46. Popper, Karl R., *Conjectures and Refutations*, New York: Basic Books, Inc. 1965.

47. Popper, Karl R., *Logic of Scientific Discovery*, New York: Basic Books, Inc. 1959.

48. Popper, Karl R., *Objective Knowledge: An Evolutionary Approach,* Oxford: Clarendon Press, 1974.

49. Pressman, Roger S., *Software Engineering: A Practitioner's Approach,* New York: McGraw-Hill, 1982.

50. Rogers, Hartley, Jr., *Theory of Recursive Functions and Effective Computability,* New York: McGraw-Hill Book Company, 1967.

51. Scott, D. S., "Outline of a Mathematical Theory of Computation," in *Proc. of 4th Ann. Princeton Conf. on Info. Sci. and Systems,* Princeton: Princeton Uni. Press. 1970. pp 169–176.

52. Spivey, J. M., *The Z Notation: A Reference Manual,* New York: Prentice-Hall, 1989.

53. Suppe, F., *The Structure of Scientific Theories,* Urbana, IL: University of Illinois Press, 1977.

54. Suppes, Patrick, *Introduction to Logic,* New York: Van Nostrand Reinhold Company, 1957.

55. Turner, A. J., private communications, July 24, 1990.

25. Hehner, Eric C. R., *The Logic of Programming,* Englewood Cliffs, NJ: Prentice-Hall, Inc. 1984

26. Hekmatpur, Sharam and Darrel C. Ince, *Software prototyping, formal methods, and VDM,* Reading, MA: Addison-Wesley, 1988.

27. Hesse, Mary B., *Models and Analogies in Science.* South Bend, IN: University of Notre Dame Press. 1966.

28. Hesse, Mary B., *Structure of Scientific Inference,* London: The Macmillan Press Ltd. 1974.

29. Hoare, C. A. R., "An Axiomatic Basis for Computer Programming," *Comm. A. C. M.,* **12(10)**, 1969. p 576–580, 583.

30. Hoare, C. A. R., "Proof of a Program FIND," *Comm. A. C. M., **14(1)***, 1971, pp. 39–45.

31. Hoare, C. A. R. and P. E. Lauer, "Consistent and Complimentary Formal Theories of Semantics of Programming Languages," *Acta Informatica,* **3**, 1974. p 135–153.

32. Jones, C. B., personal communication.

33. Kattsoff, Louis, O., *A Philosophy of Mathematics,* Freeport, NY: Books for Libraries, 1969. Originally published by Iowa State University Press, Ames, Iowa, 1948.

34. King, J. C., "A Program Verifier," Ph. D. thesis, Carnegie-Mellon University, 1969.

35. Knuth, Donald E., "Algorithmic Thinking and Mathematical Thinking," *The American Mathematics Monthly,* **92(3)**, March, 1985, pp. 170-181.

36. Kruse, Robert L., *Programming with Data Structures,* Englewood Cliffs, NJ: Prentice-Hall, 1989.

37. Lakatos, E., *Proof and Refutation,* New York: Cambridge University Press, 1976.

38. Manna, Z., *Mathematical Theory of Computation,* New York: McGraw-Hill, 1974.

39. Manna, Z. and Pnueli, "Axiomatic Approach to Total Correctness of Programs," *Acta Informatica,* **3**, 1974. p 243–263.

40. .Manna, Z. and J. Vuilllemin, "Fixedpoint Approach to the Theory of Computation," Artificial Intelligence Project Memo *AIM-164,* Stanford University, March, 1972.

41. Martin-Löf, Per, *Notes on Constructive Mathematics,* Stockholm: Almqvist & Wiksell. 1970.

42. Martin-Löf, Per, "Constructive Mathematics and Computer Programming" in *Proc. 6th Conf. of Logic, Methodology, and Phil. of Sci.,* Amsterdam: North Holland Publishing Company. 1979.

43. McCarthy, John, "A Basis for a Mathematical Theory of Computation," in *Computer Programming and Formal Systems,(*P. Braffort and D. Hurschberg , editors), Amsterdam: North Holland Publishing Company. 1963. p 33–70.

44. McCarthy, John, "Towards a mathematical science of computation," in *Information Processing 1962 (Proceedings IFIP Congress),* 1962. pp 21–29.

45. Mili, Ali, *An Introduction to Formal Program Verification,* New York: Van Nostrand Reinhold Company, 1985.

5. Berg, H. K., W. E. Boebert, W. R. Franta, and T. G. Moher, *Formal Methods of Program Verification and Specification*, Englewood Cliffs, NJ: Prentice-Hall, Inc. 1982.
6. Bjørner, D. and C. B. Jones, *The Vienna Development Method: The Meta Language,* in *Lecture Notes in Computer Science, number 61,* Berlin: Springer-Verlag, 1980.
7. Bjørner, D. and C. B. Jones, *Formal Specifications and Software Development,* Englewood Cliffs, NJ: Prentice-Hall, 1982.
8. Brent R. P., "Algorithms for Minimization without Derivatives," Prentice-Hall, 1972.
9. Brooks, Fred, *The Mythical Man Month: essays on software engineering,* Reading MA: Addison-Wesley Publ. Co., 1975.
10. Cohn, Avra, "The notion of proof in hardware verification," *J. Automatic Reasoning,* **5**, *1989.* pp 127–139.
11. Dekker, T. J., "Finding a zero by means of successive linear interpolation," in "Constructive aspects of the Fundamental Theorem of Algebra," (B. Dejon and P. Henrici, editors) Wiley-Interscience, pp. 53-61, 1969.
12. DeMillo, R, R. Lipton, and A. Perlis, "Social Processes and Proofs of Theorems and Programs," *Comm. A. C. M., **22(5)**,* May 1979. pp 271–280.
13. Dijkstra, E. S., "The Cruelty of Teaching Computer Science," *Comm. A. C. M.*, **32(12)**, Dec., 1989. pp 1397–1404.
14. Dobson, J. and B. Randell, "Program verification: public image and private reality," *Comm. A.C.M.*, **32(4)**, April, 1989. pp 420–422.
15. Dudley, Richard, *Correspondence* in *Computers and Mathematics, Notices Amer. Math. Soc.,* (J. Barwise, editor), **37(2)**, Feb, 1990. pp 123 – 124.
16. Fetzer, James H., "Program Verification: The Very Idea," *Comm. A. C. M.*, **31(9)**, September, 1988. pp 1048–1063.
17. Fetzer, James H., (letter), *Notices Amer. Math. Soc.,* **36**, 1989. pp 1352–1353.
18. Floyd, Robert W., "Assigning Meaning to Programs," in *Mathematical Aspects of Computer Science,(*J. Schwartz, editor), *Proceedings of Symposia in Applied Mathematics, 19.* Providence, RI: Amer. Math. Soc., 1967. p 19–32.
19. Forsythe, G. E., "What is a satisfactory quadratic equation solver," in "Constructive aspects of the Fundamental Theorem of Algebra," (B. Dejon and P. Henrici, editors) Wiley-Interscience, pp. 53-61, 1969.
20. Gerhart, Susan, private communication.
21. Gries, D, *Programming Methodology, A Collection of Articles by Members of IFIP WG 2.3,* Berlin: Springer-Verlag. 1978.
22. Gries, D., *The Science of Programming,* New York: Springer-Verlag. 1981.
23. Hamming, Richard, *Numerical Analysis for Scientists and Engineers,* 2d ed., McGraw-Hill Book Company, 1973.
24. Heise, David R., *Causal Analysis,* New York: John Wiley. 1975.

