to decreased efficiency or to more complex components. The increase in complexity is caused by the need for each component to compensate for the loss of information otherwise obtained from other components. As an example, if battle managers cannot exchange track information with each other, then they must maintain more tracks individually to do their jobs as efficiently. They may also have to do their own RV/decoy discrimination. Note that an architecture that requires exchange of a small amount of track information would be nearly as difficult to design and implement as one that required exchange of a large amount. The reason is that the communications procedures for the reliable exchange of small quantities of data are about the same as those for large quantities.

The Eastport group estimated that for an SDI BMD system the penalty for not exchanging track information among battle managers during boost phase would be about a 20 percent increase in the number of SBIs needed.[62]

The improvement in software dependability that might be obtained by architectural variations is not quantifiable.

Software Dependability and System Dependability

It is desirable to find some way of combining software and hardware dependability measures. As indicated earlier, MTBF, a traditional hardware reliability measure, is not appropriate as a sole measure of dependability of BMD software. Certainly it will still be desirable to measure hardware reliability in terms of MTBF in order to schedule hardware maintenance and to estimate repair and replacement inventory needs. The only components of both hardware and software dependability for which there may be some common ground for estimation are trustworthiness and availability. However, there have been few or no attempts to estimate trustworthiness for systems that are composites of hardware and software.

In summary, there are no established ways to produce a computer (hardware and software) system dependability measure. Furthermore, there are few good existing proposals for potential system dependability measures.

Software Dependability and the SDI

Although it is not possible to give a quantitative estimate of achievable software dependability for SDI software, it is possible to gain an idea of the difficulty of producing BMD software known to be dependable. We can do so by comparing the characteristics of a BMD system with characteristics of large, complex systems that are considered to be dependable. In an earlier section those characteristics were described. We apply them here to potential SDI BMD systems, using the architecture described in chapter 3 as a reference. Table 9-1 is a summary of the following sections. It shows whether or not each characteristic can be applied to SDI software, and provides a comparison with SAFEGUARD and the AT&T telephone system software, both often mentioned as comparable to SD I BMD software.[63]

SAFEGUARD and telephone system software represent different ends of the spectrum of large systems that could reasonably be compared to SDI BMD systems. The telephone system:

- is not a weapon system,
- has evolved over a period of a hundred years,

[62] Eastport Study Group Report, op. cit., footnote 4. The analysis and assumptions behind this claim have not been made available.

[63] Cf. Dr. Solomon Buchsbaum, Executive Vice President for Customer Systems for AT&T Bell Laboratories and former chair of the Defense Science Board and the White House Science Council:

> . . . most if not all of the essential attributes of the BM/C³ system have, 1 believe, been demonstrated in comparable terrestrial systems.
>
> S. Hrg. 99-933, op. cit., footnote 3, p. 275.

The system most applicable to the issue at hand is the U.S. Public Telecommunications Network.

Table 9-1 .—Characteristics of Dependable Systems Applied to SDI, SAFEGUARD, and the Telephone System

Characteristic	SDI	SAFEGUARD	Telephone system
Extensively used & abused	No	No	Yes
Predictable environment.	No	No	Yes
Low cost of a failure	No	No	Yes
Stable requirements.	No	Yes	Yes
Well-understood predecessors	No	Yes	Yes
Simple design	Unknown	7	Yes
Disciplined development'	Unknown	Yes	Yes

SOURCE Office of Technology Assessment 1988

- operates in a predictable environment with well-understood technology,
- is kept supplied with spare hardware parts that can be quickly installed, and
- is not designed to be resistant to an attack aimed at destroying it (although it can be reconfigured in hours by its human operators to circumvent individual damaged switching centers).

The SAFEGUARD system was a missile defense system that used well-understood technology, was never used in battle, would have had to operate in an environment that was not easily predictable, and was designed to make its destruction by an enemy attack costly.

