Maths adds safety to computer programs

Bill Sanderson

Computers control the operation of nuclear power stations and the launch of missiles, so we all have a vested interest in keeping computer programs free of errors. But few programmers know that mathematics holds the key to safety

Tony Hoare

MATHEMATICS COUNTS

DIGITAL COMPUTERS must be the most reliable mechanisms built by the human race. Millions of computers throughout the world, and thousands in space, execute billions of instructions per second for billions of seconds without a single error in any of the millions of bits that comprise each computer. Yet few of us would trust our lives to a computer.

The fault lies not in the computer's hardware but in the programs which control it. Programs faithfully reproduce the errors, oversights, inadequacies and misunderstandings of the programmers who compose them. There are some large and widely used programs, operating systems and compilers in which hundreds of new errors are discovered each year. Even when programmers correct errors, the rate at which users continue to discover new ones remains constant over several decades. Indeed, some suspect that each correction introduces more than one new error. And only a few of the errors in these programs will ever be discovered before the programs are superseded by new products. These new products are, of course, equally unreliable.

Most of the errors that are found in general computer programs are extremely subtle: their effects are not serious, and it is easy to avoid them until the software's supplier gets round to correcting them. But computers are beginning to play an increasing role in "life-critical applications", situations where the correction of errors on discovery is not an acceptable option—for example, in control of industrial processes, nuclear reactors, weapons systems, oil rigs, aero engines and railway signalling. The engineers in charge of such projects are naturally worried about the correctness of the programs performing these tasks, and they have suggested several expedients for tackling the problem. Let me give some examples of four proposed methods.

The first method is the simplest. I illustrate it with a story. When Brunel's ship the SS Great Britain was launched into the River Thames, it made such a splash that several spectators on the opposite bank were drowned. Nowadays, engineers reduce the force of entry into the water by rope tethers which are designed to break at carefully calculated intervals.

When the first computer came into operation in the Mathematish Centrum in Amsterdam, one of the first tasks was 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. They accepted and they survived.

A similar solution has been proposed for programs that control the propeller and steering of a ship which has to keep station in rough seas close to the leg of an oil drilling rig. The

action of the wind and waves is so sudden that no human helmsman could avoid collision, and the task must be delegated to a computer program. But if we require the programmer to demonstrate the reliability of his program by joining the crew of the ship, a question arises when he resigns his highly paid post. Is this because of boredom, seasickness or fear of something worse?

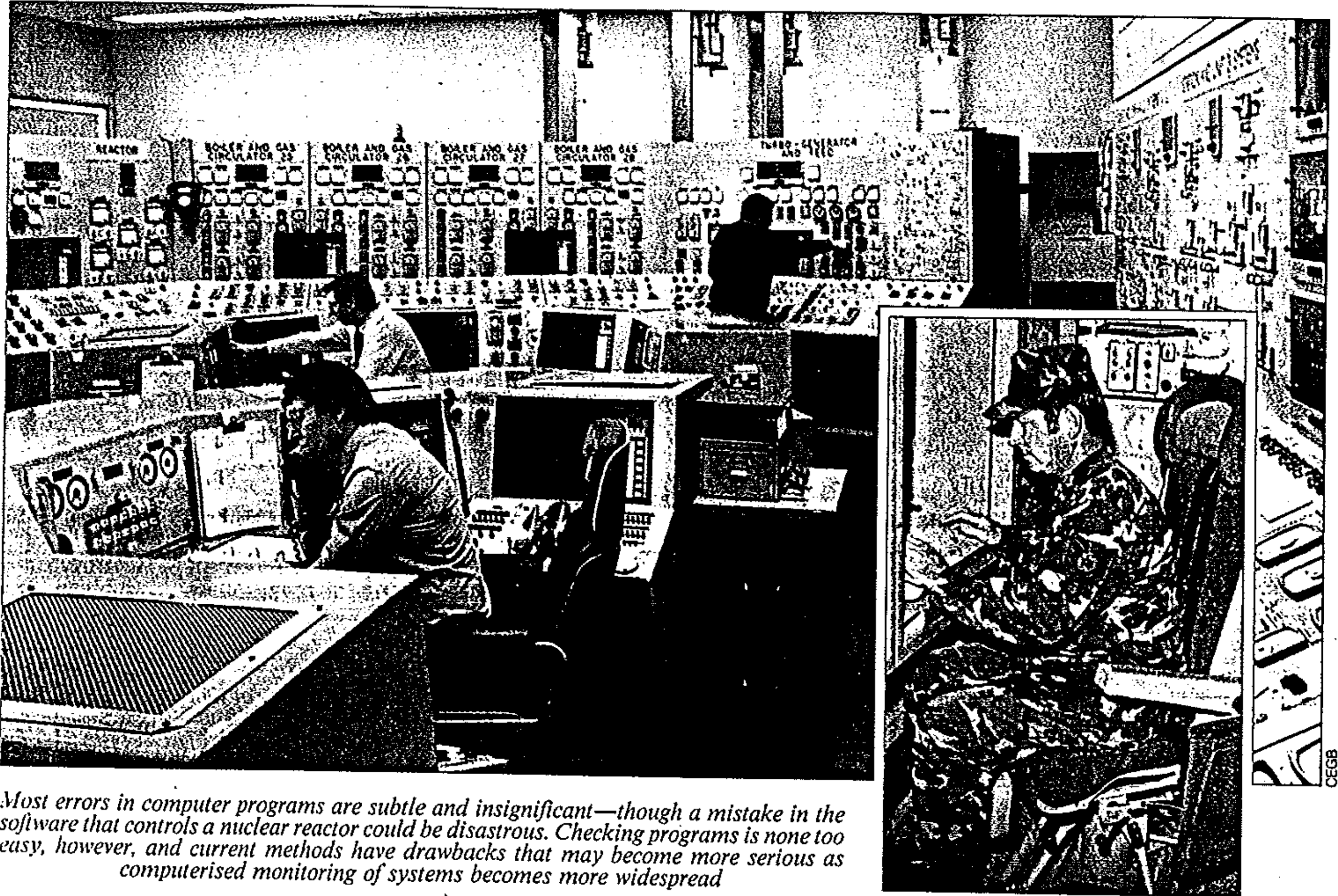
When the early American satellites were first controlled by on-board computers, outside contractors wrote the programs. On delivery, checkers visually inspected the absolute binary code of the programs: rows and rows of raw binary digits. They could not use any higher level programming language, since assemblers and compilers are large programs and, therefore, even less trustworthy than the programs that they compile.