ple claim that all functional code is warped.) No one in the class, not even the most brilliant mathematical minds managed to get a plausible proof, and if they did I contend that it would be of no real assistance because, to my thinking, a proof's first purpose is to convince someone of correctness, and a proof-by-intimidation is not really convincing. (Keith Duddy)"

Proof Systems. Current proof techniques are undeniably difficult for the part time practitioner to master and use. The romanticizing aspect arises through the expectation that proving things 'for real' is a mechanical exercise. To paraphrase Richard Hamming [23], "the purpose of a proof is insight, not formulae." While the purist may not approve of informal methods, their worth is undeniable.

The current practice suffers from the tyranny of exactness. The proof techniques basically mimic the sequence and semantics line for line. This approach is useful in many cases but does not lend itself well to informal reasoning. A problem similar to this arises in real and numerical analysis. For example, in analysis, the question of convergence is similar to termination. Except in numerical problems, the question of how quickly an approximation converges is not important and so the path of least resistance can be taken. In numerical work, however, the exact rates become important and are much harder to derive.

5.3. Epilogue

The final thought that the author wants to leave is a little story. The original idea of this survey was to get programmer reactions. The author had hoped, as well, to get some humorous remarks; often humor provides unique and deep insights. Such a remark was forthcoming and it puts everything into perspective: "If you were the first patient to a robot dentist, would you want the robot debugged or proven correct?"

6. Acknowledgment

The author wishes to acknowledge the very able assistance of Eugene N. Miya, RIACS, NASA Ames, Sunnyvale, California. Eugene is a particularly fine source of materials and historical notes. Thanks also to David Sykes, Department of Computer Science, Clemson University, Clemson, SC. Dave proof read the paper early on and provided many valuable ideas for arranging the material.

7. Bibliography

1. *ACM*, "Scaling Up: A Research Agenda for Software Engineering," *Comm. A. C. M.*,***33(3)***, March, 1990, pp 281–293.
2. Barker, Stephen F., *Philosophy of Mathematics,* in *Foundations of Philosophy* series, (Elizebeth and Monroe Beardsley, editors), Englewood Cliffs, NJ: Prentice-Hall, 1964.
3. Barringer, Howard, *A Survey of Verification Techniques for Parallel Programs,* New York: Springer-Verlag. *Lecture Notes in Computer Science, 191.*
4. Barwise, Jon, "Mathematical Proofs of Computer System Correctness," *Notices of Amer. Math. Soc.*,**36**, 1989. pp 844–851. See also **37(2)**, Feb, 1990, p 124.

[1]. Proponents of functional programming tout the correctness properties are easy to establish due to the functional paradigm.

there are oracles (programmers) who must develop programs without controls. There are two ways in which specification can be useful:

1. If the specification is significantly shorter/simpler than the implementation. If this is the case, then the specification is much easier to comprehend as a whole than the program, and it is easier to tell whether the specification captured the problem than it is to tell if the implementation solves it. Mathematical programs, because of the formal nature of their subject matter, often admit to succinct specifications.
2. If the specification may be used to short-circuit the testing process. The use of the specification as intermediate form of agreement between user and definer has some merit. However, the specification must capture the ultimate environment. Testing at this level might make the situation clearer—it does not, however, obviate any verification requirements for the code.

Academic Perspective. The academic perspective reinforces the unrealistic view of program design and development. This is a long running debate in mathematics. Do you teach the justification of theorems or do you teach the methods of discovery [36]. The reasonable answer is, "Both." However, to look at texts, it would appear that only the justification is important and that the long, arduous process of putting the theory together is uninteresting. Looking at the current curriculum from the *ACM/IEEE*, it is apparent that what little logic and mathematics the students get will only make matters worse.

One opinion was "....at least in the United States, many programmers have a weak math background—from the formal side, that is—which disinclines them to formal methods of ANY kind. They first have to understand formal methods, then get into proofs of code, then see if it has any real impact on their systems. When the goal is to look at technology to address the latter, I believe people find the first two steps a bit imposing in a production environment."

Programming Languages. Just what is the role of the language in all this? Is language part of the solution or part of the problem? For example, if we cannot define the common sense notion of language semantics in a usable way, are we addressing a correct problem? For the most part, experienced programmers are able to determine the semantics of a construct with little or no confusion, so naive semantics must be quite close to the more formal versions. In fact, programmers probably deal with the vagaries of machine and compiler dependent "features" in the same manner they are able to deal with inconsistencies in natural language.

Given a new role of specifications, is it necessary to deal with programming language at all? Are the questions raised by the semantics argument—semantics is not well specified for programming languages that are commonly available—really an indictment of the programming language idea itself. The original program semantic model was "a program is what it computes." That has long since fallen by the wayside. What has happened is the emergence of several layers of 'programmers.' Each layer has distinct needs an standards of 'correctness.' These layers are often termed *virtual machines.*

Are programming languages part of the solution space or the problem space. One person with first hand experience would say the problem space. "The other subject (of an experiment in proving) was a functional programming[1] subject, and we attempted a proof of a simple interactive mastermind program, using fixed point theory. Here it is also really a process of refining the program to make it easy to prove—ending up with some drastically warped code (some [unenlightened] peobetween physical structure and mathematical behavior. Barwise called the failure to recognize that the system under investigation is a model the *fallacy of identification*. Proponents would do well to read Kattsoff [33] or even Suppes [54].

Complexity Issues. The common approach is to describe programs a theorems to be proven. If we instead identify program *statements* with *theorems*, then a program is more like a *theory* than a *theorem*. The De Millo, *et al.*, argument still holds: certainly mathematical theories are subject to the repeated scrutiny of the community. But this scrutiny now must address the efficacy of the underlying premises of the model. An emphasis on modeling as the basis of programming would support the *algorithm versus program* distinction. An algorithm is a model for all its implementations.

