SOFTWARE ASPECTS OF STRATEGIC DEFENSE SYSTEMS

A former member of the SDIO Panel on Computing in Support of Battle Management explains why he believes the "star wars" effort will not achieve its stated goals.

DAVID LORGE PARNAS

On 28 June 1985, David Lorge Parnas, a respected computer scientist who has consulted extensively on United States defense projects, resigned from the Panel on Computing in Support of Battle Management, convened by the Strategic Defense Initiative Organization (SDIO). With his letter of resignation, he submitted eight short essays explaining why he believed the software required by the Strategic Defense Initiative would not be trustworthy. Excerpts from Dr. Parnas's letter and the accompanying papers have appeared widely in the press. The Editors of American Scientist believed that it would be useful to the scientific community to publish these essays in their entirety to stimulate scientific discussion of the feasibility of the project. As part of the activity of the Forum on Risks to the Public in the use of computer systems the Editors of Communications *are pleased to reprint these essays.**

This report comprises eight short papers that were completed while I was a member of the Panel on Computing in Support of Battle Management, convened by the Strategic Defense Initiative Organization (SDIO). SDIO is part of the Office of the U.S. Secretary of Defense. The panel was asked to identify the computer science problems that would have to be solved before an effective antiballistic missile (ABM) system could be deployed. It is clear to everyone that computers must play a critical role in the systems that SDIO is considering. The essays that constitute this report were written to organize my thoughts on these topics and were submitted to SDIO with my resignation from the panel.

My conclusions are not based on political or policy judgments. Unlike many other academic critics of the SDI effort, I have not, in the past, objected to defense efforts or defense-sponsored research. I have been deeply involved in such research and have consulted extensively on defense projects. My conclusions are based on more than 20 years of research on software engineering, including more than 8 years of work on real-time software used in military aircraft. They are based on familiarity with both operational military software and computer science research. My conclusions are based on characteristics peculiar to this particular effort, not objections to weapons development in general.

I am publishing the papers that accompanied my letter of resignation so that interested people can understand why many computer scientists believe that systems of the sort being considered by the SDIO cannot be built. These essays address the software engineering aspects of SDIO and the organization of engineering research. They avoid political issues; those have been widely discussed elsewhere, and I have nothing to add.

In these essays I have attempted to avoid technical jargon, and readers need not be computer programmers to understand them. They may be read in any order.

The individual essays explain:

1. The fundamental technological differences between software engineering and other areas of engineering and why software is unreliable;
2. The properties of the proposed SDI software that make it unattainable;
3. Why the techniques commonly used to build military software are inadequate for this job;
4. The nature of research in software engineering, and why the improvements that it can effect will not be

* Reprinted by permission of *American Scientist*, Journal of Sigma Xi: Software Aspects of Strategic Defense Systems, David Lorge Parnas, Vol. 73, No. 5, pp. 432-440.

© 1985 ACM 0001-0782/85/1200-1326 75¢

sufficient to allow construction of a truly reliable strategic defense system;

5. Why I do not expect research in artificial intelligence to help in building reliable military software;

6. Why I do not expect research in automatic programming to bring about the substantial improvements that are needed;

7. Why program verification (mathematical proofs of correctness) cannot give us a reliable strategic defense battle-management system;

8. Why military funding of research in software and other aspects of computing science is inefficient and ineffective. This essay responds to the proposal that SDIO should be funded even if the ABM system cannot be produced, because the program will produce good research.

Why software is unreliable

I. Introduction

People familiar with both software engineering and older engineering disciplines observe that the state of the art in software is significantly behind that in other areas of engineering. When most engineering products have been completed, tested, and sold, it is reasonable to expect that the product design is correct and that it will work reliably. With software products, it is usual to find that the software has major "bugs" and does not work reliably for some users. These problems may persist for several versions and sometimes worsen as the software is "improved." While most products come with an express or implied warranty, software products often carry a specific disclaimer of warranty. The lay public, familiar with only a few incidents of software failure, may regard them as exceptions caused by inept programmers. Those of us who are software professionals know better; the most competent programmers in the world cannot avoid such problems. This section discusses one reason for this situation.