Several other systems lie within the spectrum defined by SAFEGUARD and the telephone system. Examples are NASA flight software systems, such as the Apollo and Space Shuttle software, and weapon systems such as AEGIS. All have some of the characteristics of BMD systems. Nearly all are autonomous within clearly defined limits, must operate in real time, and are large. Some that are viewed as successful developments, such as AEGIS, have only been used under simulated and test conditions, but are thought to be sufficiently dependable to be put on operational status.

None of the examples known to OTA have been developed under the combined constraints imposed by SD I requirements, i.e., an SDI system would have to:

- control weapons autonomatically;
- incorporate new technology;
- be partly space-based, partly ground-based;
- defend itself from active and passive attacks;
- defend against threats whose characteristics cannot be well-specified in advance;
- operate in a nuclear environment, whose characteristics are not well-understood;
- be designed so that it can be changed to meet new threats and add new technology; and
- perform successfully in its first operational use.

Even a system such as AEGIS, which is perhaps DoD's most technologically advanced deployed system, was not developed under such stringent constraints, and its success is not yet fully determined.

Extensively Used and Abused

Although it might undergo considerable testing in a simulated environment, a BMD system cannot be considered to have been used in its working environment until it has been used in an actual battle. The working environment for a BMD system would be a nuclear war. Thus, the first time it would be used would also likely be the only time. In the telephone system, components that are put into use even after extensive testing often fail. A letter to Congress from designers and maintainers of AT&T Bell Laboratories switching systems stated:

> Despite rigorous tests, the first time new equipment is incorporated into the telephone network, it rarely performs reliably.
>
> Adding new equipment is just the tip of the iceberg; even the simplest software upgrade introduces serious errors. Despite our best efforts, the software that controls the telephone network has approximately one error for every thousand lines of code when it is initially incorporated into the system. Extensive testing and simulation cannot discover these errors.[64]

134A copy of the letter also appears as "SDI Software, Part II: The Software Will Not Be Reliable," ***Physics and Society,*** 16(2), April 1987.

Predictable Environment

Two aspects of the BMD battle environment will remain unpredictable until the outbreak of war. The first is the effect of the nuclear background caused by the battle and the second is the type and extent of the countermeasures employed against the system. In contrast, the telephone system environment is well-known and predictable. Call traffic can be measured and compared to mathematical models. Furthermore, much of the environment, such as the signals used in calling, is controlled by the designers of the system, so they are well-acquainted with its characteristics. Those who seek to defeat telephone systems want to use the environment for their own ends, and generally do not try to disrupt it. Therefore, although countermeasures are not all known in type and extent, neither are they intended to destroy the operation of the system.

Low Cost of a Failure

Software errors manifested as failures during a battle would not be repairable until after the battle. Catastrophic failures could result in unacceptably high numbers of warheads reaching their targets; there is no way to guarantee or predict that catastrophic failures will not occur. Even minor failures may result in failure to intercept some enemy warheads, causing loss of human life. Telephone switching centers experiencing catastrophic software failures generally can be removed from service and the software repaired while calls are rerouted. Minor failures are at most likely to cause difficulties for a few subscribers.

Stable Requirements

As new threats arose, new strategies devised, new countermeasures found, and new technology introduced, the requirements for BMD systems would change, and change continually. Although some changes could be planned and introduced gradually, changing threats and, particularly, countermeasures would impose changes beyond the control of the system developers and maintainers. BMD countermeasures are not subject to close scrutiny by the opposition, and new ones might appear quickly, requiring rapid response. Because changes in threat and the development of countermeasures would depend on Soviet decisionmaking and technology, the rate at which the U.S. would have to make changes to its BMD software would partly depend on Soviet actions. Delays in responding to countermeasures might have serious consequences, including the temptation for the side that had anew, effective countermeasure to strike first before a counter-countermeasure could be devised and implemented.

Well-Understood Predecessors

Earlier BMD systems, such as SAFEGUARD, can be characterized as terminal or late mid-course defense systems. The terminal and late mid-course defense part of an SD I BMD system could benefit from experience with these predecessors. There has been no experience, however, with boost phase and post-boost phase, and little experience with early mid-course defenses.[65] They are new problems that will take new technologies to solve. Most demanding of all, a system to solve these problems must be trusted to work properly the first time it is used. There have been approximately 100 years of experience with telephone switching systems. Each new system is a small change over its predecessor. If a newly-installed switching system does not work acceptably, it can be replaced by its predecessor until it is repaired.

