Social Processes and Proofs of Theorems and Programs

Richard A. De Millo
Georgia Institute of Technology

Richard J. Lipton and Alan J. Perlis
Yale University

It is argued that formal verifications of programs, no matter how obtained, will not play the same key role in the development of computer science and software engineering as proofs do in mathematics. Furthermore the absence of continuity, the inevitability of change, and the complexity of specification of significantly many real programs make the formal verification process difficult to justify and manage. It is felt that ease of formal verification should not dominate program language design.

Key Words and Phrases: formal mathematics, mathematical proofs, program verification, program specification

CR Categories: 2.10, 4.6, 5.24

This work was supported in part by the U.S. Army Research Office on grants DAHC 04-74-G-0179 and DAAG 29-76-G-0038 and by the National Science Foundation on grant MCS 78-81486.

Authors' addresses: R.A. De Millo, Georgia Institute of Technology, Atlanta, GA 30332; A.J. Perlis and R.J. Lipton, Dept. of Computer Science, Yale University, New Haven, CT 06520.

> I should like to ask the same question that Descartes asked. You are proposing to give a precise definition of logical correctness which is to be the same as my vague intuitive feeling for logical correctness. How do you intend to show that they are the same? ... The average mathematician should not forget that intuition is the final authority.
>
> J. Barkley Rosser

Many people have argued that computer programming should strive to become more like mathematics. Maybe so, but not in the way they seem to think. The aim of program verification, an attempt to make programming more mathematics-like, is to increase dramatically one's confidence in the correct functioning of a piece of software, and the device that verifiers use to achieve this goal is a long chain of formal, deductive logic. In mathematics, the aim is to increase one's confidence in the correctness of a theorem, and it's true that one of the devices mathematicians *could* in theory use to achieve this goal is a long chain of formal logic. But in fact they don't. What they use is a proof, a very different animal. Nor does the proof settle the matter; contrary to what its name suggests, a proof is only one step in the direction of confidence. We believe that, in the end, it is a social process that determines whether mathematicians feel confident about a theorem—and we believe that, because no comparable social process can take place among program verifiers, program verification is bound to fail. We can't see how it's going to be able to affect anyone's confidence about programs.

Outsiders see mathematics as a cold, formal, logical, mechanical, monolithic process of sheer intellection; we argue that insofar as it is successful, mathematics is a social, informal, intuitive, organic, human process, a community project. Within the mathematical community, the view of mathematics as logical and formal was elaborated by Bertrand Russell and David Hilbert in the first years of this century. They saw mathematics as proceeding in principle from axioms or hypotheses to theorems by steps, each step easily justifiable from its predecessors by a strict rule of transformation, the rules of transformation being few and fixed. The *Principia Mathematica* was the crowning achievement of the formalists. It was also the deathblow for the formalist view. There is no contradiction here: 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.

Believing Theorems and Proofs

> Indeed every mathematician knows that a proof has not been "understood" if one has done nothing more than verify step by step the correctness of the deductions of which it is composed and has not tried to gain a clear insight into the ideas which have led to the construction of this particular chain of deductions in preference to every other one.
>
> N. Bourbaki

> Agree with me if I seem to speak the truth.
> Socrates

Stanislaw Ulam estimates that mathematicians publish 200,000 theorems every year [20]. A number of these are subsequently contradicted or otherwise disallowed, others are thrown into doubt, and most are ignored. Only a tiny fraction come to be understood and believed by any sizable group of mathematicians.

The theorems that get ignored or discredited are seldom the work of crackpots or incompetents. In 1879, Kempe [11] published a proof of the four-color conjecture that stood for eleven years before Heawood [8] uncovered a fatal flaw in the reasoning. The first collaboration between Hardy and Littlewood resulted in a paper they delivered at the June 1911 meeting of the London Mathematical Society; the paper was never published because they subsequently discovered that their proof was wrong [4]. Cauchy, Lamé, and Kummer all thought at one time or another that they had proved Fermat's Last Theorem [3]. In 1945, Rademacher thought he had solved the Riemann Hypothesis; his results not only circulated in the mathematical world but were announced in *Time* magazine [3].

Recently we found the following group of footnotes appended to a brief historical sketch of some independence results in set theory [10]:

(1) The result of Problem 11 contradicts the results announced by Levy [1963b]. Unfortunately, the construction presented there cannot be completed.
(2) The transfer to ZF was also claimed by Marek [1966] but the outlined method appears to be unsatisfactory and has not been published.
(3) A contradicting result was announced and later withdrawn by Truss [1970].
(4) The example in Problem 22 is a counterexample to another condition of Mostowski, who conjectured its sufficiency and singled out this example as a test case.
(5) The independence result contradicts the claim of Felgner [1969] that the Cofinality Principle implies the Axiom of Choice. An error has been found by Morris (see Felgner's corrections to [1969]).

The author has no axe to grind; he has probably never even heard of the current controversy in programming; and it is clearly no part of his concern to hold his friends and colleagues up to scorn. There is simply no way to describe the history of mathematical ideas without describing the successive social processes at work in proofs. The point is not that mathematicians make mistakes; that goes without saying. The point is that mathematicians' errors are corrected, not by formal symbolic logic, but by other mathematicians.

