acm forum

Comments on Social Processes and Proofs

There follow a number of letters commenting on the recent paper "Social Processes and Proofs of Theorems and Programs" by De Millo, Lipton and Perlis. Some related letters arising out of commentary on a previous paper by Geller appear also in this issue, in the Technical Correspondence section. It has been brought to the attention of the editors that the dates published on the De Millo, Lipton and Perlis paper are incomplete; the paper was first actually submitted to the Programming Languages editor in April 1977, as one of the set of papers derived from the 1977 ACM Symposium on the Principles of Programming Languages. When it was later decided that the paper should appear under the Reports and Articles heading, the earlier dates were not included, for which omission the editors apologize.

☐ My heartiest congratulations for a job well done in the article on Proof of Theorems [1].

As a practitioner (MBA) and manager in the field of computers in business, I have found *Communications of the ACM* to be arcane, difficult to follow, and above all, of no practical use. Despite the theoretical nature of this subject, the writing was clear, interesting, and devoid of mathematical symbolism. In addition, its practical value to me was to explain to me why I should not look for development of formal proof of the "correctness" of our programs.

Again, thanks for giving me the first article in *Communications* that I have enjoyed since I joined ACM in 1973. May it signal the beginning of a new trend for at least some of the future articles.

ALLAN G. POMERANTZ
ARA Services, Inc.
Philadelphia, PA 19106

1. De Millo, R.A., Lipton, R.J., and Perlis, A.J. Social processes and proofs of theorems and programs. *Comm. ACM 22*, 5 (May 1979), 271-280.

☐ "Social Processes and Proofs of Theorems and Programs" by De Millo, Lipton and Perlis in the May 1979 *Communications* is the best article I have read in a computer publication and one of the best articles I have read anywhere. Thank you for publishing it, and thanks to the authors for their wisdom, fairness, style, rigor, and wit. Such an article makes me delight in being an ACM member, and, indeed, in being a member of the human race.

DANIEL GLAZER
2147 "O" St. N.W. #402
Washington, DC 20037

☐ I read with amusement De Millo, Lipton and Perlis's fine article, "Social Processes . . ." whose title I think could easily have been "Formal Verification Considered Harmful." What I found particularly interesting was the authors' admonition to "make a sharp distinction between program reliability and program perfection." While I agree with the spirit of most everything in the paper, I feel it is important to consider both reliability and validity. It is not always enough to demonstrate reliability—there are cases where a convincing demonstration of validity is also important. I remember my statistics professor's analogy of the important relation between reliability and validity. "Picture," he said, "the following cartoon, which I actually saw in a magazine: A small boy is standing on a scale and with a large smile is saying, 'I am thirty pounds tall'." That scale was very reliable, however . . .

STEWART A. DENENBERG
SUNY at Plattsburgh
Plattsburgh, NY 12901

☐ Marvelous, marvelous, marvelous! I refer of course to "Social Processes and Proofs of Theorems and Programs" by De Millo, Lipton and Perlis in the May 1979 *Communications*. This is exactly the kind of article that belongs in the flagship publication and nowhere else. It addresses a broad issue of importance to the computing community. It puts forth powerful and convincing arguments. To have on top of that an article that is literate, readable, and stimulating is almost too much. I want more! I realize that won't be easy, dear editor, I can only urge you to try. Meantime, a drink like that will set me up for a long dry spell. Another one like that will be worth more than a little patience.

I have only two general thoughts to suggest as addenda to what was said. First, I think it would be an immense contribution to the health of mathematics if the authors would take the parts of the paper about the nature of mathematics, expand them, round them off a bit, and submit the resulting article to *Harper's* or the *Atlantic* or other such general magazine. Those insights should be disseminated as widely as possible.

Second, though a couple of bows were made in that direction, I think more could have been made of the whole "programming style" movement. In the terms of the paper, these are nothing more or less than attempts to construct programs that will be elegant proofs of the specifications that are the theorem statements. The programs of a book like Kernighan and Plauger [1] can be read, and the reading gave me a much better handle on the whole programming process. If the discontinuities mentioned in the paper can be bridged, this is the approach that will bridge them.

LEONARD F. ZETTEL, JR.
3820 Brookshire Drive
Trenton, MI 48183

1. Kernighan B.W., and Plauger, P.J. *The Elements of Programming Style*. McGraw-Hill, New York, 1974.

☐ On the basis of ten years experience in the design, implementation and use of software for numerical applications (statistics and econometrics), I agree completely with the views of De Millo, Lipton and Perlis.

I cannot recall a single instance in which a proof of a program's correctness would have been useful. That is not to say that I have been involved in the production and use of error-free software; rather, I find that there are mainly two kinds of software errors:

(1) Actual errors in the implementation of a program (bugs).

(2) Errors in the specification of a program, or, more commonly, of a system which comprises many programs.

The first kind of error can be quite serious, but it is typically easy to fix when identified. In my experience, serious errors of the first kind can always be found by adequate testing. It is unfortunately true that some programmers (including, sometimes, myself) do not test adequately, so the user winds up discovering many bugs. This is regrettable and can be avoided by setting up a testing function or department. (See Huang [1] for a review of testing and Myers [2] for the results of controlled experiment.) In any case it is my feeling as a user that user detection of bugs can sometimes be cost-effective and acceptable to users if they are warned in advance that the system is likely to contain some errors. As a user, I would much rather be given a program with two bugs now than a perfect program next year.

The second kind of error, on the other hand, is much more serious. By definition it cannot be detected by testing or proving techniques, since the program meets its specifications. It is invariably detected very quickly by users, who inform the programmer that the formal program specifications do not meet their informal specifications (needs). In my experience these errors are often exceedingly expensive to fix; often the fix requires an extensive rewrite of the system.