The requirements of a strategic defense system

In March 1983, President Reagan said, "I call upon the scientific community, who gave us nuclear weapons, to turn their great talents to the cause of mankind and world peace; to give us the means of rendering these nuclear weapons impotent and obsolete."

To satisfy this request the software must perform the following functions:

- **—Rapid and reliable warning of attack**
- **—Determination of the source of the attack**
- **—Determination of the likely targets of the attack**
- **—Determination of the missile trajectories**
- **—Coordinated interception of the missiles or warheads during boost, midcourse, and terminal phases, including assignment of responsibility for targets to individual sensors or weapons**
- **—Discrimination between decoys and warheads**
- **—Detailed control of individual weapons**
- **—Evaluation of the effectiveness of each attempt to destroy a target.**

II. System types

Engineering products can be classified as discrete state systems, analog systems, or hybrid systems.

Discrete state or digital systems are made from components with a finite number of stable states. They are designed in such a way that the behavior of the system when not in a stable state is not significant.

Continuous or analog systems are made from components that, within a broad operating range, have an infinite number of stable states and whose behavior can be adequately described by continuous functions.

Hybrid systems are mixtures of the two types of components. For example, we may have an electrical circuit containing, in addition to analog components, a few components whose descriptive equations have discontinuities (e.g., diodes). Each of these components has a small number of discrete operating states. Within these states its behavior can be described by continuous functions.

III. Mathematical tools

Analog systems form the core of the traditional areas of engineering. The mathematics of continuous functions is well understood. When we say that a system is described by continuous functions we are saying that it can contain no hidden surprises. Small changes in inputs will always cause correspondingly small changes in outputs. An engineer who ensures, through careful design, that the system components are always operating within their normal operating range can use a mathematical analysis to ensure that there are no surprises. When combined with testing to ensure that the components are within their operating range, this leads to reliable systems.

Before the advent of digital computers, when discrete state systems were built, the number of states in such systems was relatively small. With a small number of states, exhaustive testing was possible. Such testing compensated for the lack of mathematical tools corresponding to those used in analog systems design. The engineers of such systems still had systematic methods that allowed them to obtain a complete understanding of their system's behavior.

The design of many hybrid systems can be verified by a combination of the two methods. We can then identify a finite number of operating states for the components with discrete behavior. Within those states, the system's behavior can be described by continuous functions. Usually the number of states that must be distinguished is small. For each of those states, the tools of continuous mathematics can be applied to analyze the behavior of the system.

With the advent of digital computers, we found the first discrete state systems with very large numbers of states. However, to manufacture such systems it was necessary to construct them using many copies of very small digital subsystems. Each of those small subsystems could be analyzed and tested exhaustively. Because of the repetitive structure, exhaustive testing was not necessary to obtain correct and reliable hardware. Although design errors are found in computer hardware, they are considered exceptional. They usually occur in those parts of the computer that are not repetitive structures.

Software systems are discrete state systems that do not have the repetitive structure found in computer circuitry. There is seldom a good reason to construct software as highly repetitive structures. The number of states in software systems is orders of magnitude larger than the number of states in the nonrepetitive parts of computers. The mathematical functions that describe the behavior of these systems are not continuous functions, and traditional engineering mathematics does not help in their verification. This difference clearly contributes to the relative unreliability of software systems and the apparent lack of competence of software engineers. It is a fundamental difference that will not disappear with improved technology.

IV. How can we understand software?
To ameliorate the problems caused by this fundamental difference in technology two techniques are available: (1) the building of software as highly organized collections of small programs and (2) the use of mathematical logic to replace continuous mathematics.

Dividing software into modules and building each module of so-called "structured" programs clearly helps. When properly done, each component deals with a small number of cases and can be completely analyzed. However, real software systems have many such components, and there is no repetitive structure to simplify the analysis. Even in highly structured systems, surprises and unreliability occur because the human mind is not able to fully comprehend the many conditions that can arise because of the interaction of these components. Moreover, finding the right structure has proved to be very difficult. Well-structured real software systems are still rare.