This puts the task into perspective. We are asking the programmers to develop and prove a whole theory. Given this view, we can question the reality of the demands that programmers treat each program as a theorem. As an example of how long it takes to get a problem solved correctly, we offer the simple concept of zero-finding. The reader is probably familiar with both the bisection and secant methods of finding zeros to a nonlinear system of equations. Each has numerical pathologies; Dekker [11] combined the two to try to eliminate these pathologies. Even so, the problems persisted [19]. Finally, Richard Brent closed the loopholes [6]. Moral: it took about three hundred years to do it right! Meanwhile, how many people had looked at this problem? The ‘specification’ had not changed—but our understanding and tools have.

Specifications. The role of specifications no longer would be the supposed cure-all. The specification merely records what is known about the model at a particular point in time. The ‘incorrectness’ of a specification is not something epistemological; rather that incorrectness is an incompleteness in the description of a system.Is not testing a natural part of verifying the specification in the same way that experimentation is used in physics? But if this is to be testing, then it is certainly unlike any normal concept of testing, as there seems to be no concept of induction; indeed, no idea of ‘logic’ at all. The problem with the specification approach is that there is no guarantee that the resulting specification can be computed at all (it might specify a non Turing computable function) or that the resulting program is useful.

Software Engineering. The view of algorithm development and verification *vis-á-vis* program development and verification is unreasonable. To ask a single person, or even a small group, to develop, on her or his own, a theory about anything is simply not reasonable. The gap between theory and practice is too wide to serve as a useful model. Homespun advice by Bert Sutherland, “Programs are like waffles. You should always throw the first one out.”

There also seems to be a sin of gradualism in the system design world. This would appear to be the result of the experiences of IBM in the introduction of their *370* lines of computers. The massive upheaval caused by the radical change in architecture has caused marketing driven organizations to be very conservative, hanging on to outdated methods and modes of thought simply to avoid a revolutionary system design.

The work on specification should be seen as interesting, but not necessarily a cure-all. As pointed out earlier, there have been many “cure-alls” proposed over the years. This is particularly true if

[1]. The literature is incredibly vast in this area. The interested reader should see texts in the area of *functional programming* and *logic programming.*

ogy that a program is a theorem is flawed. A large program is more like establishing a *theory* than proving a *theorem*.

Another legitimate question is, “Are we asking too much?” In particular, are the development time frames of large software systems too short relative to the amount of work to be done. From the above analogy, a 10,000 line program is compared with a theory with 10,000 theorems. Certainly we do not see mathematical theories done in such short time frames.

5.1. Conclusions

Conclusion 1. Mathematics (and lately, physics) has been romanticized by the proponents of formal methods in computer science. Certainly, no one will quarrel with their goal. The criticisms of Barwise, De Millo *et al.*, and Fetzer pointed to difficulties in approach. The error appears in not understanding how mathematics (or the physical sciences) is actually done in practice. An colleague may have put it right: “Every constructive proof is an algorithm waiting to be discovered.”

Conclusion 2. Despite fifty years of computability theory, there is little help to the professional programmer in the same way that fifty years of control theory has helped the engineer. There is no real classification of models and algorithms. Even though formal languages have matured, few practitioners are cognizant of their use anywhere except in compilers. It would seem that we need more structure on the realm of computation. For example, where is the book “Counter examples in Programming?” or “Canonical Problems in Algorithms.” A European colleague once asked me, “Is complexity theory all there is to computability?”

Conclusion 3. There is no classification of problems by method of attack. This would seem to indicate that the wrong abstractions are being used. If a program is more like a theory than a theorem, then we should look toward ways to classify useful distinctions. If algorithms are like models, then we need to develop and classify useful models in the same why the physical sciences do.

Conclusion 4. The very fundamental concepts of computer science are not yet well-defined. In so far as there is a philosophy of science and a philosophy of mathematics, there is no philosophy of computation. Until fundamental concepts are well-defined, fundamental misunderstandings and errors will continue to abound.

5.2. Future Directions

Modeling Issues. The days of the simple von Neumann machine seem to be long gone. With it, some cherished ideas about what is an algorithm have changed. For example, we now have probabilistic algorithms; these would not have been allowed twenty-five years ago[50]. New and exotic architectures have also changed ideas about what is and is not an algorithm. Massively parallel machines are far different than the von Neumann architecture and languages used to develop these proof ideas. Here, then, is the essence of *model.*

The model of computation most used for development is the so-called *von Neumann model.* This model is the deterministic memory-bus-computation unit model which so easily describes the personal computer. This model simply does not address the realities of advanced architectures. Not only are these architectures physically different (multiple memories and *CPU*s), but they are primarily non-deterministic. This has great ramifications on languages[1] and proof systems[3].

Even though we see the conduct of logic as a symbolic ‘game,’ with axioms and rules of inference, the pragmatic use of logic in mathematics and science means we do establish some identification

4.6. Other Comments

The verification milieu fails to make a distinction between design and discovery on the one hand and coding and justification on the other. The Fetzer arguments reinforces this distinction by differentiating between the algorithm and the program. The Hoare model suffers from the problem of any deductive system: the premises limit what can be discussed. The discovery of an algorithm often rests on insights of relationships not originally stated or even implied by the problem statement.

Given that the discovery of an algorithm is often inductive and creative, we see that some ‘testing’ is inevitably required in the actual development. This engenders the following question: are testing and proving necessarily mutually exclusive. Put another way, are the differences in algorithm and program such that one is provable but the other is only testable. Therefore, one of the goals of further research should be to quantify the difference or remove the question altogether.

There is an obvious use of testing— ferreting out of errors in the specifications. And how are we to deal with the specification problem? The current drive toward formal methods and specification systems seems only to be shifting the focus rather than actually addressing the problems. What is the difference of a prototyping system like *VDM* and *Z* and any other language. Is it not just the programming language problem all over? For example, if the specification has a ‘bug’ and the program does not is there still a bug? Are specifications iterative by their very nature since they represent an agreement among people? Iteration occurs in science—we might consider the evolving model as equivalent to a specification.

There seems to be an “all or nothing” mentality among programmers. While it is true that the idea of a “partially working program” is unacceptable, it should not be true that partial understanding is also unacceptable or unusable. Coupled with this mentality is the demand that there be some sort of tool which does all the ‘hard work’. A common complaint about proofs revolve around how hard they are and that there should be a *tool* to do all the hard work. Such a desire is hardly new [34]. This seeming need to have the machine assistance brings up the question of proof checkers *versus* proof generators. The halting problem properly limits the ability of proof generators. But there is no characterization of these limits.

5. The Crux of the Matter

The place to start is to ask “how valid are the analogies drawn by computer scientists and their view of how other disciplines actually work.” Looking at the standard proposed, it appears that the proponents take their ideas from mathematical—and perhaps physical science— texts, not from their practice. The Barwise argument—that the problems of program proofs are the same as in any applied mathematics[1] work—seems essentially correct as far as it goes. Barwise rightly discusses *models*. Indeed, models are the grist for the applied mathematical mill; however, formulation of models is more the purview of science[2] as a whole rather than just applied mathematics. The anal-

[1]. One needs to be careful as to the definition of *applied mathematics*. For this work, consider applied mathematics as the use of mathematics to investigate models. Hence, it is seen as a subject and not a title.
[2]. Like applied mathematics, *science* is a loaded term. Our use here is that science deals with formulating and validating models.

