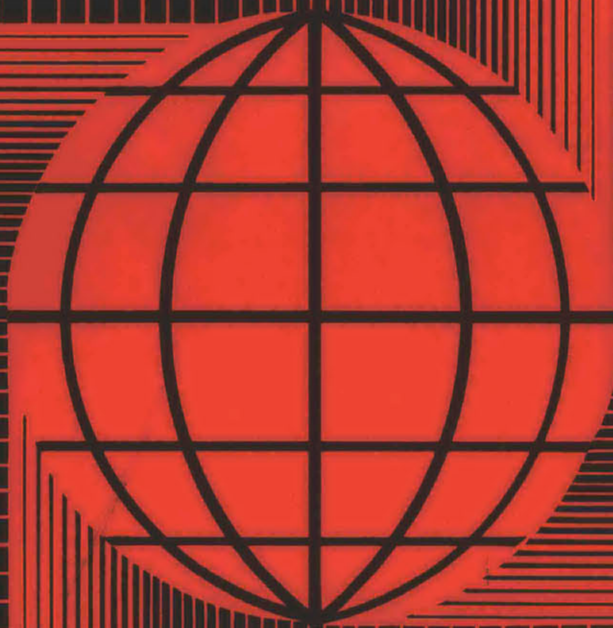
C/85-2



# SOFTWARE FOR BALLISTIC MISSILE DEFENSE

# 2513

Herbert Lin

CENTER FOR
INTERNATIONAL STUDIES
MASSACHUSETTS INSTITUTE
OF TECHNOLOGY
Cambridge, Massachusetts 02139

# Software for Ballistic Missile Defense

Herbert Lin
Center for International Studies
E38-616
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139
(617) 253-8076

June 1985
Copyright Herbert Lin
All Rights Reserved

## Abstract

A battle management system for comprehensive ballistic missile defense must perform with near perfection and extraordinarily reliability. It will be complex to an unprecedented degree, untestable in a realistic environment, and provide minimal time for human intervention. The feasibility of designing and developing such a system (requiring upwards of ten million lines of code) is examined in light of the scale of the project, the difficulty of testing the system in order to remove errors, the management effort required, and the interaction of hardware and software difficulties. The conclusion is that software considerations alone would make the feasibility of a "fully reliable" comprehensive defense against ballistic missiles questionable.

IMPORTANT NOTE: this version supersedes a widely circulated but earlier draft entitled "Military Software and BMD: An Insoluble Problem?" dated February 1985.

## Acknowledgements

The assistance of dozens of reviewers (many of whom prefer to remain anonymous) is gratefully acknowledged. The following people made especially critical and useful comments: Paul Anderson, Herb Bennington, Alan Borning, Mark Day, Richard Garwin, Richard Hilliard, Richard King, Jack Ruina, John Shore.

Herbert Lin is a post-doctoral research fellow in the Arms Control and Defense Policy group of the M.I.T. Center for International Studies. He received his doctorate in physics from M.I.T. in 1979. His primary research interest is the relationship of technology to national security policy.

# Software for Ballistic Missile Defense

Herbert Lin
Center for International Studies
E38-616
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139
(617) 253-8076

June 1985

"I can call spirits from the vasty deep."
-- Glendower

"Why, so can I, or so can any man,
but will they come when you do call for them?"
-- Hotspur

King Henry IV, Part I, Act 3, Scene 1, Line 53
William Shakespeare, 1597

To an ever-increasing degree, modern weapon systems depend critically on the technology of micro-electronics and computers. Software -- the instructions that specify to the controlling computers what operations must be performed at what times and in what sequence (as well as supporting elements such as proper documentation and operating procedures) -- is one of the most important components of computer technology. However, despite its importance, the subject of software (military or otherwise) is one that is usually reserved for computer specialists. The purpose of this paper is to present an overview of military software issues as they relate to ballistic missile defense, without assuming any particular knowledge of computers.

## 1 -- Introduction

Computer technology is generally divided into hardware and software. Computer hardware is tangible; examples include a digital watch, a pocket calculator, and a personal computer. By contrast, computer software is essentially intangible. Software fundamentally concerns procedure, the sequence of operations that the computer hardware must perform to accomplish a given task. Examples include the procedure one must perform to set the alarm on a digital watch, the sequence of keys that one must push to perform a calculation on a calculator, and the word-processing program that enables one to use a personal computer as a word processor rather than as a music generator. In addition, a large software system requires documentation that will allow human beings to understand the limits of its capabilities and to change its operation through re-programming, and a clear set of operating procedures that specify exactly what human beings must do in order that the computer programs do what they are supposed to do.[1]

Computers can process large amounts of information very quickly. It is this capability that makes computers indispensable to modern warfare, which is characterized by a fast-paced environment and large amounts of information relevant to a given battle. Computers can help to control the

----------

1. One way of thinking about hardware and software is that software is whatever can be sent over a telephone, hardware is whatever can't.

function of individual weapons (e.g., the guidance of a missile, the aiming of a gun). They can also help to perform battle management: the monitoring of the threat and the progress of the battle, and the coordination of the weapons available to meet the threat. Examples of systems that perform battle management include the Airborne Warning and Control System (AWACS) and the AEGIS air defense system for carrier battle groups.

For many people, military computing is hardware-oriented: capacity, speed, and survivability are some of the parameters of interest. It is also in computer hardware that the most dramatic advances in micro-electronics have occurred. Indeed, these advances have helped to drive the hardware cost of computing down at an average rate of 25% per year. It is widely believed that these advances will continue, and will be limited only by fundamental physical constraints such as the finite speed of light.[2]

However, computing hardware without software is useless, and software is an increasingly important aspect of computer technology. For example, in the 1960's, the average hardware cost of a typical computer (both military and non-military) project was about 40% of the total cost; in 1984, the hardware costs constituted only 15% of the total project cost over its entire life cycle.[3] Indeed, it is becoming increasingly plausible to regard software as the webbing which binds individual components into an integrated system. Software can also present a more challenging

----------

2. cf., Lewis Branscomb, "Electronics and Computers: An Overview", Science, Volume 215, 12 February 1982, page 755.

3. Branscomb, op cit.

intellectual task. Admiral Bobby Inman notes that

> The hardware [of a weapons system] might not be right the first time. But in almost every case it is right the second time. [For software], it is somewhere between the third and fifth iteration before you are close to the performance that originally had been desired, or thought to be desired, by the would-be user.[4]

This is not to say that software technology has stood still. The productivity of systems programmers over the last ten years has increased at about 10% per year.[5] Newly developed and implemented programming techniques offer the promise of making make large programming jobs easier to accomplish.[6] Entire sub-fields of computer science (e.g., expert systems and automatic programming, discussed later) appear to promise a fundamental alteration of the relationship between man and machine.

The Department of Defense has not been unaware of the increasing importance of software. It notes that

> Development and support for major military systems is one of the most complex human endeavors, often requiring hundreds of people for five or more years... These projects require the resolution of complex systems issues using techniques and management approaches that are poorly defined and not well understood.[7]

----------

4. Bobby Inman, "Technology and Strategy", Proceedings of the U.S. Naval Institute, Sea Link 1984, page 48.

5. Branscomb, op cit.

6. cf., Grady Booch, Software Engineering in Ada, Benjamin Books, 1983, Chapter 4.

7. Department of Defense, Software Technology for Adaptable, Reliable Systems Programming Strategy, 15 March 1983, page 4. Hereafter referred to as STARS; published in ACM SIGSOFT Software Engineering Notes, Volume 8(2), April 1983, page 58.

One institutional response has been the STARS program -- Software Technology for Adaptable, Reliable Systems -- whose stated goal is "to improve [software] productivity while achieving greater system reliability and adaptability" by "improving the skills, tools and business practices that constitute the environment in which software is developed and supported."[8] A second response has been the Defense Department's support over the last several years for research in expert systems and automatic programming; these are discussed in a later section. These responses (and other advances on which they have been built) give some analysts the hope that very complex software systems can indeed be developed.

Whether past and future advances in software technology will allow the successful development of large systems of military software is open to debate. In this paper, I will focus on one of the most ambitious software tasks ever imagined: the planning, design and implementation of software to control and manage a comprehensive defense against nuclear ballistic missiles (BMD). I will focus mostly on battle management issues, but I will also address the issue of controlling individual weapons.

## 2 -- The Nature and Scope of the BMD Problem

"The ultimate goal of the Strategic Defense Initiative (SDI) is to eliminate the threat posed by nuclear ballistic missiles." Thus states the

----------

8. STARS, pages 15-16.

charter of the Strategic Defense Initiative Organization.[9] To achieve this goal, there have been proposals for a multi-layered defense that would intercept boosters and re-entry vehicles at various points along their trajectories; see Figure 1. A variety of technologies have been proposed for performing these interceptions, including lasers, particle beams, electromagnetic rail guns, and homing interceptors. However, interception technologies alone do not constitute a _system_ for ballistic missile defense (BMD); many other technologies are needed.

In particular, computer technology (including both hardware and software) is critical to the success of the SDI, for the control of individual weapons and for battle management; of these two tasks, the latter is by far the most demanding. The Defensive Technologies Study Team (DTST), chartered by the Department of Defense to examine the technical feasibility of BMD, concluded that

> Developing hardware will not be as difficult as developing appropriate software. Very large (order of ten million lines) software that operates reliably, safely, and predictably will have to be developed.[10]

For comparison, Figure 2 lists various software systems and the amount of programming that they require. Assuming that the amount of "real-time" programming is a rough measure of its complexity, it is clear that the

----------

9. Strategic Defense Initiative Organization Interim Charter, April 24, 1984. Reprinted in Hearings on Department of Defense Appropriations for FY 1985, House Appropriations Committee, Subcommittee on Defense, Part 5, page 696.

10. Defensive Technologies Study Team, _The Strategic Defense Initiative_, Under Secretary of Defense for Research and Engineering, Department of Defense, March 1984, page 19. Unclassified summary of Fletcher Commission report. Hereafter referred to as DTST.

software required for BMD is considerably more complex than other software systems. Indeed, the DTST noted that

> [The development of] software for a battle management system [for BMD] 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]

A battle management system for comprehensive BMD might be organized as described in Figure 3; in tens of minutes, it must be capable of handling without error and in an enormously complex and unpredictable environment information on thousands of missiles, tens of thousands of warheads, and hundreds of thousands of decoys. In addition, it must be highly automated; this allows minimal time for human intervention to correct unexpected failures. Indeed, in most cases, it would "replace human decision-making".[12] Finally, given the potentially enormous consequences of even partial failure, it must function with near-perfect reliability.

## 3 -- Software Development for BMD

On the basis of models of the software development process calibrated on a historical basis, we can estimate the scale of effort required to produce a software system of a given size using conventional programming

----------

11. Battle Management, Communications, and Data Processing, Study on Eliminating the Threat Posed by Nuclear Ballistic Missiles, Volume 5, February 1984, SRI International, page 4. Hereafter referred to as Volume 5, FC report.

12. James Fletcher, "The Technologies for Ballistic Missile Defense", Issues in Science and Technology, Fall 1984, page 21.

techniques. The model used here, the COnstructive COst MOdel (COCOMO) [13], takes into account various factors relevant to software development (see Figure 4), and provides the estimates listed in Figure 5 for a software system for BMD -- 10,000,000 lines of code, which will be characterized by strong interdependence among hardware, computer programs, regulations, and operational procedures, coordination of program operation and events occurring independent of the computing hardware ("real-time" operating conditions, a simple example of which is in Figure 6), and most of all, the need for extraordinary reliability.[14] (Note that the effort required to produce a functional system is non-linear in program size; COCOMO uses an empirical fit of program size to the 1.2 power.[15]) For comparison, Defense Department expenditures on software development totaled about $5-6 billion per year in the early 1980's [16], or about 15% of all software costs in the U.S.[17] Assuming annual personnel plus equipment costs of about $100,000 per person per year, fifty thousand man-years used in ten years would account for about 1-2% of all programming and analyst manpower currently available to the nation.[18]

----------

13. Barry Boehm, <u>Software Engineering Economics</u>, Prentice-Hall, 1981, Chapters 23-27.

14. Boehm, page 79.

15. Thus, a system twice as large takes more than twice the effort to produce. See Boehm, page 117.

16. STARS, page 8.

17. Boehm, page 17.

18. This estimate is roughly corroborated by the U.S. Census estimate of about 534,000 computer specialists in 1979 (Bureau of the Census, <u>Statistical Abstract of the United States</u>, 101st Edition, 1980, page 418.)

The multi-tiered character of proposed BMD systems would allow software development to be separated into semi-autonomous packages, some of which would operate with layers and others of which would coordinate the operation of the entire system; see Figure 7. This separation would reduce the effects of the non-linear relationship between effort and program size. If the project can be broken into forty independent programming jobs, the entire job would take about half as long as indicated in Figure 5.

On the other hand, the 10,000,000 line estimate is most likely conservative. For example, the accuracy of quantitative software development models depend on the existence of reasonably well-defined phases of conceptual design, detailed design, coding and testing. F.J. Corbato (Professor of Computer Science at M.I.T.) points out that these phases have meaning only within the context of well-understood problems; a poorly understood problem requires greater effort than that predicted by a development model based on well-understood problems. He further suggests that growth factors of ten (from projected to actual program size) are not unheard of, and that factors of two are not uncommon.[19] Indeed, informal "word-of-mouth" estimates of the necessary software discussed since the release of the Fletcher Commission report are three to five times higher, and one report suggests that the required programming may run as high as 100 million lines.[20]

----------

19. personal communication, November 1984.