Logic is a branch of mathematics that can deal with functions that are not continuous. Many researchers believe that it can play the role in software engineering that continuous mathematics plays in mechanical and electrical engineering. Unfortunately, this has not yet been verified in practice. The large number of states and lack of regularity in the software result in extremely complex mathematical expressions. Disciplined use of these expressions is beyond the computational capacity of both the human programmer and current computer systems. There is progress in this area, but it is very slow, and we are far from being able to handle even small software systems. With current techniques the mathematical expressions describing a program are often notably harder to understand than the program itself.

V. The education of programmers
Worsening the differences between software and other areas of technology is a personnel problem. Most designers in traditional engineering disciplines have been educated to understand the mathematical tools that are available to them. Most programmers cannot even begin to use the meager tools that are available to software engineers.

Why the SDI software system will be untrustworthy

I. Introduction
In March 1983, the President called for an intensive and comprehensive effort to define a long-term research program with the ultimate goal of eliminating the threat posed by nuclear ballistic missiles. He asked us, as members of the scientific community, to provide the means of rendering these nuclear weapons impotent and obsolete. To accomplish this goal we would need a software system so well-developed that we could have extremely high confidence that the system would work correctly when called upon. In this section, I will present some of the characteristics of the required battle-management software and then discuss their implications on the feasibility of achieving that confidence.

II. Characteristics of the proposed battle-management software system

1. The system will be required to identify, track, and direct weapons toward targets whose ballistic characteristics cannot be known with certainty before the moment of battle. It must distinguish these targets from decoys whose characteristics are also unknown.
2. The computing will be done by a network of computers connected to sensors, weapons, and each other, by channels whose behavior, at the time the system is invoked, cannot be predicted because of possible countermeasures by an attacker. The actual subset of system components that will be available at the time that the system is put into service, and throughout the period of service, cannot be predicted for the same reason.
3. It will be impossible to test the system under realistic conditions prior to its actual use.
4. The service period of the system will be so short that there will be little possibility of human intervention and no possibility of debugging and modification of the program during that period of service.
5. Like many other military programs, there are absolute real-time deadlines for the computation. The computation will consist primarily of periodic processes, but the number of those processes that will be required, and the computational requirements of each process, cannot be predicted in advance because they depend

on target characteristics. The resources available for computation cannot be predicted in advance. We cannot even predict the "worst case" with any confidence.

6. The weapon system will include a large variety of sensors and weapons, most of which will themselves require a large and complex software system. The suite of weapons and sensors is likely to grow during development and after deployment. The characteristics of weapons and sensors are not yet known and are likely to remain fluid for many years after deployment. The result is that the overall battle-management software system will have to integrate a software system significantly larger than has ever been attempted before. The components of that system will be subject to independent modification.

III. Implications of these problem characteristics

Each of these characteristics has clear implications on the feasibility of building battle-management software that will meet the President's requirements.

1. Fire-control software cannot be written without making assumptions about the characteristics of enemy weapons and targets. This information is used in determining the recognition algorithms, the sampling periods, and the noise-filtering techniques. If the system is developed without the knowledge of these characteristics, or with the knowledge that the enemy can change some of them on the day of battle, there are likely to be subtle but fatal errors in the software.

2. Although there has been some real progress in the area of "fail-soft" computer software, I have seen no success except in situations where (a) the likely failures can be predicted on the basis of past history, (b) the component failures are unlikely and are statistically independent, (c) the system has excess capacity, (d) the real-time deadlines, if any, are soft, i.e., they can be missed without long-term effects. None of these is true for the required battle-management software.

3. No large-scale software system has ever been installed without extensive testing under realistic conditions. For example, in operational software for military aircraft, even minor modifications require extensive ground testing followed by flight testing in which battle conditions can be closely approximated. Even with these tests, bugs can and do show up in battle conditions. The inability to test a strategic defense system under field conditions before we actually need it will mean that no knowledgeable person would have much faith in the system.

4. It is not unusual for software modifications to be made in the field. Programmers are transported by helicopter to Navy ships; debugging notes can be found on the walls of trucks carrying computers that were used in Vietnam. It is only through such modifications that software becomes reliable. Such opportunities will not be available in the 30–90 minute war to be fought by a strategic defense battle-management system.

