Lengthly, indeed, but conceptually simple I prefer this to rules similar to those of algol 60.

Finally you may appreciate that in passing actual parameters an array type variable may be allowed as an aggregate of variables.

The syntactical details should be rather obvious, judged by the examples, and are left as an exercise for an implementor.

[1] A.N. Habermann: Critical Comments on the Programming Language PASCAL.
Acta Informatica, Vol 4, fasc. 1, 1973

[2] N. Wirth: Comments on a Note on Dynamic Arrays in PASCAL
SIGPLAN Notices, Vol 11, no 1, January 1976

In Defense of Program Testing

or

Correctness Proofs Considered Harmful

Andrew S. Tanenbaum
Vakgroep Informatica
Wiskundig Seminarium
Vrije Universiteit
Amsterdam, The Netherlands

Dijkstra' remark (Dijkstra, 1972) to the effect that testing can only demonstrate the presence of errors and not their absence in unquestionably true. This observation, together with the recent development of techniques for formally proving programs to be correct, has led some computer scientists to believe that correctness proofs can replace testing as a means for insuring that programs do what they are supposed to do. The purpose of this note is to point out several reasons why even programs rigorously proven to be correct should be nevertheless thoroughly tested. Correctness proofs can supplement, but cannot replace comprehensive testing.

1. All Programs are Correct

There is no such thing as an incorrect program. All programs perform some computation correctly. Unfortunately the computation performed is not always the one the "customer" (the person who will ultimately use the program) wanted. Since problems are invariably initially specified in a natural language (e.g. English), a correctness proof must not only prove that the program's results obey certain axioms, but also that these axioms capture the essence of what the customer wanted. If formal specifications are used, the proof must demonstrate that they correspond to what is desired, i.e. that the formal specifications are themselves correct.

To illustrate the subtlety of this point, consider the specification of an operating system. One of the design criteria is: "No one shall be able to acquire information to which he/she has no right". This is usually formalized as a model consisting of a matrix whose rows are processes and whose columns are objects (including processes). Each matrix element specifies the access rights of one process to one object. Based on this model one might be able to rigorously prove that process A could not leak information to process B because neither A nor B could access the other in any way, either directly, or indirectly via a chain of other processes and objects.

Unfortunately, this proof does not guarantee that the design criterion has been satisfied, since process A can leak information to process B in devious ways, such as degrading system performance in a clocked manner (Lampson, 1973). The trouble here is that the formal specification (the access matrix) fails to model the system completely (it neglects the implicit sharing of CPU and memory). No correctness proof can ever demonstrate that the formal specifications have missed a subtle, but crucial point. In contrast, extensive testing by an experienced penetration team might discover this kind of flaw.

Journal articles about correctness proofs invariably assume that the problem to be solved is well understood. While this is certainly true of some problems, it is not true of all problems (e.g. write the control software for a robot whose task is to tidy up the mess in my office without throwing away anything important). In such cases the formal specifications are likely to be very tentative and constantly changing as more experience is acquired. A proof that the current version of the program is internally consistent with the current version of the specifications (and that is all a proof can ever show) does not imply that the software will perform as it should. Again, extensive testing is needed.

Even if the problem does lend itself to a well defined mathematical specification, (e.g. sorting a list of integers) there is always the danger that the programmer will misunderstand the customer and write the wrong program, (e.g. sorting the integers in ascending instead of descending order) something a test would catch immediately.

2. Do You Trust the Proof?

The experience of the past 2 decades suggests that there are precious few human beings on the face of the earth who are capable of sitting down and writing a 10 page program that contains no errors whatsoever. Proving a program is far more difficult than writing a program. If writing correct programs is close to the limits of human abilities, is there any reason to believe that programmer's proofs will be more trustworthy than their programs? I think not.

Of course, if the computer can check the proof, the situation is brighter. Nevertheless I would venture the guess that most proofs offered to proof checkers will fail for either syntactic or "run time" errors (probably resulting in a hexadecimal dump of all the intermediate theorems generated by the checker). In any event proof checkers are still the stuff Ph.D. theses are made of. It will be many years before computer manufacturers routinely provide proof checkers along with compilers for their major languages. Automatic proof generators will also no doubt come some day, but that day is far in the future.

There is also the nasty problem of verifying the verifier. One might try hand proving it, but then how do you verify that proof? One might also try feeding it to itself, but if the verifier contains an error, it may incorrectly announce that it is flawless. There may even be undecidability problems lurking here.

3. Why Proofs may be Nonproofs

Proofs may simply contain mistakes, but there are also some more subtle issues that must be dealt with. First, the programmer may not correctly understand the semantics of the programming language. For example, he may mistakenly believe that the end-of-file indication is turned on when the last record of a file is read, whereas in fact it is only turned on by the next read attempt (or vice versa). Any file i/o programs written by such a programmer are likely to be incorrect, and his/her proofs will surely not bring these defects to light.

Second, an incorrect understanding of the semantics of an operating system call can lead to similar errors that proof techniques will not detect, unless the proof also includes the operating system. (This is usually not possible because most present day operating systems are, in fact, not correct.)

Third, an incorrect understanding of how the hardware works can lead to "proven" programs that fail to perform as expected. The proof of an operating system disk driver module may be invalidated when its author belatedly discovers that under certain peculiar conditions the controller can lose interrupts. We recently discovered that our real time clock is subject to race conditions, and if you read it while it is changing from 2^n to 2^n-1 you may get semi-random numbers. Since the computer's manufacturer is typically either unaware of, or embarrassed by, these kinds of problems, they are rarely mentionned in the written documentation. They can be programmed around, but only if you know about them, and that knowledge can be acquired only by testing, not by making proofs.