20. Washington Post, March 3, 1985, page 7.

Finally, the DTST estimate is for a purely defensive system. However, since the transition to a purely defensive mode will not occur instantaneously, the activities of a defensive system would have to be coordinated with those of the remaining offensive weapons, thereby introducing additional complexity into the system that will be reflected in even more software. For example, it would have to be able to distinguish between Soviet and American submarine-launched missiles, both of which can be launched from large and overlapping areas of the ocean. This task is no easier during boost-phase than the long-unsolved problem of distinguishing between friendly and hostile airplanes during wartime.[21] Alternatively, a system designed for boost-phase intercept can be used to suppress an opponent's missile-based defensive systems in conjunction with an offensive attack (for whatever reason).

Underestimates of software size are more common that overestimates for several reasons.[22] One is that people are reluctant to face confrontations brought on by high estimates of software size. In addition, people tend to underestimate the amount of support software required for large jobs (e.g., diagnostics that are used for identify hardware difficulties, simulators that are used to "exercise" the system under assumed threat conditions). Finally, and most significantly, people are often not familiar with the entire software job. Figure 5 also presents

----------

21. On the importance and unavailability of a functional IFF (Identification, Friend or Foe) system for tactical airpower in NATO, see "Interview With General Billy Minter, Commander in Chief, USAF Europe", Armed Forces Journal International, Volume 121(6), January 1984, pages 42-47.

22. cf., Boehm, page 320.

development effort for a system that is twice as large as the DTST estimate.

The COCOMO model has two significant limitations. The first is that the data base used to calibrate the COCOMO equations includes software systems up to 1.5 million lines; therefore, extrapolations beyond this range (especially by factors greater than 10) are inherently suspect. The second is that any model based on historical data cannot take into account software development techniques not yet used or developed. These two limitations drive the predicted effort in different directions. The first most likely leads to underestimates of the actual effort needed to complete a project, since it is intuitively implausible that a system significantly more complex than all previous projects might become suddenly simpler as its size increased. The second most likely leads to overestimates of the actual effort, since software development techniques are not adopted if they do not increase productivity. As the relative effects of these two limitations tend to cancel each other, they will not be discussed further, but they should not be ignored.

The number of lines of code and the actual effort required for a comprehensive BMD system are not the major issues (though these are interesting questions significant to program managers), but rather that these estimates provide a measure of the complexity of the BMD problem. Therefore, claims that programmer productivity can vary by a factor of 25 given a clearly understood and specified problem and that programming tools can improve programmer productivity by a factor of 10 are both true and essentially irrelevant to the issue of complexity toward which much of this

paper is addressed.

## 4 -- Impediments to BMD Software Development

In the pages that follow, it is essential to keep in mind that the stated ultimate goal of the SDI is to eliminate nuclear ballistic missiles as a threat to the U.S. and its allies. Other goals, such as the strengthening of deterrence or the protection of strategic military assets will be discussed separately.

### 4.1 Planning and Design

The planning and design phases are the most difficult phases of a project, since it is in these phases that people decide in detail exactly what the software system should do, how it should do it, and how to plan for future changes (see Software Maintenance below). While recent developments in software engineering do make it easier to articulate requirements and specifications that are unambiguous and consistent, they cannot (and should not) assume the burden of human decision-making. A more complex problem means that human beings must make more design decisions, while the probability that all of them will be made (let alone made correctly!) decreases exponentially with the number of decisions.

#### 4.1.1 Planning

The first conceptual step in all software development is planning:

specifying the functions that the software is to perform, and the environments in which the software must perform these functions.

These specifications become more difficult to articulate as the task grows or becomes more complex, since more human decisions must be made correctly. Figure 8 illustrates how the effects of scale can affect the complexity of specifying an "everyday" task. Another example is the collection of federal taxes. The U.S. Tax Code occupies 3000 pages that can be regarded as the detailed specification of tax law; there are many times that number of pages of precedents, regulations, and opinions that govern the actual implementation of Federal tax law. The complexity of the tax system arises largely from the need to specify exactly how much tax must be collected under all possible circumstances.[23]

Similar problems arise in the BMD task, but with much more complexity. For example, the flight control program for the A-7 attack airplane consists of 12,000 lines. *After* significant operational experience with this program was acquired, a software engineering team attempted to generate detailed specifications for the functional behavior of the system; they regarded their effort as successful when they were able to compress many shelves of manuals into a single 500 page manual.[24] For a software system on the order of 10,000,000 lines, it is not unreasonable to expect

----------

23. It would not be facetious to point out that taxpayer effort in finding loopholes in the tax laws is somewhat analogous to the Soviets' trying to find flaws in a BMD.

24. Kathryn Heninger, "Specifying Software Requirements for Complex Systems: New Techniques and Their Application", *IEEE Transactions on Software Engineering*, Volume SE-6(1), January 1980, page 2.

that its detailed specification would occupy hundreds of thousands of pages.

For example, the specification of "shoot down Soviet missiles" is adequate if the world contains only Soviet missiles and all Soviet missiles should be shot down under all circumstances. It is not an adequate specification in the real world. What defines a Soviet missile? What about non-Soviet missiles? What if the missile is not headed for a target of interest to the United States? What if the missile is actually a Soviet space shuttle launcher on a non-military mission? Which, if any, civilian authorities should have the opportunity to exercise judgment in the shooting down of a Soviet missile? When and under what observable circumstances should a Soviet missile attacked once be attacked again?

These questions only scratch the surface of a requirements specification, but they illustrate a major difficulty of large scale software development: it is often difficult to decide if some particular behavior of a program is desirable or undesirable. All of the questions above could be answered in ways over which reasonable people could disagree.

In order to specify the requirements for BMD, it would be necessary to predict accurately the characteristics of all possible environments in which the defense would have to function, and the appropriate response in these environments. Robert Cooper, Director of the Defense Advanced Research Projects Agency, has commented on this issue; in response to a question about the problems pacing the development of ballistic missile defense, he noted that

> we have no way of understanding or dealing with the problems of battle management in a ballistic missile attack ranging upward of many thousands of launches in a short period of time.[25]

To illustrate the difficulty of predicting operating environments, consider the World Wide Military Command and Control System (WWMCCS), used by civilian and military authorities to communicate with U.S. military forces in the field. In November 1978, a power failure interrupted communications between WWMCCS computers in Washington, D.C. and Florida. When power was restored, the Washington computer was unable to reconnect to the Florida computer, because no one had anticipated a need for the same computer (i.e., the one in Washington) to "sign on" twice. Human operators had to find a way to bypass normal operating procedures before being able to restore communications.[26] A second example is found in one report on the Falklands War which asserts that British warning systems were programmed to ignore transmissions from the homing devices carried by Exocet missiles, because British forces also carried Exocets. Only after an Exocet missile sunk the British destroyer Sheffield were the warning systems reprogrammed to recognize Exocet missiles as hostile.[27] Still a third example is the 767 airliner whose two engines shut down in flight. The computers controlling engine operation were programmed for "maximum fuel efficiency"; as a result, the engines ran too slowly to keep ice from

----------

25. Robert Cooper, Director of DARPA, "Strategic and Theater Nuclear Forces", Hearings before the Senate Committee on Armed Services, Department of Defense Authorization for Appropriations for FY 1984, page 2892.

26. William Broad, "Computers and the U.S. Military Don't Mix", Science, Volume 207, 14 March 1980, page 1183.

27. New Scientist, Volume 97(1334), 10 February 1983, page 353.

forming, and they overheated.[28]

4.1.2 Design

The second conceptual step in software development is design, the process by which analysts decide in detail how the software system should do what the specifications say it should do, and how to plan for future changes. A design error is one that results from misjudgment about how to implement some part of the system. For example, a decision to write a certain program to facilitate future modification might result in a program that does not run fast enough during a massive attack.

Since even designs for BMD battle management software are not yet available, we may examine the records of other software systems. For example, Gemini V missed its landing point by 100 miles because its guidance program implicitly ignored the motion of the earth around the sun.[29] A false warning of missile attack on the U.S. was reported on June 3, 1980 when a faulty computer chip resulted in improper messages that should have been detected by monitoring software.[30] Five nuclear reactors were shut down temporarily, because a program used to test their design for resistance to earthquakes used an arithmetic sum of variables

----------

28. Associated Press, "Jet Engine Failure Tied to Computer: It's Too Efficient". Reported in the Los Angeles Times, August 24, 1983, page 1.

29. Joseph Fox, Software and Its Development, Prentice Hall, 1982, pages 187-188.

30. Recent False Alerts from the Nation's Missile Attack Warning System, Report of Senators Gary Hart and Barry Goldwater to the Senate Armed Services Committee, October 9, 1980.

when it should have used the square root of the sum of the squares of these variables.[31] A guided missile frigate fired its computer-aimed gun in a direction 180 degrees from the target.[32] The rising of the moon was interpreted as a missile attack on the United States.[33] Minuteman III missiles were less accurate than the limits inherent in the hardware of their guidance systems.[34] All of these difficulties occurred in software systems less complex than any comprehensive BMD system would be, and resulted from erroneous design decisions or omissions.

#### 4.1.3 Sources of Error in Planning and Design

Both planning and design errors arise from the deficiencies in human thought processes, and are therefore hard to examine in the absence of tangible demonstrations. Software developers are particularly susceptible to a basic faith that "all will go well"; because their products are essentially ideas, expressed as lines of computer program, developers can postpone the confrontation between ideas and reality to a point much later than can others who build tangible objects. The result is that the design

----------

31. Evars Witt, "The Little Computer and the Big Problem". AP Newswire, 16 March 1979. Specifics of the software error are described by Peter Neumann in "An Editorial on Software Correctness and the Social Process", Software Engineering Notes, Volume 4(2), April 1979, page 3.

32. United Press International, "Navy Blames Computer". Reported in the San Francisco Chronicle, August 11, 1983, page 12.

33. J.C. Licklider, "Underestimates and Overexpectations", in ABM: An Evaluation of the Decision to Deploy an Anti-Ballistic Missile, Abram Chayes and Jerome Wiesner (eds.), Harper and Row, 1969, page 122-3.

34. Deborah Shapley, "Technology Creep and the Arms Race: ICBM Problem a Sleeper", Science, Volume 201, 22 September 1978, page 1103.

clarity perceived by software developers is not matched by the actual existence of clarity, and errors of planning and design occur.[35] Neither the frequency nor the nature of these errors is predictable. They can be reduced -- but not eliminated -- only by very hard thinking about possible failure modes, a task whose difficulty increases with system size.

### 4.2 Assuring Reliable Operation

Software is not "reliable" or "not reliable"; rather, it is more or less reliable depending on the care with which it was constructed, the number of errors it contains, and the degree to which it is properly validated both analytically and empirically. Moreover, "reliability" cannot be specified precisely in the absence of operational experience, though the concept retains its significance as the extent of the match between its actual performance and its proper behavior. Thus, it is reasonable to speak in terms of developing confidence that a given software system will be reliable when placed in actual operation.

#### 4.2.1 Proofs of Program Correctness

One method of developing confidence in a software system is analytical. Analytical methods, known as proofs of program correctness, require a formal (i.e., a mathematically complete and consistent) specification of the task to be performed, as the full Fletcher Commission report recognizes: "A complete set of system requirements is needed...

----------

35. For more discussion of this point, see F.P. Brooks, The Mythical Man-Month, Addison-Wesley, 1978, page 15.

[involving] a formal set of requirements [that can be] specified and verified for completeness and consistency."[36] An example of a formal specification is a mathematical equation that specifies the output behavior of a function given inputs to the function.

However, even the new Department of Defense programming language Ada, based on thirty years of operational experience, does not have a complete set of formal specifications; rather, the "specifications" are presented in natural English, and conformity to these specifications is assured not by formal proof but by the repeated testing of language compilers (i.e., programs that translate programs written in the language into machine-understandable form) against a set of test programs that grows with time.[37]

Thus, it seems unlikely that the much more poorly understood task of battle management for BMD can be specified formally in light of Cooper's statement on page 16. More importantly, the existence of formal specifications is not an assurance that these specifications themselves specify what should be done.

Even if formal specifications could be developed, proofs of program correctness would still not insure the adequacy of the software system. In particular, a proof of correctness involves a proof that a program, once started, will terminate in a finite time, a set of assumptions about the

----------

36. Volume 5, FC report, page 46.

37. cf., Daniel Klein, "A Buyer's Guide to Ada Procurement", Defense Electronics, Volume 17(1), January 1985, page 103.

expected input, and a mathematical proof that input conforming to these assumptions will result in output characterized by the formal specifications of the program. None of these features guarantee adequacy.

For example, though a program proof specifies that the program will eventually terminate, it does not specify that a given computation can be completed in an appropriate amount of time. In addition, it does not specify the nature of the output when the input does not conform to the assumptions. Consequently, one module of a program proven correct may still generate incorrect output if it is given inappropriate input.

Other peripheral issues arise as well. For example, complete proofs of program correctness are at least comparable in size to the programs they are trying to prove correct; one published proof of correctness for a code segment a half page long (19 lines of code) requires 7 pages of proof.[38] Thus, analysts attempting to check the proof are faced with understanding a proof that is at least as complicated as understanding the program. It is possible that automated proof checkers capable of handling million-line proofs will be developed someday, but the correctness of the proof checker may also be suspect.[39]

Moreover, program proofs can be wrong. For example, a program "proved

----------

38. D.I. Good and R.L. London, "Computer Interval Arithmetic: Definition and Proof of Correct Implementation", Journal of the ACM, Volume 17, October 1970, pages 603-612.