5. Programs of this type must meet hard real-time deadlines reliably. In theory, this can be done either by scheduling at runtime or by preruntime scheduling. In practice, efficiency and predictability require some preruntime scheduling. Schedules for the worst-case load are often built into the program. Unless one can work out worst-case real-time schedules in advance, one can have no confidence that the system will meet its deadlines when its service is required.

6. All of our experience indicates that the difficulties in building software increase with the size of the system, with the number of independently modifiable subsystems, and with the number of interfaces that must be defined. Problems worsen when interfaces may change. The consequent modifications increase the complexity of the software and the difficulty of making a change correctly.

IV. Conclusion

All of the cost estimates indicate that this will be the most massive software project ever attempted. The system has numerous technical characteristics that will make it more difficult than previous systems, independent of size. Because of the extreme demands on the system and our inability to test it, we will never be able to believe, with any confidence, that we have succeeded. Nuclear weapons will remain a potent threat.

Why conventional software development does not produce reliable programs

I. What is the conventional method?

The easiest way to describe the programming method used in most projects today was given to me by a teacher who was explaining how he teaches programming. "Think like a computer," he said. He instructed his students to begin by thinking about what the computer had to do first and to write that down. They would then think about what the computer had to do next and continue in that way until they had described the last thing the computer would do. This, in fact, is the way I was taught to program. Most of today's textbooks demonstrate the same method, although it has been improved by allowing us to describe the computer's "thoughts" in larger steps and later to refine those large steps to a sequence of smaller steps.

II. Why this method leads to confusion

This intuitively appealing method works well—on problems too small to matter. We think that it works because it worked for the first program that we wrote. One can follow the method with programs that have neither branches nor loops. As soon as our thinking reaches a point where the action of the computer must depend on conditions that are not known until the program is running, we must deviate from the method by labeling one or more of the actions and remembering how we would get there. As soon as we introduce loops into the program, there are many ways of getting to some of the points and we must remember all of those

ways. As we progress through the algorithm, we recognize the need for information about earlier events and add variables to our data structure. We now have to start remembering what data mean and under what circumstances data are meaningful.

As we continue in our attempt to "think like a computer," the amount we have to remember grows and grows. The simple rules defining how we got to certain points in a program become more complex as we branch there from other points. The simple rules defining what the data mean become more complex as we find other uses for existing variables and add new variables. Eventually, we make an error. Sometimes we note that error; sometimes it is not found until we test. Sometimes the error is not very important; it happens only on rare or unforeseen occasions. In that case, we find it when the program is in use. Often, because one needs to remember so much about the meaning of each label and each variable, new problems are created when old problems are corrected.

III. What is the effect of concurrency on this method?
In many of our computer systems there are several sources of information and several outputs that must be controlled. This leads to a computer that might be thought of as doing many things at once. If the sequence of external events cannot be predicted in advance, the sequence of actions taken by the computer is also not predictable. The computer may be doing only one thing at a time, but as one attempts to "think like a computer," one finds many more points where the action must be conditional on what happened in the past. Any attempt to design these programs by thinking things through in the order that the computer will execute them leads to confusion and results in systems that nobody can understand completely.

IV. What is the effect of multiprocessing?
When there is more than one computer in a system, the software not only appears to be doing more than one thing at a time, it really is doing many things at once. There is no sequential program that one can study. Any attempt to "think like the computer system" is obviously hopeless. There are so many possibilities to consider that only extensive testing can begin to sort things out. Even after such testing, we have incidents such as one that happened on a space shuttle flight several years ago. The wrong combination of sequences occurred and prevented the flight from starting.

V. Do professional programmers really use this approach?
Yes. I have had occasion to study lots of practical software and to discuss programs with lots of professional programmers. In recent years many programmers have tried to improve their working methods using a variety of software design approaches. However, when they get down to writing executable programs, they revert to the conventional way of thinking. I have yet to find a substantial program in practical use whose structure was not based on the expected execution sequence. I would be happy to be shown some.