One might legitimately ask how much of the resistance is due to lack of familiarity and practice. The current ACM/IEEE curriculum recommendations do not emphasize formal methods. It is certainly true that the subject is daunting at first blush and a lack of practice would explain many of the reported problems in choosing a usable approach. And in all fairness, having a one course introduction would hardly pass for a curriculum which requires the students to verify *all* their programs.

4.4. Programming Language and Systems

Language systems should help here, but they suffer from problems of implementation, optimization, and uniform availability. An obvious objective is to have a 'compilation system' which everyone uses and agrees on. Such proposals have been made before—and still there is no standard. The current situation works against such a system: as the newer, exotic architectures become available, the language paradigms must change. To put the language problem in context, the situation in programming is the same as it would be in mathematics if every journal had its own notation and did not provide a description of how its own notation related to others.

The concept of 'development' or 'programming' environments is becoming more acceptable. Environments were almost exclusively associated with artificial intelligence, with *Lisp* being the motivator. Now, with the complexity of parallel and distributed systems, it is necessary to have such tools as 'smart' editors and interactive / incremental compilers. Such tools hold some hope for more formal development support.

One of the underlying realities of system development is its evolutionary nature. Once a system has been implemented and if it is accepted in a 'commercial' sense, then such a system rarely is recreated *de novo.* Scientific programs tend to evolve with knowledge of the physical system. Rewrites of such systems tend to occur only when the system is moved to a new computer. Such moves will become more common as the more exotic architectures come into general use.

While much effort has be put into formal semantics—both axiomatic and denotational—there has been little impact on real languages. For example, neither *C* nor *Fortran* have a universally recognized semantic definition. The current revisions of both languages do not even pretend to have a formal definition in concert with either approach. *VDM* is used for the work for the new ISO standard *Modula* [32].

4.5. Proofs and Proof Systems

We must see the complaints concerning proof systems as a central issue. Regardless of the particular methodology, proving or verifying nontrivial programs correct is difficult and error prone. This difficulty is escalated by the newer architectures, such as hypercubes, where there might be on the order of one hundred processors working on the same problem. Such programs are side effects at their worst. Indeed, the whole side-effects issue works against programming languages as well.

After relating a "horror story,' David Thomas of Hewlett-Packard made the following observation, ".... The point is that no matter how accurate the 'proof of correctness' is, it's still based on a model of the real usage of the program, and like all models it suffers from the 'close world theorem', what isn't considered isn't important."

4. Observations and Analysis

4.1. Economic Concerns

There is no denying the economic argument. We should also see that routine analysis of programs and routine verification will not become a fact of life until one can overcome the economic argument and make benefits perceptible. For the near term, it is unlikely that quality demands will push the technology. Economic viability will come about only with software assistance or the development technology.

There is no denying that management in commercial programming is faced more with economic pressures than with quality issues. An oft heard lament is summed up by the phrase “There’s not enough time to do it right, but there’s time to do it over.”

4.2. Software Engineering Concerns

The complexity issues are not going to go away if the current design and implementation technologies are used. By and large, new systems start *de novo*—although scientific programs are more likely to start with an old program. Therefore, every system must recreate and solve all the problems that have been solved in the past. Competitive forces work against doing otherwise. People, too, work towards implementing their own vision, often believing that they will not fall into the same traps as their predecessors.

A great deal of research activity is going into the specification problem and the associated concepts of *prototyping.* It is hard to see how this will change the outcome. It appears that if one were able to get a correct specification, then one could verify program correctness. But then the problem is to verify the specification and we have a regress problem. While the specification/prototyping approach may be a good engineering practice, it does not necessarily solve the fundamental problem. What we see is support for the Barwise contention that verification is an applied mathematics problem wherein models of the physical situation must be the basis of the mathematical equations. “I’d rather spend my time building prototypes. After all, a prototype *is* a complete specification—one that can be proved directly against the objective rather than against an abstraction of what the objective is imagined to be.”

After searching the literature, the author cannot find any long term empirical study on the efficacy of the various software engineering techniques—reviews, walk-throughs, testing, formal methods, *etc.* In a recent report on the state of software engineering [1], it was noted “... [users and producers of complex software systems should] adopt a more realistic vision of the complex software system development process.”

4.3. Educational Factors

Educational factors are harder to judge. It is true—in the United States—that there seems to be a reluctance on the part of the professional educators (which is not necessarily the same as the faculty) to place heavy emphasis on program verification issues. European practice, on the other hand, seems to emphasize—or at least promote—formal elements. The new ACM/IEEE curricula recommendations will specify that the students must at least be introduced to the various approaches [55]. Most of the apparent misunderstandings seem to be educational in nature. Any new curricula are unlikely to alleviate the situation; such must come from industrial pressures.

Partial *versus* total correctness figured prominently as a purported stumbling block. The difficulties of coming up with the total correctness proof was often cited in terms of the complexity argument presented earlier and the obscure tricks argument presented above. More than one person complained that there should be some ‘enlightenment’ at the end of the proof; usually, there is no increase in understanding.

A number of respondents gave the impression that the proof was an end in itself. This seemed to be connected most often with the idea that the proof is a redundant derivation of the program. This redundancy is not seen as a positive aspect of the system. For example, many alluded to the problem of *debugging* a proof, and that debugging a proof is harder than debugging a program.

3.5. Apparent Misunderstandings

This section contains some misunderstandings and misconceptions concerning the milieu of program proofs. While these statements do not represent valid criticisms, it might indicate the breadth of the ‘public relations’ problem.

3.5.1 Computability and Logic

There seemed to be a great deal of confusion as to just what a proof is supposed to be and do. Most of the comments simply showed a poor background in logic. Given the curricula in computer science, computer engineering, and related areas, such lack of background is not surprising. Several comments were made that mathematicians do not ever formally prove anything, so it was not important that computer programs be proven correct.

The halting problem figures prominently: the halting problem is undecidable, thus you cannot prove programs correct. Enough said. There is one comment, however: many seemed to believe that program proofs could only be done by another program. Such a view might legitimately use the halting problem as a counter example. Even deeper, “A theoretical question for you: since a programming language is Turing-complete, isn’t it possible to write a correct program which cannot be proven and/or write an incorrect program which cannot be disproven?”

3.5.2 Semantics

There were many comments relating to semantics. Those which were specific to languages or language definitions were lumped under Programming Language Design and Implementation on page 10. There were many who felt that proofs could only be done if *axiomatic* semantics were available. No one mentioned the lack of understanding of the underlying algebraic (object) structures as a cause for concern.

