How Did Software Get So Reliable Without Proof?

C.A.R. Hoare
Oxford University Computing Laboratory,
Wolfson Building, Parks Road, Oxford, OX1 3QD, UK

FME ‘96 *Proceedings of the Third International Symposium of Formal Methods Europe on Industrial Benefit and Advances in Formal Methods*, March 18—22, 1996, Pages 1-17

Abstract. By surveying current software engineering practice, this paper reveals that the techniques employed to achieve reliability are little different from those which have proved effective in all other branches of modern engineering: rigorous management of procedures for design inspection and review; quality assurance based on a wide range of targeted tests; continuous evolution by removal of errors from products already in widespread use; and defensive programming, among other forms of deliberate over-engineering. Formal methods and proof play a small direct role in large scale programming; but they do provide a conceptual framework and basic understanding to promote the best of current practice, and point directions for future improvement.

1 Introduction

Twenty years ago it was reasonable to predict that the size and ambition of software products would be severely limited by the unreliability of their component programs. Crude estimates suggest that professionally written programs delivered to the customer can contain between one and ten independently correctable errors per thousand lines of code; and any software error in principle can have spectacular effect (or worse: a subtly misleading effect) on the behaviour of the entire system. Dire warnings have been issued of the dangers of safety-critical software controlling health equipment, aircraft, weapons systems and industrial processes, including nuclear power stations. The arguments were sufficiently persuasive to trigger a significant research effort devoted to the problem of program correctness. A proportion of this research was based on the ideal of certainty achieved by mathematical proof.

Fortunately, the problem of program correctness has turned out to be far less serious than predicted. A recent analysis by Mackenzie has shown that of several thousand deaths so far reliably attributed to dependence on computers, only ten or so can be explained by errors in the software: most of these were due to a couple of instances of incorrect dosage calculations in the treatment of cancer by radiation. Similarly predictions of collapse of software due to size have been falsified by continuous operation of real-time software systems now measured in tens of millions of lines of code, and subjected to thousands of updates per year. This is the software which controls local and trunk telephone exchanges; they have dramatically improved the reliability and performance of telecommunications throughout the world. And aircraft, both civil and military, are now flying with the aid of software measured in millions of lines—though not all of it is safety-critical. Compilers and operating systems of a similar size now number their satisfied customers in millions.

So the questions arise: why have twenty years of pessimistic predictions been falsified? Was it due to successful application of the results of the research which was motivated by the predictions? How could that be, when clearly little software has ever has been subjected to the rigours of formal proof? The objective of these enquiries is not to cast blame for the non-fulfilment of prophecies of doom. The history of science and engineering is littered with false predictions and broken promises; indeed they seem to serve as an essential spur to the advancement of human knowledge; and nowadays, they are needed just to maintain a declining flow of funds for research. Nixon's campaign to cure cancer within ten years was a total failure; but it contributed in its time to the understanding on which the whole of molecular medicine is now based. The proper role for an historical enquiry is to draw lessons that may improve present practices, enhance the accuracy of future predictions, and guide policies and directions for continued research in the subject.

The conclusion of the enquiry will be that in spite of appearances, modern software engineering practice owes a great deal to the theoretical concepts and ideals of early research in the subject; and that techniques of formalisation and proof have played an essential role in validating and progressing the research.

However, technology transfer is extremely slow in software, as it should be in any serious branch of engineering. Because of the backlog of research results not yet used, there is an immediate and continuing role for education, both of newcomers to the profession and of experienced practitioners. The final recommendation is that we must aim our future theoretical research on goals which are as far ahead of the current state of the art as the current state of industrial practice lags behind the research we did in the past. Twenty years perhaps?

2 Management

The most dramatic advances in the timely delivery of dependable software are directly attributed to a wider recognition of the fact that the process of program development can be predicted, planned, managed and controlled in the same way as in any other branch of engineering. The eventual workings of the program itself are internal to a computer and invisible to the naked eye; but that is no longer any excuse for keeping the design process out of the view of management; and the visibility should preferably extend to all management levels up to the most senior. That is a necessary condition for the allocation of time, effort and resources needed for the solution of longer term software problems like those of reliability.

The most profitable investment of extra effort is known to be at the very start of a project, beginning with an intensified study not only of the requirements of the ultimate customer, but also of the relationship between the product and the environment of its ultimate use. The greatest number (by far) of projects that have ended in cancellation or failure in delivery and installation have already begun to fail at this stage. Of course we have to live with the constant complaint that the customers do not know what they want; and when at last they say they do, they constantly change their mind. But that is no excuse for abrogating management responsibility. Indeed, even stronger management is required to explore and capture the true requirements, to set up procedures and deadlines for management of change, to negotiate and where necessary invoke an early cancellation clause in the contract. Above all, the strictest management is needed to prevent premature commitment to start programming as soon as possible. This can only lead to a volume of code of unknown and untestable utility, which will act forever after as a dead weight, blighting the subsequent progress of the project, if any.