Just increasing the number of mathematicians working on a given problem does not necessarily insure believable proofs. Recently, two independent groups of topologists, one American, the other Japanese, independently announced results concerning the same kind of topological object, a thing called a homotopy group. The results turned out to be contradictory, and since both proofs involved complex symbolic and numerical calculation, it was not at all evident who had goofed. But the stakes were sufficiently high to justify pressing the issue, so the Japanese and American proofs were exchanged. Obviously, each group was highly motivated to discover an error in the other's proof; obviously, one proof or the other was incorrect. But neither the Japanese nor the American proof could be discredited. Subsequently, a third group of researchers obtained yet another proof, this time supporting the American result. The weight of the evidence now being against their proof, the Japanese have retired to consider the matter further.

There are actually two morals to this story. First, a proof does not in itself significantly raise our confidence in the probable truth of the theorem it purports to prove. Indeed, for the theorem about the homotopy group, the horribleness of all the proffered proofs suggests that the theorem itself requires rethinking. A second point to be made is that proofs consisting entirely of calculations are not necessarily correct.

Even simplicity, clarity, and ease provide no guarantee that a proof is correct. The history of attempts to prove the Parallel Postulate is a particularly rich source of lovely, trim proofs that turned out to be false. From Ptolemy to Legendre (who tried time and time again), the greatest geometricians of every age kept ramming their heads against Euclid's fifth postulate. What's worse, even though we now know that the postulate is indemonstrable, many of the faulty proofs are still so beguiling that in Heath's definitive commentary on Euclid [7] they are not allowed to stand alone; Heath marks them up with italics, footnotes, and explanatory marginalia, lest some young mathematician, thumbing through the volume, be misled.

The idea that a proof can, at best, only probably express truth makes an interesting connection with a recent mathematical controversy. In a recent issue of *Science* [12], Gina Bari Kolata suggested that the apparently secure notion of mathematical proof may be due for revision. Here the central question is not "How do theorems get believed?" but "What is it that we believe when we believe a theorem?" There are two relevant views, which can be roughly labeled classical and probabilistic.

The classicists say that when one believes mathematical statement A, one believes that in *principle* there is a correct, formal, valid, step by step, syntactically checkable deduction leading to A in a suitable logical calculus such as Zermelo–Fraenkel set theory or Peano arithmetic, a deduction of A à la the *Principia*, a deduction that completely formalizes the truth of A in the binary, Aristotelian notion of truth: "A proposition is true if it says of what is, that it is, and if it says of what is not, that it is not." This formal chain of reasoning is by no means the same thing as an everyday, ordinary mathematical proof. The classical view does not require that an ordinary proof be accompanied by its formal counterpart; on the contrary, there are mathematically sound reasons for allowing the gods to formalize most of our arguments. One theoretician estimates, for instance, 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 [14]. But the classicist believes that the formalization is in principle a possibility and that the truth it expresses is binary, either so or not so.

The probabilists argue that since any very long proof can at best be viewed as only probably correct, why not state theorems probabilistically and give probabilistic proofs? The probabilistic proof may have the dual advantage of being technically easier than the classical, bivalent one, and may allow mathematicians to isolate the critical ideas that give rise to uncertainty in traditional, binary proofs. This process may even lead to a more plausible classical proof. An illustration of the probabilist approach is Michael Rabin's algorithm for testing probable primality [17]. For very large integers N, all of the classical techniques for determining whether N is composite become unworkable. Using even the most clever programming, the calculations required to determine whether numbers larger than 10^{10^4} are prime require staggering amounts of computing time. Rabin's insight was that if you are willing to settle for a very good probability that N is prime (or not prime), then you can get it within a reasonable amount of time—and with vanishingly small probability of error.

In view of these uncertainties over what constitutes an acceptable proof, which is after all a fairly basic element of the mathematical process, how is it that mathematics has survived and been so successful? If proofs bear little resemblance to formal deductive reasoning, if they can stand for generations and then fall, if they can contain flaws that defy detection, if they can express only the probability of truth within certain error bounds—if they are, in fact, not able to *prove* theorems in the sense of guaranteeing them beyond probability and, if necessary, beyond insight, well, then, how does mathematics work? How does it succeed in developing theorems that are significant and that compel belief?

First of all, the proof of a theorem is a message. A proof is not a beautiful abstract object with an independent existence. No mathematician grasps a proof, sits back, and sighs happily at the knowledge that he can now be certain of the truth of his theorem. He runs out into the hall and looks for someone to listen to it. He bursts into a colleague's office and commandeers the blackboard. He throws aside his scheduled topic and regales a seminar with his new idea. He drags his graduate students away from their dissertations to listen. He gets onto the phone and tells his colleagues in Texas and Toronto. In its first incarnation, a proof is a spoken message, or at most a sketch on a chalkboard or a paper napkin.