3.5.3 Numerical Analysis

There seems to be complete confusion over the correctness of a program *vis-á-vis* correctness of an approximation. There were many comments to the effect that the programs implementing an approximation cannot possibly be correct.

There was also the confusion—totally unanticipated—over numerical error and its role. The number of comments showing complete confusion over the (analytic) real numbers and the floating point numbers was entirely unexpected. In this vein, however, is the legitimate concern that the ‘standard’ compilers are anything but standard when it comes to implementation of floating point operations and built-in mathematical functions.

several spoke to ‘literate programming’ techniques as not being explored. “Too much entropy in the language field,” noted one respondent.

As if to underscore the problems with language, several mentions were made of improper optimizations done by compilers. One respondent asked rhetorically, “When will compiler writers find out that you cannot rearrange floating point expressions?” One can also point out that most compilers have different semantics based on the level of optimization invoked. Over the years, most *Fortran* programmers have learned that the optimizer plays by exact rules while unoptimized code may even ignore basic rules. For example, *Fortran* functions are not allowed to alter their arguments; optimizers tend to enforce this even though the non-optimized code does not.

The question of support at the language level for program proofs arose frequently. It seems to be a given in computer science—no tool, no work. Someone commented, “If the theorem provers could really run over REAL code, that’d be one thing. But the only ones I’ve heard much about [although I admit I don’t listen to this kind of stuff very attentively] required *extra random* statements to be mixed in with the code....” Certainly, this is a legitimate criticism of the current state of proving programs correct during development. Even though there is some movement in this area, most environments for ‘production languages’ are quite primitive.

3.4. Problems with the Proof Systems

Some respondents doubted the completeness of the proof system as presented in the literature. The impression is that these respondents were questioning the application, not the proof systems themselves; *i. e.,* what can and cannot be proven with these systems. This was made clear as one person wanted to talk about proof of completeness without reference to specifications; some sort of *absolute* correctness. There seems to be a high degree of confusion over completeness *versus* soundness *versus* just don’t know.

Many comments were made relative to the proofs being more subtle and complicated than the programs. There are several categories of comments.

1. There is a dearth of models to choose from as far as the proof itself goes. For example, in mathematics there are certain more or less standard approaches one takes based on the class of theorem or function.
2. There are few informal techniques. Closely coupled with this is the difficulty in formulating the pre- or post-conditions, invariants, or assertions (depending of ones approach). The lack of heuristics is seen as a deterrent.
3. Along with technique complaints, there were complaints about the difficulty in dealing with semantics of implemented languages. In particular, the difficulty of dealing with side effects was cited.
4. There were several comments relating to the fact that the proof is a ‘redundant’ development of the program or system. Within that, there is mention of complementary errors—the same error in the program and the proof. A good point made often is that the ‘tricks’ needed obscure the algorithm and are just as fuzzy as the program. “Proving that a program is right is like having parity check. In the case of the parity, there is one more bit that can be wrong; in the case of the proof, there is one more piece of logic that can go wrong.”

one seems to know how to preserve the proof across the various systems, any proof is probably altered in the transfer.

Porting also impinges on language issues. Since compilers for different manufacturers have different characteristics, it is impossible to port code without some rewriting. Those with experience porting numerical code have quite a difficult time: the different floating point representations give different—but not necessary ‘wrong’—results.

Environmental Factors. An standard comment was that the *environment* is uncontrolled. In this case, the environment consists of everything outside the program: compiler, libraries, operating system and hardware. None of these is subjected to the rigor being demanded of the system. This is, of course, the current situation. Perhaps in better times “People prefer to get the job done, rather than spend a whole heap of time learning new techniques, investing in the tools to make it tolerable and building up a library of proven trusted code. I’d love to see it someday, but I’m not holding my breath.”

As a software / system engineering question, what role should the *dicta* of structured programming, modularity, *etc*. take? Many of the *dicta* are introduced with no clear theoretical grounds. These questions seem to straddle system engineering and language issues.

Testing. As should be expected, there were a large number of points given *for* testing over proofs. Virtually all proponents stated that it was easier to do and more productive. Such messages usually tied themselves to the view that 100 percent perfection was not needed. Of the program testing comments, however, no one raised the complexity issue nor the inductive issue.

3.3. Programming Language Design and Implementation

Programming languages played a prominent role in the comments. Most importantly, many felt that current programming language designs hinder if not confute program proofs. The basis for this feeling is that semantic specifications are missing or too cumbersome to use in practice. A large number alluded to the fact that the language of the algorithm and the language of the proof are too different to be of use. In this regard, a common complaint was that the resulting proof was harder to understand than the program itself.

The question of semantic specification of languages has a long history. Many respondent seemed to believe that only an axiomatic specification would work. Others pointed out work, started by Bjørner and Jones [7], which has defined several languages with an effective operational definition. The same comments can be made of axiomatic and denotational methods. All these definitions are *post hoc* as far as the author has been able to find out. This means that the compiler and the specification are not connected in any causal way.

Some wondered if there should be program schemata which would aid in the proof process. In this same vein, some questioned the role of *dicta* in programming practice. Some mentioned that these might be without grounds and would actually confute the proof process. Several mentions of interfacing systems such as *IDL* might solve some of the specification / interfacing problems. A point often mentioned was the inability of current programming languages to accept information about the operational environment.

Another common complaint had to do with the unsettled nature of languages: either that they were always changing or that they lacked the features to make program proofs feasible. For example,

Early Activities. Early activities in development are aimed at understanding the environment in which the product will operate. Often these activities revolve around the human interactions and activities. In systems with human interactions, it is virtually impossible to complete specify how the input may be entered. In those cases where the input is from another program, there is some hope. Unfortunately, the complexity is extremely high.

The principle activity early on is that of design. There have been many approaches to designing software systems. Many of these have been attempts to emulate another successful discipline like physics or engineering. There have been design languages which emulate engineering, specification systems, and the so-called *CASE (*computer aided software engineering*)* tools. Among the obvious criticisms of these approaches has to be the fact that the all are independent of the final code.

Coding. One effective issue in coding is the adherence to the specification. If there is not a direct link to the specification, then one cannot guarantee the result. As one comment puts it, "If the program is correct, the bugs are in the spec."

The one development issue raised was that of *defensive coding*, although it was not always clear what the term meant. Usually, it means explicitly checking all assumptions before proceeding; but no one ever seems to know what to do if the assumptions are not met. Those who championed defensive coding seemed to be saying that this technique put all the force of a proof into the code, thereby obviating the necessity of a separate proof. In an ideal world, the specification does not change during coding activities; such is hardly ever the case in reality.