The transition from an analysis of requirements to the specification of a program to meet them is the most crucial stage in the whole project; the discovery at this stage of only a single error or a single simplification would fully repay all the effort expended. To ensure the proper direction of effort, the management requires that all parts of the specification must be subjected to review by the best and most experienced software architects, who thereby take upon themselves an appropriate degree of responsibility for the success of the project. That is what enables large implementation teams to share the hard-won experience and judgement of the best available engineers.

Such inspections, walkthroughs, reviews and gates are required to define important transitions between all subsequent phases in the project, from project planning, design, code, test planning, and evaluation of test results. The individual designer or programmer has to accept the challenge not only of making the right decisions, hut also of presenting to a group of colleagues the arguments and reasons for confidence in their correctness. This is amazingly effective in instilling and spreading a culture conducive to the highest reliability. Furthermore, if the review committee is not satisfied that the project can safely proceed to its next phase, the designer is required to rework the design and present it again. Even at the earliest stage, management knows immediately of the setback, and already knows, even if they refuse to believe it, that the delivery will have to be rescheduled by exactly the same interval that has been lost. Slack for one or two such slippages should be built into the schedule; but if the slack is exhausted, alternative and vigorous action should be no longer delayed.

At the present day, most of the discussion at review meetings is conducted in an entirely informal way, using a language and conceptual framework evolved locally for the purpose. However, there is now increasing experience of the benefits of introducing abstract mathematical concepts and reasoning methods into the process, right from the beginning. This permits the consequences of each proposed feature and their possible combinations to be explored by careful and exhaustive mathematical reasoning, to avoid the kind of awkward and perhaps critical interactions that might otherwise be detected only on delivery. At the design stage, the mathematics can help in exploring the whole of the design space, and so give greater assurance that the simplest possible solution has been adopted. Even stricter formalisation is recommended for specifying the interfaces between the components of the design, to be implemented perhaps in different places at different times by different people. Ideally, one would like to see a proof in advance of the implementation that correctness of the components, defined in terms of satisfaction of the interface specifications, will guarantee correctness of their subsequent assembly. This can greatly reduce the risk of a lengthy and unpredictable period of integration testing before delivery.

At the final review of the code, judicious use of commentary in the form of assertions, preconditions, postconditions and invariants can greatly help in marshalling a convincing argument that a program actually works. Furthermore, it is much easier to find bugs in a line of reasoning than it is in a line of code. In principle, correctness of each line of reasoning depends at most on two preceding lines of reasoning, which are explicitly referenced. In principle, correctness of each line of code depends on the behaviour of every other line of code in the system.

Success in the use of mathematics for specification, design and code reviews does not require strict formalisation of all the proofs. Informal reasoning among those who are fluent in the idioms of mathematics is extremely efficient, and remarkably reliable. It is not immune from failure; for example simple misprints can be surprisingly hard to detect by eye. Fortunately, these are exactly the kind of error that can be removed by early tests. More formal calculation can be reserved for the most crucial issues, such as interrupts and recovery procedures, where bugs would be most dangerous, expensive, and most difficult to diagnose by tests.

A facility in formalisation and effective reasoning is only one of the talents that can help in a successful review. There are many other less formal talents which are essential. They include a wide understanding of the application area and the marketplace, an intuitive sympathy with the culture and concerns of the customer, a knowledge of the structure and style of existing legacy code, acquaintance and professional rapport with the most authoritative company experts on each relevant topic, a sixth sense for the eventual operational consequences of early design decisions, and above all, a deep sense of personal commitment to quality, and the patience to survive long periods of intellectual drudgery needed to achieve a thoroughly professional result. These attributes are essential. The addition of mathematical fluency to the list is not going to be easy; the best hope is to show that it will enhance performance in all these other ways as well.

3 Testing

Thorough testing is the touchstone of reliability in quality assurance and control of modern production engineering. Tests are applied as early as possible at all stations in the production line. They are designed rigorously to maximise the likelihood of failure, and so detect a fault as soon as possible. For example, if parameters of a production process vary continuously, they are tested at the extreme of their intended operating range. Satisfaction of all tests in the factory affords considerably increased confidence, on the part of the designer, the manufacturer, and the general public, that the product will continue to work without fail throughout its service lifetime. And the confidence is justified: modern consumer durables are far more durable than they were only twenty years ago.

But computing scientists and philosophers remain sceptical. E.W. Dijkstra has pointed out that program testing can reveal only the presence of bugs, never their absence. Philosophers of science have pointed out that no series of experiments, however long and however favourable can ever prove a theory correct; but even only a single contrary experiment will certainly falsify it. And it is a basic slogan of quality assurance that “you cannot test quality into a product”. How then can testing contribute to reliability of programs, theories and products? Is the confidence it gives illusory?