39. A.S. Tanenbaum, "In Defense of Program Testing", SIGPLAN Notices, Volume 11(5), 1976, pages 64-68.

correct" [40] was subsequently found to have errors.[41] In another case, an algorithm "proved correct" was in actual day-to-day use without error for a few years, but a counter-example illustrating the an error in the proof was subsequently presented. The inventor of the algorithm and proof was initially unable to find an error in either his own algorithm and proof, _nor_ in the counter-example presented. The "error" turned out to be a crucial ambiguity in the description of the algorithm. Until the counter-example was found, everyone had resolved the ambiguity in a way that preserved the correctness of the algorithm. The person who had discovered the counter-example had resolved the ambiguity in a way that made the algorithm incorrect.[42]

These comments are not meant to imply that proofs of program correctness are useless; they _do_ increase program reliability and help to eliminate errors. Indeed, programs as correct as mathematical proof can make them would be highly desirable would be an enormous step forward in the state of the computing art. However, proofs of program correctness would not guarantee that no errors exist.

----------

40. R.L. London, "Software Reliability Through Proving Programs Correct", _International Symposium on Fault-Tolerant Computing: 1971_, Institute of Electrical and Electronics Engineers, New York City, 1971, IEEE Catalog Number 71 C 6-C, pages 125-129.

41. J.B. Goodenough and S.L. Gerhart, "Toward a Theory of Test Data Selection", IEEE Transactions on Software Engineering, SE-1(2), page 159.

42. Henry Lieberman, M.I.T. Artificial Intelligence Laboratory. Personal communication, April 1985.

### 4.2.2 Empirical Testing

Without formal proofs of consistency and completeness for the entire software system, it is necessary to perform empirical testing under a variety of unexpected circumstances large enough that one can begin to have confidence that it will perform as expected when other unlikely events occur. Note that for most large programs, it is essentially impossible to test the correctness of all possible sequences of program execution, since the number of these sequences grows roughly exponentially with the size of the program involved.

Moreover, when the software system in question is a ballistic missile defense system, it cannot be subjected even once to even a single large-scale empirical test under realistic conditions short of nuclear war. The Fletcher Commission concluded that

> The problem of realistically testing an entire system, end-to-end, has no complete technical solution. The credibility of a deployed system must be established by credible testing of subsystems and partial functions, and by continuous monitoring of its operations and health during peacetime.[43]

The extreme difficulty of large-scale empirical testing for BMD is recognized, but weapons designers generally believe that small-scale testing (e.g., a BMD system intercepting one or two missiles) and the use of simulators to mimic by computer large-scale threats would make actual large-scale tests unnecessary. These two points have some validity, but in

----------

43. Volume 5, FC report, page 7.

the context of a system for BMD they are misleading.

In particular, while successful small-scale tests are necessary for a system to work, they are far from sufficient.[44] The underlying reason is that difficulties of "system integration" (ensuring that all pieces of the system work well together even when operating near the limits of their capabilities) tend to appear only when the system is tested near its limits. For example, during routine non-crisis operations, when the message traffic is low, WWMCCS performs adequately. However, when message traffic has been high, the performance of WWMCCS has suffered. A 1977 exercise in which WWMCCS was connected to the command and control systems of several regional American commands resulted in an average success rate in message transmission of only 38%.[45]

The use of computer-driven simulators can provide some large-scale testing opportunities not otherwise available. Simulators do provide valuable information and confidence in system operation beyond that achievable by small-scale testing alone; nevertheless, simulation testing is limited for two reasons. One is that an actual attack may involve many events occurring in very short times; simulators may not be able to reproduce signatures of multiple events (e.g., explosions, missile intercepts, signal transmissions) quickly enough or with sufficient

----------

44. Dr. Donald Latham, current Deputy Under Secretary of Defense for $C^3I$, has pointed out that "a one-on-one test is going to be meaningless; [the problem is] how do we conduct a many-on-many test?" Quoted in James Canan, "Fast Track for $C^3I$", _Air Force Magazine_, July 1984, page 43.

45. Broad, _Science_, page 1184.

accuracy to test the defensive system subject to all of the timing constraints of an actual large-scale attack. Therefore, simulated events must be "pre-processed", and fed to the defensive system during the course of a system-wide test. Since defense and offense will interact (by assumption), pre-processed data must be corrected for factors that would not be known in advance (e.g., the sudden disappearance of a battle station, the failure of a communications link) [46] leading to a test environment of reduced realism.

The development of massively parallel computing architectures may increase significantly processing speed [47], though a strongly nuclear environment presents simulation developers with important uncertainties about the underlying physical processes that cannot be resolved with confidence in the absence of multi-burst atmospheric testing of nuclear weapons. More importantly, simulators cannot mimic adequately all plausible attacks, because a determined and clever opponent will decide the parameters of an actual attack. Consequently, the confidence resulting from simulated tests would rest on the _assumption_ that those responsible for the simulation can predict the range of tactics that an attacker might use.

Two examples of software failure due to testing limitations will

----------

46. cf., G.W. Suhy, "Digital Interface Simulation for Large System Development", _RCA Engineer_, Volume 27(6), November/December 1982, pages 40-41.

47. Parallel computing involves the partitioning of a computing task into separate sub-tasks on which independent processors can operate, thus reducing the time required to solve the original problem by a factor on the order of the number of processors involved.

illustrate these points. One is the October 1980 "crash" of the national computer network known as the ARPANET (incidentally, a Defense Department funded project); the ARPANET was entirely unusable due to software difficulties, after operating for several years without a network-wide disturbance. The circumstances leading to the crash were rare enough that the problem slipped through the testing cracks.[48]

A second example is the first operational test of the cruiser Ticonderoga carrying the AEGIS air defense system in April 1983, after the cruiser was commissioned for operational use. AEGIS is a battle management system designed to track hundreds of airborne objects in a 300 km radius, and allocate weapons sufficient to destroy about 20 targets within the range of its defensive missiles.[49] A threat environment simulator such as the one described in the previous paragraph is used to debug its real-time software.[50]

In this test, the AEGIS system failed to shoot down six out of sixteen targets due to system failures [51] later associated with faulty software

----------

48. Eric Rosen, "Vulnerabilities of Network Control Protocols: An Example", Software Engineering Notes, Volume 6(1), January 1981, pages 6-8.

49. Norman Polmar, The Ships and Aircraft of the U.S. Navy, 13th Edition, Naval Institute Press, 1984, page 477.

50. Suhy, pages 39-43.

51. Admiral James Watkins, Chief of Naval Operations, Letter to Congressman Denny Smith, February 7, 1984. Reprinted in Department of Defense Authorization for Appropriations for FY 1985, Hearings before the Senate Committee on Armed Services, page 4337.

52. Vice Admiral Robert Walters, Deputy Chief of Naval Operations. Department of Defense Authorization for Appropriations for FY 1985, Hearings before the Senate Committee on Armed Services, page 4379.

[52]; these errors were not caught in small-scale or simulation testing. Moreover, in no individual test were more than three targets presented simultaneously due to test range limitations.[53] Thus, even while engaging a threat much smaller than the design limit of the AEGIS system, the system did not meet its intended performance goals. For an attack whose size was comparable to the design limit, it is reasonable to suppose that the results would have been worse.

The purpose of operational testing is to find errors that will emerge during the initial shake-down of a new system. Errors were found and apparently corrected; in a second set of operational tests in April 1984, none of the problems recurred. As with all other weapons systems, future AEGIS exercises can be expected to uncover additional errors; these will be fixed, and the weapon system will approach its maximum capabilities.

Nevertheless, the testing for AEGIS suffers (and will continue to suffer) from one major deficiency; it has not been, and will not be tested near its saturation limits for the indefinite future; consequently, the performance of AEGIS against saturation-level threats will be unknown in the absence of an actual large attack.

Actual empirical testing of a BMD system will be similar to that of AEGIS with respect to the lack of extensive large-scale test. Even disregarding the staggering cost of a large-scale test, the U.S. will not be _able_ to launch simultaneously a large number of test missiles, because this test would be indistinguishable from a U.S. first strike against the

----------

53. Vice Admiral Robert Walters, op. cit., page 4378.

Soviets. Thus, the first large-scale empirical test for a BMD system would be an actual attack involving many missiles, and it seems implausible that a more complex battle management system for BMD keeping track more targets and operating under tighter timing constraints will have a better performance record than that of AEGIS, whose software is an order of magnitude smaller.

It is important to note significant differences between the testing of an individual weapon platform (such as a Minuteman missile or an F-16 fighter) and the testing of a battle management system such as AEGIS or AWACS. An individual weapon platform can be tested nearly "end-to-end", i.e., with all components in place, and at the margins of their expected performance; indeed, testing sometimes is required to establish the precise performance "envelope". Even so, operational tests (as opposed to developmental tests) for ICBMs take place at the rate of a few per year; these tests exercise the full capability the missile, short of the actual detonation of a nuclear charge. By contrast, small-scale tests of battle management systems do not exercise at the margins of capability. Thus, analogies between platform testing and battle management system testing [54] are potentially misleading.

#### 4.2.3 A Minimum Floor of Reliability

No matter what combinations of analytical and empirical techniques are

----------

54. cf., Testimony of General James Abrahmson, in Hearings on Department of Defense Appropriations for FY 1985, House Appropriations Committee, Subcommittee on Defense, Part 5, page 707.

used to develop confidence in reliability, decision-makers must decide upon the particular standard or standards of reliability to which a software system should be held; a software system that is "reliable" to a certain extent may well be suitable for one application but not for another, more demanding application. In the case of the Strategic Defense Initiative, it is reasonable to suggest that software for a defensive system should be at least as reliable as that of the offensive system that the defense will replace.

The software responsible for controlling strategic offensive weapons has a variety of functions. For example, software can improve the efficiency of a weapon (e.g, as a component of the navigational computers that guide a ballistic missile warhead to its intended target). Software is also an integral part of the command and control system that enforces positive control on the strategic forces, i.e., the requirement that the nuclear forces should be used only when a properly authorized message is received.

However, failure of the first function is qualitatively different from failure of the second. A failure in the first function is a failure of efficiency (e.g., a warhead might not land as near to its target as intended). A failure in the second function (command and control) might well be catastrophic (e.g., an unauthorized party able to penetrate the command and control system might be able to start World War III). Consequently, the reliability requirements associated with command and control are much greater than those associated with efficiency.

Are these distinctions equally applicable to software controlling a defensive system? For software in the offensive system, an efficiency

failure leads to inefficient (but authorized) use of weapons, whereas a command and control failure leads to the use of weapons where no weapons should be used at all; thus, command and control failures and efficiency failures drive in different directions. By contrast, the parameters of the defensive system are different. An efficiency failure leads to fewer than expected missiles being destroyed. The most critical command and control failure of a defensive system would involve not an unauthorized activation of the defensive system (which at worst would lead to the destruction of missiles that should not have been destroyed), but a _failure_ of the system to activate when needed, resulting in _many_ fewer than expected missiles being destroyed. Thus, command and control failures and efficiency failures drive in the same direction, and are simply differences of magnitude (rather than differences of sign).

Thus, for software controlling offensive systems, the central reliability issue revolves around security: enforcing positive control on weapons whose unauthorized use would be catastrophic. For software controlling defensive systems, the central reliability issue revolves around performance; the actual ability of the system to destroy missiles, and whose catastrophic non-use when authorized would be merely a special case of loss of efficiency.

These two reliability issues are qualitatively different in complexity. System security is a problem that can be specified in relatively narrow and well-defined terms; consequently, theoretical proofs of program correctness are applicable and can indeed increase the reliability (security) of such software, just as they have increased the security of various multi-user

computer operating systems.[55]

The performance of defensive systems is not nearly so well-defined. Indeed, it depends on choices of tactical doctrine and rules of engagement that different reasonable individuals might make differently. Consequently, theoretical proofs of correctness are far less applicable, and a defensive system must rely much more heavily on empirical testing of actual hardware. Indeed, since the performance issue for defense subsumes both efficiency and command/control functions, all parts of a defensive system must perform as expected with the reliability required of the most critical parts of the system, or else enormous redundancy and excess capability must be built into different parts of the system (e.g., the terminal defense layer might have to be built to handle the utter failure of boost-phase and mid-course intercepts).

These considerations apparently underlie the requirement that the entire system for comprehensive BMD must function with extraordinary reliability. For example, James Fletcher, Director of the DTST, has noted that

> Battle management for a multi-layered defense is clearly one of the largest software problems ever tackled, requiring an enormous and *error-free* program.[56] (emphasis added)

----------

55. Peter Neumann, "Experiences with Formality in Software Development", in Theory and Practice of Software Technology, D. Ferrari et al (eds), North Holland Publishing Company, 1983, pages 203-219.

56. Fletcher, page 21.

## 4.3 Software Maintenance

Software, once delivered, is not static, though in principle it could be. Testing must continue in order to discover errors. Any errors discovered after delivery must be eliminated. New software capabilities are added in response to the needs of the users. New hardware is added to the system, for which new controlling software must be developed and integrated into the existing software system. These upgrades must be transmitted to a system that must run 24 hours a day in real time, with the possibility that adding the upgrade might interfere with the functioning of the original software. New staff members must be introduced to the project. In short, software maintenance is a large fraction of the total life-cycle cost of a project; experience with Air Force command and control software suggests that this fraction is about 70%.[57]

The environment in which the maintenance of large and complex software systems must take place is described in the following quotation:

> [A programmer faced with a complex program that does not produce the desired results] would have the options of thoroughly understanding the existing program and 'really fixing' the trouble, or of entering a new advice statement describing what he imagines the defective situation to be... When a program grows in power by an evolution of partially understood patches and fixes, the programmer begins to lose track of internal details, loses his ability to predict what will happen, begins to hope instead of know, and watches the results as though the program were an individual whose range of behavior is uncertain. This is already true in some big programs, [and] it will soon be much more acute. ... [Programs] will be developed and modified by several programmers, each [acting] independently... The program will

----------

57. Boehm, Figure 30-1, page 534.

> grow in effectiveness, but no one of the programmers will understand it all. (Of course, this won't always be successful -- the interactions might make it get worse, and no one might be able to fix it again!)[58]

Such an environment is not conducive to long-term maintenance of complex software systems. In the discussion below, some of the consequences of this environment are discussed.