That spoken stage is the first filter for a proof. If it generates no excitement or belief among his friends, the wise mathematician reconsiders it. But if they find it tolerably interesting and believable, he writes it up. After it has circulated in draft for a while, if it still seems plausible, he does a polished version and submits it for publication. If the referees also find it attractive and convincing, it gets published so that it can be read by a wider audience. If enough members of that larger audience believe it and like it, then after a suitable cooling-off period the reviewing publications take a more leisurely look, to see whether the proof is really as pleasing as it first appeared and whether, on calm consideration, they really believe it.

And what happens to a proof when it is believed? The most immediate process is probably an internalization of the result. That is, the mathematician who reads and believes a proof will attempt to paraphrase it, to put it in his own terms, to fit it into his own personal view of mathematical knowledge. No two mathematicians are

likely to internalize a mathematical concept in exactly the same way, so this process leads usually to multiple versions of the same theorem, each reinforcing belief, each adding to the feeling of the mathematical community that the original statement is likely to be true. Gauss, for example, obtained at least half a dozen independent proofs of his "law of quadratic reciprocity"; to date over fifty proofs of this law are known. Imre Lakatos gives, in his *Proofs and Refutations* [13], historically accurate discussions of the transformations that several famous theorems underwent from initial conception to general acceptance. Lakatos demonstrates that Euler's formula $V - E + F = 2$ was reformulated again and again for almost two hundred years after its first statement, until it finally reached its current stable form. The most compelling transformation that can take place is generalization. If, by the same social process that works on the original theorem, the generalized theorem comes to be believed, then the original statement gains greatly in plausibility.

A believable theorem gets used. It may appear as a lemma in larger proofs; if it does not lead to contradictions, then we are all the more inclined to believe it. Or engineers may use it by plugging physical values into it. We have fairly high confidence in classical stress equations because we see bridges that stand; we have some confidence in the basic theorems of fluid mechanics because we see airplanes that fly.

Believable results sometimes make contact with other areas of mathematics—important ones invariably do. The successful transfer of a theorem or a proof technique from one branch of mathematics to another increases our feeling of confidence in it. In 1964, for example, Paul Cohen used a technique called forcing to prove a theorem in set theory [2]; at that time, his notions were so radical that the proof was hardly understood. But subsequently other investigators interpreted the notion of forcing in an algebraic context, connected it with more familiar ideas in logic, generalized the concepts, and found the generalizations useful. All of these connections (along with the other normal social processes that lead to acceptance) made the idea of forcing a good deal more compelling, and today forcing is routinely studied by graduate students in set theory.

After enough internalization, enough transformation, enough generalization, enough use, and enough connection, the mathematical community eventually decides that the central concepts in the original theorem, now perhaps greatly changed, have an ultimate stability. If the various proofs feel right and the results are examined from enough angles, then the truth of the theorem is eventually considered to be established. The theorem is thought to be true in the classical sense—that is, in the sense that it *could* be demonstrated by formal, deductive logic, although for almost all theorems no such deduction ever took place or ever will.

The Role of Simplicity

> For what is clear and easily comprehended attracts; the complicated repels.
>
> David Hilbert

> Sometimes one has to say difficult things, but one ought to say them as simply as one knows how.
>
> G.H. Hardy

As a rule, the most important mathematical problems are clean and easy to state. An important theorem is much more likely to take form A than form B.

A: Every ----- is a -----.

B: If ----- and ----- and ----- and ----- and ----- except for special cases
  a) -----
  b) -----
  c) -----,

  then unless
  i) ----- or
  ii) ----- or
  iii) -----,
  every ----- that satisfies ----- is a -----.

The problems that have most fascinated and tormented and delighted mathematicians over the centuries have been the simplest ones to state. Einstein held that the maturity of a scientific theory could be judged by how well it could be explained to the man on the street. The four-color theorem rests on such slender foundations that it can be stated with complete precision to a child. If the child has learned his multiplication tables, he can understand the problem of the location and distribution of the prime numbers. And the deep fascination of the problem of defining the concept of "number" might turn him into a mathematician.

The correlation between importance and simplicity is no accident. Simple, attractive theorems are the ones most likely to be heard, read, internalized, and used. Mathematicians use simplicity as the first test for a proof. Only if it looks interesting at first glance will they consider it in detail. Mathematicians are not altruistic masochists. On the contrary, the history of mathematics is one long search for ease and pleasure and elegance—in the realm of symbols, of course.

Even if they didn't want to, mathematicians would have to use the criterion of simplicity; it is a psychological impossibility to choose any but the simplest and most attractive of 200,000 candidates for one's attention. If there are important, fundamental concepts in mathematics that are not simple, mathematicians will probably never discover them.

Messy, ugly mathematical propositions that apply only to paltry classes of structures, idiosyncratic propositions, propositions that rely on inordinately expensive mathematical machinery, propositions that require five blackboards or a roll of paper towels to sketch—these are unlikely ever to be assimilated into the body of

mathematics. And yet it is only by such assimilation that proofs gain believability. The proof by itself is nothing; only when it has been subjected to the social processes of the mathematical community does it become believable.

In this paper, we have tended to stress simplicity above all else because that is the first filter for any proof. But we do not wish to paint ourselves and our fellow mathematicians as philistines or brutes. Once an idea has met the criterion of simplicity, other standards help determine its place among the ideas that make mathematicians gaze off abstractedly into the distance. Yuri Manin [14] has put it best: A good proof is one that makes us wiser.