The resolution of the paradox is well known in the theory of quality control. It is to ensure that a test made on a product is not a test of the product itself, but rather of the methods that have been used to produce it—the processes, the production lines, the machine tools, their parameter settings and operating disciplines. If a test fails, it is not enough to mend the faulty product. It is not enough just to throw it away, or even to reject the whole batch of products in which a defective one is found. The first principle is that the whole production line must be re-examined, inspected, adjusted or even closed until the root cause of the defect has been found and eliminated.

Scientists are equally severe with themselves. To test a theory they devise a series of the most rigorous possible experiments, aimed explicitly and exclusively to disprove it. A single test with a negative result may occasionally be attributed to impure ingredients or faulty apparatus; but if the negative outcome is repeated, parts of the theory have to be rethought and recalculated; when this gets too complicated, the whole theory has to be abandoned. As Popper points out, the non-scientist will often die with (or even for) his false beliefs; the scientist allows his beliefs to die instead of himself.

A testing strategy for computer programs must be based on lessons learned from the successful treatment of failure in other branches of science and engineering. The first lesson is that the test strategy must be laid out in advance and in all possible detail at the very earliest stage in the planning of a project. The deepest thought must be given to making the tests as severe as possible, so that it is extremely unlikely that an error in the design of the program could possibly remain undetected. Then, when the program is implemented and passes all its tests the first time, it is almost unbelievable that there could be any inherent defect in the methods by which the program has been produced or any systematic lapse in their application. This is the message of Harlan Mill's "clean room" strategy.

The earliest possible design of the test strategy has several other advantages. It encourages early exploration, simplification and clarification of the assumptions underlying use of the program, especially at edges of its operating range; it facilitates early detection of ambiguities and awkward interaction effects latent in the specification; and it concentrates attention from the earliest stage on central problems of assuring correctness of the system as a whole. Many more tests should be designed than there will ever be time to conduct; they should be generated as directly as possible from the specification, preferably automatically by computer program. Random selection at the last minute will protect against the danger that under pressure of time the program will be adapted to pass the tests rather than meeting the rest of its specification. There is some evidence that early attention to a comprehensive and rigorous test strategy can improve reliability of a delivered product, even when at the last minute there was no time to conduct the tests before delivery!

The real value of tests is not that they detect bugs in the code, but that they detect inadequacy in the methods, concentration and skills of those who design and produce the code. Programmers who consistently fail to meet their testing schedules are quickly isolated, and assigned to less intellectually demanding tasks. The most reliable code is produced by teams of programmers who have survived the rigours of testing and delivery to deadline over a period of ten years or more. By experience, intuition, and a sense of personal responsibility they are well qualified to continue to meet the highest standards of quality and reliability. But don't stop the tests: they are still essential to counteract the distracting effects and the perpetual pressure of close deadlines, even on the most meticulous programmers.

Tests that are planned before the code is written are necessarily "black box" tests; they operate only at the outermost interfaces of the product as a whole, without any cognizance of its internal structure. Black box tests also fulfil an essential role as acceptance tests, for use on delivery of the product to the customer's site. Since software is invisible, there is absolutely no other way of checking that the version of the software loaded and initialised on the customer's machine is in fact the same as what has been ordered. Another kind of acceptance test is the suite of certification tests which are required for implementations of standard languages like COBOL and ADA. They do little to increase confidence in the overall reliability of the compiler, but they do at least fairly well ensure that all the claimed language features have in fact been delivered; past experience shows that even this level of reliability cannot be taken for granted.

Another common kind of black box test is regression testing. When maintaining a large system over a period of many years, all suggested changes are submitted daily or weekly to a central site. They are all incorporated together, and the whole system is recompiled, usually overnight or at the week end. But before the system is used for further development, it is subjected to a large suite of tests to ensure that it still works; if not, the previous version remains in use, and the programmer who caused the error has an uncomfortable time until it is mended. The regression tests include all those that have detected previous bugs, particularly when this was done by the customer. Experience shows that bugs are often a result of obscurity or complication in the code or its documentation; and any new change to the code is all too likely to reintroduce the same bug—something that customers find particularly irksome.

4 Debugging

The secret of the success of testing is that it checks the quality of the process and methods by which the code has been produced. These must be subjected to continued improvement, until it is normal to expect that every test will be passed first time, every time. Any residual lapse from this ideal must be tracked to its source, and lead to lasting and widely propagated improvements in practice. Expensive it may be, but that too is part of the cure. In all branches of commerce and industry, history shows dramatic reduction in the error rates when their cost is brought back from the customer to the perpetrator.