Simple Design and Disciplined Development

Since the SDI BMD system has not yet proceeded to the point of a system design, much less a design for battle management or other software, one cannot judge whether or not the

[65] The Spartan missile, used by SAFEGUARD, can be considered a late mid-course defense component. However, SAFEGUARD was designed to discriminate reentry vehicles from aircraft, satellites, aurora, and meteors, but not from decoys of the types expected to be available for use against BMD systems within the next 10-20 years. The only discriminators available to SAFEGUARD were phased-array radars. Potential countermeasures against modern BMD systems are discussed in chapters 10 and 11, and discriminators in chapter 4. Options considered for both include technologies considerably different from anything available for or against SAFEGUARD.

design will be simple. Similarly, one cannot judge whether or not the development process will be appropriately disciplined.

Development Approaches That Have Been Suggested

In the middle ground between those who believe that an SDI BMD system could never be made trustworthy, and those who are sure that it could, are some software developers who are unsure about the feasibility. The view some of them take is that it would be worthwhile to try to develop BMD software, given that one were prepared to abandon the attempt if the system could not be shown to be trustworthy. The approaches they suggest have the following characteristics:

- The purpose of the system would have to be clearly stated so that the requirements were known before development started.
- The development would have to start with what was best known, i.e., should build upon the knowledge and results of earlier U.S. efforts to build BMD systems.
- The development would have to be phased, so that each phase could build upon the results of the previous one. The system architecture would have to be consistent with such phasing.
- Simulation would be needed at every stage, and the simulations would have to be extremely realistic.
- Realistic tests would have to be Performed at each stage of development:

Because failure is a clear possibility, those who advocate this approach recognize that options to deal with the possibility must be left open. If this approach were adopted, and failed, the cost of the attempt, including maintaining other options, could be high.

SUMMARY

Estimating Dependability

Most of the indices of dependability for large, complex software systems would be missing in BMD software systems. In particular, the telephone switching system, often cited as an example of a large, complex system, is quite unlike BMD systems.

The characteristics associated with dependability in large, complex systems include:

- a history of extensive use and abuse,
- operation in a predictable environment,
- a low cost of failures to the users,
- stable requirements,
- evolution from well-understood predecessor systems,
- a simple design, and
- a disciplined development effort.

The absence of many of these factors means that technology beyond the present state of the art in software engineering might have to be developed if there is to be a chance of producing dependable BMD software. It might be argued that such technology will be invented, but traditionally progress has been slow in software engineering technology development. It appears that the nature of software causes progress to be slow, and that there is no prospect for making a radical change in that nature.

There is no highly reliable way to demonstrate that BMD software would operate properly when used for the first time. One of the long-term purposes of the National Test Bed is to provide a means of simulating operation of BMD software after deployment. Such tests could simulate a variety of threats and countermeasures, as well as the conditions existing in a nuclear environment. On the other hand, actual environments often exhibit characteristics not reproduced in a simulator. Simulations of battles involving BMD would have to reproduce enemy countermeasures—a particularly difficult task. The usual technique for validating simulations—making predictions based on the simulation and then verifying their accuracy—would be particularly difficult to use. This

would especially be true when one considers the complexity of the atmospheric effects of nuclear explosions and the speculation involved in determining countermeasures. Repeated failures in simulation tests would demonstrate a lack of dependability. Successful performance in a simulation would give some confidence in the dependability of the system, but neither the dependability nor the confidence could be measured. Subjective judgments based on simulations would probably be highly controversial.

Traditional Reliability Measures

Traditional measures of reliability, such as mean time between failure, are insufficient to characterize dependability of software. Appropriate software reliability measures have yet to be fully developed. Furthermore, in the debate over BMD software dependability there is often confusion over the meaning of reliability. Error rate, e.g., number of errors per KLOC, is often misapplied as a definition of software reliability. There is no single figure of merit that would adequately quantify the dependability of BMD software. A potentially useful view is that dependability can be considered to be a combination of qualities such as trustworthiness, correctness, availability, fault tolerance, security, and safety. Unfortunately, there are no good ways of quantifying some of these properties and dependability would have to be a subjective judgment.

