What can we learn from Edsger W. Dijkstra?

by Tony Hoare.

0. Preamble

I would like to start by joining all of you in the expression of deep gratitude to Schlumberger for the grant which set up a series of lectures to honour the memory of Edsger W. Dijkstra. My next thanks go to the University of Texas at Austin, for inviting me to deliver the first lecture in the series. For forty years Edsger W. Dijkstra was my close professional colleague and my close personal friend. The goals and direction of both my academic and my industrial career as a scientist, have been inspired not only by his teaching but also by his personal example. So I will drop the formalities, and even I will drop his family name and his middle initial W (standing for Wybe), of which he confessed himself to be inordinately fond. I will refer to him simply, as I did in life, as Edsger.

Edsger spent the latest and longest and happiest period of his career as a scientist, with the title of Schlumberger Chair of Computer Sciences at the University of Texas at Austin. He spent the earliest and incredibly productive part of his career as a software engineer at the Mathematisch Centre in Amsterdam. The two main topics of my lecture will be Science and Engineering, and my examples will be drawn from Computer Science and Software Engineering. I will first contrast their many striking differences in culture that divide scientists from engineers, and then describe the immense benefits to both of them that arise when they work together in harmony. First I will describe the branch of Computer Science to which Edsger and I devoted our main attention. It takes computer programs in general, rather than particular applications, as the subject of its study. Next, I will describe the qualities which Edsger most strongly recommended and most conspicuously displayed as a scientist. Then I will turn to engineering, with a survey of his early achievements in the construction of operating systems and compilers. Edsger always believed that the concerns of the scientist are best separated from those of an engineer, and I will explain many reasons why this should be. But in the end, scientists and engineers derive the greatest benefit from working together with each other, and with the industries that can benefit from the advances in their research. In the last part of my lecture, I will look forward to the prospects of exploiting and bringing into widespread application many of the inspirational scientific concepts and laws discovered by Edsger as a

scientist, for the benefit of the computer hardware industry, the software industry, and all their millions of customers.

1. Science

Edsger studied theoretical physics at the University of Leiden. This gave him a strong taste for the goals and ideals and practices of a mature branch of pure science. On graduation, he made a considered decision to move to the new science of computing. After a Doctoral thesis on the architecture of the X1 computer, he rapidly realised that “Computer Science is no more about computers than astronomy is about telescopes”. It is computer software, the programs that control the computer, that he made the subject of his lifelong scientific study. And in this I was his follower. I followed him in the belief that we were in fact engaged in a branch of science. So I will start my presentation by summarising the essential similarity of computer science with those longer established branches of science, whose history stretches back to the Greek Philosophers, particularly Aristotle in the fourth century bce. They include Physics, Medicine, and Biology. The development and maturation of a new branch of science can be described as a series of four steps, consisting of description, analysis, explanation, and finally prediction.

1.1. Description. The earliest goal of a new branch of science is description. Whatever the subject matter of the new science, whether it is a natural occurring object like an ant or an antelope or an artificial product like an automobile or an aeroplane, the first questions addressed by the scientist are ‘What are its properties, and what does it do?’ For artificial products, a description of properties and behaviour can serve as a specification, describing an appropriately precise interface between its purchaser and its supplier.

Scientific description often requires the development of a conceptual framework and terminology different from that of natural language. Mathematics often provides the inspiration, though sometimes the relevant branch of mathematics needs to be further developed in the direction of its application. In the case of programs in general, a highly relevant branch of mathematics is logic. Logic has long been studied and extended by philosophers, including Aristotle, Occam, Leibniz. More recently, it has been significantly extended by philosophers of mathematics, like Boole, Frege and Russel. Logic has now found extensive application to programming and programming languages; and again it has been significantly developed in its new application. A good example is the logical calculus of weakest

preconditions, which we owe to Edsger. I expect that many of these extensions will find their way back to the core of logic.

1.2 Analysis. As a branch of science matures, its questions become deeper. We move from a purely descriptive and classificatory phase to one that seeks more convincing explanations. We want to know not only what the thing does, but how it does it. A scientific answer to this question is often provided by dissection, which divides the object into parts, and examines the ways in which the parts interact with each other. In the case of an engineering product, the interactions are specified in an interface, which can serve as a contract between engineers designing and constructing components to meet the same interface from both sides.