But there is an entirely different and very common response to the discovery of an error by test: just correct the error and get on with the job. This is known as debugging, by analogy with the attempt to get rid of an infestation of mosquitoes by killing the ones that bite you—so much quicker and cheaper and more satisfying than draining the swamps in which they breed. For insect control, the swatting of individual bugs is known to be wholly ineffective. But for programs it seems very successful; on removal of detected bugs, the rate of discovery of new bugs goes down quite rapidly, at least to begin with. The resolution of the paradox is quite simple; it is as if mosquitoes could be classified into two very distinct populations, a gentle kind that hardly ever bite, and a vicious kind that bite immediately. By encouraging the second kind, it is possible to swat them, and then live comfortably with the yet more numerous swarm that remains. It seems possible that a similar dichotomy in software bugs gives an explanation of the effectiveness of debugging.

The first tests of newly written code are those conducted by the programmer separately on isolated segments. These are extraordinarily effective in removing typographical errors, mis-keying, and the results of misunderstanding the complexity of the programming language, the run-time library or the operating system. This is the kind of error that is easily made, even by the most competent and diligent programmer, and fortunately just as easily corrected in today's fast-turn-round visual program debugging environments. Usually, the error is glaringly obvious on the first occasion that a given line of code is executed.

For this reason, the objective of the initial test suite is to drive the program to execute each line of its code at least once. This is known as a coverage test; because it is constructed in complete knowledge of the object under test, it is classified as an "open box" test. In hardware design a similar principle is observed; the suite of tests must ensure that every stable element makes at least one transition from high voltage to low and at least one transition from low voltage to high. Then at least any element that is stuck at either voltage level will be detected.

The cheapest way of testing a new or changed module of code in a large system is simply to insert the module in the system and run the standard suite of regression tests. Unfortunately, the coverage achieved in this way does not seem adequate: the proportion of code executed in regression tests has been reported to be less than thirty per cent. To improve this figure, a special test harness has to be constructed to inject parameters and inspect results at the module level. Unfortunately, for a module with many parameters, options and modes, to push the coverage towards a hundred percent gets increasingly difficult; in the testing of critical software for application in space, comprehensive testing is reported to increase costs by four times as much as less rigorously tested code. Equally unfortunately, total coverage is found to be necessary: more errors continue to be discovered right up to the last line tested.

In hardware design, exhaustive testing of stuck-at faults has also become impossible, because no sufficiently small part of a chip can be exercised in isolation from the rest. Nevertheless, quite short test sequences are adequate to identify and discard faulty chips as they come off the production line. It is a fortunate property of the technology of VLSI that any faults that are undetected by the initial tests will very probably never occur; or at least they will never be noticed. They play the role of the gentle kind of mosquito: however numerous, they hardly ever bite.

Returning to the case of software, when the program or the programmer has been exhausted by unit testing, the module is subjected to regression testing, which may throw up another crop of errors. When these are corrected, the regression tests soon stop detecting new errors. The same happens when an updated system is first delivered to the customer: nearly all the errors are thrown up in early runs of the customer's own legacy code. After that, the rate at which customers report new errors declines to a much lower and almost constant figure.

The reason for this is that even the most general-purpose programs are only used in highly stereotyped ways, which exercise only a tiny proportion of the total design space of possible paths through the code. Most of the actual patterns of use are explored by the very first regression tests and legacy tests, and beta testing enables the customer to help too. When the errors are removed from the actually exercised paths, the rate at which new paths are opened up is very low. Even when an anomaly is detected, it is often easier to avoid it by adapting the code that invokes it; this can be less effort and much quicker than reporting the error. Perhaps it is by this kind of mutual adaption that the components of a large system, evolving over many years, reach a level of natural symbiosis; as in the world of nature, the reliability and stability and robustness of the entire system is actually higher than that of any of its parts.

When this stable state is reached, analysis of a typical error often leads to an estimate that, even if the error were uncorrected, the circumstances in which it occurs are so unlikely that on a statistical basis they will not occur again in the next five thousand years. Suppose a hundred new errors of this kind are detected each year. Crude extrapolation suggests that there must be about half a million such errors in the code. Fortunately, they play the same role as the swarms of the gentle kind of mosquito that hardly ever bites. The less fortunate corollary is that if all the errors that are detected are immediately corrected, it would take a thousand years to reduce the error rate by twenty percent. And that assumes that there are no new errors introduced by the attempt to correct one which has already been detected. After a certain stage, it certainly pays both the customer and the supplier to leave such errors unreported and uncorrected.

Unfortunately, before that stage is reached, it often happens that a new version of the system is delivered, and the error rate shoots up again. The costs to the customer are accepted as the price of progress: the cost to the supplier is covered by the profit on the price of the software. The real loss to the supplier is the waste of the time and skill of the most experienced programmers, who would otherwise be more profitably employed in implementing further improvements in the functionality of the software. Although (surprisingly) the figures are often not officially recorded, the programmers themselves estimate that nearly half their time is spent in error correction. This is probably the strongest commercial argument for software producers to increase investment in measures to control reliability of delivered code.

