The Mathematics of Programming

C.A.R. Hoare

Cambridge 4th April, 1985

Summary

Computer programming is a profession in which the talents of a mathematician are most urgently needed and can be most effectively employed. The mathematics of programming is an exciting topic for research, with great promise of original and useful results. Its subject matter and methods are those of discrete pure mathematics, but the challenge and excitement are those of applied mathematics and engineering - to design and deliver a reliable product performing a valuable service of a kind that is usually new and sometimes spectacular.

2009 CARH draft for ? New Scientist article?

1. The Problem

Stored program digital computers must be the most reliable mechanisms ever built by man. Millions of computers throughout the world, and thousands in space, are executing millions of instructions per second for millions of seconds without a single error in any of the millions of bits from which each computer is made. In spite of this, nobody trusts a computer, and their lack of faith is amply justified. The fault lies not in the computer hardware, but in the programs which control it - programs replete the errors, oversights, inadequacies and misunderstandings of the programmers who compose them. There are some large and widely used programs in which hundreds of new errors are discovered each month; and even when they are corrected, the error rate remains constant over several decades. Indeed, it is suspected that each correction introduces on average more than one new error. Statistical estimation offers the dubious comfort that only a negligible proportion of all the errors in these programs will ever be discovered, before the programs are superceded by new products, which are, of course, equally unreliable.

Now computers are beginning to be used increasingly in life-critical applications, where the correction of errors on discovery is not an acceptable option - for example in industrial process control, nuclear reactors, weapons systems, station-keeping of ships at sea, aero engines, and railway signalling. The engineers in charge of these projects are naturally worried about the correctness of the programs performing these tasks, and they have suggested a number of expedients for tackling the problem. Unfortunately, many of these methods are of limited efffectiveness, because they are based on false analogies rather than a true appreciation of the nature of computer programs and the activity of programming. Let me give some examples of four methods which have been proposed and used to increase confidence in the reliability of programs.

1.1 The sword of Damocles

The first method is the simplest, and requires no technical insight. I illustrate it with a story. When Brunel's ship the SS Great Britain was launched into the river Thames, it made such a big splash that several spectators on the opposite bank were drowned. Nowadays on launching a ship, the force of entry into the water is reduced by rope tethers which are designed to break at carefully calculated intervals.

When the first computer was installed in the Mathematish Centrum in Amsterdam it was used to calculate the appropriate intervals and breaking strains of these tethers. In order to ensure the correctness of the program which did the calculations, the programmers were invited to watch the launching from the first row of the ceremonial viewing stand set up on the opposite bank; fortunately they accepted and they survived.

Contd/.....3

A similar solution has been proposed for programs which control the propeller and steering of ship which has to keep station in rough seas very close to the leg of an oil drilling rig. Unfortunately, when the programmer resigns his highly paid residential sinecure on board the ship, is this because of boredom or sea-sickness, or from fear of something worse?

1.2 Eyeballing

When the early American artificial satellites were controlled by computer, the programs were written by outside contractors. Like all such products, when delivered they were subjected to quality control. This was done by visual inspection of the absolute binary code of the programs, rows and rows of raw digits, zeroes and ones. They could not use any higher level programming language, since assemblers and compilers are large programs, and therefore even less to be trusted than the programs which they compile.

In spite of this, a massive suite of programs was constructed to assist in the eyeballing, for example to reconstruct the assembly code and flow charts from the binary. These charts were then annotated by human checkers with assertions about scaling factors of the arithmetic operations; and machine assistance was provided to check that the scaling factor would not vary each time the program loops.

The fundamental flaw in this approach is that when checking something you should always check it against something else, which is either known to be reliable, or which is similarly checked. To check the absolute binary code against absolutely nothing is a fearsome task, and requires inspired guesswork to reconstruct the documentation, designs and specifications. No wonder checking can be even more expensive than the original production of the code, which progresses in the more natural direction from abstract to concrete, from specification through design to the code.

The basic mistake in eyeballing is that the checking is done far too late: it is a fundamental principle of quality control that what should be checked is not the product but the methods by which it is produced. It is only by improving methods that quality can be assured.

1.3 Heroic Testing

Programs that control critical processes can often be tested by running them initially in a simulated environment, for example inside a fast mainframe computer. Suppose that the simulation runs many times faster than real time. Thus in one year it may be possible to simulate say a thousand year's operation of the process, and check all the answers given by the program. If there are only a few errors detected, it is then quite unlikely that any such error will occur within the first ten years of actual live running, and this constitutes the whole design life of the program.

Contd/....4

This appealing method suffers from a number of devastating drawbacks. The first and least of them is that the delay before installation of a working program is usually unacceptable.

The second flaw is philosophical: it is morally very difficult to risk people's lives on a program with known bugs. And yet to 'correct' the known bugs would not only be wholly ineffective; it would be quite disastrous, since it could introduce a completely unknown batch of new bugs into the population.