Lientz, Swanson and Tompkins [3] present the results of a survey of 69 EDP organizations. This survey confirms that maintenance due to inadequate or changing program specifications is an important component of today's EDP costs. On average the surveyed users allocated 48 percent of their annual personnel hours to maintenance and enhancement. Within this category, 60 percent of the time was devoted to user enhancements, improved documentation and recoding for computational efficiency (40 percent to user enhancements alone). Only 17 percent was devoted to emergency fixes and routine debugging. Table V of their article clearly shows that management in the surveyed organizations perceived user demands for enhancements and extensions to be the number one problem area.

It is my opinion that methods for avoiding program misspecification are much more urgently needed than methods for proving that a program meet its specifications. For example, I consider Michael Jackson's work to be a step in the right direction [4], as are the various concepts known by the buzzwords chief-programmer team, top-down design, iterative refinement, etc. (see Zelkowitz [5] for a review).

RICHARD HILL
A.C. Nielsen Management
Services S.A.
CH-6002 Lucerne
Switzerland

1. Huang, J.C. An approach to program testing. *Computing Surveys 7*, 3 (Sept. 1975), 113-128.
2. Myers, G.J. A controlled experiment in program testing and code walkthroughs/inspections, *Comm. ACM 21*, 9 (Sept. 1978), 760-768.
3. Lientz, B.P., Swanson, E.B., and Tompkins, G.E. Characteristics of application software maintenance. *Comm. ACM 21*, 6 (June 1978), 466-471.
4. Jackson, M.A. Information systems: Modeling, sequencing and transformations. Proc. Third International Conference on Software Engineering, Atlanta, Ga., May 1978, 72-81.
5. Zelkowitz, M.V. Perspectives on software engineering. *Computing Surveys 10*, 2 (June 1978), 197-216.

□ It was time somebody said it—loud and clear—the formal approach to software verification does not work now and probably never will work in the real programming world. In their well-written paper the authors stress the obvious parallel to mathematics: Little to nothing of the immense wealth of present day mathematics came out of formal reasoning like the predicate calculus, etc., nor are theorems usually proved by such means.

The one sentence that was quoted from Poincaré really says it all: "... if it requires twenty-seven equations to establish that 1 is a number, how many will it require to demonstrate a real theorem?"

Automatic verification by the computer, much heralded only a few years ago, has turned out to be a totally impossible approach for any program that lies outside the toy-program world of the "greatest common divisors," etc. Some of the deeper reasons for this may be found in Weizenbaum's excellent book [1]. But the fact that the verification route has failed does not mean that the need for correct, reliable software has disappeared. Quite the contrary: With the ever increasing complexity of computer systems, particularly of real-time process control systems, the problem has become more aggravated than ever.

The authors at one point identify "real life programming" with hourly changing specifications, with "patches," "glue," "spit," etc. It seems to me that with such a chaotic programming style one would be as remote from reality as are their chastised colleagues of the "verification guild."

It is both interesting and fruitful to draw parallels between the programming world and the world of mathematics. As the authors point out there exist important differences, too.

There are many deep mathematical theorems that may be written down in a couple of lines; programs may take thousands of lines but exhibit a certain "shallowness." But does it have to be that way? No doubt, mathematics can be very concise and deep—but must programs be shallow? That this is now usually the case may be one reason for the often quoted "software problem." There is one dimension that is crucial in "real-life" programs: complexity. The problem of software engineering is usually not the finding of "deep theorems" but rather the highly nontrivial task of mastering complexity. This is what system engineering—be it for hardware or software—is all about.

To attack this problem successfully nature can teach us a lesson or two, if one is willing to see and to listen. Biological systems are extremely sophisticated; we can hardly guess their immense complexity. How does nature "engineer" these extremely complex systems? In his wonderful and very readable works Arthur Koestler [2, 3] gives us some hints.

These systems are deeply structured and highly redundant. Many years ago Koestler introduced the concept of a holon that is gradually finding acceptance in many disciplines. The modules of Portal (a Pascal based real-time programming

language developed at our research lab) are something like holons. When programming in Portal one is forced to provide a certain amount of redundant information. In his newest book [3] Koestler very eloquently shows how natural systems can be represented as a hierarchy of holons (as a "holarchy").

We believe that complex software should also be developed as a hierarchy of such holons (or modules, as they are now appearing more and more in modern programming languages). Deep structuring and redundancy: This is, in the long run, the only way to come to grips with complexity in software that can reliably operate in a real-life environment.

H. LIENHARD
LGZ Landis & Gyr Zug AG
CH-6301 Zug
Switzerland

1. Weizenbaum, J. *Computer Power and Human Reason*. Freeman, San Francisco, 1978.
2. Koestler, A. *The Ghost in the Machine*. Henry Regnery Co., Chicago, 1967.
3. Koestler, A. *Janus—a Summing Up*. Hutchinson, 1978.

□ It is with great satisfaction that I noted that for the first time a paper on the philosophy of computer science, in this case the methodology of program verification, has been published in *Communications*.

I think the authors make a good case in demonstrating that computer science is at best like mathematics, and that unfortunately not all mathematics is classical, *id est* Euclidean mathematics. The notion that mathematics is a fallible, quasi-empirical science has been defended very persuasively by Lakatos [1, 2]. Many of the arguments the authors put forward can be found in the first reference and it is only sad that the authors do not seem to be aware of its existence.

Once one accepts the quasi-empiricism in mathematics, and by analogy in computer science, one can either become an adherent of the Popperian school of conjectures (theories) and refutations [3], or one may believe Kuhn [4] who claims that the fate of scientific theories is decided by a social forum, a thesis the authors also appear to defend and which can be directly derived from Kuhn's views. Again an appropriate reference would have shown that the ideas in the paper are only novel on the level of analogy with computer science.

Personally I have to admit to feeling ill at ease with Kuhn's philosophy because its direct application implies that sociology, and worse, astrology and scientology, are on a par with "normal" science. I think it is therefore preferable to state that a program is corroborated as long as it is producing results in agreement with the program specification. When a negative result occurs the core of the program could still be correct but then surely something is wrong in one of the layers around the core [5].