5 Over-engineering

The concept of a safety factor is pervasive in engineering. After calculating the worst case load on a beam, the civil engineer will try to build it ten times stronger, or at least twice as strong, whenever the extra cost is affordable. In computing, a continuing fall in price of computer storage and increase in computer power has made almost any trade-off acceptable to reduce the risk of software error, and the scale of damage that can increasingly result from it. This leads to the same kind of over-engineering as is required by law for bridge-building; and it is extremely effective, even though there is no clear way of measuring it by a numeric factor.

The first benefit of a superabundance of resource is to make possible a decision to avoid any kind of sophistication or optimisation in the design of algorithms or data structures. Common prohibitions are: no data packing, no optimal coding, no pointers, no sharing, no dynamic storage allocation. The maximum conceivably necessary size of record or array is allocated, and then some more. Similar prohibitions are often placed on program structures: no jumps, no interrupts, no multiprogramming, no global variables. Access to data in other modules is permitted only through carefully regulated remote procedure calls. In the past, these design rules were found to involve excessive loss of efficiency; up to a factor of a hundred has been recorded on first trials of a rigorously structured system.

This factor had to be regained by relaxing the prohibitions, massaging the interfaces between modules, even to the extent of violating the structural integrity of the whole system. Apart from the obvious immediate dangers, this can lead to even greater risk and expense in subsequent updating and enhancing of the system. Fortunately, cheaper hardware reduces the concern for efficiency, and improved optimisation technology for higher level languages promises further assistance in reconciling a clear structure of the source code with high efficiency in the object code.

Profligacy of resources can bring benefits in other ways. When considering a possible exceptional case, the programmer may be quite confident that it has already been discriminated and dealt with elsewhere in some other piece of code; as a result in fact the exception can never arise at this point. Nevertheless, for safety, it is better to discriminate again, and write further code to deal with it. Most likely, the extra code will be totally unreachable. This may be part of the explanation why in normal testing and operation, less than twenty per cent of the code of a large system is ever executed; which suggests an overengineering factor of five. The extra cost in memory size may be low, but there is a high cost in designing, writing and maintaining so much redundant code. For example, there is the totally pointless exercise of designing coverage tests for this otherwise unreachable code.

Another profligate use of resources is by cloning of code. A new feature to be added to a large program can often be cheaply implemented by making a number of small changes to some piece of code that is already there. But this is felt to be risky: the existing code is perhaps used in ways that are not at all obvious by just looking at it, and any of these ways might be disrupted by the proposed change. So it seems safer to take an entirely fresh copy of the existing code, and modify that instead. Over a period of years there arise a whole family of such near-clones, extending over several generations. Each of them is a quick and efficient solution to an immediate problem; but over time they create additional problems of maintenance of the large volumes of code. For example, if a change is made in one version of the clone, it is quite difficult even to decide whether it should be propagated to the other versions, so it usually isn't. The expense arises when the same error or deficiency has to be detected and corrected again in the other versions.

Another widespread over-engineering practice is known as defensive programming. Each individual programmer or team erects a defensive barrier against errors and instabilities in the rest of the system. This may be nothing more than a private library of subroutines through which all calls are made to the untrusted features of a shared operating system. Or it may take the form of standard coding practices. For example, it is recommended in a distributed system to protect every communication with the environment, or with another program, by a timeout, which will be invoked if the external response is not sufficiently prompt. Conversely, every message accepted from the environment is subjected to rigorous dynamic checks of plausibility, and the slightest suspicion will cause the message to be just ignored, in the expectation that its sender is similarly protected by timeout.

A similar technique can be applied to the global data structures used to control the entire system. A number of checking programs, known as software audits, are written to conduct plausibility checks on all the records in the global system tables. In this case, suspicious entries are rendered harmless by a reinitialization to safe values. Such audits have been found to improve mean time between crashes of an embedded system from hours to months. The occasional loss of data and function is unnoticed in a telephone switching application: it could hardly be recommended for air traffic control, where it would certainly cause quite a different kind of crash.

The ultimate and very necessary defence of a real time system against arbitrary hardware error or operator error is the organisation of a rapid procedure for restarting the entire system. The goal of a restart is to restore the system to a valid state that was current some time in the recent past. These warm starts can be so efficient that they are hardly noticeable except by examining the historical system log. So who cares whether the trigger for a restart was a rare software fault or a transient hardware fault? Certainly, it would take far too long to record information that would permit them to be discriminated.