Technology for Preventing Catastrophic Failure

OTA found no evidence that the software engineering technology foreseeable in the near future would make large improvements in the dependability of software for BMD systems. In particular there would be no way to ensure that BMD software would not fail catastrophically when first used. It might be argued that the most important part of dependability is fault tolerance, and that there exist large, complex systems that are fault-tolerant, such as the telephone switching system. On the other hand, the fault tolerance of such systems is small compared to what would be needed for BMD, since they are not under attack by an intelligent. adversary interested in destroying their usefulness. A further complication of the argument over fault tolerance is that quantifications of software fault tolerance are not easily translated into measures of performance. At the same time, there is no generally accepted subjective standard of fault tolerance.

Confidence Based on Peacetime Testing

Confidence in the dependability of a BMD system would have to be derived from simulated battles and tests conducted during peacetime. Getting a BMD system to the point of passing realistic peacetime tests would most likely require a period of stability during which there were few changes made to the software. Unfortunately, the system developers and maintainers would have to respond to changes in threats and countermeasures put into effect by the Soviets. That is, the Soviets would partly control the rate at which changes would have to be made to the system. As changes were made, the system would again have to pass tests in order for the United States to maintain confidence in it.

Accommodating Changes During Peacetime

Experience with complex systems shows that changes eventually start introducing errors at a rate faster than they can be removed. At such a point all changes must be stopped and new software developed. The extent to which changes could be made would depend on the foresight of the developers during the design of the software. The better the requirements were understood at that time, and the better the potential changes were predicted, the more the chance that the software could accommodate changes as they occurred. The appearance of an unforeseen threat or countermeasure, or simply the advent of new, unexpected technology, might require redevelopment of all or substantial parts of the software. In a sense, the useful lifetime of the software would be determined by how well the software developers understood the requirements initially.

Establishing Goals and Requirements

Explicit performance and dependability goals for BMD have not been established. Consequently, one cannot set explicit software dependability goals. Even when BMD goals have been set, it will be difficult to derive explicit software dependability goals from them; there is no clear mapping between system dependability and software dependability. All agree that perfect software dependability is unattainable. Only arguments by analogy, e.g., as dependable as an automobile or telephone, have been proposed. There is no common agreement on what the dependability needs to be, nor how to measure it, except that it must be high.

There is common agreement that standard DoD procedures for developing software are not adequate for producing dependable BMD software in the face of rapidly changing requirements. There are few convincing proposals as yet on how to improve the procedures. The developers should not be expected to produce an adequate system on the first try. As Brooks says in discussing large software systems:

> In most projects, the first system built is barely usable. It maybe too slow, too big, too awkward to use, or all three. There is no alternative but to start again, smarting but smarter, and build a redesigned version in which these problems are solved. . . . all large-system experience shows that it will be done. Where a new system concept or new technology is used, one has to build a system to throw away, for even the best planning is not so omniscient as to get it right the first time. . . .
>
> Hence plan to throw one away: you will, anyhow.[66]

BMD software may be an order of magnitude larger than any software system yet produced. Early estimates of software size for projects are notoriously inaccurate, often by a factor of 3 or more. Some argue that the use of an appropriate systems architecture can make SDI software comparable in size to the largest existing systems. On the other hand, none of the intermediate or far-term architectures yet proposed would appear to have this effect, and previous experience with large software systems indicates that the size is likely to be larger than current estimates. Such an increase in scale could cause unforeseen problems to dominate the development process.