Finally, I wish to take issue with the remarks made by the authors about probabilistic proofs. The fact that, for the time being, a proof is considered a working proof, does not mean that one can attach a probabilistic truth value to it. What value would you assign it? And is it depending on the successful use of the theorem? Probability is a meaningless concept here (for a more detailed argument see [5]). It is safer to say that the probability is zero, just as for any other empirical theory—we use it until it is replaced by a better one, and so on, and so forth. The most we can say is that we have faith or confidence in a program, terms which are, however, unknown in the theory of knowledge.

J. VAN DEN BOS
University of Nijmegen
Nijmegen, The Netherlands

1. Lakatos, *Mathematics, Science and Epistemology*. Cambridge University Press, London, 1978.
2. Lakatos, I. *Proofs and Refutations*. Cambridge University Press, London, 1976.
3. Popper, K. *Conjectures and Refutations*. Routledge and Kegan Paul, London, 1974.
4. Kuhn, T.S. *The Structure of Scientific Revolutions*. Chicago University Press, 1970.
5. Lakatos, I. *The Methodology of Scientific Research Programmes*. Cambridge University Press, London, 1977.

□ Congratulations on finally publishing a sensible paper on program verification, namely "Social Processes and Proofs of Theorems and Programs" by De Millo, Lipton and Perlis. If that paper doesn't demolish program verification, and the verification freaks manage to make some progress toward their goals, there still will be a number of questions which ultimately must be answered.

Consider the following thought experiment. The designers of Euclid (the name "Euclid" may be replaced by the name of any other language of similar intent) expect all Euclid programs to be verified. Euclid is a general purpose programming language, so it is reasonable to expect important large scale programs to be written with it; one of these programs might well be an automatic Euclid program verifier which accepts as input a Euclid program and a formal specification of what that program does, and which returns a result of "Verified" or "Not Verified." Although not all Euclid programs may in fact be verified, certain important software ought to be verified, for example, the automatic program verifier itself. It makes sense then to apply the verifier to itself and to a specification of what an automatic Euclid program verifier does. Questions arise:

(1) Are there any grounds for believing such a self-application of the verifier would terminate?

(2) If such a self-application can be shown to terminate, would it terminate within the lifetime of anyone currently interested in program verification?

(3) If it terminates with a result of "Verified," would anyone believe that the result is correct, and why?

(4) If it terminates with a result of "Not Verified," perhaps lots of people would believe it informally, but again, what would be the formal grounds for believing such a result? Furthermore, would not such a correct result of self-application verge on the paradoxical?

If for some technical reason self-application is not allowed, how would one decide to which Euclid programs the verifier is applicable? Would a "higher level" verifier be necessary to verify the verifier? How would that verifier be verified? And so on, *ad infinitum*.

I suspect that no major programs written in Euclid, whether an automatic Euclid program verifier, a Euclid compiler, a database management system, or an operating system, would ever be verified by man, woman, child, beast, or machine.

HARVEY ABRAMSON
The University of British Columbia
Vancouver, B.C.
Canada V6T 1W5

□ The article by De Millo, Lipton and Perlis presents among other issues some theoretical limitations in the possible use of automatic program verifiers when applied to real-world systems in which the specifications are often of the same order of magnitude as the programs themselves.

There is another aspect of the theoretical limitations of such verifiers the authors have overlooked. Before an automatic program verifier can determine whether a program realizes the specification from which it was written, the specification itself must be encoded in machine readable form for input to the verifier. This encoding is in fact nothing more or less than a program. Assuming that the language used to encode the specification has approximately the same power as the language in which the program to be verified is written, it follows that the encoding of the specification must be of the same order of magnitude as the program to be verified.

Even if the encoding of the specification were developed separately from the program to be verified, it would be subject to the same programming errors to which all programs are liable.

Thus even assuming it were theoretically possible to develop a verifier capable of handling a real-world program, all it could really demonstrate would be that one encoding of a specification accomplished the same function as another encoding of the specification. It could never demonstrate that either encoding accomplished the actual functional requirement of the specification itself, much less that an encoding fulfilled the much more elusive intentions of those responsible for requesting the system in the first place.

I am afraid these limitations ultimately confine the role of the automatic verifier to that of a historical curiosity.

JOSHUA TURNER
Penn Mutual Life
Philadelphia, PA 19172

□ In "Social Processes and Proofs of Theorems and Programs," De Millo, Lipton and Perlis decry the attention paid to program verification at the expense of other, unnamed techniques. It appears that the "ACM Algorithms Policy" was published in the same issue (May 1979, pp. 329-330) to assure them that they need not worry. This two-page policy statement describes in detail the criteria that a program must meet in order to be published—such things as what dialects of Fortran are acceptable, where indentation should be used, and what kinds of comments are needed.* Nowhere is it stated that the correctness of the program be demonstrated in any way. What a wonderfully democratic social process: All programs are innocent until proven guilty. There is even provision for a jury of peers; the publication of "certifications" and "remarks."

I'm afraid that I am one of those "classicists" who believe that a theorem either can or cannot be derived from a set of axioms. I don't believe that the correctness of a theorem is to be decided by a general election. To err is human. False theorems will be published. Yet must we rely so heavily upon divine forgiveness? Surely we should try to prove our theorems as best we can.

For years, we did not know any better way to check programs than by testing them to see if they worked. We were in the position of geometers before Euclid: To see if a theorem was true, all they could do was test it on some diagrams. But the work of Floyd and others has given us another way. They taught us that a program is a mathematical object, so we can apply the reasoning methods of mathematics to deduce its properties. (Of course, there were geometric proofs before Euclid, and program verification before Floyd. I hope the reader will forgive my rhetorical simplification of history.)

After Euclid, a theorem could no longer be accepted solely on the basis of evidence provided by drawing pictures. After Floyd, a program should no longer be accepted solely on the basis of how it works on a few test cases. A program with no demonstration of why it is correct is the same as a conjecture—a statement which we think may be a theorem. A conjecture must be exceptionally interesting to warrant publication. An unverified program should also have to be exceptional to be published.

