

Program Verification: Public Image And Private Reality

Editor's Note: Shortly after Communications published James Fetzer's "Program Verification: The Very Idea" [September 1988, p. 1048], a large volume of negative mail began to arrive from researchers engaged in program verification. The writers raised questions about the responsibilities of the author, editors, and reviewers, and about the publications policies of ACM. The issue of accurate reporting of science and accurate representation of research has heated up in recent years throughout all disciplines [See Selling Science, by Dorothy Nelkin, Freeman, 1987], and it is no surprise to encounter it in ACM. At my invitation, Dobson and Randell prepared this analysis; they offer a new context for observing the situation, a context that applies to many other similar situations as well. The Forum and Technical Correspondence sections of this issue contain the letters and response from the author.—

PJD

When we first read Professor Fetzer's paper, our initial somewhat uncritical reaction was to view it as an interesting, and unusually literate, contribution to the literature on the theoretical limitations of program verification—a topic which we regard as being equally as valid as discussion of its achievements and practical limitations. Perhaps naively we did not regard the paper as being of much relevance or interest to readers whose main concern was these latter topics. Clearly program verification, and especially

computer-based verification, can make major contributions to reducing the risk of residual design errors, but equally clearly there are obvious theoretical limitations on the types of design errors that can be uncovered using these techniques, and these theoretical limitations have to be understood in order that alternative measures can be taken to prevent or tolerate execution errors of the sort that cannot in principle be handled by program verification.

The furor that the article has aroused in the program verification community has caused us to go back and re-read Fetzer's paper more carefully, to see whether it really deserves the opprobrium which the program verification community has been heaping on it. Part of the problem seems to us to be the (unfortunately justifiable) fear that the paper will be misinterpreted by laymen, particularly those involved in research funding. We have good reason to sympathize with such reactions, given the misunderstandings that arose in some quarters from the Knight and Leveson paper on design diversity [3]. Indeed, both papers demonstrate the care that needs to be taken in assessing how different readerships will react to one's writings. We now believe however, that there are also significant philosophical issues underlying the present debate, which is not helped by the terms in which some of the reactions have been couched.

The issues we have in mind are not so much concerned with the theoretical aspects of program verifi-

cation, as with the relation between verification considered purely as a logical or mathematical activity—the formal manipulation of symbols—and the (spoken or written) claims made by the program verifier in defense or explanation of the manipulations. As the current controversy has shown, the examination of this relation is not only overdue, but requires a more detailed philosophical analysis than it has so far received. It is, however, not only Professor Fetzer who has failed to give this topic the careful scrutiny it so clearly needs; the program verification community has not done so either. Although this scrutiny, which would need to draw on insights from the philosophy of mathematics, the philosophy of language, and theories of truth, is not one we propose to undertake in detail here, there is one facet which seems to us to shed some light on the current dispute.

There is an important distinction, well-known to medieval school men, and surely to Fetzer, between "*rationes essendi*" or why-it-is-so reasons (ontological or explanatory reasons) and "*rationes cognoscendi*," or why-we-hold-it-to-be-so reasons (epistemological or evidential reasons). To see the difference, consider the height of yonder tree. We say it is roughly 100 feet. The explanatory reason for our claim will be, roughly, that it is a tree of such-and-such a sort, growing in this sort of climate in this sort of soil, and that under such conditions trees of that type can sustain a height of about

100 feet. On the other hand, an evidential reason (why-we-hold-it-to-be-so) for our claim might be that it casts a shadow of 50 feet at a time of day when a 10-foot pole casts a shadow of 5 feet. One set of reasons deals with the *explanation* of our claims (why-it-is-so), the other deals with the *substantiation*, or the *evidence* we have for making them (why-we-hold-it-to-be-so). The difference is that the first set of reasons derive from the phenomenon itself ("treehood" in our example), whereas the second derives from how we find out about the phenomenon ("measuring shadows" in our example). Mathematical logic, for example, provides the first sort of reasons whereas program testing can only provide the second sort of reasons.