In software engineering, the interfaces are specified as assertions, which also serve as contracts. For example, an assertion written before a program component (a precondition) describes what the designer of the component may assume to be true before execution of the component. Similarly, an assertion written after a program component (its postcondition) expresses an obligation in the design of the component itself — to make it true after execution. This use of assertions as guide to design has been illustrated in many elegant and efficient algorithms, developed and described by Edsger.

The principle of top-down design is now well established in Software Engineering practice. Assertions have been included in most standard computer programming languages. They are widely used as oracles to assist in program testing, to check whether a test run has gone wrong, and if so, to indicate as closely as possible exactly where in the program the error has occurred. In Microsoft, assertions have been renamed as contracts, to emphasise their potential role in the definition of interfaces at the design stage of the product.

1.3. Explanation. The next question probes even more deeply than the last. It asks for an explanation why the program works. Why is it that any assembly of components, conforming individually to these particular internal interface descriptions, will in combination evince the behaviour described by the external specification? That is the question addressed by the theory of programming. The answer to the question is codified in one or more presentations of the semantics for a programming language. The semantics acts as an interface between the user of the language and its implementer.

Although the search for valid explanation is the province of pure scientific research, it is the explanation discovered by this pure research that is most

helpful to practicing engineers. It helps them to determine good boundaries at which to define the interfaces between components. It permits them to replace components by better or cheaper ones that conform to the same interface. It gives them confidence that the experiments on a structurally similar prototype will give results applicable to the eventually manufactured product.

1.4. Prediction. My fourth and final question is often taken as the criterion of maturity of a branch of science, that it can make accurate predictions of natural phenomena and experimental observations. The calculations required to make successful predictions in science are based on mathematics. In the computer era, they are usually carried out by a computer program, which has been based on the relevant scientific theory. It is the consistent and repeated success of prediction that accumulates convincing evidence of the validity of the scientific theory on which the prediction is based. If the same theory predicts phenomena of many apparently different kinds, the evidence becomes overwhelming. No-one can doubt a theory of gravitation that applies equally to falling apples, flying cannon-balls, and planets wandering in the sky.

In the case of computer programs, the checking of conformity to an external specification and to internal interfaces is a matter of proof. For the engineer, proofs predict the properties of a computer program even before it runs. The details of the proof are these days carried out by computer, using the fascinating technologies of constraint satisfaction and model checking, which are improving at a phenomenal rate. It has been this progress that has brought the science of computing, and Edsger's contributions to it, much closer to application in the large-scale engineering of software as well as computer hardware, as I will describe in the final section of my talk.

That concludes my brief survey of four stages in the development of a mature branch of science. It appears that Computer Science has passed through the first three of them, and is making rapid progress in the fourth.

Edsger was fortunate in his opportunity to contribute at its very start to the new science of computing. He was blessed with the scientist's gift for formulation of fundamental questions, and for finding elegant answers to them. He pursued the scientific ideal of total correctness of computer programs, with the same rigour and the same vigour as a physicist pursues the ideal of accuracy of measurement, and as a chemist seeks absolute purity of the materials that they refine and use. He recommended (in the title ofEWD 288, July 1970) the "concern for correctness as a guiding principle for program construction." Like Newton, (EWD 273, 1969) he recognized "for the human mind the mathematical method is indeed the most effective way to

come to grips with complexity” . His constant advice was to follow the general scientific maxim of separation of concerns. For example (EWD 273) he tells us to “separate for each program component clearly “what it does” from “how it works””. And finally, (EWD 709, 1979) “separation of concerns for efficiency and correctness—to get rid of our operational thinking habits—that is what I regard as Computer Science’s major task.”

To these important scientific principles he added a complete refusal to play the part of a salesman. He wrote up his work in a clear and scholarly style, and he left it to the reader to decide whether it had any value. He never compared the work of others unfavourably to his own. He rarely submitted his work to conferences or to Journals for publication. Instead, he revived the early tradition of modern science, that of sending his work in a neat manuscript to other scientists whose opinion he respected. The only concession to modernity was the use of a photocopier as a method of reproduction, to reach a wider circle of friends.