### 4.3.1 Error Removal

Nearly all software contains errors, some of which are critical and some of which are not. Critical errors must be removed, and for large systems, this task is difficult. During software development (but before system testing), the average program contains ten to one hundred errors per thousand lines.[59] A linear extrapolation suggests that the BMD battle management system will contain between 100,000 and 1,000,000 errors. Some of these errors will be easy to find and fix (e.g., a minus sign omitted, a set of parentheses left out), and thus relatively easy to fix. (This is not to say that such errors are not significant; the Mariner 1 space probe failed due to a single incorrect symbol in one line of its control program, which contained only a few thousand lines.[60]) Others may be conceptual, requiring enormous effort even to describe.

----------

58. Marvin Minsky, "Why Programming is a Good Medium for Expressing Poorly Understood and Sloppily Formulated Ideas", in Design and Planning 2, Martin Krauper and Peter Seitz, Hastings House, 1967, page 121. A personal communication with Professor Minsky confirms that he still holds this view.

59. Ivars Peterson, "Superweapon Software Woes", Science News, Volume 123, May 14, 1983, page 312.

60. Henry Tropp, "Fortran Anecdotes", Annals of the History of Computing, Volume 6(1), page 61.

Programs contain two types of error. One type of error is structural; for example, the design of a program might provide no guidance under a set of unexpected circumstances. A second type of error is random; for example, a program might fail because of a typographical error, or an algorithm for computing some function may have been implemented improperly.

It is possible to guard against random error by exploiting their statistical nature by "seeding" a program with known errors. Debugging is then assigned to individuals with no previous knowledge of these errors, and if their efforts uncover all of the known errors, then it is likely that they will have found most of the unknown errors in doing do.[61] However, "seeding" is inherently a local process that does not introduce large-scale changes in the system; structural errors requiring large-scale changes are most likely to be errors of planning or design.

However, even random errors are difficult to eliminate in large real-time software systems. One major reason is that a large system may exhibit an error under one set of circumstances at a given instant, and not exhibit it at another instant under apparently "identical" circumstances. Indeed, the precise set of circumstances that gives rise to a given error may not recur even with people _trying_ to make it recur.

It is useful to draw an analogy to a coin toss. Assuming that all of the parameters that determine the outcome of a coin toss (e.g., a coin's

----------

61. Harlan Mills, "On the Statistical Validation of Computer Programs", IBM Federal Systems Division, Gaithersburg, MD, 1972. FSC-72-6015.

initial condition, the impact that drives it into the air, local atmospheric characteristics) were duplicated with sufficiently high accuracy from toss to toss, the outcome of the various tosses would be identical. However, in practice, it is impossible to set up successive coin tosses so precisely. Instead, one resorts to probabilistic statements about the outcomes, based on a few parameters known to be central in determining the outcome (e.g., if the coin is weighted, if it has two heads). The use of probability enables us to replace complex calculations and the uncertainties about the details of the parameter list with (predictable) uncertainty about the outcome.

Similarly, the detailed sequence in which computer instructions are executed is determined very strongly by many parameters (e.g., arrival times of sensor inputs, particular data previously stored in memory; Figure 9 provides an example.). If these determinants of a program's behavior could be reproduced to sufficient accuracy, the error could be located and then eliminated. However, for large real-time software systems, such accuracy is often not possible. Since debugging often requires a clear understanding of the precise circumstances leading to an error, it may be possible to identify errors only in a probabilistic sense.

There is one crucial point at which the analogy to coin-tossing fails. There is consensus on the few significant variables that determine the probabilistic outcome of a coin toss. No such consensus exists for determining even the probabilistic outcome of computer programs; indeed, one reason for using computer programs in the first place is to determine how many variables interact to lead to a given result.

A final complication is that an attempt to remove a given error may not be entirely successful. Indeed, it may eliminate the error at the expense of introducing one or more additional errors.[62] The probability of such an occurrence varies, but estimates range from 15 to 50 percent.[63] Moreover, the majority of software design errors take a long time to appear (after the software is initially released), and therefore are unlikely to be discovered in anything but extensive operational use. Experience with large control programs (between 100,000 and 2,000,000 lines) suggests that the chance of introducing a severe error during the correction of original errors is large enough that only a small fraction of the original errors should be corrected, accepting the consequences of the uncorrected but relatively infrequent errors.[64]

These points are well illustrated by the the failure of the first operational launch attempt of the Space Shuttle, whose real-time operating software is about 500,000 lines. The software error responsible for the failed launch would reveal itself -- on average -- once in 67 times.[65]

----------

62. M.L. Shooman and S. Natarajan, "Effect of Manpower Deployment and Bug Generation on Software Error Models", in Computer Software Engineering, Jerome Fox (ed.), Polytechnic Press, 1976, page 155-170.

63. E.N. Adams, "Optimizing Preventing Service of Software Products", IBM Journal of Research and Development, Volume 28(1), January 1984, page 8. See also Brooks, page 122.

64. Adams, page 12.

65. Alfred Spector and David Gifford, "Case Study: The Space Shuttle Primary Computer System", Communications of the ACM, Volume 27(9), September 1984, page 897.

This error was itself introduced in the fixing of another problem two years earlier.[66]

### 4.3.2 Other Maintenance Issues

A second maintenance issue involves staff turnover on long projects. For example, assuming the DTST estimate is low by only a factor of two, even a very optimistic software development project would involve 3000 programmers and analysts for about 10 years, long compared to the characteristic promotion or moving time for most people.[67] Significant staff turnover reduces the institutional memory of the project with the result that many essential details (such as documenting a particular module or actually performing a software update on a given battle station) may be lost or forgotten during staff transitions. One tragic example of management error is the Air New Zealand airliner that crashed into an Antarctic mountain because its crew had not been informed that the input data to its navigational computer, describing its flight plan, had been changed.[68]

A third maintenance issue is that nearly every large-scale weapon system that has been deployed has undergone significant change after its first

----------

66. John Garman, "The 'Bug' Heard 'Round The World", Software Engineering Notes, Volume 6(5), October 1981, page 8-9.

67. Note also that people and years are interchangeable only when they can work independently. For a project as complex as that of comprehensive BMD, this assumption is hardly warranted.

68. David Brown, "Incorrect Computer Route Cited in Antarctic Crash", Aviation Week and Space Technology, May 18, 1981, page 26.

deployment: components are changed in order to correct problems discovered in operational use or realistic rehearsals, or to improve combat performance. Thus, at any particular time, a defense against ballistic missiles that involved (for example) 150 laser battle stations in space might well include different models of the "same" battle station; on June 27, 2016, the U.S. might have deployed 140 Mark 3-A BMD lasers, and 10 Mark 3-B BMD lasers, with the expectation that in three years, all Mark 3-A lasers would be upgraded to Mark 3-B lasers. A conceivable difference between these two models might be the more precise mechanisms for moving the laser mirror in the Mark 3-B model. This "minor" difference might well involve changes in the software controlling the mirror moving mechanisms. Therefore, two different versions of the software would have to be maintained. Maintenance of multiple functional versions of the "same" software can be a logistical nightmare, as anyone who has tried to write simultaneously different versions of the "same" article can testify. It is not difficult to imagine that a change (e.g., to remove an error) intended for the Mark 3-A laser might inadvertently be made instead on software for the Mark 3-B laser. Moreover, to ensure reliability, the addition of a new component or the removal of an old one would require the re-validation of the complete software system against the same battery of tests which validated the original configuration. A current example of this difficulty is illustrated by the capability of the high-speed anti-radiation missile (HARM) to be launched at supersonic speeds by the F-4G Wild Weasel but not by the F/A-18 Hornet; this difficulty is caused by software.[69]

----------

69. Jane's Defence Weekly, Volume 3(21), 25 May 1985, page 887.

A fourth maintenance issue concerns the actual replacement of one version of software with a later version. Since the defensive software system must be functional 24 hours a day, how does one update it while it is running? Though this problem can be solved, it is not easy; communications with the Viking 1 lander on Mars were unexpectedly lost, apparently due to a software change transmitted to the lander that was accidentally overlaid upon some mission-critical software already in the lander's computer.[70] This issue is not critical for a software update to an individual battle station, but it is enormously important for the space-based portions of the global battle management system. Also, how does one transmit the update when the Soviets may be attempting to interfere with the transmission from earth to space? Indeed, a satellite jammer a few kilometers from a defensive satellite (battle station or command and control or sensor) has a relative power advantage over ground transmitting stations of at least 60 dB (a factor of a million) due to its proximity, perhaps enabling it to drown out any transmission from earth. A related issue is that of guaranteeing the integrity of programs installed remotely; the Soviets must not be able to insert a program of their own into the memories of an American computing system.

### 4.4 Comparisons to Large-Scale Hardware Projects

The yet-to-be-designed battle management system for BMD can also be compared to other engineered entities whose safety, reliability, and

----------

70. Bruce Smith, "JPL Tries to Revive Link with Viking 1", Aviation Week and Space Technology, April 4, 1983, Volume 118(14), page 16.

performance records are better known. In particular, the effort required to _design_ any engineered entity is an indirect metric for the complexity (and therefore the difficulty) of the design task. This leads to an analogy between hardware design flaws that result in unexpectedly poor performance records and software design flaws that cause computer systems to function more poorly than expected. General support for this analogy is found in the STARS report:

> The cost and time required to _design_ a software change is comparable to the cost and time to design a hardware change, since both are human-intensive, intellectual tasks of comparable complexity...[71]

The COCOMO model explicitly provides breakdowns of effort by phase; for the nominal case described above, COCOMO predicts that about 16,000 man-years setting system requirements and planning system design. By comparison, the total design time of a nuclear reactor is on the order of several hundred man-years. Specifically, Bowers lists for "home office non-manual man-hours" in building pre-1976 reactors 2.2 million man-hours (about 1200 man-years) for plants requiring a total of about 9100 man-years to bring on-line.[72]

Design efforts of this magnitude have resulted in nuclear power plants that do not live up to their design promises. For example, in 1974, the Energy Research and Development Administration suggested that availability factors of at least 75% were a reasonable basis for planning nuclear power

----------

71. STARS, page 4.

72. H.I. Bowers, et al, _Trends in Nuclear Power Plant Capital-Investment Cost Estimates: 1976-1982_, Nuclear Regulatory Commission, NUREG/CR-3500, August 1983, page 12.

generation facilities.[73] Data presented in 1978 suggest that actual availability factors for large nuclear power plants averaged about 69% between 1974 and 1978.[74]

Complex electronic systems, in particular aircraft radar subsystems, are another class of engineered entities whose reliability can be examined. Detailed statistics on the design of these systems are not available, but an upper bound would be hundreds of man-years. In 1974, the General Accounting Office described for these systems the difference between actual and specified (i.e., design) mean times to failure. In six of the ten radars examined, the ratio of the former to the latter was less than 33%; in no case was the ratio better than one.[75]

More generally, over-optimism pervades most predictions regarding military hardware. For example, an Air Force report cited by the General Accounting Office states that "reliability predictions tend to be generally optimistic by a factor of two to six, but sometimes for substantially greater factors."[76]

Is it really reasonable to compare the design of hardware to the design

----------

73. The Nuclear Industry, Office of Industry Relations, Energy Research and Development Administration, 1974, WASH 1174-74, page 21

74. A.D. Rossin and T.A. Rieck, "The Economics of Nuclear Power", Science, Volume 201, 18 August 1978, Table 5, page 585. For additional commentary, see also National Academy of Sciences, Energy in Transition: 1985-2010, W.H. Freeman, 1979, pages 266-267.

75. General Accounting Office letter to Senator Mike Gravel. Untitled, circa 1974-5, Accession Number B-164105, page 5.

76. Cited in GAO Letter to Senator Gravel.

of software? If anything, this comparison is more favorable to software than is warranted, for a variety of reasons.[77] Specifically, hardware is often assembled from standard (and debugged!) components; software almost never is. Indeed, it is often so difficult to adapt existing software to the required task that it is easier simply to start from scratch. This factor contributes significantly to the empirical observation that the number of different ways of implementing a given design are much smaller for hardware than for software. In addition, the phases of hardware design and construction are naturally sequential (or cyclic); design must be frozen when construction begins. By contrast, software has fewer "natural" breakpoints, and human discipline, rather than nature, must impose design freezes.

The much larger system design effort for a BMD battle management system suggests that the system will be conceptually more complex than nuclear reactors or aircraft radars. Furthermore, BMD computer technologies will be pushing the state-of-the-art; by contrast, radar and nuclear reactor technologies are relatively mature and have long histories of exercise under actual operating conditions; these should in principle make them easier design media in which to work.