With this distinction in mind, we can now analyze the dispute in the following way: *it seems to us that Fetzer thinks the verificationists have been representing their work as providing explanatory reasons for program correctness, whereas they claim their work provides merely evidential reasons.* If this is so, it raises the important question, *Why* does Fetzer appear to believe that the program verification community hold that program verification technology guarantees execution correctness simply by virtue of the nature of verification?

We believe that a partial answer is that although the program verifiers claim their reasons are evidential (why-they-hold-it-to-be-so), they often speak, write and behave (and thus give the impression) as if their reasons are explanatory (why-it-is-so). This is not always a matter of simple misrepresentation (though it sometimes is), but lies deeper and in the very terminology used. We argue that, although program proofs are indeed proofs in the logical sense, their purpose is not to demonstrate new and unsuspected properties, but rather to show that certain properties—i.e., certain forms of incorrectness—are conspicuous by their absence. In other words, their purpose is not proof but refutation.

From the point of view of the theory of knowledge, the hypothesis

'This program will execute correctly' is one that can never be proven, only falsified. Program verification is a fruitful way of demonstrating certain very common types of falsification in certain sorts of programs, but can never provide complete coverage of reasons why the program will not work (e.g., errors in the compiler, the microcode, the hardware . . .).

From the point of view of mathematics, on the other hand, the important concept is not that of refutation but of proof. Mathematical proof is the demonstration of existence of certain abstract objects, or objects with certain abstract properties. In short, mathematics is concerned with explaining existence, whereas refutation is concerned with providing concrete (counter-) evidence. Refutation thus stimulates the mathematics but is not itself necessarily mathematical.

In fact the widespread use of 'proof,' and cognate terms, seems to indicate that formal methods are considered to be explanatory, not evidential in nature. It might have been better if 'program proving' had been called 'program refutation' and 'program verification' had been termed 'program falsification,' since the fact that a program has been verified (i.e., is theoretically correct) in no way reduces the obligation to demonstrate that a program execution does in fact produce the proven result (i.e., is empirically correct). Unfortunately, though, there are those who believe this obligation is in fact removed by the construction of a mathematical proof.

For example, we can vouch for the essential accuracy of the following exchange at the University of Erehwon [not its real name].

Professor M: Tell me, Dr. N, why do you not teach program testing in your course on software engineering?

Dr. N: Because to teach testing would be to admit to a deficiency in my course on program construction. Correctly constructed programs do not need testing, therefore there is no need to teach it.

All we can respond to that is summed up in a characteristically caustic passage from the philosopher Austin ([1], p. 2):

One might well want to ask how seriously this doctrine is intended, just how strictly and literally the philosophers who propound it mean their words to be taken . . . It is, as a matter of fact, not at all easy to answer, for, strange though the doctrine looks, we are sometimes told to take it easy—really it's just what we've all believed all along. (There's the bit where you say it and the bit where you take it back.)

Austin is here referring to the doctrine that we never *directly* perceive or sense material objects; but he could well have been speaking of many of the program verificationists' claims, for their response to Fetzer's article seems to be a retreat from things we (and Fetzer) might have thought they said. (Though it has to be admitted that many of the positions most notoriously associated with the originators of the idea of program verification are in fact vulgarizations by their less gifted but more publicity-conscious followers.)

But the distinction that we have drawn between the two sorts of reasons is not just a philosophical sophistry. It has arisen in one form or another many times before, and frequently gives rise to controversy. One example is the dispute a generation ago in AI, between those who thought that AI could in principle explain something about the workings of human intelligence and those who thought that AI could at most merely provide evidence that human intelligence was capable of being simulated mechanically. Another example, from quite a different field, is in the area of literary theory. The issue, well discussed in the very readable and well-argued polemic by Tallis [4] written—unusually for the field—both by and for an intelligent layman, is whether post-Saussurean theory is a new description of how language actually works (as arguably it is not) or a new way of thinking in

a language (as it almost certainly is). We could cite further examples from disciplines even more removed from computing.

In all these disputes, however, there is a common pattern: a book or paper is published that outrages the sensibilities of some closely-knit and often somewhat distinctive group. Their response is to accuse the author of lack of understanding, lack of responsibility, lack of intellectual integrity and so on. The two sides hurl grenades at each other for a time, then things quiet down. But the matter is not settled; it is merely dormant, since the underlying issues are neither identified nor resolved.