The limitation of over-engineering as a safety technique is that the extra weight and volume may begin to contribute to the very problem that it was intended to solve. No one knows how much of the volume of code of a large system is due to over-engineering, or how much this costs in terms of reliability. In general safety engineering, it is not unknown for catastrophes to be caused by the very measures that are introduced to avoid them.

6 Programming Methodology

Most of the measures described so far for achieving reliability of programs are the same as those which have proved to be equally effective in all engineering and industrial enterprises, from space travel to highway maintenance, from electronics to the brewing of beer. But the best general techniques of management, quality control, and safety engineering would be totally useless by themselves; they are only effective when there is a general understanding of the specific field of endeavour, and a common conceptual framework and terminology for discussion of the relationship between cause and effect, between action and consequence in that field. Perhaps initially, the understanding is based just on experience and intuition; but the goal of engineering research is to complement and sometimes replace these informal judgements by more systematic methods of calculation and optimisation based on scientific theory.

Research into programming methodology has a similar goal, to establish a conceptual framework and a theoretical basis to assist in systematic derivation and justification of every design decision by a rational and explicable train of reasoning. The primary method of research is to evaluate proposed reasoning methods by their formalisation as a collection of proof rules in some completely formal system. This permits definitive answers to the vital questions: is the reasoning valid? is it adequate to prove everything that is needed? and is it simpler than other equally valid and adequate alternatives? It is the provably positive answer to these simple questions that gives the essential scientific basis for a sound methodological recommendation—certainly an improvement on mere rhetoric, speculation, fashion, salesmanship, charlatanism or worse.

Research into programming methodology has already had dramatic effects on the way that people write programs today. One of the most spectacular successes occurred so long ago that it is now quite non-controversial. It is the almost universal adoption of the practice of structured programming, otherwise known as avoidance of jumps (or gotos). Millions of lines of code have now been written without them. But it was not always so. At one time, most programmers were proud of their skill in the use of jumps and labels. They regarded structured notations as unnatural and counter-intuitive, and took it as a challenge to write such complex networks of jumps that no structured notations could ever express them.

The decisive breakthrough in the adoption of structured programming by IBM was the publication of a simple result in pure programming theory, the Bohm-Jacopini theorem. This showed that an arbitrary program with jumps could be executed by an interpreter written without any jumps at all; so in principle any task whatsoever can be carried out by purely structured code. This theorem was needed to convince senior managers of the company that no harm would come from adopting structured programming as a company policy; and project managers needed it to protect themselves from having to show their programmers how to do it by rewriting every piece of complex spaghetti code that might be submitted. Instead the programmers were just instructed to find a way, secure in the knowledge that they always could. And after a while, they always did.

The advantages of structured programming seem obvious to those who are accustomed to it: programs become easy to write, to understand, and to modify. But there is also a good scientific explanation for this judgement. It is found by a formalisation of the methods needed to prove the correctness of the program with the aid of assertions. For structured programs, a straightforward proof always suffices. Jumps require a resort to a rather more complex technique of subsidiary deductions. Formalisation has been invaluable in giving objective support for a subjective judgement: and that is a contribution which is independent of any attempt to actually use the assertional proof rules in demonstrating the correctness of code.

Another triumph of theory has been widespread appreciation of the benefits of data types and strict type-checking of programs. A type defines the outer limits of the range of values for a program variable or parameter. The range of facilities for defining types is sufficiently restricted that a compiler can automatically check that no variable strays outside the limits imposed by its declared type. The repertoire of operations on the values of each type are defined by simple axioms similar to those which define the relevant branch of mathematics. Strict typechecking is certainly popular in universities, because of the help it gives in the teaching of programming to large classes of students with mixed abilities; it is even more widely beneficial in modern mass consumer languages like Visual Basic; and in very large programs which are subject to continuous change, it gives a vital assurance of global system integrity that no programmer on the project would wish to forego.

Another triumph of theoretical research has been widespread adoption of the principles of information hiding. An early example is found in the local variables of Algol 60. These are introduced by declaration and used as workspace for internal purposes of a block of code which constitutes the scope of the declaration; the variable name, its identity, and even its existence is totally concealed from outside. The concept of declaration and locality in a program was based on that of quantification and bound variables in predicate logic; and so are the proof methods for programs which contain them.

The information hiding introduced by the Algol 60 local variable was generalised to the design of larger-scale modules and classes of object-oriented programming, introduced into Algol 60 by Simula 67. Again, the scientific basis of the structure was explored by formalisation of the relevant proof techniques, involving an explicit invariant which links an abstract concept with its concrete representation as data in the store of a computer.

The value of a foundation in formal logic and mathematics is illustrated by the comparison of Algol 60 with the Cobol language, brought into existence and standardised at about the same time by the U.S. Department of Defence. Both languages had the highly commendable and explicit objective of making programs easier to understand. Cobol tried to do this by constructing a crude approximation to normal natural English, whereas Algol 60 tried to get closer to the language of mathematics. There is no doubt which was technically more successful: the ideas of Algol 60 have been adopted by many subsequent languages, including even Fortran 90. Cobol by comparison has turned out to be an evolutionary dead end.