The ACM seems to have gone too far in its eagerness to reassure De Millo, Lipton and Perlis. After all, they did write of "the benefits that could result from accepting a standard of correctness like the standard of correctness for real mathematical proofs." Let us heed their advice and settle for the frail, human standards of mathematicians. The ACM should require that programmers convince us of the correctness of the programs that they publish, just as mathematicians must convince one another of the correctness of their theorems. Mathematicians don't do this by giving "a sufficient variety of test cases to exercise all the main features," and neither should computer scientists.

LESLIE LAMPORT
SRI International
Menlo Park, CA 94025

*The word "algorithm" usually means a general method for computing something, and "program" means code that can be executed on a computer. In the tradition of Computer Science, the ACM has tried to make itself seem more erudite by calling the programs it publishes "algorithms." The "ACM Alogrithms Policy" would not have allowed Euclid to publish his algorithm, although he could have published a Fortran program to compute GCDs which was based upon it. In the interest of restoring meaning to our language, I will call programs "programs." I urge the ACM to do the same.

□ The catalog of criticisms of the idea of proving programs correct which was published in the May 1979 issue of *Communications* ("Social Processes and Proofs of Theorems and Programs," by R.A. De Millo, R.J. Lipton and A.J. Perlis, pp. 271-280) demands a catalog of responses by someone who believes in verification, as I do. The following catalog is organized as a series of responses to quotations, or sequences of quotations, from the De Millo, Lipton and Perlis paper cited above.

(1) "The aim of program verification . . . is to increase dramatically one's confidence in the correct functioning of a piece of software" (p. 271). "There is a fundamental logical objection to verification . . . Since the requirement for a program is informal and the program is formal, there must be a transition, and the transition itself must necessarily be informal" (p. 275). ". . . the monolithic view of verification is blind to the benefits that could result from accepting . . . a standard of reliability like the standard for real engineering structures" (p. 279).

This is a criticism of a viewpoint on verification held for a long time by most people in the field—this author included—but which has been obsolete for at least two years. The fact is that [6] "there are really two kinds of software correctness, only one of which can be proved. There is *program correctness* (does a program satisfy its specifications?) and *specification correctness* (are the specifications what the users wanted?). Debugging a program involves the finding and elimination of *program bugs* (ways in which the program failed to satisfy its specifications) as well as *specification bugs* (ways in which the users decided that the specifications were wrong). Usually, in practice, there are more of the latter than of the former. The trouble is that, although we can prove that a program meets its specifications, we have no way of proving what users want."

Software correctness, in other words, is neither entirely formal, as the older view would have it, nor entirely informal, as one might infer from reading De Millo, Lipton, and Perlis. A proof of correctness consists of two steps, one formal, the other informal; and neither of the two is valid without the other one.

(2) "[Bertrand] Russell did succeed in showing that ordinary working proofs can be reduced to formal, symbolic deductions. But he failed, in three enormous, taxing volumes, to get beyond the elementary facts of arithmetic. He showed what can be done in principle and what cannot be done in practice. If the mathematical process were really one of strict, logical progression, we would still be counting on our fingers" (p. 272).

It is easy to say that Russell did not have a computer and that the 1,907 pages of mathematical formulas in *Principia Mathematica* could be verified in less than two hours of computing time, assuming 1,000,000 instructions per second, 100,000 instructions per line of theorem to be verified, and 35 lines per page. This facile reply, however, hides an important fact: One of the major goals of verification is to provide a new dimension in the way we do mathematics, as well as in the way we do computer science. Mathematical facts can, in theory, be encoded in machine readable form and verified by computer in terms of other mathematical facts, until we have a compendium of known mathematical knowledge roughly equivalent to Gmelins' 70-volume, over 100,000-page *Handbook of Inorganic Chemistry,* or Beilstein's even larger *Organic Chemistry.* (Actually, all these works suffer from the inclusion of massive amounts of useless information; better examples might be the 2400-page *Handbook of Chemistry and Physics,* or Boss's *General Catalogue of 33,342 Stars For The Epoch 1950* in astronomy.)

These works are not perfect, either, but they do provide a dimension in chemistry, physics, and astronomy that is totally lacking in mathematics and computer science today, and which would be impossible to implement in either mathematics or computer science, as Russell's example shows us, without the aid of the computer. The tools for checking mathematical proofs by computer have been with us for at least fifteen years (see e.g.[1]), and a number of attempts to codify various branches of mathematics have been made since then. Many mathematicians, however, remain disdainful of routine work and unwilling to investigate the capabilities that computer science can give them.

We do not argue that strict logical deduction should be the only way that mathematics should be done, or even that it should come first; rather, it should come last, after the theorems to be proved, and their proofs, are well understood informally.

(3) "Stanislaw Ulam estimates that mathematicians publish 200,000 theorems every year. . . . A number of these are subsequently contradicted or otherwise disallowed. . . . The theorems that get ignored or discredited are seldom the work of crackpots or incompetents . . . increasing the number of mathematicians working on a given problem does not necessarily insure believable proofs. . . . Even simplicity, clarity, and ease provide no guarantee that a proof is correct" (pp. 272-273).

Again, this is because mathematicians do not use the computer in verifying their proofs. They simply throw together loosely constructed arguments and hope that these stand up to each other's scrutiny. A number of chapters of my own Ph.D. thesis [3], in fact, were devoted to the re-expression of just such a proof (originally formulated by my thesis supervisor) into a rigorous form.

The stage of manipulating loosely constructed arguments, in fact, seems to be necessary in mathematics (unless we ask the computer to prove our theorems as well as to verify their correctness, an enterprise whose main defect appears to be that it takes all the fun out of the game). Once a loosely formulated argument has been committed to paper, however, it should, in the ideal world we are striving for, be verified by computer.

(4) "One theoretician estimates . . . that a formal demonstration of one of Ramanujan's conjectures assuming set theory and elementary analysis would take about two thousand pages; the length of a deduction from first principles is nearly inconceivable" (p. 273).