This pattern has been observed so often that, as a psychological phenomenon, it has been given a name: groupthink. Our observation is that many cases of intellectual groupthink are of just the pattern we have identified. The group, and its opponents, fail to appreciate the differences in their respective perceptions of the nature of the underlying arguments propounded by the group. Mutual respect and understanding are frustrated or prevented by a failure on both sides to make the distinction between 'this is the way the world is' and 'this is a useful way of thinking about the world.'

In essence, then, our view is that the dispute has arisen as a discrepancy between the program verification community's view of their practice and objectives on the one hand, and the outside world's view of those objectives on the other. It is in fact a public image problem, and public image is not to be corrected simply by the writing of intemperate letters. We suspect that a major underlying reason for the undesired image is that there has been a good measure of overselling of the concepts. (This is not of course the first occurrence of this practice in computer science, nor will it be the last.) A recent and all too clear example of this overselling can be seen in the case of hardware verification by

contrasting exaggerated advertising claims for the VIPER microprocessor (e.g., "VIPER is the first commercially available microprocessor with both a formal specification and a proof that the chip conforms to it") with the careful and balanced discussion, in Avra Cohn's report on her (very impressive) work on formal verification of the VIPER 'block model' [2], of what was actually proven, and what was out of scope of the proving techniques. (The above advertisement is in fact quoted by Cohn, who rightly goes on to state that "such assertions, taken as assurances of the impossibility of design failure in safety-critical applications, could have catastrophic results.")

The point is that the sort of *a posteriori* proofs of limited design correctness which program verification provides, though of undoubted importance, do not necessarily provide a *major* contribution to the reliability, safety or security of real machinery. We can only endorse the following quotation from Cohn's report as applying to software as well as hardware verification [*op. cit.*, p. 13; emphasis in original]:

... using a hardware design verified only down to the register-transfer level (and then only partially verified and only in 'normal' situations) as part of the control system in extraordinarily hazardous applications does not seem significantly safer than using any other design. The use of the word 'verified' must *under no circumstances* be allowed to confer a false sense of security.

If our belief that formal verification has been oversold is correct, then articles such as "Program Verification: The Very Idea," misconceived though it undoubtedly is, might be seen in the longer term as providing a stimulus for a strategy of corrective measures necessary to counter the false sense of security that is, we fear, all too commonly

promoted by vendors of verification technology and/or verified programs.

Having said that, however, we would not wish to be thought to subscribe to the view that the blame for the current misunderstanding can be laid solely on the interests of commercial promotion. We wish to reiterate our view that program verification can have a significant role to play in the vital task of clarifying specifications and removing design faults. It can provide supporting evidence for the claim of program correctness, though it can never by itself explain or guarantee execution correctness. But there is a well-known class of philosophical pothole which comes from ignoring the dictum "To understand what a proof has proved, study the proof." Some fall into this pothole through ignorance or oversight. Some make a living out of falling into it and encouraging others to do so. Some, though they may not fall into it themselves, are aware of it but signally fail to put any fences or warning notices round it. Of these, all, it seems to us, are culpable; but the last maybe more so.

REFERENCES

1. Austin, J.L. *Sense and Sensibilia*. Oxford University Press, 1962.
2. Cohn, A.A. Correctness Properties of the Viper Block Model: The Second Level. Tech. Rept. 134, University of Cambridge Computer Laboratory, May 1988.
3. Knight, J.C., and Leveson, N.G. An Empirical Study of Failure Probabilities in Multi-version Software. In *Proceedings of the 16th IEEE International Symposium on Fault-Tolerant Computing (FTCS-16)*, (Vienna, Austria, July 1986) pp. 165-170.
4. Tallis, R. *Not Saussure: A Critique of Post-Saussurean Literary Theory*. Macmillan, Basingstoke, England, 1988.

Authors' Present Address: Computing Laboratory, University of Newcastle upon Tyne NE1 7RU, England.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.