Disbelieving Verifications

> On the contrary, I find nothing in logistic for the discoverer but shackles. It does not help us at all in the direction of conciseness, far from it; and if it requires twenty-seven equations to establish that 1 is a number, how many will it require to demonstrate a real theorem?
>
> Henri Poincaré

> One of the chief duties of the mathematician in acting as an advisor to scientists ... is to discourage them from expecting too much from mathematics.
>
> Norbert Weiner

Mathematical proofs increase our confidence in the truth of mathematical statements only after they have been subjected to the social mechanisms of the mathematical community. These same mechanisms doom the so-called proofs of software, the long formal verifications that correspond, not to the working mathematical proof, but to the imaginary logical structure that the mathematician conjures up to describe his feeling of belief. Verifications are not messages; a person who ran out into the hall to communicate his latest verification would rapidly find himself a social pariah. Verifications 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. Being unreadable and—literally—unspeakable, verifications cannot be internalized, transformed, generalized, used, connected to other disciplines, and eventually incorporated into a community consciousness. They cannot acquire credibility gradually, as a mathematical theorem does; one either believes them blindly, as a pure act of faith, or not at all.

At this point, some adherents of verification admit that the analogy to mathematics fails. Having argued that A, programming, resembles B, mathematics, and having subsequently learned that B is nothing like what they imagined, they wish to argue instead that A is like B′, their mythical version of B. We then find ourselves in the peculiar position of putting across the argument that was originally theirs, asserting that yes, indeed, A does resemble B; our argument, however, matches the terms up differently from theirs. (See Figures 1 and 2.)

Fig. 1. The verifiers' original analogy.

Mathematics	*Programming*
theorem	... program
proof	... verification

Fig. 2. Our analogy.

Mathematics	*Programming*
theorem	... specification
proof	... program
imaginary formal demonstration	... verification

Verifiers who wish to abandon the simile and substitute B′ should as an aid to understanding abandon the language of B as well—in particular, it would help if they did not call their verifications "proofs." As for ourselves, we will continue to argue that programming is like mathematics, and that the same social processes that work in mathematical proofs doom verifications.

There is a fundamental logical objection to verification, an objection on its own ground of formalistic rigor. 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. We have been distressed to learn that this proposition, which seems self-evident to us, is controversial. So we should emphasize that as antiformalists, we would not object to verification on these grounds; we only wonder how this inherently informal step fits into the formalist view. Have the adherents of verification lost sight of the informal origins of the formal objects they deal with? Is it their assertion that their formalizations are somehow incontrovertible? We must confess our confusion and dismay.

Then there is another logical difficulty, nearly as basic, and by no means so hair-splitting as the one above: The formal demonstration that a program is consistent with its specifications has value only if the specifications and the program are independently derived. In the toy-program atmosphere of experimental verification, this criterion is easily met. But in real life, if during the design process a program fails, it is changed, and the changes are based on knowledge of its specifications; or the specifications are changed, and those changes are based on knowledge of the program gained through the failure. In either case, the requirement of having independent criteria to check against each other is no longer met. Again, we hope that no one would suggest that programs and specifications should not be repeatedly modified during the design process. That would be a position of incredible poverty—the sort of poverty that does, we fear, result from infatuation with formal logic.

Back in the real world, the kinds of input/output specifications that accompany production software are seldom simple. They tend to be long and complex and peculiar. To cite an extreme case, computing the payroll for the French National Railroad requires more than

3,000 pay rates (one uphill, one downhill, and so on). The specifications for any reasonable compiler or operating system fill volumes—and no one believes that they are complete. There are even some cases of black-box code, numerical algorithms that can be shown to work in the sense that they are used to build real airplanes or drill real oil wells, but 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, known empirically, in laboratory tests and field trials and in production. Only recently, however, has its superiority been theoretically demonstrable [6], 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.

It might be countered that while real-life specifications are lengthy and complicated, they are not deep. Their verifications are, in fact, nothing more than extremely long chains of substitutions to be checked with the aid of simple algebraic identities.

All we can say in response to this is: Precisely. 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 run into a friend's office with a program verification. Nobody is going to sketch a verification out on a paper napkin. 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.

It has been suggested that very high level languages, which can deal directly with a broad range of mathematical objects or functional languages, which it is said can be concisely axiomatized, might be used to insure that a verification would be interesting and therefore responsive to a social process like the social process of mathematics.