The third is a practical flaw. What happens if ten errors are detected in the thousand year test? This gives a quite unacceptable risk that an error will occur in the ten years of actual use. But the only remedy is to rewrite the whole program and start the test again. By that time, the whole project will have lost its value and relevance.

The fourth is a logical flaw: the method depends on the correctness of the simulation environment and on that of the checking program. Yet if the checking program is correct, why not use it as part or whole of the program which controls the real industrial process?

The fifth drawback fortunately makes the previous four irrelevant: it is only in very simple and increasingly rare applications that it is possible to run a simulation, even on the fastest supercomputers, at a rate faster than real time; and it is ridiculous to simulate a program for a hundred times its design life before putting it into service.

1.4 Diversity

In many life-critical applications, the problem of hardware reliability requires installation of three or more identical computers, with a voting circuit at their output, to ensure that every action has the agreement of at least two of them. The likelihood of two or more computers going wrong simultaneously is very much smaller than just one. Since the hardware is available, the same redundancy technique can be cheaply applied to software, by writing three or more independent programs, and loading a different one into each computer.

In hardware, redundancy is designed to counter transient errors, such as might be caused by an occasional alpha-particle impinging on the silicon. It does not deal with persistent faults, which must be cured by manual (or automatic) replacement of components. Unfortunately, in software there is no reason to suppose errors are transient; a single erroneous subscript can cause the program to be overwritten, so that it never works again. To guard against this, the hardware is usually designed to clear the store and reload the program after each cycle of operation. Unfortunately this severely restricts the range of applications and algorithms which may be protected by redundancy. For example, it is impossible to accumulate past readings, to integrate or to smooth them.

Contd/.....5

A second weakness in this method is that there is no reason to suppose that errors in programs produced by independent programmers will be independent. Quite the reverse, there is good evidence that programmers are subject to the same kinds of misunderstandings and oversights - for example, forgetting to test for an extreme case, or omitting to provide for zero iterations of a loop.

But there is one new circumstance that will make diversity impractical on large systems. In many real time applications, the response of a computer depends on details of the timing of the signals which it responds to - for example, the arrival of an interrupt. Thus two correct programs which receive signals at slightly different times can give different results, both of them correct. Unfortunately, a hardware voting circuit cannot know this, and will invoke unnecessary alarms. In spite of vigorous efforts to prevent it, this is what actually occurred on the first attempted launch of the first space shuttle, the Columbia, in the first highly spectacular public failure of a computer program. And it was actually caused by the very technique designed to increase reliability. Yet this is the technique which is nowadays commonly required and implemented in life-critical applications.

1.5 Summary

I have described four methods which have been used in the past to tackle the problem of program reliability; and I have described and even mocked their limitations and inadequacies. But please don't think I am advocating abandonment of these methods. Certainly not: they have proved reasonably effective so far, and they must continue to be used until some better solution is established, and preferably for some time after. In matters of safety, I recommend belt, braces, as well as thick underwear. But when proper effective methods for achieving program reliability have become established, I predict that the methods I have described so far will play a very minor role.

Cont/.....6

2. The Solution

A proper solution to the problem of program reliability must be based not just on current practice, or on analogies with other branches of engineering, but on a full understanding of the nature of computers and the programs that control them. I claim that this understanding is to be obtained from mathematics; and I present this claim in the following four theses:

1. Computers are Mathematical machines. Every aspect of their behaviour is defined with mathematical precision; and every detail can in principle be deduced from this definition by the laws of pure mathematics.

2. Computer programs are mathematical expressions describing the instructions which the computer will follow with unprecedented accuracy, reliability and speed.

3. A computer programming language is a mathematical theory, including concepts, notations, axioms, and theorems which enable the consequences of every design decision taken by a programmer to be rigorously deduced from nothing but the text of the program.

4. From these premises I conclude that programming is a mathematical activity, whose reliable practice requires only the determined application of traditional methods of mathematical understanding, calculation and proof.

I shall illustrate this conclusion by showing a small fragment of the elementary mathematics of computer programming, nothing more advanced than the simple kind of algebra of arithmetic operations which we learn at school. First, we must learn to say:

'Let P and Q be programs'

But how can they be? - programs can be enormous - millions of lines of FORTRAN code! How can all that complexity be squashed into just a single letter or even two letters P and Q?

Well numbers can be pretty big too; yet we can't even start doing mathematics until we allow them to be represented by a few little letters.

If P andQ are programs, so is (P;Q). A computer executes this program by first executing P; and when P successfully terminates, it executes Q. In familiar programming languages like FORTRAN and BASIC the operator of sequential composition is represented by a change to a new line

P
Q

so that the program P is written above the program Q.