[66] Frederick P. Brooks, Jr., *The Mythical Man-Month; Essays on Software Engineering (New York, NY: Addison-Wesley, 1975), p. 116.*

SDIO INVESTMENT IN BATTLE MANAGEMENT, COMPUTING TECHNOLOGY, AND SOFTWARE

SDIO's battle management program serves as the focus for addressing many of the communications, computing, and software technology problems discussed in chapters 7,8, and 9. Based on funding and project description data supplied by SDIO, this section analyzes how SDIO is spending its money to try to solve these problems. The battle management program is organized into eight areas:

1. *software technology program plan:* developing and implementing a software technology program for the SDIO;
2. *algorithms:* development of algorithms for solving battle management problems such as resource allocation, track data handing over, discrumination, and coordination of actions within a distributed system;
3. *communications:* identifying the requirements and technology for establishing a communications system to link SD I components together into a BMD system;
4. *experimental systems:* proposing and evaluating system and battle management architectures and the technologies for implementing them;
5. *networks:* the design and development of distributed systems and of communications networks that could be used to support BMD;

6. The National Test Bed: procurement of hardware and software needed for the National Test Bed;
7. *processors:* development of computers that would be sufficiently powerful, radiation-hardened, fault-tolerant, and secure for BMD needs, and of the software required to operate them; and
8. *software engineering* the technology for developing and maintaining software for SDI, including techniques and tools for requirements specification, design, coding, testing, maintenance, and management of the software life-cycle.

Table 9-2 is a snapshot of the funding for these areas as of June 1987. Rather than showing the fiscal year 1987 SDIO battle management budget, it shows money that at that time had been spent since the inception of the program, that was then under contract, or that was expected soon to be under contract. It is a picture of how the SDIO was investing its money to solve battle management problems over the first few years of the program. Not shown is money invested by other agencies, such as the Defense Advanced Research Projects Agency, in joint projects. The leverage attained by SDIO in some areas is therefore greater than might appear from the table.

The SDIO battle management program clearly emphasizes experimental systems. Examination of the individual projects in this area shows a concentration on the development and maintenance of simulations and simulation facilities, such as the Army's Strategic Defense Command Advanced Research Center Test Bed, used to run battle simulations; on architecture analyses, such as the phase I and II battle management/C^3 architecture studies; and on the first two Experimental Validation 88 (EV88) experiments.

The funding categories shown in table 9-2 permit considerable overlap; projects in each category could easily be assigned to a different category. To try to draw clearer distinctions among categories and to try to identify funding targeted specifically at the problems discussed in chapters 7 through 9, OTA reorganized the funding data supplied by SDIO. Table 9-3 shows just those funds aimed at exploring solutions to some of the more significant problems noted in chapters 7 through 9. It does not include all funds shown in table 9-2, but does show percentages of total funding. The categories are defined as follows:

1. *battle management and system simulations:* the development of particular simulation algorithms or specialized hardware for battle management simulations;
2. *simulation technology development:* the development of the hardware and software for bigger, faster simulations, and for improving techniques for evaluating the results of simulations;
3. *automating existing software engineering technology:* the development of software and hardware that would be used to improve the software development and maintenance process, which is now based on existing manual techniques;

Table 9-2.—SDIO Battle Management Investment

Area	Funding ($M)	Percent of total
Software technology program plan	2.5	1
Algorithms	25.3	9
Communications	8.1	3
Experimental systems	117.5	42
National Test Bed	13.0	5
Networks	29.6	11
Processors	47.1	17
Software engineering	32.8	12
Total	275.9	100

SOURCE: Office of Technology Assessment, 19S8; and SDIO.

Table 9-3.—Funding for OTA Specified Problems

Problem	Funding ($M)	Percent of
Battle management and system simulations	42.5	15
Simulation technology development	35.7	13
Automating existing software engineering technology	14.2	5
Computer security	10,3	4
Communications networks	7,8	3
Software verification	4.6	2
Fault tolerance (hardware and software)	3.1	1
Software engineering technology development	2.5	1
Total	120.6	44

[a] Percentage of total battle management funding, i.e., of $275.9M

SOURCE Office of Technology Assessment, 1988, and S010

4. *computer security:* techniques for detecting and preventing unauthorized access to computer systems;
5. *communications networks:* the organization of computer-controlled communications equipment into a network that could meet SDI communications requirements;
6. *software verification:* the development of practical techniques for mathematically proving the correctness of computer programs;
7. *fault tolerance (hardware and software):* the development of hardware and software that continues to work despite the occurrence of failures; and
8. *software engineering technology development:* the development of new techniques for improving the dependability of software and the rate at which dependable software can be produced.