Maintenance. At the 'other' end of the cycle, maintenance provides some insight as to why the cycle breaks down. These points are expanded below. The cycle breaks down because of

- Changes in the environment that the users institute outside the computer system.
- Need for 'porting' the system to a completely new computer system. This system may be the same type (von Neumann, parallel, *etc.)* or a completely different type.
- Changes in use brought about by business changes in the users' needs.
- Multiple versions of the same program due to local needs.

There have been many studies, reports, *etc.* on the cost of maintenance for systems. All these pressure work against expending effort to prove correctness unless there is a way of transforming the proof.

Evolution and the Changing System. More than anything else, systems evolve. This evolution may be intellectually smooth: changes are logical and incremental. These changes are usually based on experience or changes in the environment. Such changes are rarely smooth from the programming standpoint; this works against proving systems since the evolution of the system will eliminate large sections of code. Trivial changes of the specification tend to propagate throughout the code, perhaps changing whole subsystems. At some point, however, the system inevitably goes through a "massive" change. These changes would swamp any attempt to carry through proofs.

Porting. The question of *porting* code is a critical one. The idea of porting is that a program which runs on a particular computer is made to run on another computer with (usually) different hardware manufacturer (inclusive) or different system software. Experience indicates that even the most carefully written software must be altered to make the program run on the new system. Since no

The specification may be made in several different ways. The issue was put into focus by the comment, "[The] proof only shows that program implements the specification. Who cares if you correctly implement an incorrect spec?" Another view is given by "The specification is the key and it is uncontrolled." Many respondents pointed out that the specifications are constantly changing. Even more to the point is the comment that getting a correct specification is even harder. "[It] may well be more error-prone than the program."

"Understanding the problem is really the hard part." There are major efforts underway in the specification (now often called 'prototyping') arena with *VDM* [26] and *Z* [52], to name two. There are two problems: (1) Getting the specification and (2) verifying that the specifications are 'correct.' It is not clear what a correct specification is. "Understanding the originally stated problem is the really hard part.[It is the] Necessary and sufficient part of getting specification." The size of the specification may be even more daunting. One comment to this effect is "V7/bin/mail source: 554 lines. 1989 X.400 specs: 2200+ pages." Some aspects or programs seem to be impossible to specify, "I wrote a program to play the game of Go. What is 'correct'? Plays perfect moves? That problem is PSPACE-hard (perhaps not even in PSPACE!). Writing a perfect Go program is easy if you can wait around 361! seconds for the answer."

There were several major areas mentioned as sources of difficulties in providing specifications.

- The first was the capturing of human interactions. B. F. Skinner not withstanding, humans are not automata and hence it is hard to specify with certainty how they will conduct themselves.
- The second area is that of exceptional conditions. Exceptional conditions alter the normal control flow and can occur at (potentially) any time. Concepts of failure are an active area of research in computer science.
- The third is capturing the input / output specifications.

Even given that one can in fact produce a specification, there are two major problems. Firstly, one must understand the specification before program design begins. Unless this specification can be processed directly, there can be no guarantee that the specification used in the proof is the original. Secondly, when the specifications change, the program and the proof both change. One does not want to reprove the entire system. Most readers who are also programmers have had program changes propagate in strange ways. "A correct program only means the bugs are in the specs."

Exceptions make the program follow non-sequential control paths. Exception handling is a subproblem of the general parallel and distributed program[3]. Experience with the more exotic systems—multiprocessor Crays, Alliants, or iPSC hypercubes—points of the extreme difficulty of dealing with non von Neumann architectures. Presumedly, the art of specifications will catch up with these different architectural paradigms. These systems have been notoriously difficult to program. Even moving a working program from a von Neumann machine to a hypercube is difficult.

3.2.2 The Development Cycle

The development cycle as portrayed in many texts (for example,[49]) may be utterly unrealistic. The texts do not even address the issue of how proofs might fit into the development of systems. As we discuss below, the problem of proving programs seems to be one related to the specification. For many reasons, formal specifications are difficult—if not impossible—to complete.

One academic claimed to have a friend who took statistics about what various people claimed to know during interviews for employment. The estimate was that less than 20% of all computer science majors have been exposed to formal methods.

Psychological and Cultural. There are cultural biases introduced through a variety of means. Many spoke of not being interested in correctness because a professor had stated a similar (or stronger) position. There was indirect mention of mathematical [1]. Several remarked that language rules were not fixed, although it was not clear what was meant. Such a complaint also belongs to Programming Language Design and Implementation on page 10.

After reading several responses, it became clear that 'math anxiety' is rampant among programmers. The fear of rejection of the proving software was quite evident. A good point was made in the observation "People resist vigorously any attempt to clean up their code." So, too, is resistance to critical thinking and evaluation: "Well, I do [prove all my programs]—the proof is in my head, that's all." It is still hard to appeal to a proof without some sort of societal review—especially when carried out in ones head. "... too much like maths (sic), programming is a sort of art that SHOULD have an element of uncertainty and danger to keep programmers from brain-death...."

Several comments fall under the human factors umbrella. These include the psychological tedium of doing proofs to statements of belief that all errors in programs were 'dumb' errors anyway which did not need such high power mathematics. There were, of course, many complaints about being too hard. Eric Freundenthal of the NYU Ultra project did admit to having been "burned by a really clever—but wrong—sync[ronization] algorithm, and now I sometimes actually do it."

Managerial Issues. There were several respondents who claimed to be a direct supervisor of programmers or a manager higher up the chain of command. There were also comments from the programmers concerning their management's understanding of the verification issue.

Programming managers are often not computer scientists but rather managers culled from throughout a company. As such, there is no particular incentive to have the software proven. The time-budget axes described in Economic Realities on page 4 are the guiding principles. Some mention was made about lack of appreciation by managers for verification or those managers who did not wish to investigate the technology.

3.2. Software Engineering

Broadly speaking, *software engineering* is an attempt to apply a discipline to the activities of developing a software product. Basically, one sees the so-called *water-fall* model[2], which speaks of activities related to feasibility, design, coding, and maintenance. While the literature is vast, an introduction can be found in [49].

3.2.1 Reasons Revolving around Specifications

As related in the Introduction, the concept of a specification is central to the proof of correctness. The specification must capture all the nuances germane to the actual functioning of the program.

[1]. Questions of algorithmic thinking *versus* mathematical thinking is an open area. See, for example, [35].
[2]. Barry Boehm, now of *DARPA*, is generally acknowledged as the inventor of this model along with Fred Brooks [9].

proposed as an answer to the long and complex nature of a long proof. On the other hand, one respondent observed, “Most (>95%) of all code is what I would call ‘banal’— that is, more or less obvious stuff, not tricky implementations of clever algorithms.”

The worst thing about the complexity problem is that it confutes a “debug and iterate” strategy. It was suggested by several respondents that perhaps some form of ‘abstraction’ might solve some of these problems, but no one elaborated an approach. Modularization has been held up over the years as an approach. Object-oriented programming has also been touted. However, neither has provided examples of actual systems and their proofs.