Other methods are discussed in advanced courses, a few good textbooks, and scientific meetings, but most programmers continue to use the basic approach of thinking things out in the order that the computer will execute them. This is most noticeable in the maintenance (deficiency correction) phase of programming.

VI. How do we get away with this inadequate approach?
It should be clear that writing and understanding very large real-time programs by "thinking like a computer" will be beyond our intellectual capabilities. How can it be that we have so much software that is reliable enough for us to use it? The answer is simple; programming is a trial and error craft. People write programs without any expectation that they will be right the first time. They spend at least as much time testing and correcting errors as they spent writing the initial program. Large concerns have separate groups of testers to do quality assurance. Programmers cannot be trusted to test their own programs adequately. Software is released for use, not when it is known to be correct, but when the rate of discovering new errors slows down to one that management considers acceptable. Users learn to expect errors and are often told how to avoid the bugs until the program is improved.

VII. Conclusion
The military software that we depend on every day is not likely to be correct. The methods that are in use in the industry today are not adequate for building large real-time software systems that must be reliable when first used. A drastic change in methods is needed.

The limits of software engineering methods

I. What is software engineering research?
We have known for 25 years that our programming methods are inadequate for large projects. Research in software engineering, programming methodology, software design, etc., looks for better tools and methods. The common thrust of results in these fields is to reduce the amount that a programmer must remember when checking and changing a program.

Two main lines of research are (1) structured programming and the use of formal program semantics and (2) the use of formally specified abstract interfaces to hide information about one module (work assignment) from the programmers who are working on other parts. A third idea, less well understood but no less important, is the use of cooperating sequential processes to help deal with the complexities arising from concurrency and multiprogramming. By the late 1970s the basic ideas in software engineering were considered "motherhood" in the academic community. Nonetheless, examinations of real programs revealed that actual programming practice, especially for military systems,

had not been changed much by the publication of the academic proposals.

The gap between theory and practice was large and growing. Those espousing structured approaches to software were certain that it would be easy to apply their ideas to the problems that they faced in their daily work. They doubted that programs organized according to the principles espoused by academics could ever meet the performance constraints on "real" systems. Even those who claimed to believe in these principles were not able to apply them consistently.

In 1977 the management of the Naval Research Laboratory in Washington, D.C., and the Naval Weapons Center in China Lake, California, decided that something should be done to close the gap. They asked one of the academics who had faith in the new approach (myself) to demonstrate the applicability of those methods by building, for the sake of comparison, a second version of a Navy real-time program. The project, now known as the Software Cost Reduction project (SCR), was expected to take two to four years. It is still going on.

The project has made two things clear: (1) much of what the academics proposed can be done; (2) good software engineering is far from easy. The methods reduce, but do not eliminate, errors. They reduce, but do not eliminate, the need for testing.

II. What should we do and what can we do?

The SCR work has been based on the following precepts.

1. The software requirements should be nailed down with a complete, black-box requirements document before software design is begun.
2. The system should be divided into modules using information-hiding (abstraction) before writing the program begins.
3. Each module should have a precise, black-box, formal specification before writing the program begins.
4. Formal methods should be used to give precise documentation.
5. Real-time systems should be built as a set of cooperating sequential processes, each with a specified period and deadline.
6. Programs should be written using the ideas of structured programming as taught by Harlan Mills.

We have demonstrated that the first four of these precepts can be applied to military software by doing it. The documents that we have written have served as models for others. We have evidence that the models provide a most effective means of technology transfer.

We have not yet proved that these methods lead to reliable code that meets the space and time constraints. We have found that every one of these precepts is easier to pronounce than to carry out. Those who think that software design will become easy, and that errors will disappear, have not attacked substantial problems.

III. What makes software engineering hard?

We can write software requirements documents that are complete and precise. We understand the mathematical model behind such documents and can follow a systematic procedure to document all necessary requirements decisions. Unfortunately, it is hard to make the decisions that must be made to write such a document. We often do not know how to make those decisions until we can play with the system. Only when we have built a similar system before is it easy to determine the requirements in advance. It is worth doing, but it is not easy.