In theory this idea sounds hopeful; in practice, it doesn't work out. For example, the following verification condition arises in the proof of a fast Fourier transform written in MADCAP, a very high level language [18]:

If $S \in \{1, -1\}$, $b = \exp(2\pi iS/N)$, r is an integer, $N = 2^r$,
(1) $C = \{2j: 0 \le j < N/4\}$ and
(2) $a = \langle a_r: a_r = b^{r \bmod (N/2)}, 0 \le r < N/2\rangle$ and
(3) $A = \{j: j \bmod N < N/2, 0 \le j < N\}$ and
(4) $A^* = \{j: 0 \le j < N\} - A$ and
(5) $F = \langle f_r: f_r = \sum_{k_1 \in R_n} k_1(b^{k_1 \lfloor r/2^{r-1} \rfloor \bmod N}), R_r = \{j: (j - r) \bmod (N/2) = 0\} \rangle$ and $k \le r$

then

(1) $A \cap (A + 2^{r-k-1}) = \{x: x \bmod 2^{r-k} < 2^{r-k-1}, 0 \le x < N\}$
(2) $\langle \triangleright a_c \triangleright a_c \rangle = \langle a_r: a_r = b^{r2^k \bmod (N/2)}, 0 \le r < N/2 \rangle$
(3) $\langle \triangleright (F_{A \cap (A+2^{r-k-1})} + F_{\{j: 0 \le j < N\} - A \cap (A+2^{r-k-1})}) \triangleright (\langle \triangleright a_c \triangleright a_c \rangle * (F_{A \cap (A+2^{r-k-1})} + F_{\{j: 0 \le j < N\} - A \cap (A+2^{r-k-1})})) \rangle = \langle f_r: f_r = \sum_{k_1 \in R_r} k_1(b^{\lfloor r/2^{r-k-1} \rfloor \bmod N}), R_r = \{j: (j - r) \bmod 2^{r-k-1} = 0\} \rangle$
(4) $\langle \triangleright (F_A + F_{A^*}) \triangleright a^* (F_A - F_{A^*}) \rangle = \langle f_r: f_r = \sum_{k_1 \in R_r} k_1(b^{k_1 \lfloor r/2^{r-1} \rfloor \bmod N}), R_r = \{j: (j - r) \bmod (N/2) = 0\} \rangle$

This is not what we would call pleasant reading.

Some verifiers will concede that verification is simply unworkable for the vast majority of programs but argue that for a few crucial applications the agony is worthwhile. They point to air-traffic control, missile systems, and the exploration of space as areas in which the risks are so high that any expenditure of time and effort can be justified.

Even if this were so, we would still insist that verification renounce its claim on all other areas of programming; to teach students in introductory programming courses how to do verification, for instance, ought to be as farfetched as teaching students in introductory biology how to do open-heart surgery. But the stakes do not affect our belief in the basic impossibility of verifying any system large enough and flexible enough to do any real-world task. No matter how high the payoff, no one will ever be able to force himself to read the incredibly long, tedious verifications of real-life systems, and unless they can be read, understood, and refined, the verifications are worthless.

Now, 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. The lower bounds on the length of formal demonstrations for mathematical theorems are immense [19], and there is no reason to believe that such demonstrations for programs would be any shorter or cleaner—quite the contrary. In fact, even the strong adherents of program verification do not take seriously the possibility of totally automated verifiers. Ralph London, a proponent of verification, speaks of an out-to-lunch system, one that could be left unsupervised to grind out verifications; but he doubts that such a system can be built to work with reasonable reliability. One group, despairing of automation in the foreseeable future, has proposed that verifications should be performed by teams of "grunt mathematicians," low level mathematical teams who will check verification conditions. The sensibilities of people who could make such a proposal seem odd, but they do serve to indicate how remote the possibility of automated verification must be.

Suppose, however, that an automatic verifier could

somehow be built. Suppose further that programmers did somehow come to have faith in its verifications. In the absence of any real-world basis for such belief, it would have to be blind faith, but no matter. Suppose that the philosopher's stone had been found, that lead could be changed to gold, and that programmers were convinced of the merits of feeding their programs into the gaping jaws of a verifier. 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."

There is a tendency, as we begin to feel that a structure is logically, provably right, to remove from it whatever redundancies we originally built in because of lack of understanding. Taken to its extreme, this tendency brings on the so-called Titanic effect; when failure does occur, it is massive and uncontrolled. To put it another way, the severity with which a system fails is directly proportional to the intensity of the designer's belief that it cannot fail. Programs designed to be clean and tidy merely so that they can be verified will be particularly susceptible to the Titanic effect. Already we see signs of this phenomenon. In their notes on Euclid [16], a language designed for program verification, several of the foremost verification adherents say, "Because we expect all Euclid programs to be verified, we have not made special provisions for exception handling ... Runtime software errors should not occur in verified programs." Errors should not occur? Shades of the ship that shouldn't be sunk.

So, having for the moment suspended all rational disbelief, let us suppose that the programmer gets the message "VERIFIED." And let us suppose further that the message does not result from a failure on the part of the verifying system. What does the programmer know? He knows that his program is formally, logically, provably, certifiably correct. He does not know, however, to what extent it is reliable, dependable, trustworthy, safe; he does not know within what limits it will work; he does not know what happens when it exceeds those limits. And yet he has that mystical stamp of approval: "VERIFIED." We can almost see the iceberg looming in the background over the unsinkable ship.

Luckily, there is little reason to fear such a future. Picture the same programmer returning to find the same 20,000 lines. What message would he really find, supposing that an automatic verifier could really be built? Of course, the message would be "NOT VERIFIED." The programmer would make a change, feed the program in again, return again. "NOT VERIFIED." Again he would make a change, again he would feed the program to the verifier, again "NOT VERIFIED." A program is a human artifact; a real-life program is a complex human artifact; and any human artifact of sufficient size and complexity is imperfect. The message will never read "VERIFIED."