7 Conclusion

This review of programming methodology reveals how much the best of current practice owes to the ideas and understanding gained by research which was completed more than twenty years ago. The existence of such a large gap between theory and practice is deplored by many, but I think quite wrongly. The gap is actually an extremely good sign of the maturity and good health of our discipline, and the only deplorable results are those that arise from failure to recognise it.

The proper response to the gap is to first congratulate the practitioners for their good sense. Except in the narrowest areas, and for the shortest possible periods of time, it would be crazy for industry to try to keep pace with the latest results of pure research. If the research fails, the industry fails with it; and if the research continues to succeed, the industry which is first to innovate runs the risk of being overtaken by competitors who reap the benefits of the later improvements. For these reasons, it would be grossly improper to recommend industry on immediate implementation of results of their own research that is still in progress. Indeed, Sir Richard Doll points out that scientists who give such advice not only damage their clients; they also lose that most precious of all attributes of good research, their scientific objectivity.

The theorists also should be accorded a full share of the congratulations; for it is they who have achieved research results that are twenty years ahead of the field of practice. It is not their failing but rather their duty to achieve and maintain such an uncomfortable lead, and to spread it over a broad front across a wide range of theories. No one can predict, with any certainty or accuracy of detail, the timescales of change in technology or in the marketplace. The duty of the researcher is not to predict the future more accurately than the businessman, but to prepare the basic understanding which may be needed to deal with the unexpected challenges of any possible future development. Provided that this goal has been met, no researcher should be blamed for failure of early predictions made to justify its original funding of the research. Mistakes made by businessmen and politicians are far more expensive.

The recognition of the appropriate timescale to measure the gap between the theory and practice of a discipline is an essential to the appropriate planning of research and education, both to fill the gap by improving practice, and to extend it again by advancing the theory. I would recommend that the best researchers in the field should simultaneously try to do both, because the influence of practice on the development of theory is more beneficial and actually quicker than the other way round.

At the extreme of the practical end, I would recommend the theorist to alternate theoretical pursuits with much closer observation and experimentation on actual working programs, with all the mass of documentation and historical development logs that have accumulated in the last ten years. These systems are now sufficiently stable, and have sufficient commercial prospects, to justify quite practical research to answer questions that will guide recommendations for future beneficial changes in their structure, content or methods of development.

For example, it would be very interesting to find a way of estimating the proportional cost of cloning and the other over-engineering practices. By sampling, it would be interesting to trace a number of errors to their root cause, and see how they might have been avoided, perhaps by better specification or by better documentation or by better structuring of code. Is my conjectured dichotomy of error populations observed in practice? Any recommendation for improved formalisation or improved structure will probably be based on other people's research ideas that are up to twenty years old. Even so, they must be backed up by trial recoding of a range of existing modules, selected on the scientific principle of being the most likely to reveal the fallacies in the recommendation, rather than its merits. Strange to relate, it has been known for a business to spend many millions on a change that has not been subjected to any prior scientific trials of this kind.

Formal methods researchers who are really keen on rigorous checking and proof should identify and concentrate on the most critical areas of a large software system, for example, synchronisation and mutual exclusion protocols, dynamic resource allocation, and reconfiguration strategies for recovery from partial system failure. It is known that these are areas where obscure time-dependent errors, deadlocks and livelocks (thrashing) can lurk untestable for many years, and then trigger a failure costing many millions. It is possible that proof methods and model checking are now sufficiently advanced that a good formal methodologist could occasionally detect such obscure latent errors before they occur in practice. Publication of such an achievement would be a major milestone in the acceptance of formal methods in solving the most critical problems of software reliability.

I have suggested that personal involvement in current practices and inspection of legacy code may lead to quite rapid benefits, both to the practitioner and to the theorist. But this is not the right permanent relationship between them; in a proper policy of technology transfer, it is for the practitioner to recognise promising results of research, and take over all the hard work of adapting them for widespread application. In software, unfortunately, the gap between practice and theory is now so large that this is not happening. Part of the trouble is that many or most of the practitioners did not study formal methods or even computing science at University. This leaves a large educational gap, that can only be filled by a programme of in-service education which will acquaint some of the best software engineers in industry with some of the important ideas of computing science. Since many of them have degrees in mathematics, or at least in some mathematical branch of science, they have the necessary background and ability: since they do not have degrees in computing, they need to start right at the beginning, for example, with context free languages and finite state machines, and simple ideas of types and functional programming.