To assist in the "eyeballing", NASA constructed a massive suite of programs—for example, to reconstruct the assembly code from the delivered binary code, to draw flow-charts and to analyse all the control paths. The human checkers then annotated these charts with assertions about scaling factors of the arithmetic operations; with the help of machines, they then checked that the scaling factor would not vary each time that the program went round a loop.

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

This kind of machine-assisted analysis is still very popular in the checking of safety-critical software. The quoted reason is far from reassuring: many of the programs are written without any specification at all; so the only thing there to check is the ultimate code. The basic mistake is that the checking is done far too late: it is a fundamental principle of quality control that what you should check is not the product but the methods by which it is produced. It is only by improving methods that it becomes possible to achieve reliability.

We can often test programs that control critical processes 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 years' operation of the process and check all the answers that the program gives. If only a few errors are detected, it is then unlikely that any such error will occur within the first 10 years of live running—which may be the length of the program's useful lifetime. This appealing method suffers from several devastating drawbacks. The first and least of them is that the delay before a working program is installed is usually unacceptable.



Most errors in computer programs are subtle and insignificant—though a mistake in the software that controls a nuclear reactor could be disastrous. Checking programs is none too easy, however, and current methods have drawbacks that may become more serious as computerised monitoring of systems becomes more widespread

CEGB

The second flaw is philosophical: it is morally very difficult to risk people's lives on a program that has known bugs. Yet to "correct" the known bugs would not only be wholly ineffective; it could be disastrous since it could introduce a completely unknown batch of new bugs. To counter this possibility, all the testing would have to start again from the beginning. The third is a practical flaw. What happens if 10 errors are detected in the thousand-year test? This result gives a quite unacceptable risk that an error will occur in the 10 years of actual use. The only remedy is to rewrite the whole program and start the test again. By that time, the project will have lost its value and relevance.

The fourth is a logical flaw: the method depends on the correctness of the simulated 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 process?

The fifth drawback fortunately makes the previous four irrelevant: it is only in the 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. So this method is applicable only to programs with a design life measured in minutes or hours; it is therefore not appropriate for most civilian applications.

In many life-critical applications, the problem of the reliability of hardware requires there to be three or more identical computers, with a voting circuit at their output which ensures that every action has the agreement of at least two of

Only a computer can react quickly enough to prevent a ship from colliding with an oil drilling rig. But how reliable are its programs?

them. The likelihood that two or more computers would go wrong simultaneously is very much smaller than the risk that just one would. Since the hardware is available, it is possible to apply the same technique to software. You get three or more independent teams of programmers to write three or more independent programs, and load a different one into each computer. If one of the programs goes wrong occasionally, the other programs, which are unlikely to go wrong on the same occasion, outvote it.