These particular examples are not meant to suggest that it is impossible for large or complex systems to function as intended. Two good examples of large systems that function reliably despite their complexity are the

----------

77. Some of the following points are taken from W. Earl Boebert, "Software Quality through Software Management", in Software Quality Management, John Cooper and Matthew Fisher (eds.), Petrocelli Books, 1979, pages 18-19.

pre-divestiture long-distance telephone network of AT&T and the life-critical air traffic control (ATC) system. Even the large software systems of WWMCCS and the Space Shuttle "work" well enough that they deliver acceptable levels of performance, and these software systems are within an order of magnitude of the lowest projected size for a comprehensive BMD system. Computer simulations of the aerodynamic behavior of modern commercial airliners are good enough to provide many people with confidence in the adequacy and safety of its design.

However, in the case of the design of commercial airliners, no one would suggest that the first flight of an airplane designed without resort to wind-tunnel testing should be involve a full load of passengers. Indeed, design simulators in widespread use today are very tightly coupled to empirically validated standards; a simulator to test the behavior of a BMD could not be similarly validated.

For the the telephone network and the ATC system, the functions that these systems are intended to perform were and are well-understood; indeed, human beings understood them well enough that they could perform these tasks without computing assistance. In addition, these systems have been (and are currently) subject to continuous test in actual use over decades under an enormously varied range of parameters, with the consequent exposure of defects that would be unlikely to appear with less intensive testing. In the case of WWMCCS and the Space shuttle, the requirements for performance and reliability of these systems are modest compared to the requirements for comprehensive BMD.

Moreover, the initial design of the telephone network contained one very

severe design flaw: the tone-signaling method used to route long-distance calls was vulnerable to "phone hackers" using "tone boxes" that duplicated in an unauthorized manner the tones that the network used, allowing them to make free long-distance calls; only recently has this initially unanticipated threat has been reduced. As for the air traffic control problem, its time scales are fortunately long enough that human intervention to correct an immediate problem is possible when the system fails.

By contrast, a BMD system must perform tasks with which no human being has any experience, and it would not be undergoing continuous test; rather, it would be quiescent most of the time, except during those rare occasions when it would have to work with near-perfection without human intervention.

### 4.5 Management

A product that takes fifty thousand person-years to deliver presents a considerable management challenge. Indeed, the full Fletcher Commission report points out that "there is little that we can do to insure proper management of the software development process".[78]

#### 4.5.1 Changing Requirements

The long time scales of large programming projects complicate the development task enormously. When development time scales are long,

----------

78. Volume 5, FC report, page 45.

requirements are likely to change. This problem is accentuated by the propensity of the military for the best-available performance. As hardware matures, mission requirements will invariably undergo significant change; this will require significant software changes. Therefore, either the development of battle management software must be deferred until system requirements are relatively well-defined, or enormous effort will be required to make changes to existing programs.[79] Even today, two thirds of all programming effort is consumed by debugging and alterations due to changing requirements.[80]

### 4.5.2 Documentation

Documentation of a large and demanding project is extraordinarily difficult to manage. In particular, most first-rate programmers find documentation an enormous burden.[81] Indeed, the sociology of the programming community is such that documentation is in the "necessary evil" category; thus, documentation is a task usually left to technical writers who lack first-hand familiarity with the software systems in question. Often, the result is a mismatch between what the documentation says and what the systems actually do.

----------

79. Errors in conceptual design are over one hundred times more expensive to fix than are errors in coding once the code has been written; see Boehm, page 40.

80. Joel Birnbaum, "Computers: A Survey of Trends and Limitations", *Science*, Volume 215, 12 February 1982, page 760.

81. cf., Gerald Weinberg, *The Psychology of Computer Programming*, Van Nostrand, 1971, page 262. Little has changed since then.

### 4.5.3 Ensuring the Actual Use of Good Development Practices

The documentation problem generalizes to essentially all aspects of software engineering. Specifically, the fact that new software development techniques and tools have been developed in the past fifteen years does not mean they will be used. For example, one group of analysts notes that

> [In an examination of] actual software products and their documentation, few showed any use of of the [recently developed good software engineering] ideas and no successful product appeared to have been designed by consistent application of the principles touted at conferences and in journals. The ideas appeared to be easier to write about than to use.[82]

### 4.5.4 Maintenance of Security and its Implications

Finally, a programming team of 3000 people might include individuals whose motives are not friendly to the U.S. Programming is fundamentally a solitary activity; actual programs are not routinely exposed to public scrutiny in the same way that outputs from these programs are, and the possibility that a program module might contain some hidden "time bomb" that would cause the software to fail at a critical moment cannot be ignored.

The usual response to this possibility would be to classify the software development project, requiring security clearance of all programmers and analysts associated with the project. Security clearances generally

----------

82. D.L. Parnas, P.C. Clements, and D.M. Weiss, "The Modular Structure of Complex Systems", Proceedings of the 7th International Conference on Software Engineering, Orlando, FL 1984, page 408.

involve a demonstrated "need to know". If one can demonstrate the "need to know", one can also probably anticipate a potentially dangerous interaction that might arise between different parts of the system; if one cannot demonstrate the "need to know", potentially dangerous interactions will remain hidden.

Systems requiring stringent security during development add approximately 5-10% to total system development effort.[83]

## 5 -- The Human Interface

The short time scale of the BMD battle management problem has raises two issues that relate software development to the role that human beings play in the system. One issue is that the software development process will no longer require as much attention to the man-machine interface as other software systems have required. The effect of this difference is that the human environment in which a BMD system must function is far more predictable, and therefore less effort must be made to build the software to withstand human mistakes in its tactical operation. On the other hand, the complexity and size of the software require that it must be designed to cope with a less predictable computational environment, in which software faults must not be able to degrade significantly the performance of the system.

----------

83. Boehm, page 490.

The second issue is that the times available for target engagements are so short that they effectively preclude the possibility that human beings can take over system functions should the system fail. Rather, the only role that one could even imagine for human beings once the engagement decision had been made would be that of *correcting* a software failure; obviously, this would be difficult under the pressure of an attack in progress.

## 6 -- Hardware/Software Interaction

Up to this point, I have concentrated mostly on software issues alone. Thus, I have implicitly assumed that all of the hardware involved works perfectly. For a complex project that takes many years to complete, this assumption is entirely unwarranted. Indeed, the advances in electronic hardware described in the Introduction virtually guarantee that the hardware with which programmers will be working will change over the course of the project. In this section, I will touch briefly on some hardware/software concerns.

Hardware changes during software development confound the process of removing errors from software, because incorrect output might have been caused by incorrect programming, by malfunctioning hardware, or some combination of the two. Computer hardware, especially when it is new, may not perform as intended. For example, an early computer changed "01" to "10" improperly only under certain very specific and rare conditions due to

a wiring error; this error was discovered only years later by programmers.[84]

Changes in computer hardware also means that some programs that ran on the old computers must be changed in order to run on the new computers. Compilers (i.e., programs that transform the "source" language in which programmers work into instructions that the machine can execute directly) are particularly sensitive to new computing hardware. In particular, since most source languages are defined only in natural English (and are thus likely to be ambiguous), different compilers can implement the same English description in different ways. These differences can introduce subtle differences that are difficult to notice, let alone correct, and yet they may have crucial significance in some application. Even Ada, a high-level computer language specifically designed for portability among different machines, is not entirely transferable; indeed, the Ada reference manual specifically includes a chapter on implementation-dependent functions, and leaves many ambiguities in the language definition to be resolved by the particular implementation.

Finally, computers that control weapons usually accept inputs from several sensors. These sensors are often delicate, requiring large amounts of maintenance if they are to provide information to the computers that properly reflects the situation in which the weapon is operating. When the combination of computer and sensor is not operating within tolerances, the results can be disastrous. For example, the flight control computers on

----------

84. Peter Neumann, SRI International, Menlo Park, California, personal communication, January 1985.

the F-4E and F-111F aircraft are programmed to take into account the presence or absence of an aircraft-carried pod used during ground attacks. Without constant maintenance, the computers provide the "with pod" solution even when no pod is present. The result is that the accuracy of weapons delivery drops essentially to zero, since the absence of the pod changes significantly the aerodynamic characteristics of the airplane.[85]

## 7 -- Alternatives to a Conventional Programming Approach

If conventional programming techniques are inadequate to the development of a 10,000,000 line system, are there alternatives? Two have been suggested: expert systems and automatic programming.

### 7.1 Expert Systems

Expert systems are computer systems designed to embody the knowledge of human experts as computer programs, albeit in restricted domains. A brief description of expert systems is contained in the Strategic Computing Initiative recently announced by DARPA:

> The term "expert system" describes the codification of any process that people use to reason, plan, or make decisions as a set of computer rules, ...[involving] a detailed description of the precise thought processes used.[86]

----------

85. Joshua Epstein, Measuring Military Power: The Soviet Air Threat to Europe, Princeton University Press, 1984, page 166.

86. Defense Advanced Projects Research Agency, Strategic Computing, 28 October 1983, page 7. Hereafter referred to as SC.

To date, expert system research has focused on well-defined areas such as biochemistry and internal medicine. However, many believe that expert systems have matured to the point that they can be used over a much broader set of applications. Indeed, one specific focus of the DARPA strategic computing initiative is the implementation of an expert system to assist in battle management for carrier battle groups, with software spin-offs from this demonstration project that would strongly support battle management for missile defense.[87]

This point of view neglects the fact that human expertise (the ultimate source of knowledge to be included in any expert system) is based on human experience. No one has "expert knowledge based on experience" concerning the battle management task for missile defense. Moreover, even ostensible experts are often confused when they encounter radically unfamiliar situations, and make judgments based on past experience that may be entirely incorrect within the context of the new situation.

In addition, expert systems often use informal reasoning procedures whose validity cannot be proven or disproven using standard tools of formal logic. (For example, the converse of a true statement is not necessarily true, but an expert system might use it anyway as a heuristic "rule-of-thumb".) Therefore, the structure of the expert system does not require of its developers the discipline to check extensively for interactions. Consequently, expert system developers are strongly motivated to program by "giving advice" to the system that "seems right"

----------

87. SC, page 27-29.

under the circumstances; this advice may lead to local improvements to performance without revealing deeply embedded conceptual contradictions on a system-wide scale that might cause catastrophic failure at some (unknowable) point in the future.

### 7.2 Automatic Programming

A second allegedly promising technology is automatic programming: the use of programs to write other (error-free) executable programs. For example, Fletcher states that battle management software for BMD "will require the development of techniques for automated software development -- in essence, a computer that can write another computer program."[88] Major Simon Worden of the Strategic Defense Initiative Office has stated that "A human programmer can't do this. We're going to be developing new artificial intelligence (AI) systems to write the software. Of course, you have to debug any program. That would have to be AI too."[89] More generally, automatic programming systems are said to enable people with little or no programming expertise to program computers, and to enable professional programmers to generate code (translate ideas into computer language) with much greater ease than if they had to generate the code themselves. Indeed, some in the military software community predict that a person will be able to "talk directly to the machine in natural language, make a few sketches, wave his arms a bit, answer a few questions, say yes

----------

88. Fletcher, page 25.

89. quoted in Washington Post, March 3, 1985, page 7.

or no a few times, and have the program tested and documented in the morning."[90]

This optimism is not justified. In particular, automatic programming systems do not (indeed, should not) relieve the human burden of decision-making imposed by the the planning and design process, which generate about two thirds of all errors in computer software.[91] Rather, their primary functions are to facilitate the implementation of design specifications into actual code and modification of currently existing code. Even the most optimistic assessments of automatic programming acknowledge that the decision-making aspects of the system design process cannot be automated to any significant degree.[92]

## 8 -- Soviet Responses

The Soviets would attempt to thwart an American BMD if they decide to attack, and it is they who choose the time and nature of attack. Therefore, in case of attack there will be no "time-outs" available (as there are in the space program) during which software difficulties can be fixed; the software must work properly the *first* time it is called into

----------

90. cf., Robert Everett, "Command, Control, and Communications", *The Bridge*, National Academy of Engineering, Volume 11(2), Summer 1981.

91. Peter Wegner and Robert Grafton, "Perspective on Software Engineering", *Naval Research Reviews*, 1982/Four, page 18.

92. cf., C. Green, et al, *Report on a Knowledge-Based Software Assistant*, Rome Air Development Center, RADC-TR-83-195, August 1983, pages 25-32.

action.

A second problem is that proposed defensive systems will require an enormous amount of data on Soviet booster, post-boost vehicle and re-entry vehicle characteristics.[93] While it is now possible to collect such data from observing Soviet missile testing, there is no reason to suppose that, facing an American BMD, the Soviets would not attempt to camouflage these characteristics during test or operation.

Finally, even empirical testing against small-scale threats might make a BMD system vulnerable to Soviet countermeasures. For example, the Soviets would be able to observe every test conducted against an actual missile launch. These observations would allow the Soviets to attack in a way that is different from what they have seen us test. Even if they did not, knowledge of significantly new Soviet tactics or hardware could force us to re-program the battle management system to meet the new threat. The history of computer control programs known as operating systems (the closest civilian counterpart to battle management systems in widespread use) suggests that these accumulated changes, over time, would probably cause many unexpected interactions that could eventually necessitate a total "ground-up" system redesign.[94]

----------

93. Dr. Latham, as quoted in Canan, page 44.

94. Brooks, page 123.

## 9 -- Possible BMD Software Failures

How might the software for BMD fail? In order to give some concrete substance to the previous discussion, it is useful to describe specific but hypothetical software errors.

1. In the aftermath of an American-Soviet crisis during which all all defensive layers were activated, the boost-phase intercept layer is turned off. However, due to a software update performed during routine maintenance, one cluster of boost-phase interceptors never receives the disable command, and remains activated, unknown to American authorities. The Soviets launch a manned space vehicle to repair a satellite disabled during the crisis, using a booster with a boost-phase infra-red signature similar to that of the SS-18. A mistakenly enabled American battle station in space interprets this launch as the launch of an SS-18. It intercepts the booster during atmospheric burn, causing the loss of the Soviet flight crew on board.