Fourth, the mathematical model used as a basis for the proof may differ the computer realization in subtle, but crucial ways. A "proven" numerical analysis program may give the wrong answers because the proof considered x, y, and z to be real numbers, rather than finite precision numbers subject to overflow and underflow. Furthermore, in the absence of testing one could easily forget that $(1.0/3.0) \times 3.0$ is usually 0.999 ... 9 due to roundoff.

Fifth, large programs are usually constructed by teams of people, resulting in the well known interface problem: each individual procedure is free of logical errors, but makes incorrect assumptions about the parameters or results of other procedures. Large proofs are subject to the same phenomenon.

Sixth, computers have only a finite amount of memory. A beautiful recursive garbage collector may be "proven" to be correct, but may unceremoniously blow up the first time it is tried due to stack overflow. Again, the only remedy is to test it extensively in the vicinity of genuine garbage.

Seventh, various charcter set difficulties can occur. For example, a compiler may assume that each operator (+ - × / < ≤ = ≠ ≥ >) can be represented as a single symbol. Although this is true for e.g. CDC Display Code, it is not true for ASCII. A compiler "proven" correct on the assumption of single character operators may need substantial changes to its lexical scan, proof or no proof. End-of-line conventions are another source of problems; cr, lf, cr+lf, and lf+cr, etc. are used on various computers. At some CDC sites consecutive colons in columns 10n+9 and 10n+10 ($0 \leq n \leq 7$) are converted to end-of-line by the operating system!

Eighth, a program which is otherwise correct may fail to work due to timing considerations. Although timing is usually not important for applications programs, it is often crucial in real time systems. A certain i/o device may only function correctly if a data transfer is initiated within N microseconds after some event occurs (e.g. device selection, interrupt etc.). A formal proof that ignores timing problems (including process switching time) may demonstrate that a program is correct when it is not.

Some mathematicians may regard these and similar points as "trivial", arguing that a program which failed completely because it incorrectly assumed that ≠ was one character is nevertheless correct "in principle". One can imagine the following scenario. The customer tries to use the program and discovers that it does not work. He confronts the programmer:

Customer: "Your program has a bug in it."
Programmer: "That is impossible. I proved it correct."
Customer (displaying test output): "Look here!"
Programmer (displaying proof): "Look here!"

This situation is called a deadlock.

Putting the problem in other words, many programming errors occur because the programmer was not aware of something he/she ought to have been aware of. Given the imperfect nature of homo sapiens, this problem is unavoidable. Endres (1975) concluded that nearly half the errors in the programming project he studied were of this type. Testing offers the hope that reality will rudely intrude and force the programmer to become aware of the error. Correctness proofs can detect inconsistencies between the input and output specifications, but cannot bring to light important issues that the programmer may have completely forgotten about.

4. But it was not My Fault

Even if the program really is correct, it may still not run. There may be errors in the compiler, linker, operating system or even in the hardware. A program which has been tested has at least some chance of discovering these errors so they can be corrected or avoided; if a program is delivered to its customer untested, this may simply have the effect of shifting the debugging from the programmer to the customer.

5. Conclusion

Correctness proofs have their place, but they can easily lull one into a false sense of security, and therein lies the potential danger. They should be regarded as an important technique for insuring internal consistency, but they can not eliminate problems having to do with the "programming environment" rather than the internal program logic. Testing can not guarantee this either, but well thought out testing can possibly detect some errors that proofs inherently miss. The two methods do not compete; rather they complement each other. If possible, all programs should be proven AND tested.

Those of us engaged in education should insure that our students learn that testing can be approached systematically (Brown and Lipow, 1975; Goodenough and Gerhart, 1975; Hetzel, 1973; King, 1975; Kopetz, 1975; Rain, 1973; van Tassel, 1974). It should certainly not be dismissed as unimportant. Testing has acquired an unsavory reputation due to the haphazard way it has been applied. If people tried to prove programs by generating assertions at random, proof techniques would not be very effective either.

For the benefit of those people who still think this is all nonsense, and that correctness proofs are sufficient, I pose the following thought exercise. Imagine that you were the project manager responsible for developing a fully automated air traffic control system whose software was all shown to be correct by formal proof, but never tested even once. Would you be willing to be a passenger on the first actual flight using the system? Would you take out flight insurance?

6. References

Brown, J.R. and Lipow, M: "Testing for Software Reliability", Sigplan Notices, vol. 10, pp. 518-527, 1975.

Dijkstra, E.W.: "The Humble Programmer", CACM, vol. 15, pp. 859-866, 1972.

Endres, A.: "An Analysis of Errors and Their Causes in Systems Programs", IEEE Transactions on Software Engineering, vol. 1, pp. 140-149, 1975.

Goodenough, J.B. and Gerhart, S.L.: "Toward a Theory of Test Data Selection", IEEE Transactions on Software Engineering, vol. 1, pp. 156-173, 1975.

Hetzel, W.C.: Program Test Methods, Prentice-Hall, Englewood Cliffs, 1973.

King, J.C.: "A New Approach to Program Testing", Sigplan Notices, vol. 10, pp. 228-233, 1975.

Kopetz, H.: "On the Connections Between Range of Variable and Control Structure Testing", Sigplan Notices, vol. 10, pp. 511-517, 1975.

Lampson, B.W.: "A Note on the Confinement Problem", CACM, vol 16, pp. 613-614, 1973.

Rain, M.: "Two Unusual Methods for Debugging System Software", Software Practice and Experience, vol. 3, pp. 61-63, 1973.

van Tassel, D.: Program Style, Design, Efficiency, Debugging and Testing, Prentice-Hall, Englewood Cliffs, 1974.