In hardware, such a method will deal with transient errors, such as might arise from an occasional cosmic ray impinging on the silicon chip. 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 that errors are transient; a single erroneous subscript can cause the program to be overwritten, so that it never works again. To guard against this possibility, it is necessary to design the hardware to clear the store and reload the program between each cycle of operation. So this technique applies only to programs whose operation is a series of independent cycles, with no long-term storage. Such a program cannot accumulate past readings, to integrate or to smooth them. Thus the technique is applicable only to simple control processes.

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. Programmers are often educated in the same "culture"; they find the same things difficult, and they 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 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 simple voting circuit in the hardware cannot know this, and it will invoke unnecessary alarms. In spite of vigorous efforts to prevent it, this is what actually occurred on the first attempted launch of the American space shuttle. The Columbia was a victim of the first highly spectacular public failure of a computer program. The cause of the failure was the very technique designed to ensure reliability.

There is more insidious danger in using diversity for long-running programs. When a software error occurs, the hardware-comparison circuit will signal an alarm; but the operators will attribute this alert to a software error, and will ignore it. As a result, when it really is the hardware that goes wrong, it will be impossible for the operators to distinguish it from an error in the software, and they will continue to ignore it. Thus intermittent and faulty hardware will not be replaced in good time. This infection of hardware by the unreliability of the software has actually been observed on a computer for which the ALGOL compiler used the hardware parity violation circuits to detect the programming error of an uninitialised variable. In a short while, as a result of lack of maintenance, the hardware of the main store of every computer which used that compiler became unreliable.

Mathematics holds the solution

These are the four methods that people have proposed and used in practice for the construction of safety-critical software. As far as I know, they have been almost completely successful so far, and no large-scale loss of human life has ever been attributed to a programming error. Nevertheless, each method suffers from several drawbacks that may become more serious with the imminent increase in the scale and sophistication of systems whose safety is monitored by computer programs. I therefore suggest that we should explore an additional method, which promises to increase the reliability of programs. The same method has assisted the reliability of designs in other branches of engineering, namely the use of mathematics to calculate the parameters and check the soundness of a design before passing it for construction and installation.

Alan Turing first made this suggestion some 40 years ago; it was put into practice, on occasion, by the other great pioneer of computing, John von Neuman. Shigeru Igarashi and Bob Floyd revived the idea some 20 years ago, providing the groundwork for a wide and deep research movement aimed at developing the relevant mathematical techniques. Wirth, Dijkstra, Jones, Gries and many others, (including me) have made significant contributions. Yet, as far as I know, no one has ever checked a single safety-critical program using the available mathematical methods. What is more, I have met several programmers and managers at various levels of a safety-critical project who have never even *heard* of the posibility that you can establish the total correctness of computer programs by the normal mathematical techniques of modelling, calculation and proof.

Such total ignorance would seem wilful, and perhaps it is. People working on safety-critical projects carry a heavy responsibility. If they ever get to hear of a method which might lead to an improvement in reliability, they are obliged to investigate it in depth. This would give them no time to complete their current projects on schedule and within budget. I think that this is the reason why no industry and no profession has ever voluntarily and spontaneously developed or adopted an effective and relevant code of safe practice. Even voluntary codes are established only in the face of some kind of external pressure or threat, arising from public disquiet, fostered by journals and newspapers and taken up by politicians.

A mathematical proof is, technically, a completely reliable method of ensuring the correctness of programs, but this method could never be effective in practice unless it is accompanied by the appropriate attitudes and managerial techniques. These techniques are in fact based on the same ideas

that have been used effectively in the past.

It is not practical or desirable to punish errors in programming by instant death. Nevertheless, programmers must stop regarding error as an inevitable feature of their daily lives. Like surgeons or airline pilots, they must feel a personal commitment to adopt techniques that eliminate error and to feel the appropriate shame and resolution to improve when they fail. In a safety-critical project, every failure should be investigated by an impartial enquiry, with powers to name the programmer responsible, and forbid that person any further employment on safety-critical work. In cases of proven negligence, criminal sanctions should not be ruled out. In other engineering disciplines, these measures have led to marked improvement in personal and professional responsibility, and in public safety. There is no reason why programmers should be granted further immunity.

Mathematical calculations and proofs are in many ways very like programs. They are long and intricate texts in which the slightest blunder leads to invalidity. In principle, a programmer could learn to write completely formal proofs, and a computer could be programmed to check these proofs. This principle has inspired some excellent research and development of proof-checking programs. But the labour of constructing proofs with sufficient formality for a machine to check them has turned out to be excessive.