3.1.2 Quality Issues

Several comments about the efficacy of correctness revolve around recognizing a need for certain types of software. Two types mentioned were “life-critical” and “military secure”. In other cases, most comments mentioning this issue noted that there is a tolerance to error. Those not supporting verification activities cited testing as the alternative of choice.

3.1.3 People Issues

Since verification is an activity conducted by people, people’s attitudes and training become major considerations. In this section, we comment on academic training, cultural biases instilled by that training as well as the personal environment of the working programmer.

Academic Issues. The most common complaint was that the subject is dull and boring. The way it was presented to the respondents “turned them off.” One respondent called the whole thing a “fantastic magic approach” to learning. Those with a positive feeling toward the classroom presentation remarked that there were not enough simple programs to practice on or that they were not encouraged in later courses to use these skills. Some noted that they had been encouraged to formulate informal conjectures about their code but there was no emphasis in later classes. Several respondents did admit to not feeling comfortable with the high degree of abstraction they felt was needed to do proofs of any type. Typical comments range from “Students aren’t convinced and won’t use the proof techniques again” to “People want to remember why [something is correct], not how [to prove it].”

The only direct evidence from teaching came from Snorri Agnarsson. “In (my) course I require a loop invariant for every loop, pre- and post conditions for every procedure and function, and usually require data invariants for each implementation of an abstract data type....uses Robert L. Kruse’s book[36]... I have found the students to fall rather neatly into three groups:

1. One group is able to understand and use pre- and post conditions and loop invariants. This is a rather small group, about 20% of those who pass the course. All in this group pass the course easily.
2. One group is able to understand and use pre- and post conditions reasonably well. Almost all of those pass the course.
3. One group never understands pre- and postconception. Everyone in this group fails the course. In this course I require a loop invariant for every loop, pre- and postconception for every procedure and function, and usually require data invariants for each implementation of an abstract data type....”

The four axes model specifies that management can only control two of the four coordinates. Programming management insists on setting the time interval for development and, being a business, the budget. Usually, then, quality and people are free parameters. Since everyone plays by the same rules, the people must be interchangeable. This has been great for salaries, but not the professional development of non management computer professionals. Since quality is not the dominant issue, there has been little or no pressure on systems developers to develop the tools or skills to produce proven code.

3.1.1 Time Issues

There are two basic time issues facing the programmer. The first type relate to the issues of how the programmer prioritizes the twenty-four hours in the day. The second relate to the time to actually produce a proof. We address these separately.

Use of Time During Development. There are time issues independent of the complexity of a proof. The observation is that the time to complete a proof is much longer than the useful life of a piece of software. Time pressures on development work against an open-ended activity. Efficacy issues are important in terms of setting priorities. If the proof does not clarify the program or indicate important optimization criteria, the time to produce a proof is not seen as a fruitful. Programmer appreciation of this fact is seen in the quote, “Most programs aren’t [correct], so why waste your time.” The budgetary constraint—and a bit of chauvinism—was exemplified by the comment, “Don’t use an expensive programmer for [proofs]; use a cheap mathematician.” Priority issues in the workplace were mentioned; in particular, documentation is cited as far more important.

“If you have time to do a formal proof of correctness, you’re working on too simple a project,” according to one comment. There is some gallows humor in this time issue. “Any program short enough to be proven correct is not interesting.” A close colleague—R. M. Panoff of the Clemson Physics Department—formalized the issue, “Theorem (somebody’s got a name for this theorem): Any non-trivial program has at least one bug. Lemma: It is a sufficient condition for triviality for a program to be proven to have no bugs.”

Complexity Issues. By far, the greatest number of respondents mentioned complexity based issues.

1. Complexity of the program. The proof is seen (by respondents) as ‘exponential’ in length.
2. Proofs more complex by almost any measure than programming.
3. Combinatorial complexity of proof of cases.
4. Combinatorial complexity of interactions.
5. Combinatorial complexity of input cases.
6. Combinatorial complexity of sequences.

Neal Johnson of Apple gave an example from a class he had taken: the proof of a 25-30 line program had run to over forty pages when written out. Another example given was a program to do Fahrenheit to Centigrade conversion. The program was 33 lines; the proof, without diagrams was thirty pages. The respondents see partial correctness as about the best one can do, and that in only a few cases. (More on this in the section titled Problems with the Proof Systems on page 11.) The long-standing question of how much faith could one put into a multi-thousand line proof cropped up as well. This is certainly a strong case for the ‘societal’ problem; without repeated reviews of the proof, it is indeed hard to put much confidence in a long proof. Higher level of abstraction was cations, ‘letters’ are not necessarily signed in the conventional way. Hence, these quotes should be seen as taken from the messages only.

3. Programming in the Real World

The survey is discussed in this and the next sections. In this section, we attempt to categorize the results of the survey into subject areas meaningful to computer scientists. Section 4 explores issues raised by the programmers responding to the posted query.

The attempt to classify responses was not altogether successful. The author has attempted to put the comments into categories which most represented the thrust of the comments. These were essentially

1. Economic Realities,
2. Software Engineering,
3. Psychological and Educational,
4. Programming Language Arguments, and
5. Proof Methodologies.

Some comments defied all attempts categorize them in one section and were repeated where appropriate. There is one final category, Apparent Misunderstandings on page 12, which is included because it seemed, to the author, that such misconceptions work against acceptance among those responding.

3.1. Economic Realities

The economics of programming certainly dictate against the use of ‘expensive’ programmers for detailed proofs of correctness of programs. The most often cited reason for this is that the product normally does not need to be 100% correct. Like most everything else in the modern world, there is a certain level of incompetence that people are willing to put up with. Most who cited this as a reason for not proving a large system correct also pointed out that life-critical systems *should* be so proven.

The author had several experiences while in industry which related to the management side of developing large systems. The development cycle and the economics of development can be tied together many ways. The model used here has four coordinates. Each point is developed below.

1. *Time.* The major component of the time axis is complexity in one form or another. The usual ideas of complexity apply: namely the number of lines of code. However, there are other issues such as the complexity of interactions which is closely related to specifications.
2. *Quality.* Three major themes affect quality: the specifications; implementation system, including language; and the development environment. These are covered in their own sections.
3. *People.* The dominant factors on this axis are the psychological and educational elements.
4. *Budget.* While not investigated here, development is certainly driven by costs and budgets.

for guaranteeing program performance is not even a theoretical possibility." Hence, Fetzer's conclusion is the same as De Millo *et al.*, but through a different route.

Barwise[4] published a review of Fetzer in 1989. Barwise concluded that "...But they're [limitations to program verification] just the limitations implicit in any applied mathematics." This view seemed to fit in with the author's experience, especially when trying to do proofs of real problems as in liveness in parallel systems. Computer science literature seemed to be lacking the type information, definitions and results which would make such proofs more direct.