The Role of Continuity

> We may say, roughly, that a mathematical idea is "significant" if it can be connected, in a natural and illuminating way, with a large complex of other mathematical ideas.
>
> G.H. Hardy

The only really fetching defense ever offered for verification is the scaling-up argument. As best we can reproduce it, here is how it goes:

(1) Verification is now in its infancy. At the moment, the largest tasks it can handle are verifications of algorithms like FIND and model programs like GCD. It will in time be able to tackle more and more complicated algorithms and trickier and trickier model programs. These verifications are comparable to mathematical proofs. They are read. They generate the same kinds of interest and excitement that theorems do. They are subject to the ordinary social processes that work on mathematical reasoning, or on reasoning in any other discipline, for that matter.

(2) Big production systems are made up of nothing more than algorithms and model programs. Once verified, algorithms and model programs can make up large, workaday production systems, and the (admittedly unreadable) verification of a big system will be the sum of the many small, attractive, interesting verifications of its components.

With (1) we have no quarrel. Actually, algorithms were proved and the proofs read and discussed and assimilated long before the invention of computers—and with a striking lack of formal machinery. Our guess is that the study of algorithms and model programs will develop like any other mathematical activity, chiefly by informal, social mechanisms, very little if at all by formal mechanisms.

It is with (2) that we have our fundamental disagreement. We argue that there is no continuity between the world of FIND or GCD and the world of production software, billing systems that write real bills, scheduling systems that schedule real events, ticketing systems that issue real tickets. And we argue that the world of production software is itself discontinuous.

No programmer would agree that large production systems are composed of nothing more than algorithms and small programs. Patches, ad hoc constructions, bandaids and tourniquets, bells and whistles, glue, spit and polish, signature code, blood-sweat-and-tears, and, of course, the kitchen sink—the colorful jargon of the practicing programmer seems to be saying something about the nature of the structures he works with; maybe theoreticians ought to be listening to him. It has been estimated that more than half the code in any real production system consists of user interfaces and error messages—ad hoc, informal structures that are by definition unverifiable. Even the verifiers themselves sometimes seem to realize the unverifiable nature of most real software. C.A.R. Hoare has been quoted [9] as saying,

"In many applications, algorithm plays almost no role, and certainly presents almost no problem." (We wish we could report that he thereupon threw up his hands and abandoned verification, but no such luck.)

Or look at the difference between the world of GCD and the world of production software in another way: The specifications for algorithms are concise and tidy, while the specifications for real-world systems are immense, frequently of the same order of magnitude as the systems themselves. The specifications for algorithms are highly stable, stable over decades or even centuries; the specifications for real systems vary daily or hourly (as any programmer can testify). The specifications for algorithms are exportable, general; the specifications for real systems are idiosyncratic and ad hoc. These are not differences in degree. They are differences in kind. Babysitting for a sleeping child for one hour does not scale up to raising a family of ten—the problems are essentially, fundamentally different.

And within the world of real production software there is no continuity either. The scaling-up argument seems to be based on the fuzzy notion that the world of programming is like the world of Newtonian physics—made up of smooth, continuous functions. But, in fact, programs are jagged and full of holes and caverns. Every programmer knows that altering a line or sometimes even a bit can utterly destroy a program or multilate it in ways that we do not understand and cannot predict. And yet at other times fairly substantial changes seem to alter nothing; the folklore is filled with stories of pranks and acts of vandalism that frustrated the perpetrators by remaining forever undetected.

There is a classic science-fiction story about a time traveler who goes back to the primeval jungles to watch dinosaurs and then returns to find his own time altered almost beyond recognition. Politics, architecture, language—even the plants and animals seem wrong, distorted. Only when he removes his time-travel suit does he understand what has happened. On the heel of his boot, carried away from the past and therefore unable to perform its function in the evolution of the world, is crushed the wing of a butterfly. Every programmer knows the sensation: A trivial, minute change wreaks havoc in a massive system. Until we know more about programming, we had better for all practical purposes think of systems as composed, not of sturdy structures like algorithms and smaller programs, but of butterflies' wings.

The discontinuous nature of programming sounds the death knell for verification. A sufficiently fanatical researcher might be willing to devote two or three years to verifying a significant piece of software if he could be assured that the software would remain stable. But real-life programs need to be maintained and modified. 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.

And it is this discontinuity that obviates the possibility of refining verifications by the sorts of social processes that refine mathematical proofs. The lone fanatic might construct his own verification, but he would never have any reason to read anyone else's, nor would anyone else ever be willing to read his. No community could develop. Even the most zealous verifier could be induced to read a verification only if he thought he might be able to use or borrow or swipe something from it. Nothing could force him to read someone else's verification once he had grasped the point that no verification bears any necessary connection to any other verification.