This sounds like the kind of estimate we used to read about fifteen

years ago concerning the length of a program to play chess. The point is that if we want to demonstrate a conjecture of Ramanujan's (say), it is *not* necessary to assume set theory and elementary analysis and nothing else. We can always use lemmas and other known theorems, whether we are doing a computer generated proof or not. Again, we are not expecting perfection, because not all the lemmas we use are necessarily right; we can only strive toward some future day when most of the important mathematical proofs in the world will have been verified by computer.

(5) "... let us suppose that the programmer gets the message 'VERIFIED'. . . . What does the programmer know? He knows that his program is formally, logically, provably, certifiably correct. . . . He does not know within what limits it will work; he does not know what happens when it exceeds those limits" (p. 277).

This is a fundamental misunderstanding of the nature of program correctness. The statement of correctness of any program says that if we start it at the beginning with its entry assertion valid, then it will end, and when it does, its exit assertion (the one that is associated with the particular exit point where it ended) will be valid. The exit assertion tells us what the program has done; the entry assertion tells us what must be true in order for the program to work properly. All the limits on the program are built into the entry assertion; and a program can always be written to test whether its input is within its own given limits, before proceeding.

It is true that a higher level language program can be proved correct with no reference to machine limitations such as word size. Such a proof, however, is not complete and must be supplemented by a proof of correctness of the object program in its actual machine environment. The theory behind such proofs is known [4] and such proofs have actually been constructed [5].

The warning which De Millo, Lipton and Perlis give here may be characterized more as misdirected than as inappropriate. Instead of saying that a programmer does not know within what limits his verified program will work, it would be more to the point to say that he knows that his verified program will do exactly what he said it would, but that he does not know whether this will continue to satisfy him. We have all heard stories of labor unions which, barred legally or financially from calling a strike, have caused an effective slowdown by simply asking their members to "work to rules"—that is, to obey every rule slavishly. Bugs in verified programs should be expected to be of this kind—sending out a check for $0.00, for example, simply because the program was never told explicitly not to do that.

(6) "Verifications are not messages; a person who ran out into the hall to communicate his latest verification would rapidly find himself a social pariah" (p. 275).

Not true. Let me give you an example. A number of years ago a fellow expert in verification was stumped by the problem of trying to figure out why the following program terminates:

```
      DIMENSION A(100)
      I = 1
1     IF (A(I) . LE . A(I + 1)) GO TO 2
      T = A(I)
      A(I) = A(I + 1)
      A(I + 1) = T
      IF (I . EQ . 1) GO TO 2
      I = I - 1
      GO TO 1
2     I = I + 1
      IF (I . NE . 100) GO TO 1
```

This program sorts 100 numbers, and in the process the index I "wanders" up and down until it finally reaches 100. Therefore $100 - I$ does not satisfy the criteria for a termination expression (i.e. an integer expression which is non-negative at an assertion point in a loop and which always decreases every time we go around the loop). Consideration of the number of adjacent pairs of elements A(I) and A(I + 1) which are out of order at any given time likewise fails to provide us with a termination expression. Yet it is clear that this should be a one-loop program, since statement number 1 is inside every closed loop of the program and hence a single termination expression should suffice.

I concluded my visit and left my colleague's office, still thinking about the problem, and was about a mile down the nearest interstate when the answer came to me: Instead of the number of adjacent pairs, use the number of all pairs (A(I), A(J)) which are out of order at any given time; this number should decrease whenever an out-of-order pair is interchanged. Full of excitement, I rushed back to my colleague's office, where he confirmed that this was indeed the right idea. It turns out that the *unsortedness* U (the number of out-of-order pairs) is not quite the termination expression we want, because it decreases only when an interchange actually takes place. A little experimentation, however, reveals that $2U - I + 99$ will work. This is an integer expression which is never negative at statement number 1, because $I \leq 99$ at statement number 1, while $U \geq 0$ by definition. There are three paths from statement number 1 around the loop and back to statement number 1, and the behavior of U, I, and $2U - I + 99$ on each of these three paths is as follows:

PATH NUMBER	1	2	3
INTERCHANGE	NO	YES	YES
PATH GOES THROUGH STATEMENT 2?	YES	YES	NO
CHANGE IN VALUE OF U	0	−1	−1
CHANGE IN VALUE OF I	+1	+1	−1
CHANGE IN VALUE OF 2U − I + 99	−1	−3	−1

With this help, a formal proof may be readily constructed.

Verifications, like any other mathematical proofs, have an informal stage at which we determine assertions and test to make sure they remain true as we go through the program from one assertion point to another. Quite often, in fact, a published program will be accompanied by a proof of its correctness which is given in informal style only. It is the *formal* verification of a program that "cannot really be read; a reader can flay himself through one of the shorter ones by dint of heroic effort,

but that's not reading" (to continue the above quote). But the same is true of any totally formal proof in mathematics—or, for that matter, of a trace (in the usual sense) of any program taking more than a few milliseconds of computer time.

(7) "There are even some cases of black-box code, numerical algorithms that . . . work for no reason that anyone knows; the input assertions for these algorithms are not even formulable, let alone formalizable. To take just one example, an important algorithm with the rather jaunty name of Reverse Cuthill-McKee was known for years to be far better than plain Cuthill-McKee. . . . Only recently, however, has its superiority been theoretically demonstrable . . . and even then only with the usual informal mathematical proof, not with a formal deduction. During all of the years when Reverse Cuthill-McKee was unproved, even though it automatically made any program in which it appeared unverifiable, programmers perversely went on using it" (p. 276).

Of course; that's called a conjecture. Mathematicians use conjectures all the time, and only later prove them; sometimes they remain unproved for long periods of time, and yet other theorems based on them continue to be believed. In computer science, we derive results based on the P/NP conjecture and proceed to act as if they were true; in mathematics, we do the same with the Riemann hypothesis. If Reverse Cuthill-McKee were used in a program before it was theoretically demonstrated, this would not make any program in which it appeared unprovable; it would merely be provable up to a conjecture, in the same way that the exponential character of the traveling salesperson problem is provable up to the P/NP conjecture.