We know how to decompose complex systems into modules when we know the set of design decisions that must be made in the implementation. Each of these must be assigned to a single module. We can do that when we are building a system that resembles a system we built before. When we are solving a totally new problem, we will overlook difficult design decisions. The result will be a structure that does not fully separate concerns and minimize complexity.

We know how to specify abstract interfaces for modules. We have a set of standard notations for use in that task. Unfortunately, it is very hard to find the right interface. The interface should be an abstraction of the set of all alternative designs. We can find that abstraction only when we understand the alternative designs. For example, it has proved unexpectedly hard to design an abstract interface that hides the mathematical model of the earth's shape. We have no previous experience with such models and no one has designed such an abstraction before.

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

IV. Will new programming languages make much difference?

Because of the very large improvements in productivity that were noted when compiler languages were introduced, many continue to look for another improvement by introducing better languages. Better notation always helps, but we cannot expect new languages to provide the same magnitude of improvement that we got from the first introduction of such languages. Our experience in SCR has not shown the lack of a language to be a major problem.

Programming languages are now sufficiently flexible that we can use almost any of them for almost any task. We should seek simplifications in programming languages, but we cannot expect that this will make a big difference.

V. What about programming environments?

The success of UNIX® as a programming development tool has made it clear that the environment in which we work does make a difference. The flexibility of UNIX® has allowed us to eliminate many of the time-consuming housekeeping tasks involved in producing large programs. Consequently, there is extensive research in programming environments. Here, too, I expect small improvements can be made by basing tools on improved notations but no big breakthroughs. Prob-

lems with our programming environment have not been a major impediment in our SCR work.

VI. Why software engineering research will not make the SDI goals attainable
Although I believe that further research on software engineering methods can lead to substantial improvements in our ability to build large real-time software systems, this work will not overcome the difficulties inherent in the plans for battle-management computing for SDI. Software engineering methods do not eliminate errors. They do not eliminate the basic differences between software technology and other areas of engineering. They do not eliminate the need for extensive testing under field conditions or the need for opportunities to revise the system while it is in use. Most important, we have learned that the successful application of these methods depends on experience accumulated while building and maintaining similar systems. There is no body of experience for SDI battle management.

VII. Conclusion
I am not a modest man. I believe that I have as sound and broad an understanding of the problems of software engineering as anyone that I know. If you gave me the job of building the system, and all the resources that I wanted, I could not do it. I don't expect the next 20 years of research to change that fact.

Artificial intelligence and the Strategic Defense Initiative

I. Introduction
One of the technologies being considered for use in the SDI battle-management software is artificial intelligence (AI). Researches in AI have often made big claims, and it is natural to believe that one should use this technology for a problem as difficult as SDI battle management. In this section, I argue that one cannot expect much help from AI in building reliable battle-management software.

II. What is artificial intelligence?
Two quite different definitions of AI are in common use today.

AI-1: The use of computers to solve problems that previously could be solved only by applying human intelligence.

AI-2: The use of a specific set of programming techinques known as heuristic or rule-based programming. In this approach human experts are studied to determine what heuristics or rules of thumb they use in solving problems. Usually they are asked for their rules. These rules are then encoded as input to a program that attempts to behave in accordance with them. In other words, the program is designed to solve a problem the way that humans seem to solve it.

It should be noted that the first definition defines AI as a set of problems, the second defines AI as a set of techniques. The first definition has a sliding meaning. In the Middle Ages, it was thought that arithmetic required intelligence. Now we recognize it as a mechanical act. Something can fit the definition of AI-1 today, but, once we see how the program works and understand the problem, we will not think of it as AI anymore.

It is quite possible for a program to meet one definition and not the other. If we build a speech-recognition program that uses Bayesian mathematics rather than heuristics, it is AI-1 but not AI-2. If we write a rule-based program to generate parsers for precedence grammars using heuristics, it will be AI-2 but not AI-1 because the problem has a known algorithmic solution.

Although it is possible for work to satisfy both definitions, the best AI-1 work that I have seen does not use heuristic or rule-based methods. Workers in AI-1 often use traditional engineering and scientific approaches. They study the problem, its physical and logical constraints, and write a program that makes no attempt to mimic the way that people say they solve the problem.