Believing Software

> The program itself is the only complete description of what the program will do.
>
> P.J. Davis

Since computers can write symbols and move them about with negligible expenditure of energy, it is tempting to leap to the conclusion that anything is possible in the symbolic realm. But reality does not yield so easily; physics does not suddenly break down. It is no more possible to construct symbolic structures without using resources than it is to construct material structures without using them. For even the most trivial mathematical theories, there are simple statements whose formal demonstrations would be impossibly long. Albert Meyer's outstanding lecture on the history of such research [15] concludes with a striking interpretation of how hard it may be to deduce even fairly simple mathematical statements. 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. 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. To insist on the formal detail would be a silly waste of resources. Both symbolic and material structures must be engineered with a very cautious eye. Resources are limited; time is limited; energy is limited. Not even the computer can change the finite nature of the universe.

We assume that these constraints have prevented the adherents of verification from offering what might be fairly convincing evidence in support of their methods.

The lack at this late date of even a single verification of a working system has sometimes been attributed to the youth of the field. The verifiers argue, for instance, that they are only now beginning to understand loop invariants. At first blush, this sounds like another variant of the scaling-up argument. But in fact there are large classes of real-life systems with virtually no loops—they scarcely ever occur in commercial programming applications. And yet there has never been a verification of, say, a Cobol system that prints real checks; lacking even one makes it seem doubtful that there could at some time in the future be many. Resources, and time, and energy are just as limited for verifiers as they are for all the rest of us.

We must therefore come to grips with two problems that have occupied engineers for many generations: First, people must plunge into activities that they do not understand. Second, people cannot create perfect mechanisms.

How then do engineers manage to create reliable structures? First, they use social processes very like the social processes of mathematics to achieve successive approximations at understanding. Second, they have a mature and realistic view of what "reliable" means; in particular, the one thing it never means is "perfect." There is no way to deduce logically that bridges stand, or that airplanes fly, or that power stations deliver electricity. True, no bridges would fall, no airplanes would crash, no electrical systems black out if engineers would first demonstrate their perfection before building them—true because they would never be built at all.

The analogy in programming is any functioning, useful, real-world system. Take for instance an organic-chemical synthesizer called SYNCHEM [5]. For this program, the criterion of reliability is particularly straightforward—if it synthesizes a chemical, it works; if it doesn't, it doesn't work. No amount of correctness could ever hope to improve on this standard; indeed, it is not at all clear how one could even begin to formalize such a standard in a way that would lend itself to verification. But it is a useful and continuing enterprise to try to increase the number of chemicals the program can synthesize.

It is nothing but symbol chauvinism that makes computer scientists think that our structures are so much more important than material structures that (a) they should be perfect, and (b) the energy necessary to make them perfect should be expended. We argue rather that (a) they cannot be perfect, and (b) energy should not be wasted in the futile attempt to make them perfect. It is no accident that the probabilistic view of mathematical truth is closely allied to the engineering notion of reliability. Perhaps we should make a sharp distinction between program reliability and program perfection—and concentrate our efforts on reliability.

The desire to make programs correct is constructive and valuable. But the monolithic view of verification is blind to the benefits that could result from accepting a standard of correctness like the standard of correctness for real mathematical proofs, or a standard of reliability like the standard for real engineering structures. The quest for workability within economic limits, the willingness to channel innovation by recycling successful design, the trust in the functioning of a community of peers—all the mechanisms that make engineering and mathematics really work are obscured in the fruitless search for perfect verifiability.

What elements could contribute to making programming more like engineering and mathematics? One mechanism that can be exploited is the creation of general structures whose specific instances become more reliable as the reliability of the general structure increases.[1] This notion has appeared in several incarnations, of which Knuth's insistence on creating and understanding generally useful algorithms is one of the most important and encouraging. Baker's team-programming methodology [1] is an explicit attempt to expose software to social processes. If reusability becomes a criterion for effective design, a wider and wider community will examine the most common programming tools.

The concept of verifiable software has been with us too long to be easily displaced. For the practice of programming, however, verifiability must not be allowed to overshadow reliability. Scientists should not confuse mathematical models with reality—and verification is nothing but a model of believability. Verifiability is not and cannot be a dominating concern in software design. Economics, deadlines, cost-benefit ratios, personal and group style, the limits of acceptable error—all these carry immensely much more weight in design than verifiability or nonverifiability.

So far, there has been little philosophical discussion of making software reliable rather than verifiable. If verification adherents could redefine their efforts and reorient themselves to this goal, or if another view of software could arise that would draw on the social processes of mathematics and the modest expectations of engineering, the interests of real-life programming and theoretical computer science might both be better served.

Even if, for some reason that we are not now able to understand, we should be proved wholly wrong and the verifiers wholly right, this is not the moment to restrict research on programming. We know too little now to sense what directions will be most fruitful. If our reasoning convinces no one, if verification still seems an avenue worth exploring, so be it; we three can only try to argue against verification, not blast it off the face of the earth. But we implore our friends and colleagues not to narrow their vision to this one view no matter how promising it

[1] This process has recently come to be called "abstraction," but we feel that for a variety of reasons "abstraction" is a bad term. It is easily confused with the totally different notion of abstraction in mathematics, and often what has passed for abstraction in the computer science literature is simply the removal of implementation details.

may seem. Let it not be the only view, the only avenue. Jacob Bronowski has an important insight about a time in the history of another discipline that may be similar to our own time in the development of computing: "A science which orders its thought too early is stifled ... The hope of the medieval alchemists that the elements might be changed was not as fanciful as we once thought. But it was merely damaging to a chemistry which did not yet understand the composition of water and common salt."