Another high barrier to technology transfer is the failure of software engineering toolsets to include a modicum of support for formality—for example to allow mathematical notations in word processors, to incorporate typechecking for specifications, and hypertext techniques for quick cross-referencing between formal and informal documentation. Improved tools should concentrate first on very simple old techniques like execution profiles and selective compilation of assertions before going on to more advanced but less mature technology, such as model checking or proof assistance. The actual construction of industrial quality tools must be done in collaboration with the industrial suppliers of these tools. Only they have the knowledge and profit motive to adapt them, and to continue adapting them, to the rapidly changing fashions and needs of the marketplace.

For long-term research, my advice is even more tentative and controversial. It pursues a hope to complement the many strengths, and compensate the single weakness, of current theoretical research in formal methods. The strengths arise from the depth and the range of the specialisation of many flourishing research schools in all the relevant areas. For example, in programming language semantics, we have reasoning based on denotational, algebraic and operational presentations. Among programming paradigms, we have both theoretical studies and applications of functional, procedural, logical and parallel programming languages. Even among the parallel languages there is a great variation between those based on synchronous or asynchronous control, shared store or distributed message passing, untimed or with timing of various kinds; even hardware and software have different models.

Specialisation involves a deep commitment to a narrow selection of presentation, reasoning methods, paradigm, language and application area, or even a particular application. The whole point of the specialisation in formal methods is to restrict the notational framework as far as necessary to achieve some formal goal, but nevertheless to show that the restrictions do not prevent successful application to a surprisingly wide range of problems. This is the reason why specialist research into formal methods can run the risk of being very divisive. An individual researcher, or even a whole community of researchers, becomes wholly committed to a particular selection of specialisations along each of the axes: say an operational or an algebraic presentation of semantics, bisimulation or term rewriting as a proof method, CCS or OBJ as a design notation. The attraction of such a choice can be well illustrated in certain applications, such as the analysis of the alternating bit protocol or the definition of the stack as an abstract data type. The perfectly proper challenge of the research is to push outwards as far as possible the frontiers of the convenient application of the particular chosen formalism. But that is also the danger: the rush to colonise as much of the available territory can lead to imperialist claims that deny to other specialisms their right to existence. Any suggestion of variation of standard dogma is treated as akin to treason. This tendency can be reinforced by the short-sightedness of funding agencies~ which encourage exaggerated claims to the universal superiority of a single notation and technique.

The consequences of the fragmentation of research into rival schools is inevitable: the theorists become more and more isolated, both from each other and from the world of practice, where one thing is absolutely certain: that there is no single cure for all diseases. There is no single theory for all stages of the development of the software, or for all components even of a single application program. Ideas, concepts, methods, and calculations will have to be drawn from a wide range of theories, and they are going to have to work together consistently, with no risk of misunderstanding, inconsistency or error creeping in at the interfaces. One effective way to break formal barriers is for the best theorists to migrate regularly between the research schools, in the hope that results obtained in one research specialisation can be made useful in a manner acceptable by the other. The interworking of theories and paradigms can also be explored from the practical end by means of the case study, chosen as a simplified version of some typical application. In my view, a case study that constructs a link between two or more theories, used for different purposes at different levels of abstraction, will be more valuable than one which merely presents a single formalisation, in the hope that its merits, compared with rival formalisations, will be obvious. They usually are, but unfortunately only to the author.

Since theories will have to be unified in application, the best help that advanced research can give is to unify them in theory first. Fortunately, unification is something that theoretical research is very good at, and the way has been shown again and again in both science and mathematics. Examples from science include the discovery of the atomic theory of matter as a unified framework for all the varied elements and components of chemistry; similarly, the gravitational field assimilates the movement of the planets in the sky and cannon balls on earth. In mathematics, we see how topology unifies the study of continuity in all the forms encountered in geometry and analysis, how logic explains the valid methods of reasoning in all branches of mathematics. I would suggest the current strength of individual specialisation in theoretical computing science should be balanced by a commitment from the best and most experienced researchers to provide a framework in which all the specialisations can be seen as just aspects or variations of the same basic ideas. Then it will be clear how both existing and new specialisations are all equally worthy of effort to deepen the theory or broaden its application. But the aim is no longer to expand and colonise the whole space but rather to find the natural boundaries at which one theory can comfortably coexist and cooperate with its neighbours. Closing a gap between one theory and another is just as important as closing the gap between theory and practice; and just as challenging.

8 Acknowledgments

I am very grateful to many programmers and managers working in industry, who have made available to me the benefits of their judgment and long experience. In particular, I would like to praise the leading practitioners of the state of the art in IBM at Hursley, in BNK at Maidenhead, and in Digital at Nashua. Many contributions to my thinking are due to members of IFIP WG2.3 on Programming Methodology, and to its chairman Cliff Jones, who made useful suggestions to the previous draft of the paper. Finally, thanks to those named in the paper, with apologies for lack of more formal reference.