III. What can we learn from AI that will help us build the battle-management computer software?
I have seen some outstanding AI-1 work. Unfortunately, I cannot identify a body of techniques or technology that is unique to this field. When one studies these AI-1 programs one finds that they use sound scientific approaches, approaches that are also used in work that is not called AI. Most of the work is problem specific, and some abstraction and creativity are required to see how to transfer it. People speak of AI as if it were some magic body of new ideas. There is good work in AI-1 but nothing so magic it will allow the solution of the SDI battle-management problem.

I find the approaches taken in AI-2 to be dangerous and much of the work misleading. The rules that one obtains by studying people turn out to be inconsistent, incomplete, and inaccurate. Heuristic programs are developed by a trial and error process in which a new rule is added whenever one finds a case that is not handled by the old rules. This approach usually yields a program whose behavior is poorly understood and hard to predict. AI-2 researchers accept this evolutionary approach to programming as normal and proper. I trust such programs even less than I trust unstructured conventional programs. One never knows when the program will fail.

On occasion I have had to examine closely the claims of a worker in AI-2. I have always been disappointed. On close examination the heuristics turned out to handle a small number of obvious cases but failed to work in general. The author was able to demonstrate spectacular behavior on the cases that the program handled correctly. He marked the other cases as extensions for future researchers. In fact, the techniques being used often do not generalize and the improved program never appears.

IV. What about expert systems?
Lately we have heard a great deal about the success of a particular class of rule-based systems known as expert systems. Every discussion cites one example of such a system that is being used to solve real problems by people other than its developer. That example is always the same—a program designed to find configurations for VAX computers. To many of us, that does not sound like a difficult problem; it sounds like the kind of problem that is amenable to algorithmic solution because VAX systems are constructed from well-understood, well-designed components. Recently I read a paper that reported that this program had become a maintenance nightmare. It was poorly understood, badly structured, and hence hard to change. I have good reason to believe that it could be replaced by a better program written using good software engineering techniques instead of heuristic techniques.

SDI presents a problem that may be more difficult than those being tackled in AI-1 and expert systems. Workers in those areas attack problems that now require human expertise. Some of the problems in SDI are in areas where we now have no human experts. Do we now have humans who can, with high reliability and confidence, look at missiles in ballistic flight and distinguish warheads from decoys?

V. Conclusion
Artificial intelligence has the same relation to intelligence as artificial flowers have to flowers. From a distance they may appear much alike, but when closely examined they are quite different. I don't think we can learn much about one by studying the other. AI offers no magic technology to solve our problem. Heuristic techniques do not yield systems that one can trust

Can automatic programming solve the SDI software problem?

I. Introduction
Throughout my career in computing I have heard people claim that the solution to the software problem is automatic programming. All that one has to do is write the specifications for the software, and the computer will find a program. Can we expect such technology to

II. Some perspective on automatic programming
The oldest paper known to me that discusses automatic programming was written in the 1940s by Saul Gorn when he was working at the Aberdeen Proving Ground. This paper, entitled "Is Automatic Programming Feasible?" was classified for a while. It answered the ques-

rectly and actually looked at the holes in the tape. I have seen programmers "patch" programs by literally patching the paper tape.

The automatic programming system considered by Gorn in that paper was an assembler in today's terminology. All that one would have to do with his automatic programming system would be to write a code such as CLA, and the computer would automatically punch the proper holes in the tape. In this way, the programmer's task would be performed automatically by the computer.

In later years the phrase was used to refer to program generation from languages such as IT, FORTRAN, and ALGOL. In each case, the programmer entered a specification of what he wanted, and the computer produced the program in the language of the machine.

In short, automatic programming always has been a euphemism for programming with a higher-level language than was then available to the programmer. Research in automatic programming is simply research in the implementation of higher-level programming languages.

III. Is automatic programming feasible? What does that mean?
Of course automatic programming is feasible. We have known for years that we can implement higher-level programming languages. The only real question was the efficiency of the resulting programs. Usually, if the input "specification" is not a description of an algorithm, the resulting program is woefully inefficient. I do not believe that the use of nonalgorithmic specifications as a programming langauge will prove practical for systems with limited computer capacity and hard real-time deadlines. When the input specification is a de-

change from our present capability.