2. During the early phases of a nuclear war, American and Soviet leaders agree to a cease-fire. American authorities are aware that one U.S. missile submarine is unable to receive the cease-fire order, but proceed anyway, believing that the missile defense satellites were designed to shoot down inadvertent launches. Only after American missiles appear is it realized that no one had anticipated the possibility that an American missile defense might have to shoot down American missiles. American missiles explode on Soviet territory after the cease-fire is declared, and the Soviets accuse the U.S. of bad faith.

3. Sensors that feed information to the boost-phase intercept screen detect the launch of a hundred ICBMs from a Soviet missile field. One cluster of laser battle stations successfully intercepts many missiles, but while doing so moves beyond its effective kill range against the remaining missiles. The design of the software designed to "hand off" targets from one cluster of battle stations to another has ignored the possibility that a battle station might move out of range *during* an engagement, and many more missiles than expected penetrate the boost-phase screen.

4. During a massive Soviet ICBM attack, the boost-phase intercept performs as advertised, destroying 90% of all launched boosters. However, some warheads explode as the boosters carrying them are destroyed. These detonations generate thermal pulses that

temporarily disable the infrared sensors searching for the surviving post-boost vehicles (PBVs). These sensors take time to recover, but eventually they find the PBVs that survive. Data transmission to the PBV interceptors occurs, but due to the much shorter time now available for post-boost intercept, some of the data on these vehicles are lost in overflowing a data reception buffer. Several PBVs are effectively lost, and since the sensors in later phases have been designed to depend primarily on hand-off data from the post-boost phase, the warheads on most of the lost PBVs penetrate to their targets.

5. Two waves of attacking ICBMs separated by five minutes cause two separate "boost-phase enable" signals to be sent. The first signal functions properly, and boosters are shot down for five minutes. However, the arrival of the second signal causes a partial reset of the boost-phase intercept software. The boost-phase software was not designed to handle a second enabling signal, and the partial re-initialization of the system causes all boost-phase computing to stop.

6. Two terminal interceptors are launched to intercept a re-entry vehicle. A ground-based tracking radar tracks both the RV and the interceptors, and an illuminating radar provides a signal that bounces off the RV, allowing the interceptors to home on the RV. However, when two interceptors are near the RV, the illuminating radar illuminates both the RV and the interceptors, and each interceptor homes on the radar return from the other interceptor. This occurs because the interceptors are closer to each other than they are to the RV, and the software responsible for terminal homing assumes that the proper target is indicated by the strongest radar return.

7. Dozens of missiles are simultaneously launched from two widely separated points A and B when one cluster of battle stations is approximately equidistant from each point. As the engagement proceeds, this cluster moves closer to B and farther from A, as a second cluster moves closer to A. The first cluster attempts to pass its track data on the missiles fired from point A to the second cluster, but the local sensors associated with the second cluster generate track data on their own. The data management programs have not considered the possibility that target hand-off might occur simultaneously with the generation of new track data. The result is a data base containing two separate entries for each missile. The second cluster destroys a missile and removes one entry from the data base. It searches the data base for another target, and finds the second entry for the now-destroyed missile. It attempts to shoot down the non-existent missile, but without sensor data indicating a change in the situation, it assumes that it has missed, and tries again. More missiles than expected pass through the boost-phase interception screen.

8. A battle station with an electromagnetic rail-gun and a large number of projectiles has successfully engaged during operational testing

consisting of one or two missiles per test. The Soviets then launch a large scale attack, and the battle station must engage many missiles during boost-phase. The first projectiles hit their targets, but an analyst designing the aiming algorithms for the gun has omitted the fact that the station lightens as more projectiles are fired. Later projectiles are fired from a station that is considerably lighter than when the engagement first began, and these projectiles do not reach their targets in time due to their reduced speed.

None of these specific errors are likely, since their description here illustrates that they can be anticipated. Moreover, many of these situations can be tested in simulation. However, one must have thought of testing these particular situations. The problem is not "How likely is it that the system will have a particular error in it?" but rather "How likely is it that the system will have any one of millions of potential errors?" The primary errors of concern are those that remain unimagined -- the "unknown unknowns".

In addition, it is worth noting that the errors described above are all systematic errors, i.e., they affect in the same way all engagements that fit the descriptions above. Thus, these are not errors that cannot be taken into account when defensive efficiencies are calculated, and consequently represent a "leakage" through the defense that is greater than expected.

## 10 -- Less-Than-Comprehensive BMD and Other Military Missions

The discussion to this point has emphasized the development of BMD software sufficiently reliable to eliminate the threat of nuclear ballistic missiles against all U.S. assets, military and civilian. The difficulty of

performing the comprehensive BMD mission arises from four considerations: the catastrophic consequences should even one missile leak through the defense (and hence the need for utter reliability), the short time scale of the ballistic missile threat (and hence the need for computers to take over functions that in other systems humans would perform), the complexity of the problem (and hence the tens of millions of lines of programming needed), and the inability to test empirically the system even once under realistic conditions (and hence the requirement that the system work properly the very first time it is called into action).

However, two less demanding goals have been widely articulated by proponents of the SDI; these are the strengthening of deterrence and the defense of strategic military assets.[95] System performance requirements for each of these goals are far less stringent than in the case of comprehensive BMD. What is the proper standard of reliability for less demanding goals?

The argument for the strengthening of deterrence rests on the fact that in the absence of realistic empirical testing, no one will know with certainty how a BMD system would perform during an actual large-scale attack. This ignorance would introduce substantial uncertainty for a Soviet leader contemplating an attack, and a prudent Soviet leader would thereby be deterred from attack. In this case, the requirement for BMD testing is that the U.S. conduct enough tests to persuade the _Soviets_ that

----------

95. Strengthening deterrence and other intermediate military goals are discussed in Fred Hoffman, "Ballistic Missile Defenses and U.S. National Security", Summary Report, prepared by the Future Security Strategy Study, October 1983, pages 2-3. Sponsored by the Institute for Defense Analyses.

a BMD system would have a reasonable chance of succeeding. This is a much less demanding standard, since conservative Soviet planners would assign to an American BMD system much higher performance and reliability ratings on the basis of a given set of American tests than would conservative American planners given the same set of tests.

In the case of intermediate military goals such as the defense of strategic military assets, the performance and reliability goals for the BMD system are also significantly relaxed. For example, consider the defense of fixed land-based ICBM silos. A defense that consistently destroys only 50% of all attacking warheads would still leave an enormously powerful retaliatory force. Even a defense that failed catastrophically (allowing attacking missiles to penetrate as though no BMD system were in place at all) would most likely leave intact a significant number of warheads, due to missile failures in launch and organizational and operational difficulties in the coordination of the attack.[96]

In the context of these less demanding goals, confidence in the reliability of the system need not be as high as in the case of comprehensive BMD, and testing is easier to perform. In particular, a smaller suite of test cases is necessary. For protecting strategic assets, losses of efficiency can be tolerated with relative ease, and therefore testing efforts can be directed entirely towards the prevention of catastrophic failures. For enhancing deterrence, a few empirical tests

----------

96. Catastrophic failures result in sudden and dramatic decreases in performance. In a real-time software system, a catastrophic failure would be the sudden cessation of all computing, rather than a small error that resulted a somewhat longer time to complete a computation.

visible to the Soviets and obviously successful are likely to induce caution in Soviet leaders. Moreover, these tests could be staged to maximize their influence on Soviet perceptions rather than their usefulness in acquiring information relevant to BMD development and testing.

Detailed comment on BMD software development with these limited goals in mind would require an extensive analysis of the sort that is not possible in the absence of a detailed system design. Nevertheless, under these circumstances, the software development task for BMD does not appear nearly so difficult; indeed, the examples cited above as illustrative of software "failure" can be viewed as successes in this light. Most of these "failures" are now reasonably well-understood, and are exceedingly unlikely to occur in the future. The systems in which they were embedded are now more capable and perform more reliably. (However, it is important to note that limited goals for BMD are NOT what the President proposed to the American public; this point is discussed at length in Conclusions.)

As an example of these points, consider the similarities between limited BMD and carrier air defense as performed by the AEGIS air defense system; Figure 10 illustrates some of these similarities. The primary structural difference between the two tasks appears to be time scale on which the tasks must take place: BMD in ten minutes, carrier air defense in an hour. ("Minor" details such as a strongly nuclear environment and the large scale of the attack for BMD are ignored for the present.) Therefore, human action and decision-making can be (and is) an integral part of carrier air defense, whereas is has essentially no tactical role in even limited BMD. The reliance on human action reduces software requirements significantly

under that required for BMD, since human beings can be regarded in this context as additional computational capability that has been pre-programmed and is capable of doing its own "on-line" debugging.

The translation of human decision-making abilities into computer programs is difficult, but it is not impossible. The development of software for limited BMD can therefore be expected to be more difficult than the development of AEGIS software, but its feasibility over the long run with extensive operational testing seems plausible. A limited BMD system, like AEGIS, will evolve and improve with time. It is likely that it would never operate with 100% effectiveness, but for most military missions short of comprehensive defense of the entire population, such effectiveness is not required. Indeed, very few military missions could fail with consequences even remotely comparable to the nuclear destruction of one U.S. city.

Military missions less consequential, complex and time-sensitive than carrier air defense are even more likely to succeed, an example of which might be battlefield defense against tactical ballistic missiles. Indeed, in software systems designed to carry out these missions, many characteristics of software that are disadvantageous with respect to software for comprehensive BMD become advantageous.

For example, software for less complex systems is often easier to change than the relevant hardware. Because the software medium is not physical, software changes are not bound by physical laws. The STARS report on page 41 noted that the intellectual input to hardware and software changes are comparable. It goes on to note that the overhead associated with

implementing these changes can be dramatically lower with software. Indeed, the lack of physical constraints allows intellectual effort to be focused much more directly on the problem rather than on the overhead, if the complexity of the system can be kept within reasonable bounds.

In addition, the cost of software duplication is essentially zero, once the intellectual problem has been solved. Therefore, software changes that implement new capabilities or that correct old difficulties can be propagated very rapidly. Moreover, since software is not tangible, it can be transmitted from afar by electronic means. Field units need not be recalled for software changes to be installed. Thus, changing field circumstances can be accommodated by experts not associated with the field units.

These advantages are significant, and will become more so as micro-electronics are integrated into modern weapon systems. These weapons will become increasingly capable as the software controlling their operation is tested and refined. However, the task of "eliminating" the ballistic missile threat is not simply somewhat more demanding than other military missions; it is qualitatively different, and the software successes that will appear in the development of weapon systems designed for simpler tasks should not mislead project managers into believing that these successes will necessarily appear in the development of comprehensive BMD.

## 11 -- Conclusions

All of the "horror stories" above are associated with software that was planned, designed, implemented, and debugged by competent software engineers who believed that their efforts were adequate for their assigned tasks. Mostly, these errors have been corrected, resulting in more reliable software whose actual capabilities better match the environment in which they must function. But this route to software improvement -- iteratively discovering errors in operational use and then correcting them -- is unlikely to be useful in the development of comprehensive BMD. Rather, comprehensive BMD requires that the software operate properly _the first time_ in an unpredictable environment and without large-scale empirical testing.

Indeed, the Administration has stated -- in a different context -- that full-scale empirical testing is an integral part of maintaining confidence in the reliable performance of a system. In particular, it opposes a comprehensive ban on the testing of nuclear weapons on the grounds that testing is essential to maintain the reliability of nuclear weapons.[97] Since comprehensive BMD is intended to assume in the future the strategic role that nuclear weapons now play, it is reasonable to believe that the failure of BMD to meet a comparable testing standard should lead to doubts

----------

97. ACDA statement to House Appropriations Committee, March 2, 1983. In Hearings, House Appropriations Committee, Departments of State, Justice and Commerce, FY1984, Part 2, page 273.

about the reliability of BMD.

Perhaps the technical requirements for nearly anything can be met, as the technological optimists assert. Indeed, the Director of the SDI Office, General James Abrahmson, has asserted that "we can do just about anything, technically, if we just decide to do it" [98]. Secretary of Defense Caspar Weinberger has cited historical claims of technical impossibility that proved inaccurate to dismiss critics who make claims of impossibility in the present.[99] Indeed, he has gone so far as to equate the desirability of a goal to its feasibility:

> ... the goal of strategic defense is so eminently desirable that we can and will find solutions to any problems that might develop along the way.[100]

It is true that the pessimists of the past have not been entirely accurate. However, the same is true of the optimists. For example, it is worth recalling Secretary of the Army Stanley Resor's statement of 1966: "The technical feasibility of the NIKE-X [ABM system] and the high probability that the design objectives will be achieved are generally accepted. All major engineering problems have been solved, and only minor design fixes are foreseen."[101] In 1976, the much more modest anti-ballistic missile system (derived from NIKE-X) at Grand Forks was

----------

98. cited in Aerospace America, July 1984, page 81.

99. quoted on "Meet the Press", NBC News, March 27, 1983. See page 8 of Kelly Press Transcript.

100. Caspar Weinberger, "Seeking a Realistic Strategic Defense", Defense/83, June 1983, page 28.

101. Hearings on Department of Defense Appropriations for FY 1967, House Appropriations Committee, Subcommittee on Defense, Part 1, page 408.

de-activated due to Congressional doubts about its technical effectiveness. This track record and the software horror stories above suggest that assurances about software reliability must be taken with a very large degree of skepticism.