Acknowledgments. We especially wish to thank those who gave us public forums—the 4th POPL program committee for giving us our first chance; Bob Taylor and Jim Morris for letting us express our views in a discussion at Xerox PARC; L. Zadeh and Larry Rowe for doing the same at the Computer Science Department of the University of California at Berkeley; Marvin Dennicoff and Peter Wegner for allowing us to address the DOD conference on research directions in software technology.

We also wish to thank Larry Landweber for allowing us to visit for a summer the University of Wisconsin at Madison. The environment and the support of Ben Noble and his staff at the Mathematics Research Center was instrumental in letting us work effectively.

The seeds of these ideas were formed out of discussions held at the DOD Conference on Software Technology in 1976 at Durham, North Carolina. We wish to thank in particular J.R. Suttle, who organized this conference and has been of continuing encouragement in our work.

We also wish to thank our many friends who have discussed these issues with us. They include: Al Aho, Jon Barwise, Manuel Blum, Tim Budd, Lucio Chiaraviglio, Philip Davis, Peter Denning, Bernie Elspas, Mike Fischer, Ralph Griswold, Leo Guibas, David Hansen, Mike Harrison, Steve Johnson, Jerome Kiesler, Kenneth Kunen, Nancy Lynch, Albert Meyer, Barkley Rosser, Fred Sayward, Tim Standish, Larry Travis, Tony Wasserman, and Ann Yasuhara.

We also wish to thank both Bob Grafton and Marvin Dennicoff of ONR for their comments and encouragement.

Only those who have seen earlier drafts of this paper can appreciate the contribution made by our editor, Mary-Claire van Leunen. Were it the custom in computer science to list a credit line "As told to ...," that might be a better description of the service she performed.

Received October 1978

References

1. Baker, F.T. Chief programmer team management of production programming. *IBM Syst. J. 11*, 1 (1972), 56–73.
2. Cohen, P.J. The independence of the continuum hypothesis. Proc. Nat. Acad. Sci., USA. Part I, vol. 50 (1963), pp. 1143–1148; Part II, vol. 51 (1964), pp. 105–110.
3. Davis, P.J. Fidelity in mathematical discourse: Is one and one really two? *The Amer. Math. Monthly 79*, 3 (1972), 252–263.
4. Bateman, P., and Diamond, H. John E. Littlewood (1885–1977): An informal obituary. *The Math. Intelligencer 1*, 1 (1978), 28–33.
5. Gelerenter, H., et al. The discovery of organic synthetic roots by computer. *Topics in Current Chemistry 41*, Springer–Verlag, 1973, pp. 113–150.
6. George, J. Alan. Computer Implementation of the Finite Element Method. Ph.D. Th., Stanford U., Stanford, Calif., 1971.
7. Heath, Thomas L. *The Thirteen Books of Euclid's Elements.* Dover, New York, 1956, pp. 204–219.
8. Heawood, P.J. Map colouring theorems. *Quarterly J. Math., Oxford Series 24* (1890), 322–339.
9. Hoare, C.A.R. Quoted in *Software Management*, C. McGowan and R. McHenry, Eds.; to appear in *Research Directions in Software Technology*, M.I.T. Press, Cambridge, Mass., 1978.
10. Jech, Thomas J. *The Axiom of Choice.* North–Holland Pub. Co., Amsterdam, 1973, p. 118.
11. Kempe, A.B. On the geographical problem of the four colors. *Amer. J. Math. 2* (1879), 193–200.
12. Kolata, G. Bari. Mathematical proof: The genesis of reasonable doubt. *Science 192* (1976), 989–990.
13. Lakatos, Imre. *Proofs and Refutations: The Logic of Mathematical Discovery.* Cambridge University Press, England, 1976.
14. Manin, Yu. I. *A Course in Mathematical Logic.* Springer–Verlag, 1977, pp. 48–51.
15. Meyer, A. The inherent computational complexity of theories of ordered sets: A brief survey. Int. Cong. of Mathematicians, Aug. 1974.
16. Popek, G., et al. Notes on the design of Euclid. Proc. Conf. Language Design for Reliable Software, SIGPLAN Notices (ACM) *12*, 3 (1977), pp. 11–18.
17. Rabin, M.O. Probabilistic algorithms. In *Algorithms and Complexity: New Directions and Recent Results*, J.F. Traub, Ed., Academic Press, New York, 1976, pp. 21–40.
18. Schwartz, J. On programming. Courant Rep., New York U., New York, 1973.
19. Stockmeyer, L. The complexity of decision problems in automata theory and logic. Ph.D. Th., M.I.T., Cambridge, Mass., 1974.
20. Ulam, S.M. *Adventures of a Mathematician.* Scribner's, New York, 1976, p. 288.