IV. Will automatic programming lead to more reliable programs?
The use of improved languages has led to a reduction in the amount of detail that a programmer must handle

extant programming languages, while far from perfect, are not that bad. Unless we move to nonalgorithmic specifications as an input to these systems, I do not

search.

On the other hand, our experience in writing non-

mistakes in writing them just as they do in writing algorithms. The effect of such work on reliability is not yet clear.

matic programming systems are greatly exaggerated.

Automatic programming in a way that is substantially different from what we do today is not likely to become a practial tool for real-time systems like the SDI battle-management system. Moreover, one of the basic problems with SDI is that we do not have the information to write specifications that we can trust. In such a situation, automatic programming is no help at all.

Can program verification make the SDI software [illegible]

I. Introduction
Programs are mathematical objects. They have meanings that are mathematical objects. Program specifications are mathematical objects. Should it not be possible to prove that a program will meet its specification? This has been a topic of research now for at least 25 years. If [illegible] prove the SDI software correct? If it was proved correct, could we not rely on it to defend us in time of need?

II. What can we prove?
We can prove that certain small programs in special [illegible] would consider a 500-line program to be large. In discussing SDI software, we would consider a 500-line program to be small. The programs whose proofs I have seen have been well under 500 lines. They have performed easily defined mathematical tasks. They have been written without use of side effects, an important tool in practical programs.

Proofs for programs such as a model of the earth's [illegible] programs are larger, their specifications are not as neat or mathematically formalizable. They are often written in programming languages whose semantics are difficult to formalize. I have seen no proof of such a program [illegible] small size with mathematical specifications, machine theorem provers and verifiers are also strictly limited in the size of the program that they can handle. The size of programs that they can handle is several orders of magnitude different from the size of the programs that would constitute the SDI battle-management sys- [illegible]

III. Do we have the specifications?
In the case of SDI we do not have the specifications against which a proof could be applied. Even if size were not a problem, the lack of specifications would make the notion of a formal proof meaningless. If we wrote a formal specification for the software, we would have no way of proving that a program that satisfied the specification would actually do what we expected it to do. The specification itself might be wrong or incomplete.

IV. Can we have faith in proofs?
Proofs increase our confidence in a program, but we have no basis for complete confidence. Even in pure mathematics there are many cases of proofs that were published with errors. Proofs tend to be reliable when they are small, well polished, and carefully read. They are not reliable when they are large, complex, and not read by anyone but their author. That is what would happen with any attempt to prove even a portion of the SDI software correct.

V. What about concurrency?
The proof techniques that are most practical are [illegible] of systems of concurrent processes has focused on message-passing protocols rather than processes that cooperate using shared memory. There are some techniques that can be applied with shared memory, but they are more difficult than proofs for sequential programs or proofs for programs that are restricted to communication over message channels.

VI. What about programs that are supposed to be robust?
One of the major problems with the SDI software is that it should function with part of its equipment destroyed or disabled by enemy action. In 20 years of watching [illegible] one attempt at proving that a program would get the correct answer in the event of a hardware failure. That proof made extremely unrealistic assumptions. We have no techniques for proving the correctness of programs in the presence of unknown hardware failures and errors in input data.

VII. Conclusion
It is inconceivable to me that one could provide a rea- [illegible] the SDI software. Given our inability to specify the requirements of the software, I do not know what such a proof would mean if I had it.

Is SDIO an efficient way to fund worthwhile research?

The subject of this section is not computer science. Instead, it discusses an issue of concern to all modern scientists: the mechanism that determines what re- [illegible] 20 years of experience with DOD funding as well as experience with other funding mechanisms in several [illegible]

I. The proposal
In several dicussions of this problem, I have found people telling me they knew the SDIO software could not be built but felt the project should continue because it might fund some good research. In this section I want to discuss that point of view.

II. The moral issue
There is an obvious moral issue raised by this position. The American people and their representatives have [illegible]