Then, in the December, 1989, issue of the *Communications of the Association of Computing Machinery,* remarks made by Edsger Dijkstra at the Computer Science Conference 1989 appeared. The remarks., "On the Cruelty of Really Teaching Computer Science"[13], had stirred up much controversy. Of particular interest are the comments on programming and proofs of correctness (page 1404):

> "From the beginning and all through the course, we stress that the programmers's task is not just to write down a program, but that his main task is to give a formal proof that the program he proposes meets the equally formal functional specifications.... Finally, in order to drive home the message that this introductory programming course is primarily a course in formal mathematics, we see to it that the programming language in question has *not* been implemented on campus so that students are protected from the temptation to test their programs."

The article was accompanied by several commentaries by noteworthy computer scientists. The Dijkstra article can be seen as the terminus of a line started by C. A. R. Hoare [29] twenty years before.

The stage is set. There is a large—and vocal—following for the Hoare-Dijkstra view of programming. And there is a large—and vocal—following against this view. But no one had ever asked programmers in the real world why they did not "prove their programs correct." But how to ask?

The survey was conducted in a very unscientific way: USENET. USENET is a computer network of 'newsgroups.' Each group comprises several concurrent discussions on a general topic, using computer communications networks. As of this writing there are about 675 groups, each having an international distribution. A non-scientific survey question was put to many—but certainly not all—groups which were likely to have readers with opinions on the subject of program verification. These groups ranged through computer science, mathematics (including logic), philosophy, and natural science. The query simply asked for reasons why the respondent did *not* prove her/his program correct. The author had expected maybe twenty responses; there were 393 respondents. Most responses had more than one reason or comment. A large number of responses were fairly lengthy (for this type communication) and reasoned.

For example, here are some comments about the general subject. One respondent wondered if the entire technology was an attempt to substitute formalism for understanding: "I'm not sure formalism eliminates error." Another comment was "Computer science is an art, not a science." One honest soul said, "Most programs aren't correct anyway, so why waste your time?" Yet another was, "It's hard enough to write a correct program, let alone prove it is correct." The ultimate expression of the 'against' argument was one wag who said, "some things you just have to take on faith." The section Programming in the Real World on page 4 is an attempt to put these responses into some order. Where useful or interesting, quotes are included. Due to the nature of electronic communi-

2. Introduction

In 1967, Floyd [18] presented concepts for assigning meaning to a program. This original work served as the foundation for, among other things, program verification concepts. Then, C. A. R. Hoare [29] in 1969 proposed that programming should be an exact science; this article is taken as the beginning of the “verificationist” movement. In another article [30], Hoare proposed that the program and the proof of correctness could be developed simultaneously.

The basic *program verification* or *program correctness* concepts are simple to present. Suppose we have program *P* written in some programming language. This program is supposed to compute some final values, say *F,* given some initial values, say *I. I* is called an *input specification* and *F* is called an *output specification.* A program may or may not *terminate* for a given argument. A program which terminates for every possible input value and always computes the prescribed answer is called *totally correct,* or just *correct.* A program which, if it terminates, does indeed compute the correct answer is called *partially correct* [45]. There have been several approaches to verification proposed. For this paper, we need not delve into these different approaches. There are many texts in this area and several are listed in the bibliography [5, 21, 22, 45].

In principle, it seems that the concepts of program proofs should be straightforward and relatively easy to apply. The hope has been to put programming on some firm, logical basis. Proponents of this approach to programming point to the use of proofs in mathematics. By suitably defining the actions of the programming language (now part of *programming language semantics*) and proper specification of the input and output (now *formal methods of specification*), every program would have a proof of correctness.Unfortunately, the verification technology has been disappointing.

By 1979, several important advances in verification technology had taken place; *e. g.,* Manna [38]and Manna and Pneuli [38] as well as many contributions by Hoare [31]. In 1979, however, DeMillo, Lipton, and Perlis attacked this formalist effort.

The position espoused DeMillo, Lipton, and Perlis in [12] is that science and mathematics are sociological as well as formal. The basic point is that a proof accrues credence through continued examination; *i.e.,* a deductive result needs inductive convincing. The philosophies of both science and mathematics demand that there be multiple verifications of any theorem or scientific discovery. Such guaranties are forthcoming only when there are many independent trials, all of which confirm the results. This naturally comes about in practice by either repeated teaching or industrial use of a theory or theorem. No doubt the reader is familiar with a story of how a theorem or theory survived many such reviews before an error was discovered. The most common societal review in computer science circles is the walk-through [49].

The reaction to DeMillo, Lipton, and Perlis was loud, but short-lived. In the early 1980’s, the ready availability of parallel and distributed architectures led to renewed interest in verification. The main force in this new area was David Gries [21, 22]. Development of various techniques for parallel and distributed techniques [3] as well as further work on sequential techniques; [5, 45], for example. Direct criticism of the verification movement was muted until 1988.

Fetzer’s article [16], “Program Verification: The Very Idea,” presents a reasoned analysis of the entire debate up to 1988. His position is succinctly put in the abstract of the article. “The notion of program verification appears to trade upon an equivocation. Algorithms, as logical structures, are appropriate subjects for deductive verification. Programs, as causal models of those structures, are not. The success of program verification as a generally applicable and completely reliable method

Preprint for *Philosophy and Computers*

1001 Reasons for not Proving Programs Correct: A Survey

D. E. Stevenson
Department of Computer Science
Clemson University
Clemson, SC 29634-1906
Internet: steve@hubcap.clemson.edu
(803)-656-5880 Fax:(803)- 656-0145

1. Overview

The ready availability of supercomputers has allowed scientists to model physical phenomena at a level only dreamed of a few short years ago. These computer models tend to be very large systems of code. Naturally, the problem of the veracity of the code *vis-á-vis* the physical model and its mathematical formulation arises. It would be tempting to dictate that the programs must be *verified* or *proven correct.* But that technology has not been particularly successful. Why has that technology been unsuccessful? Is there any hope of scientific computing being put on a firm logical basis? As a comment given during the survey shows, we have a long way to go, "... I work in science, and the proof is less necessary than knowing the results are correct."[1]

We present the results of an investigation into program verification. The presentation is divided into three components.

1. A short history of program verification is presented. This history traces a debate which began in 1979 and continues to the present. This serves as a prologue for a survey of programming professionals and academics.
2. The survey was conducted by asking a question of certain newsgroups on USENET. The responses were categorized to try to discover various impediments to program verification activities. The report is largely anecdotal and presented without comment.
3. Finally, a short discussion of the survey's points is presented. We close with some directions that new philosophical investigations might take.

1. For the rules relating to quotes contained herein, see page 2.