Table 9-3 shows that SDIO is investing considerably more in simulations and simulation technology than any of the other problem areas in battle management and computing identified by OTA. Of **some concern is the smallness of the investment in especially challenging areas** such as computer security, communications networks, fault tolerance, and new software engineering technology development.

CONCLUSIONS

Based on both the preceding analysis, and the further exposition in appendix A, OTA has reached eight major conclusions.

1. The dependability of BMD software would have to be estimated subjectively and without the benefit of data or experience from battle use. The nature of software and our experience with large, complex software systems, including weapon systems, together indicate that there would always be irresolvable questions about how dependable the BMD software was, and also about the confidence to be placed in dependability estimates. Political decision-makers would have to keep in mind that there would be no good technical answers to questions about the dependability of the software, and no well-founded technical definition of software dependability.

   It is important to note that the Soviets would have similar problems in trying to estimate the dependability of the software, and therefore the potential performance of the system. Technical judgments of dependability would rely on peacetime tests that would be unlikely to apply to battle conditions. Political judgments about the credibility of the defense provided would therefore rest on very uncertain technical grounds.
2. No matter how much peacetime testing were done, there would be no guarantee that the system **would not fail catastrophically dur**ing battle as a result of a software error. Furthermore, experience with large, complex software systems that have unique requirements and use technology untested in battle, such as a BMD system, **indicates** that there is a significant probability that a catastrophic failure caused by a software error would occur in the system's first battle.
3. It is possible that an administration and a Congress would reach the political decision to "trust" software that passed all the tests that could be devised in peacetime, despite the irresolvable doubts about whether such software might fail catastrophically the first time it was used in an actual battle. Such a decision could be based upon the argument that the purpose of strategic forces—even defensive strategic forces—is primarily deterrence, and that a defensive system passing all its peacetime tests would be adequate for deterrence. If deterrence succeeded, we would never know, and never need to know, whether the system would function in wartime.
4. The extent to which BMD software would differ from complex software systems that have proven to be dependable in the past raises the possibility that software could not be created that ever passed its peacetime tests. This a possibility exacerbated by the prospect of

changing requirements caused by Soviet actions. We might arrive at a situation in which fixing problems revealed by one test created new problems that caused the software to fail the next test.

5. No adequate models exist for the development, production, test, and maintenance of software for full-scale BMD systems. Current DoD models of the software life-cycle and methods of software procurement appear inadequate for the job of building software as large, complex, and dependable as BMD software would have to be.
6. The system architecture, the technologies to be used in the system, and a consistent set of performance requirements over the lifetime of the system must be established before starting software development.[67] Otherwise, the system is unlikely even to pass realistic peacetime tests.
7. As the strategic goals for a BMD system became more stringent, confidence in one's ability to produce software that would meet those goals would decrease as a result of the increased complication required in the software design. Even for modest goals, such as improved deterrence, the United States could not have high confidence that the software would not fail catastrophically, whether faced with a modest threat or a severe threat. Put another way, there is no good way of knowing that BMD software would degrade gracefully rather than fail catastrophically when called on to face increasing levels of threat. Current techniques for identifying problems and detecting errors, such as simulations, would not help, although they could help to reduce the failure rate. Furthermore, foreseeable improvements in software engineering technology would not change this situation.
8. The SDIO is investing relatively small amounts of money in software technology research in general, and in software engineering technology, computer security, communications networks, and fault tolerance in particular. This investment strategy is of some concern, since particularly challenging BMD software development problems lie in these areas.

[67] Note that this does not preclude a phased system, with both capabilities and requirements growing over time, provided that the final architecture and final performance requirements are clear before initial software development begins. Even then, considering the uniqueness of BMD defense, one would expect to spend considerable time finding a workable design.