(8) "Every programmer knows that altering a line or sometimes even a bit can utterly destroy a program or mutilate it in ways that we do not understand and cannot predict. . . . There is no reason to believe that verifying a modified program is any easier than verifying the original the first time around. There is no reason to believe that a big verification can be the sum of many small verifications. There is no reason to believe that a verification can transfer to any other program—not even to a program only one single line different from the original" (p. 278).

This is one of the most important fallacies in the paper. The first statement above is true, but it does not imply the other three. The reason is that the first statement has to do with modifying a correct program in a reasonable seeming, but wrong way, so as to produce an incorrect program. If a program is incorrect, no amount of verification can prove it correct. And it is true that correct programs can be made to exhibit wildly erratic behavior—in fact, they normally will do so—if only a single bit is changed.

But contrast this with a carefully worked out, informally verified change in a correct program so as to produce another correct program. We have a proof of correctness of the first program and we need to derive a proof of correctness of the second. Under these conditions, large amounts of the first proof remain unmodified in the second; in particular, all the unmodified subroutines will still have the same entry and exit assertions and will still remain correct with respect to these. The change in the proof should be expected to be bigger than the change in the program, but not that much bigger.

The statement about big verifications not being the sums of smaller ones was apparently made in simple ignorance of the relations between the proofs of correctness of subroutines and the proofs of correctness of the programs which call them. It is perfectly true that the use of procedures as parameters, and even the use of call by reference, can lead to situations in which the semantics of the given program are imperfectly understood and proof techniques are of questionable validity. But in the case of call by value and result, the simplest and most used of parameter passing methods, several researchers have derived ways of handling subroutine calls in a program to be proved correct, provided that the subroutine has already been proved correct. In this way, if a program is broken up into a main program and n subroutines, we have $n + 1$ verifications to do, and that is all we have to do in proving program correctness. (Let us always remember, of course, that the specification correctness problem is still with us.)

(9) "Verifications are long and involved but shallow; that's what's wrong with them. The verification of even a puny program can run into dozens of pages, and there's not a light moment or a spark of wit on any of those pages. . . . Nobody is going to buttonhole a colleague into listening to a verification. Nobody is ever going to read it. One can feel one's eyes glaze over at the very thought" (p. 276).

We can make an analogy here with compiling a higher level language program into machine language. Originally, this was done by hand; people wrote out programs in sequences of steps specified informally in English and then proceeded to translate these into machine languages. Then compilers came along, and started to do this job automatically. At first people were against this, and for much the same reasons as given above. Compilers did what had previously been a fascinating human job in a machine-like, humorless manner. (They also produced overly long object code, in much the same way that a verifier produces overly long proofs.) Nobody is ever going to read the object code produced by a compiler, either; one simply trusts the compiler and goes about one's business. What we hope for in verifiers is that we will at least be able to trust them to show program correctness.

(10) "The formal demonstration that a program is consistent with its specifications has value only if the specifications and the program are independently derived" (p. 275).

It is true that many people used to look on verification as a purely mathematical process, whereas now we look on it as two processes, only one of which is mathematical. But the statement above seems to be denying the value of the part which is mathematical, when in fact both parts are necessary. It may be true that showing, informally, that the specifications of a given program will satisfy the users of that program is a formidable task. But the point is that, even when we are finished with that

task, we are not done; we still have to go through the mathematical part, to show that the program satisfies its specifications.

It may be argued, of course, that there is another way. Let us forget entirely about proving anything about our programs and concentrate on testing them as thoroughly as we can. All we can say in response to such an argument is that this is the way other sciences and engineering disciplines used to function, with disastrous results. The Tacoma Narrows Bridge collapsed because people were designing bridges, in those days, with no thought whatever to proving that they would not collapse. That state of affairs is now changed; and yet nobody claims that proofs in bridge engineering (or chemistry or physics or electrical engineering) are in any sense perfect. They are approximations, for a different reason than proofs of correctness are approximations; and yet they do increase our confidence in the proper functioning of the bridges, or the circuits, or the chemical reactions, with which they are concerned.

(11) "For even the most trivial mathematical theories, there are simple statements whose formal demonstrations would be impossibly long. . . . Suppose that we encode logical formulas as binary strings and set out to build a computer that will decide the truth of a simple set of formulas of length, say, at most a thousand bits. Suppose that we even allow ourselves the luxury of a technology that will produce proton-size electronic components connected by infinitely thin wires. Even so, the computer we design must densely fill the entire observable universe" (p. 278).

This is the same sort of argument that has been used before against using computers at all. Back when the unsolvability of the halting problem was a relatively new result, people used to use arguments which, stripped to their bare essentials, seemed to be saying that because we cannot prove that all halting programs halt, it is futile to try to prove that any halting program halts. In the same way, the argument above seems to be saying that because we cannot prove the correctness of all correct programs less than a thousand bits in length, it is futile to try to prove the correctness of any correct program.

(12) [continuing the quotation above] "This precise observation about the length of formal deductions agrees with our intuition about the amount of detail embedded in ordinary, workaday mathematical proofs. We often use 'Let us assume, without loss of generality' or 'Therefore, by renumbering, if necessary' to replace enormous amounts of formal detail" (p. 278).

Those who have been concerned with formalizing mathematical proofs are well aware of the fact that large amounts of formal detail are, indeed, passed over when writing out an informal proof. But they are just as aware that the complete specification of this formal detail—as long as known results in mathematics may be used, and theorems do not have to proceed from first principles—involves an increase in size of perhaps one or even two orders of magnitude (that is, ten or a hundred times the original size) but nowhere near the difference between an actual computer and a computer that would fill up the universe: "The proofs presented in Chapter VI are considerably more formal than the proofs of the same theorems that appear in mathematical textbooks. The textbook version of the algebra proof is a short paragraph; the formalized version runs to 63 steps" [1].

(13) ". . . it might be argued that all these references to readability and internalization are irrelevant, that the aim of verification is eventually to construct an automatic verifying system. Unfortunately, there is a wealth of evidence that fully automated verifying systems are out of the question" (p. 276).