Many of the difficulties described above are not new; indeed, some have plagued large-scale software development for nearly thirty years. But the knowledge of these difficulties does not necessarily imply that these difficulties can be overcome. The lessons learned from the history of large software systems suggest while it is possible that a software system for comprehensive BMD will have the necessary robustness and freedom from error in the absence of large-scale empirical testing, this possibility is by no means a certainty.

At the very least, the proponents should prove or at least plausibly argue (rather than merely assert) that a robust and error-free battle management system can be built. One possible test of feasibility would be to build a new commercial airliner entirely on the basis of computer simulations and designs, and then to fly it on its very first flight with a full passenger load on a trans-oceanic flight, the software methodologies used in that project may be usable to construct the software needed for comprehensive BMD. An unwillingness to conduct such a test would be demonstration enough that these new methodologies are inadequate to provide the required confidence in the operation of a BMD software system.

The achievement of intermediate goals for BMD cannot be ruled out, but they should not be pursued through an SDI program whose stated purpose is the effective and comprehensive defense of both civilian and military

targets. Indeed, intermediate goals of SDI are irrelevant if the ultimate goal of the SDI is very unlikely to be achieved and if the SDI will pursue these intermediate goals in a manner that is more difficult, risky, and uncertain than pursuing them outside the SDI program.

Unfortunately, both of these conditions are true. On the basis of software considerations alone, the ultimate goal of eliminating the threat of nuclear ballistic missiles against civilian and military targets is highly unlikely to be achieved by unilateral steps. As for intermediate goals, these are more easily and reliably achieved with research aimed specifically at those intermediate goals; for example, it is easier to protect strategic retaliatory forces with hard point terminal defenses that are already feasible with today's technology than it is to do so with lasers or rail-guns yet to be developed.

However, the most important issue is not the feasibility of BMD, whether comprehensive or limited. Instead, the most important issue is the public policy suggested by the stance that the nation should develop a system to defend strategic assets or one that that can be tested empirically only against small-scale threats in order to dissuade Soviet leaders from attacking. In particular, the President articulated a vision of hope in his March 23, 1983 speech -- that American security would not depend on Soviet decisions, and that it would no longer rest on the cornerstone of nuclear retaliation in case of attack.

The argument that deterrence is enhanced by the uncertainty that BMD introduces leaves unstated the fact that the U.S. would still rely on the threat of massive nuclear retaliation to deter the Soviets from attacking

in the first place. This argument does not support the President's promise, and therefore it perpetrates intentionally or unintentionally a fraud on those Americans who share the President's vision.

In addition, empirical testing against small-scale threats is a testing scenario more similar to defending the U.S. against Soviet retaliation after a largely successful American first strike than it is to defending against a Soviet first strike. Thus, our confidence in missile defense in the first case would be greater than in the second. Is this a desirable route to follow?

Thirdly, the SDI seeks ultimately to instill public confidence that people the world over are free from the threat of nuclear missile attack. What kind of confidence should the public have in an system that is so complex that no individual can understand it? That cannot be certified as error-free? That is untestable? A world in which "computer errors" annoy many and make life miserable for a few suggests that the answer might be "very little".

Finally, some SDI proponents argue that the SDI program is merely a research program to investigate the feasibility of comprehensive defenses against missiles, and that no decisions regarding deployment have yet been made. In a narrow sense, this is true: no specific technologies have been identified as being capable of bearing the burden that comprehensive defenses would impose. However, statements of high-ranking officials do suggest a belief that some technologies will indeed be able to support these defensive systems, and they give every indication that they will base future arms control and defense policy decisions on these systems. The

words of George Keyworth, President Reagan's Science Adviser, are illustrative:

> There are [a few] critical questions about how ... [the Soviets] will respond, either now or a few years from now *when our research programs have made even clearer that SDI will work*.[102] (emphasis added)

The conclusion of this study is that software considerations make questionable the feasibility of a "fully reliable" comprehensive defense against ballistic missiles. Research on these considerations is warranted, but today's state of the computing art and that of the foreseeable future is well-described by the words of an Office of Technology Background Paper: "the prospect that emerging technologies [including software technologies] will provide a near-perfect defense system is so remote that it should not serve as the basis of public expectation or national policy."[103]

----------

102. George Keyworth, "The Case for Arms Control and the Strategic Defense Initiative", Arms Control Today, Volume 15(3), April 1985, page 2.

103. "Directed Energy Missile Defense in Space -- A Background Paper, Office of Technology Assessment, OTA-BP-ISC-26, April 1984.

| | |
|---|---|
| boost | 0-5 minutes after launch. Target characterized by a very visible hot exhaust plume. Value of target is very high, since destruction of target during boost can reduce by a factor of up to 1000 the number of targets and decoys that subsequent layers must handle. With most boost-phase intercept schemes, the atmosphere shields the target, so that attack must occur after it has cleared the atmosphere. |
| post-boost | 5-8 minutes after launch. Target characterized by a relatively dim exhaust plume associated with the post-boost vehicle. Value of target is very high initially, but declines with time, since additional warheads and decoys are released to their targets. Post-boost intercept takes place above the atmosphere. |
| mid-course | 8-25 minutes after launch. Actual target (re-entry vehicle) characterized by room temperature thermal emissions; difficult to detect against background of earth's surface. Since many more decoys than warheads are present, average value of target is low unless warheads can be discriminated from decoys by methods that do not rely on trajectory information. |
| terminal | 25-30 minutes after launch. Target characterized by increasing thermal emissions as the warhead enters the atmosphere and is heated by friction. Decoys gradually burn out in the atmosphere. |

Figure 1: The Different Layers of A Multi-Layer BMD

| SYSTEM | operational software | support software | total |
|---|---|---|---|
| | | (in thousands of lines) | |
| multi-layered BMD (DTST-estimated size) | 3,000 | 7,000 | 10,000 |
| Space Shuttle | 500 | 9,500 | 10,000 |
| NORAD space track and warning | < 2,500 | 1,600 | 4,100 |
| SAFEGUARD (terminal ABM defense) | 789 | 1,500 | 2,490 |
| AEGIS | 400 | 1,400 | 1,800 |
| Airborne Warning and Control System (AWACS) | 70 | 367 | 437 |
| word processing program for home computer [104] | 5 | --- | --- |

Several military systems that employ large amounts of software are described above. The last entry is for civilian software, and is provided for comparative purposes. All program sizes are approximate, and are measured in terms of the instructions written in the original language, i.e., the "source" language. In general, support software is roughly three to five times the size of the real-time operational software.[105]

**Figure 2: Program Sizes (approximate number of lines)**

----------

104. author estimate.

105. cf., Boehm, page 322.

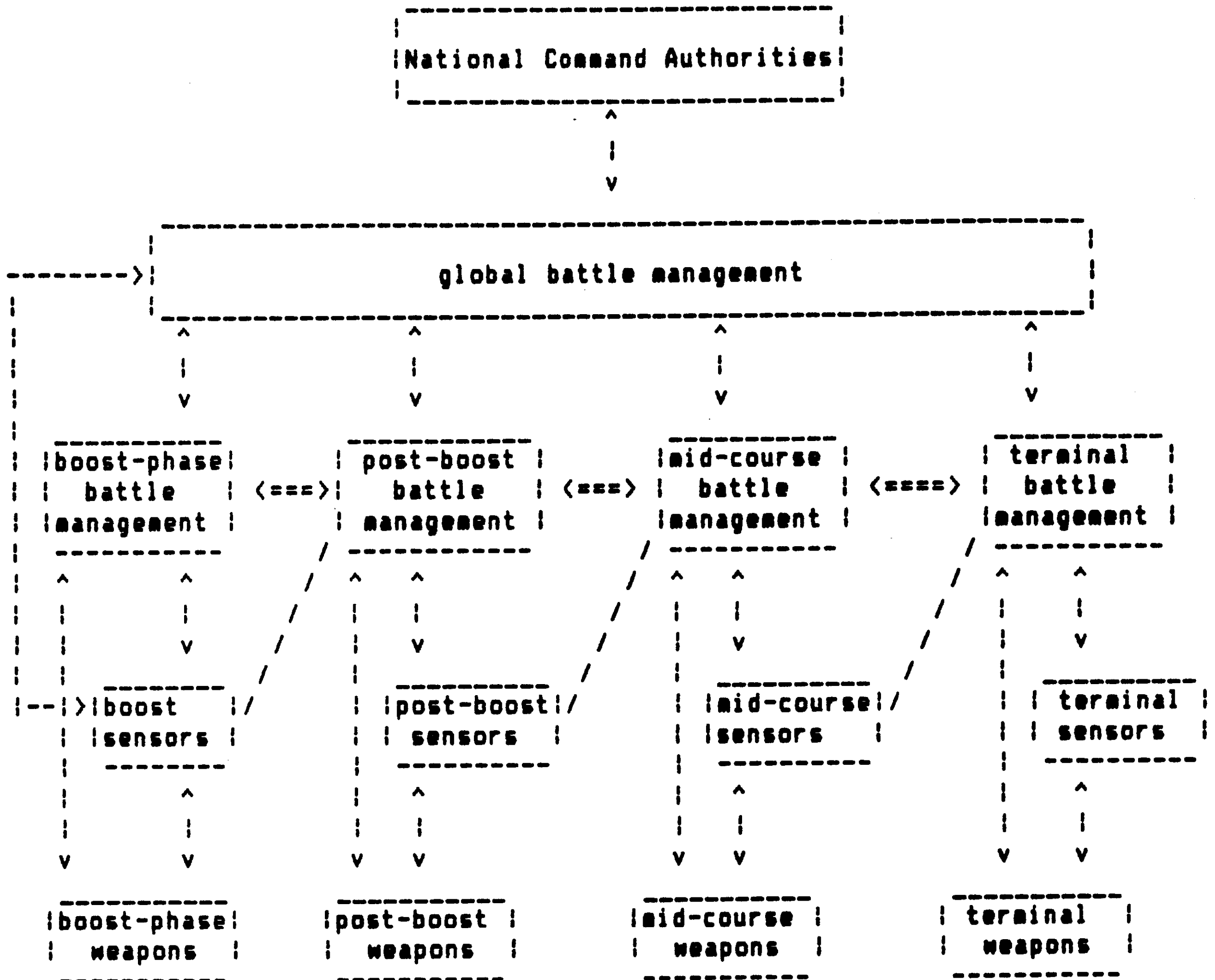
**Sources for Program Sizes**

| | |
|---|---|
| Multi-layered BMD | breakdown provided by transparencies of Albert Nerath, AT&T. |
| Space Shuttle | Alfred Spector and David Gifford, "Case Study: The Space Shuttle Primary Computer System", Communications of the ACM, Volume 27(9), September 1984, page 897. |
| NORAD | General Accounting Office, NORAD's Missile Warning System: What Went Wrong?, page 11. Operational and support estimates from C.A. Zraket. |
| SAFEGUARD | Boehm, page 322. |
| AEGIS | J.T. Threston, "Managing the Future: The Expanding Role of Systems Engineering in a High-Technology Society", RCA Engineer, Volume 28(4), July/August 1983, page 16. Support software estimate from Sig Kopinetz, RCA Missile Systems Division, personal communication, June 1985. |
| AWACS | Boehm, page 322. |

Some of these sources provide software sizes in terms of equivalent "machine" instructions. In these cases, I have assumed for convenience a conversion factor of about three or four "machine" instructions per program line.[106]

----------

106. Boehm, page 478.




-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-

A layered defense has several stages, each with its own battle manager to control its sensors and weapons, and responsible for targets in its own sector. Each operates under the coordination of the global battle management system, which in turn is under the control of the National Command Authority (the President and the Secretary of Defense). In this hypothetical battle management architecture, each layer "hands off" to the next layer data on the status of each target for which it is responsible (e.g., probability that each target has been destroyed, signature data) as well as sensor data (to assist in locating targets that pass through that layer). In addition, the boost-phase sensors provide early warning information to the NCA.

Figure 3: A possible architecture of a battle management system for BMD

| | 10,000,000 lines | | | 20,000,000 lines | | |
|---|---|---|---|---|---|---|
| | LEVEL OF OPTIMISM | | | | | |
| | high | nominal | low | high | nominal | low |
| Phase in development | (man-years) | | | | | |
| planning | 1,500 | 3,900 | 7,000 | 3,400 | 9,000 | 16,200 |
| conceptual design | 3,300 | 8,800 | 15,800 | 7,500 | 20,300 | 36,400 |
| detailed design | 3,400 | 10,300 | 16,700 | 7,700 | 23,600 | 38,400 |
| coding & unit test | 2,100 | 5,800 | 11,700 | 4,900 | 13,300 | 26,900 |
| integration & test | 3,100 | 11,700 | 30,400 | 7,200 | 26,800 | 69,900 |
| Total Project Effort | 13,400 | 40,500 | 81,700 | 30,700 | 93,000 | 187,700 |

These figures unrealistically assume entirely serial development; see text for details. This table also includes 8% of total project effort for the development of plans and requirements, which must take place even before conceptual design.[107]

These figures are roughly corroborated by calculations provided by Paul Anderson (personal communication) of the Naval Electronic Systems Command using the Software Life-cycle Management model (SLIM) for software cost estimation, and by models discussed by Brooks.[108]

Levels of optimism are characterized by assumptions about the ability of the programming teams assigned, their previous experience with battle management systems and the language they will use, the extent to which modern software development tools and practices are employed, and so on. More precise definitions are described in Figure 4 below.