For the time being, the most effective method of checking proofs is to submit them to the gaze of another programmer or mathematician. The checker then joins the programmer in taking responsibility for the correctness of the program. The principle that the work of an engineer should be inspected and signed off by another more experienced and competent engineer lies at the heart of the codes of safe practice in all branches of engineering. The checking of proofs has much in common with the eyeballing of code, but it is in principle more effective, since it is easier to detect a hole in a well-presented proof than it is to find an oversight in a program. This is because a proof checker only needs to check the validity of each line of the proof, comparing it only with one or two previous lines. For a program, the checker has to check each line in the context of every other line of code in the program—a task which is quite impossible for large programs.

Testing for realism

However carefully an engineer has specified, designed and implemented a product, it would be extremely foolish to put the item into service without subjecting it to thorough tests that are as realistic as possible. Such tests are necessary to check the adequacy of the original specification, and the general assumptions on which it is based. We need these tests in order to check our understanding of the relations between the hardware, the software and their working environment. Evaluation of this kind also checks the adequacy of the methods by which the system was specified, designed and constructed.

The vast majority of all tests should succeed—otherwise there is something so seriously wrong with the product that it would be best to throw it away. But inevitably there will be an occasional failure even in programs which were thought to be proved correct. On these occasions the proper response is first to find the cause of the error: for example, carelessness, misunderstanding or use of inadequate methods. Second, you must assess whether that same cause might have led to other errors in other parts of the program. All such doubtful areas must be checked again. Only then should the detected error be corrected, and the correction itself undergo more rigorous checks against the possibility that it will introduce further errors. These common-sense principles are standard practices in many branches of engineering, yet their application to programming has been slow to gain recognition.

Most computer programs today are written in high-level programming languages which have to be translated into the language of the machine before they can be executed. The program which does the translation is itself large, complex and subject to error. This has inhibited the use of high-level languages for safety-critical programming. This is a mistake. Some compilers for simple high-level languages have proved reliable in widespread use and the chance that they will make a mistake in compiling a particular safety-critical program is very small. The chance that such a mistake would escape detection in routine testing is far smaller still. In the last resort, a visual check of the machine code against the original high-level program is still a possibility—easier and safer than writing the original program.

The principle of diversity is fundamental to improving the reliability of programs. The people who check proofs should be independent of those who construct them. Those who design test regimes should be independent of those who design the objects undergoing the tests. Finally, for ultimate confidence, it would be ideal to have two independent proofs, preferably using different methods of proof. It is by independent experiment that the laws of physics are confirmed; and even mathematicians are happier with fundamental theorems that have been proved more than once.

The principle of diversity when applied to proof will be more effective than when applied to programs, and will probably be no more expensive. It does not suffer from any of the problems or dangers which arise from the use of hardware error checks as a protection against software bugs.

Mathematics is a traditionally unpopular subject, among programmers as in the general population. Even programmers with a university degree in pure and applied mathematics have no idea how to apply their mathematical skills to the practice of their profession. Fortunately, the mathematical aspects of programming are now beginning to find their way into university curricula. I look forward to the day when these topics find their way into specialist teaching in secondary schools. My prediction is that the mathematical study of programming will contribute to wider appreciation, understanding and love of mathematics; for mathematics is constantly rejuvenated by discovery of new applications. By that time, the professional use of mathematics in safety-critical programming will ensure that computer programs are the most reliable component of any system in which they are embedded.

To achieve this desirable goal, I believe it is necessary to raise the alarm publicly about the inadequacy of some of the methods that are currently proposed and in use. But it is important not to exaggerate the risk or to provoke over-reaction. Critical processes controlled by computer programs are probably far more reliable than those controlled by older analogue devices and certainly more reliable than manual methods. This is mainly due to the greatly increased reliability of the hardware of computers. Indeed, at present, the fear of errors in programming is the main reason for delay in the introduction of computerised control in safety-critical applications. This delay itself represents a risk to public safety. The introduction of mathematical methods to prove programs correct will therefore bring double benefit—it will improve the reliability of existing applications, and it will give people confidence to extend these benefits to new applications. □

Professor C. A. R. Hoare, FRS, is professor of computation in the programming research group, Oxford University Computing Laboratory.

References

Roland C. Backhouse, *Program Construction and Verification*, Prentice Hall (1986). An introductory text for assertional methods of program development and proof. These methods form the basis of a standard of rigour and accuracy for safety-critical programs.

Software, Vital Key to UK Competitiveness, Report of ACARD, HMSO, 1986, £6·00. This report emphasises the importance of improved methods in the construction of software for all purposes. It suggests that mathematical methods introduced for safety-critical programs will bring improvements throughout the software industry.