It might be argued that it is futile to answer this criticism in this particular paper, because the authors, in their next paragraph, go on to assume that a fully automated verifying system could indeed be built, and attempt to show why such a system could not produce the results expected of it. But I should like to answer it anyway, because it is a point quite commonly encountered. The fact is that, in one sense, the authors are right—fully automated verifying systems, for any but the simplest classes of programs, are out of the question, roughly because we cannot expect the computer to be a universal theorem prover. But that doesn't matter because, in practice, a verifier does not have to be fully automated in order to be useful. In fact, for real machine language programs, a wealth of information about the program to be proved correct (normally as long as or longer than the program itself) must be supplied as input to the verifier along with the program (see e.g. [7]). If this information is incorrect or incomplete, the program will not be proved correct, even though it may be correct. Other researchers have experimented with verification as an interactive process, and to a certain extent with program construction and verification as simultaneous interactive processes.

(14) "It seems to us that the scenario envisioned by the proponents of verification goes something like this: The programmer inserts his 300-line input/output package into the verifier. Several hours later, he returns. There is his 20,000-line verification and the message 'VERIFIED'." (p. 277).

This scenario is off in at least four respects. It will not take several hours; it will take about as long as a compilation. It will not normally happen on the first run, any more than a compilation produces usable results on the first run; the programmer must stay with the computer until the verification is right, and may have to communicate interactively with it. There will not be 20,000 lines of output; there is no need for the intermediate steps in a verification to be printed out at all. Most importantly, the message does not read simply "VERIFIED": *All the assumptions that are made* (the entry assertion) *are stated, and all the actions of the program* (the exit assertions) *are likewise.* If the program still has bugs in it after verification (which it very likely will, if it is being written for the first time) these will be specification errors, not software errors.

I should like to conclude by citing a number of statements made by De Millo, Lipton and Perlis with which I heartily agree:

(15) ". . . we would still insist that verification renounce its claim on all other areas of programming; to teach students in introductory programming classes how to do verification, for instance, ought to be as farfetched as teaching students in introductory biology how to do open-heart surgery" (p. 276).

I have been teaching programming for thirteen years and verification for eight, and never have I attempted to teach verification to anyone in an elementary programming class, or advised others to do so. Verification is a mathematical subject requiring mathematical maturity for its understanding; it is properly taught as part of a course on the analysis of algorithms, the philosophy of programming (as I teach it, along with structured programming), or compiler construction (since verifier construction is so similar).

(16) ". . . there has never been a verification of, say, a Cobol system that prints real checks" (p. 279).

This is true (although my student Harry Keeling has been working on Cobol program verification for some time), but not for the reasons the authors seem to imply. Cobol involves a great wealth of programming language features, whereas the inductive assertion method of proving programs correct, as it was originally formulated [2], applied only to programs containing simple assignment (variable = expression), goto, conditional goto, and halt statements. It has taken researchers in verification a long time to learn how to handle all the features of a language as sophisticated as Cobol. But it has now been done; program correctness has changed from being impossible to being merely hideously expensive. I can prove the program corectness of a two-page check writing program, today, for about $10,000; yes, I know that this is not cost-effective, and the only reason I offer to do it is because the proof would involve quite a bit of general work that would make succeeding proofs easier. (And there are provisos; you have to explain to me, informally, how the program works, and I take no responsibility for the Cobol manuals being wrong, which has happened to me in the past.)

(17) "The desire to make programs correct is constructive and valuable . . . this is not the moment to restrict research on programming" (p. 279).

Let no one forget that these words were also written by De Millo, Lipton and Perlis.

W.D. MAURER
School of Engineering
and Applied Science
George Washington University
Washington, DC 20052

1. Abrahams, P.W. Machine verification of mathematical proof. Sc.D. Th., Dept. of Mathematics, MIT, Cambridge, Mass., June 1963.
2. Floyd, R.W. Assigning meanings to programs. Proc. Symp. Applied Math. 19 (Mathematical Aspects of Computer Science), Amer. Math. Soc., Providence, R.I., 1967, pp. 19-32.
3. Maurer, W.D. On minimal decompositions of group machines. Ph.D. Th., Dept. of Mathematics, U. of California, Berkeley, Jan. 1965.
4. Maurer, W.D. Some correctness principles for machine language programs and microprograms. Proc. 7th Annual Workshop on Microprogramming (MICRO 7), Palo Alto, Calif., Sept. 1974, pp. 225-234.
5. Maurer, W.D. Proving the correctness of a flight-director program for an airborne minicomputer. Proc. ACM SIGMINI/SIGPLAN Interface Meeting, New Orleans, La., March 1976, pp. 103-108.
6. Maurer, W.D. Software systems design and correct software. Proc. IEEE COMPCON 77, Spring (14th IEEE Computer Soc. International Conf.), San Francisco, Calif., Feb. 1977, pp. 194-197.
7. Maurer, W.D. A microcomputer program verifier and its assertion language. Proc. 12th Hawaii International Conference on System Sciences, Honolulu, Hawaii, Jan. 1979.

Authors' Response:

Many of the points raised above are addressed directly in our paper, so in deference to conciseness we will not attempt a paragraph-by-paragraph response to the correspondents.

Van den Bos raises the possibility that our stance is in some sense anticipated by more conventional philosophies of science and mathematics. Indeed, we referred in the article to the beautiful monograph by Lakatos [1]. Since writing the paper, we have uncovered several other sources of essentially the same notions in computer science and mathematics. The most striking of these is the discursive chapter on proof in [2]. Van den Bos is incorrect, however, in his assertion concerning "normal science": Scientific theories should have tests in reality!

Lienhard and Denenberg both raise the concept of reliability coupled to validity. If by validity, they mean knowledge in fact of correctness, if they mean certainty, then we do not believe that validity is possible. In a manner of speaking we hold a view which is exactly opposite to Lamport's. Lakatos [1] has put it beautifully: " 'certainty' is far from being a sign of success, it is only a symptom of lack of imagination, of conceptual poverty. It produces smug satisfaction and prevents the growth of knowledge."