**Figure 5: Estimates for BMD Battle Management Software**

----------

107. Boehm, Chapter 4, and Table 6-8, page 90.

108. Brooks, Chapter 8.

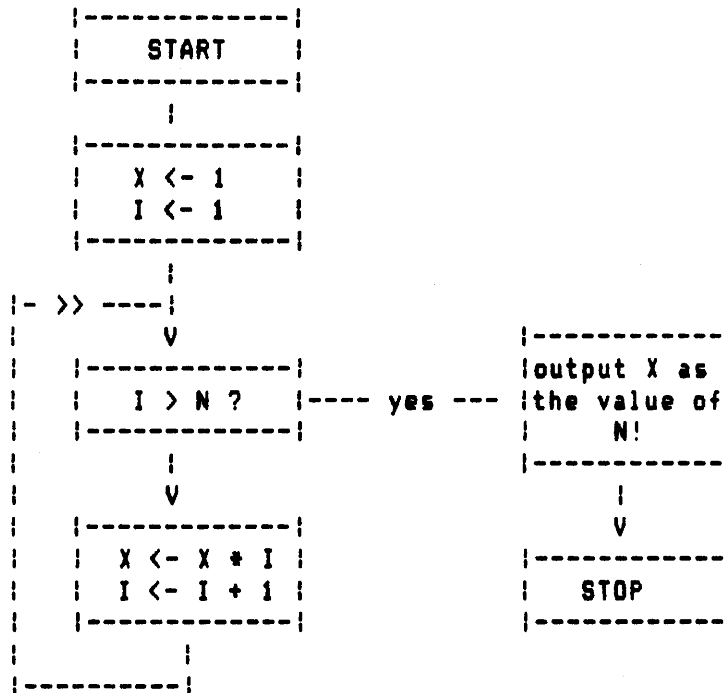
| | high | nominal | low |
|---|---|---|---|
| analyst ability | top 10% | top 45% | top 65% |
| programmer ability | top 10% | top 45% | top 65% |
| team experience in developing BMD software | > 12 years | 3 years | 1 year |
| experience with real-time BMD hardware used in system | > 3 years | 1 year | 4 months |
| experience with language used in system development | > 3 years | 1 year | 1 year |
| use of modern programming practices (MPPs) (top-down planning & design program design languages design & code inspections structured programming incremental development program librarian) | routine use of all MPPs | moderate experience in use of most MPPs | moderate experience in use of some MPPs |
| use of programming tools | use of very advanced mainframe tools (e.g., requirements specification language and analyzer, documentation control automated verification system, performance measurement) | | use of basic mainframe tools (e.g., interactive edit and debug, data base mgmt) |

Other Assumptions for All Phases

- Necessary reliability is very high.
- Complexity of the software product is very high.
- Fraction of processing time used by system is 70%.
- Hardware with which software system must operate experiences major change every six months.
- Software development takes place in an interactive environment.
- Schedule of development is stretched out by 30%.
- Requirements change in a major way occasionally.

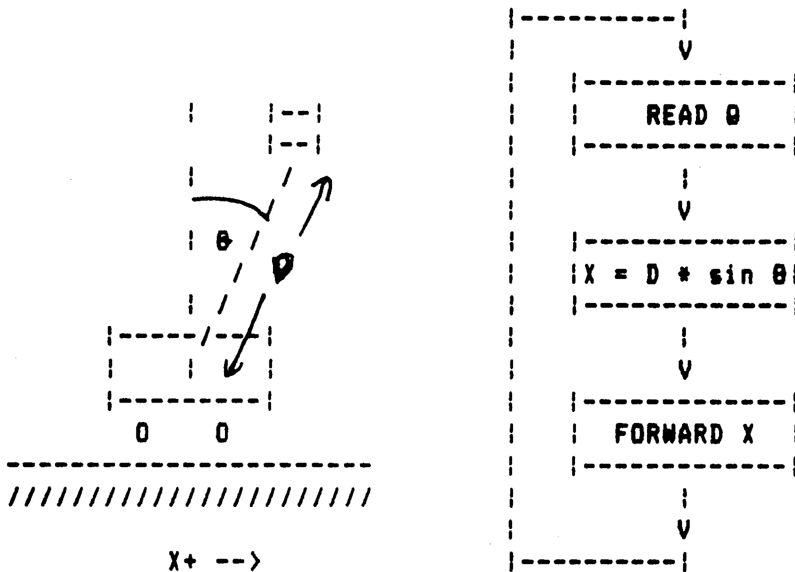
**Figure 4: Definitions for Optimism**

**Program to Calculate N!**




In a non-real-time software system, the rate at which a given set of instructions is executed depends only on a clock internal to the computer's processing unit. For example, a typical micro-processor clock runs at 4 million machine cycles per second (with one instruction equivalent to a few machine cycles). By replacing this clock with another that runs at 2 million cycles per second, a program that does not interact with anything outside the processing unit (such as a keyboard or a disk) will take exactly twice as long to complete.

**Figure 6a: A Non-Real-Time Software System**




In a real-time software system, instructions must be executed at a rate determined by the timing of events external to the computing hardware. In the example above, a motorized cart can move forward or backward. Above it is mounted an inverted weighted pendulum. The task that must be performed is to balance the pendulum above the cart; when the pendulum drops to the right (left), the cart moves to the right (left), in an attempt to place the arm of the pendulum directly under the weight.

A sensor determines the angle θ between the arm and the vertical (the first box). The computation in the second box determine the amount the cart should move (Positive values of X are to the right). The third box directs the cart to move the proper amount, after which the program returns to the first box for another angle reading.

The times relevant to this real-time processing task are the time the cart takes to move the required distance (t(1)), the time it takes to complete the computations (t(2)), and the time that the pendulum takes to drop a significant amount (t(3)). If the cart can move very fast (so that t(1) is always small compared to t(2) and t(3)), then the requirement for successful performance of the task is that t(2) < t(3), i.e., the computation must be completed before the pendulum drops so far that the cart <u>cannot</u> be moved to the required location. In other words, computations must be fast enough that the angle θ is always kept small.

**Figure 6b: A Real-Time Software System**

## Local Battle Management Functions

Local functions are functions that are logically performed within a given defensive layer, though their computational requirements may vary.

acquisition/track — locate all potential targets of relevance; generates a "track file" that contains all known information about every target.

discrimination — identifies actual targets from decoys, debris, or other "junk".

resource allocation — coordinates track file information, operating status of components in the defensive layer, and tactical doctrine to assign specific weapons platforms to specific targets at specific times in order to maximize the efficiency of the defense.

kill assessment — determines of the extent to which an attacked target has been damaged. May also provide information on how to attack target again.

## Global Battle Management Functions

Global functions are functions that logically connect different layers of the system.

rules of engagement — specify the precise circumstances under which given classes of target may be attacked, and in what manner. May differ from layer to layer: the boost-phase might be forbidden to engage targets unless they exceed a minimum threshold, whereas the mid-course might engage anything.

surveillance — assesses sensor output to determine if an attack is in progress and if so, its nature and extent.

"hand-off" — transmits track file information and damage kill assessment from layer to layer, to assist subsequent layers in their own acquisition/tracking and resource allocation.

self-defense — coordinates elements of the defense to maximize its own survival to the maximum extent consistent with the goal of destroying attacking missiles.

**Figure 7: Functions of Battle Management**

Task: count the number of people in a room.

Level:

1. Scale of task: 30 or so people in room.

   Level of difficulty: simple task, intuitively understood by most adults. Most people attempting this task would accomplish it successfully.

2. Scale of task: 300 or so people in room.

   Level of difficulty: more difficult task. Still intuitively understood by most adults, but more difficult to implement. Different people attempting this task would most likely arrive at different counts. More formal specification of task required to guide implementation: count each person in the room, and count no person more than once. New procedure: develop more formal procedure to count systematically (e.g., ask people to sit down, tag a person once counted).

3. Scale of task: 3000 or so people in room.

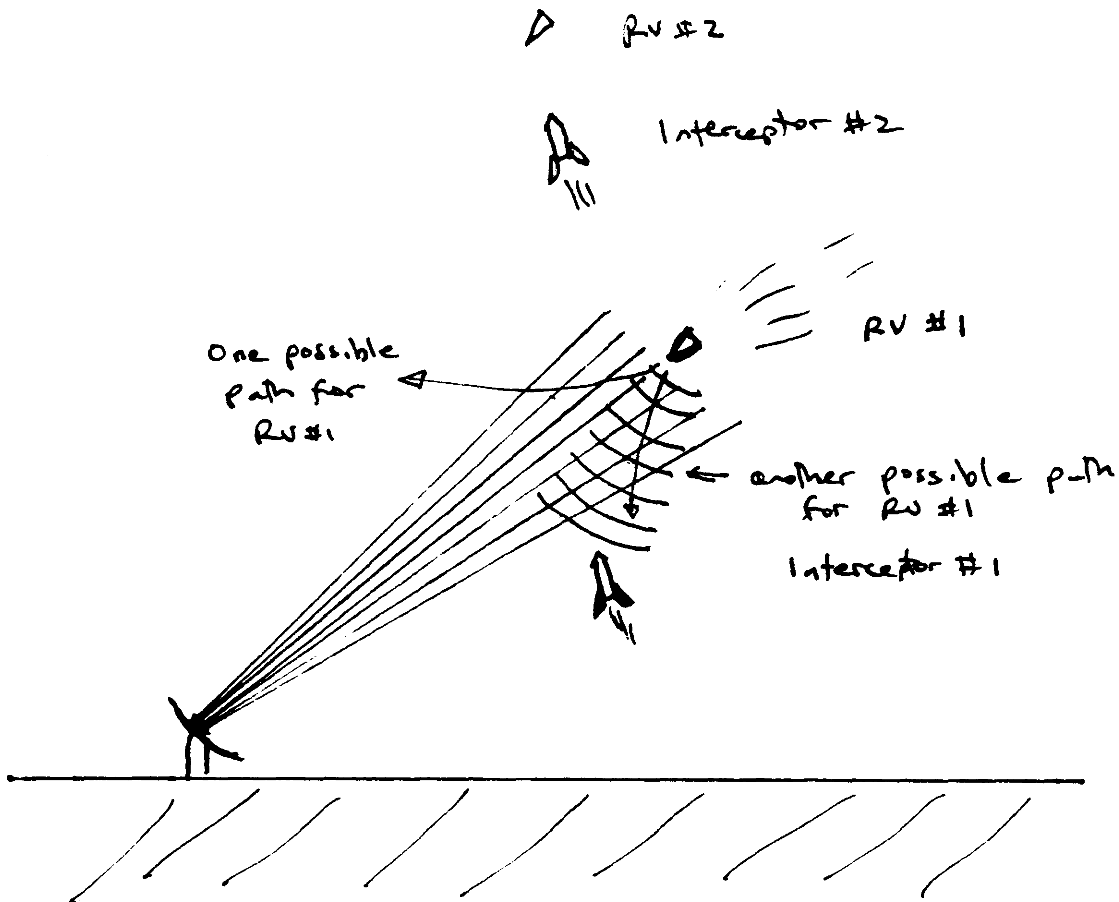
   Level of difficulty: still more difficult task. Still intuitively understood by most adults, but task definition begins to become ill-defined: what does "in room" mean? Formal specification of task given above still necessary, but "room" must be defined more precisely; entrances and exits to room must be sealed.

4. Scale of task: 30,000 or so people in room.

   Level of difficulty: exceedingly difficult task. Task definition begins to become ill-defined: what does "in room" mean? Formal specification of task given above and formal "room" boundaries still necessary, but "people" now needs definition. In a full sports stadium, it is often that a woman will be pregnant or that a man will have a fatal heart attack. How should these entities enter the count, and by what criteria?

Each level of the "same" task is more difficult than the previous one in both quantitative and qualitative ways. Quantitatively, a level of higher difficulty is one that requires the counter to count higher with all of those attendant difficulties (e.g., losing one's count). Qualitatively, factors that are unlikely to be relevant at easier levels (e.g, due to their infrequent occurrence) must be taken into account formally as the problem grows in scale.

Under most circumstances, some error in the count of people can be tolerated. However, if the task is instead intercepting warheads destined for American cities, no error is permissible.




A ground-based radar illuminates an incoming target, in this case a maneuvering re-entry vehicle, by directing a continuous radar signal at the target. A homing device on the intercepting missile seeks out the radar signal reflected by the target. If the illuminating radar turns away from target #1 too soon in order to illuminate target #2, the first intercepting missile may "lose" its target, thereby missing it. A computer program directing the illuminating radar to switch from target #1 to target #2 must do so after target #1 is destroyed, but soon enough that it can guide interceptor #2 to its target. Depending on the precise maneuvers of target #1, the indication that target #1 had been destroyed might or might not arrive in time to guide interceptor #2 to its target.

**Figure 9: Effects of Timing in Real-Time Systems**

| | Ballistic Missile Defense | Carrier Air Defense |
|---|---|---|
| multi-layered defense | boost, post-boost, mid-course, terminal intercept phases | three rings around carrier: outer (F-14), middle (SM-2 missiles), terminal (Close-In Weapon System |
| multiple sensors | radar, ladar, short-wave infra-red, long-wave IR | SPY-1 ship-based radar, E-2C airborne radar |
| target | hardened missile silos | highly survivable ship |

Both the AEGIS air defense system and currently envisioned BMD system architectures involve a multi-layered defense, receiving sensor information from a variety of platforms. In both cases, battle management concerns the allocation of weapons to targets, kill assessment, threat assessment, and tracking targets in the presence of noise and decoys.

Figure 10: Similarities between AEGIS and BMD