Contd/.....7

However, this vertical formatting is not conducive to mathematical reasoning, and the semicolon of PASCAL and the mathematical calculus of relations is much to be preferred. It enables us to formulate our first algebraic law, the associativity of sequential composition

$$(P;Q);R \;=\; P;(Q;R)$$

Our next notation is a constant, the program 'skip' which terminates successfully without doing anything or changing anything. Thus it has the same effect as a CONTINUE in FORTRAN or a remark in BASIC:

```
REM IS JUST LIKE SKIP
```

The algebraic law which defines the essential property of 'skip' is that it is unit of sequential composition

$$skip;P \;=\; P;skip \;=\; P$$

The conditional I write with the condition b in the middle

$$P \lhd b \rhd Q \qquad \text{"P } \underline{\text{if}}\text{ b } \underline{\text{else}}\text{ Q"}$$

This is executed by first testing the condition b. If b is true, P is executed, and if b is false, Q is executed. Conditionals are written in BASIC using jumps and labels.

```
410 IF b THEN GOTO 560
411 Q
...
550 GOTO 593
560 P
...
593 REM TO FILL IN THE LINE
```

The reason why I have preferred an infix notation $\lhd\ b\ \rhd$ is that it simplifies the expression and use of algebraic laws. For example, $\lhd b \rhd$ is idempotent and associative; it distributes through other conditionals $\lhd c \rhd$, and it admits distribution of ; from the right.

$$P \lhd b \rhd (Q \lhd c \rhd R) \;=\; (P \lhd b \rhd Q) \lhd c \rhd (P \lhd c \rhd R)$$

$$(P \lhd b \rhd Q);R \;=\; (P;R) \lhd b \rhd (Q;R)$$

Enthusiasts for BASIC or FORTRAN are invited to translate these laws into their favourite notations - and then try using them to reason about their programs.
My final example is the assignment

$$x:=f(x)$$

which is executed by evaluating the function f at the current value of x and assigning the result as the new value of x. An important example of an algebraic law of assignment is that successive assignments to the same variable can be amalgamated by substitution

$$(x:=f(x);\ x:=g(x)) \;=\; x:=g(f(x))$$

Contd/.....8

In most programming languages this law is subject to a number of complicated qualifications. When choosing or designing a programming language, it is a good idea to avoid such complications, to simplify reasoning about programs and their correctness.

In BASIC and FORTRAN for assignment you have to use a plain equal sign = instead of the := sign adopted in PASCAL, for example

X = 3 x Y

or even X = X + 1

If the designers of these programming languages were deliberately trying to make it difficult to reason mathematically about computer programs, they could not have chosen better notations for their nefarious purposes.

Programming in BASIC is like doing long division in Roman numerals. Roman numerals are very convenient for carving on stone, and they are not too bad for addition. For multiplication, they present problems; but division is next to impossible. The only known method is to guess the answer, multiply out, and correct it if it is wrong - in exactly the same way as BASIC programmers are taught now to write their program, test it and correct it if it is wrong.

Once you have seen the mathematical basis of computer programming, it is clearly foolish to seek reliability of programs by any other method than by sound and familiar methods of calculation and proof, checked where necessary by other mathematicians or even by computer. In mathematics we don't use the sword of Damocles. Did they offer Isaac Newton free accommodation in an elegant mansion built for the purpose in the middle of a firing range, just to ensure that canon balls fly in a parabola? In mathematics we rarely use a test to destruction. Did they test the binomial theorem on a hundred thousand numbers, to ensure it would work on the hundred occasions it was needed? In mathematics we do not use diversity. We know the grocer's assistant who adds the figures five times, and compares the different results, but we don't regard him as a mathematician. We know topologists who give five different definitions of continuity, but that's not to ensure that at least three of them will work when the others fail? An in mathematics we do not use Roman numerals any more. ~~An~~ Should we go on teaching BASIC to schoolchildren, on the grounds that it runs better on stone-age Computers?

Before this happens, we need to know much more about the mathematics of programming. In this talk I have given a sample of half a dozen algebraic laws about sequential programs. I know about eighty laws in all, and these are sufficient to characterise a complete theory of sequential programming. Further laws are required for concurrent programming, of the kind likely to be required in life-critical applications.

This is only a start. You would not expect to solve serious problems in science and engineering just by algebraic laws, though these form a very necessary basis, taken for granted in all subsequent reasoning on the subject.

To discover and formulate and prove the validity of new methods of developing correct programs is like discovering new methods of solving differential equations. It is not something which programmers and computer scientists can do by themselves. Like other scientists and engineers before us, we need the utmost assistance of good research mathematicians who are willing to turn away from the traditional branches of mathematics which they have studied at length from books, and start thinking from first principles about a new branch of their subject. It is only by continuous exploration of such new branches that mathematics can retain its vigour and intellectual rewards.

I believe that this will be an exciting and important branch of mathematics. Its subject matter and methods appear to be those of discrete pure mathematics, but the challenge is that of applied mathematics and engineering - to design, develop and deliver a reliable product meeting requirements of a kind that are often new and sometimes spectacular.