The points raised by Lienhard, Hill and Zettel concerning program style have considerable merit. We do not advocate sloppy or intellectually inefficient programming techniques; on the other hand style must not be credited with more than it is capable of delivering. Programming technique cannot take one far in the absence of talent. We are reminded of Gauss's rebuke of a colleague: ". . . [he] needs notions not notations!"

The arguments raised by Abramson and Turner are of course theoretical variations on the "who will verify the verifier?" theme. Although Abramson's point is technically correct, it does not necessarily imply that verification is meaningless. Recognizing early the theoretical intractability of automated verification, the verification community abandoned the uniform applicability of their techniques, concentrating on programs that are "verifiable" and languages which encourage their construction. It is not necessary to invoke so esoteric an example of a program which is not a verifiable program; the fact that there are valuable programs to which the social process cannot apply seems to us an insurmountable difficulty. Turner's concern over specification raises another problem. If formal specification of the type required by these techniques is any less error prone than programming, why bother with the programming step at all? As Turner points out, such a specification is really an executable object, and if it is more reliable than the program with which it will be proved consistent, then the program is downright dangerous!

Lamport and Maurer display an amazing inability to distinguish between algorithms and programs. Of course, the social processes of math-

ematics will apply to algorithms and even to their proofs! One has only to glance at [3] or [4] or the proceedings of any recent SIGACT conference to see the processes at work—with noticeable disregard of the trappings of program verification. Maurer makes our point. He was able to mull over his 11-line sorting program precisely because it is compact and clean and interesting—the fact that [4] is devoted largely to sorting algorithms and their mathematical treatment testifies to the inherent attractiveness of such problems. Would Maurer be as motivated to mull over the properties of a report writer embedded deep within someone's Cobol fixed assets package? Maurer would probably claim that he would, but we have seen some of this code (out of necessity, not choice) and there is nothing interesting about it; it is detailed and baroque and valuable, but it is also very boring. Number theorists and accountants both do interesting and valuable things with numbers. But to a number theorist, the numbers are personal friends to be cultivated and dealt with individually. The folklore of mathematics is filled with stories of mathematical discoveries resulting from an "idle" consideration of a number. It is hard to imagine an accountant sustaining the same interest in his ledger figures. We think that the implications for program proving are clear. Computer scientists enjoy computers and programming, but to the rest of the world the computer is a tool.

Maurer also expounds a view of mathematics which is fairly far removed from our experience. In our paper we attempted to present mathematical proof in a setting which is much closer to what we perceive is mathematical practice. A still more current example can be found in the *Mathematical Intelligencer* (Volume 1, No. 4, 1979). In an article entitled "A Proof that Euler Missed," A. van der Poorten describes a "proof" that the Riemann Zeta function, when evaluated at 3, is irrational. Some excerpts:

1. It seems that Apery has shown that zeta(3) is irrational.
2. What on earth is going on here?
3. Apery's incredible proof appears to be a mixture of miracles and mysteries.

It is hard to imagine such a field benefiting at all from the approach suggested by Maurer.

Maurer also seems to have missed the point that there is no notion of continuity which makes "scaling up" or approximation at all sensible. His characterization of the Tacoma Narrows Bridge disaster as the result of not proving that bridges don't collapse is a complete distortion of fact, and to suggest that engineers do so now is simply false.

It seems to us that the only potential virtue of program proving lies in the hope of obtaining perfection. If one now claims that a proof of correctness can raise confidence even though it is not perfect or that an incompleted proof can help one locate errors, then that claim must be justified! There is absolutely no objective evidence that program verification is as effective as, say, *ad hoc* program testing in this regard. Indeed, all we have to go on are the testimonials of the verifiers—hardly a disinterested group.

Finally, pervading several of the letters is the sense that if only we did things this way or that way or if we drastically shifted our activities, then program proving would work. Perhaps, but we are troubled by Thoreau's advice: "Beware of any enterprise that requires new clothes."

RICHARD A. DE MILLO
Georgia Tech
Atlanta, GA 30332

RICHARD J. LIPTON
University of California
Berkeley, CA 94720

ALAN J. PERLIS
Yale University
New Haven, CT 06520

1. Lakatos, I. *Proof and Refutations: The Logic of Mathematical Discovery.* Cambridge, 1978.
2. Manin, Yu I. *A Course in Mathematical Logic.* Springer-Verlag, 1977.
3. Aho, A., Hopcroft, J., and Ullman, J. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.
4. Knuth, D.E. *The Art of Computer Programming, Vol. 3, Sorting and Searching:* Addison-Wesley, 1973.

acm news — official administrative chapter sig

ACM Contributes to CSA Exam Study. President Daniel D. McCracken CCP has presented ACM's check for $7,500 to the Institute for Certification of Computer Professionals for development work on the Certificate in Systems Analysis examination. ACM's contribution should carry the program forward through its first year.

William W. Cotterman, Georgia State University, is chairman of the ICCP *ad hoc* Committee on CSA. Committee members are J. Daniel Couger, University of Colorado; Norman L. Enger, Applied Management Systems; Frederick G. Harold, Florida Atlantic University; and Clement L. McGowan, SofTech.

ACM Announces College Consulting Service. The Association has initiated a consulting service for the purpose of providing knowledgeable computer personnel to assist colleges and universities in planning for and utilizing computers in three main areas: (1) the uses of computers in education; (2) computer science and information systems curricula; and (3) planning, selection, and administration of computing resources.

The service, which is intended primarily for undergraduate institutions, is to be adminstered by Gerald L. Engel of the Christopher Newport College, and Richard H. Austing of the University of Maryland, through ACM's Curriculum Committee on Computer Education.

Colleges and universities desiring to obtain the help of a consultant through this service during the 1979/80 academic year must submit an application form including information on current facilities and plans as well as preferred visit dates. Selected applicants will be provided with the name of a qualified computer professional and will be expected to make appropriate arrangements with that