He certainly looked forward to the time when the results of his work would be widely exploited by engineers in industry: (EWD 209, 1967) “many debugging aids that are in vogue now are invented as a compensation for the shortcomings of a programming technique that will be denounced as obsolete in the near future”. But he never tried to sell his results for immediate application. Because to the true scientist, his most precious virtue is that of scientific integrity, and his most precious emotion is that of scientific doubt. Both of these qualities would be grave impediments to the success of a salesman. The qualities of a salesman are equally incompatible with those of a scientist.

The idealistic view that I have put forward of computing as a science is taken as the basis of policy by my current employers. The policy was laid down fifteen years ago on the foundation of Microsoft Research Division by Bill Gates and Nathan Myhrvold. The policy is to recruit the best scientists in the world, provided that they are passionate in their commitment to an area of scientific research that might conceivably be relevant, some time in the future, to the software industry. Recruits should also welcome the opportunity that Microsoft provides, to bring the eventual results of their research to fruition, in applications of long-term beneficial impact on the welfare and prosperity of a significant fraction of mankind. Having chosen a promising recruit, Microsoft would then trust him or her to do excellent research to the advancement of science. No need for research proposals in advance. No need to set timescales and deadlines. No external evaluation of success or failure. The Company trusted that the individual researcher would report to the Company

when he or she believed that the research results were sufficiently mature to be incorporated into Microsoft's products or practices. At this stage, many of the researchers would personally assist, engage or even manage the transfer of technology, perhaps moving to a development division for a few years, before moving back into the next cycle of research.

I heard Nathan Myhrvold give a lecture in Cambridge on this policy in December 1998. It was the decisive factor in my decision to take up Roger Needham's offer of a job with Microsoft Research and move to Cambridge in 1999, on retirement from Oxford University. My experience in the last ten years has amply demonstrated Microsoft's continued commitment to its policy; it has also demonstrated an extraordinary degree of success in improving the quality and extending the range of Microsoft products.

2. Engineering.

I would like now to move on from the discussion of pure science to deal with applied science and engineering. I will make a case that the activity and culture of engineering is in polar antithesis to that of the science, along almost every axis of evaluation. Engineers and Scientists engage in projects spanning different timescales, they pursue different goals and ideals, they require different levels of evidence for their judgements, they have opposite requirements for generality or specificity of knowledge, and they place opposite emphases on separation or amalgamation of concerns. They have opposite requirements for originality or conformity, and they strike a different balance between mathematical formalisation and intuitive judgement based on experience.

This polarisation in no way prevents a brilliant scientist from being also a brilliant engineer. This is not in spite of but actually because of the polarisation. Indeed, many spectacular breakthroughs in both science and engineering have resulted from a personal move from science to engineering or vice versa. Edsger himself provides an excellent example. In his earlier years at the Mathematisch Centrum and later at the Technical University at Eindhoven, he was a remarkably successful Software Engineer.

He wrote and delivered the very first compiler for the new international algorithmic language ALGOL 60. He devised the algorithms which were necessary in the implementation of a complete version of the language, and they were later widely copied. Incidentally, he invented and first practiced the technique of 'pair programming', now beloved of the advocates of Extreme Programming. His colleague Jaap Zonneveld sat at the opposite side of his

working table, and every instruction in the compiler was only written (by both of them) when they had discussed and agreed that it was correct. In the evening each of them took his own copy of the code back to his separate home, to protect against the danger of fire.

His next achievement was to lead a team of research students in the design and implementation of the THE operating system, for use in the Technical University of Eindhoven. It ran on the Electrologica X8 computer, having 48K of 27-bit words of core memory and, as a backing store, a drum of 512K words. On this it implemented a purely software virtual memory, spooling, batch processing and multiprogramming. It was the first operating system built in a rational design process, based on explicit separation of concerns in a logical hierarchy of abstraction and implementation.

In the early 1970's I visited Edsger in Eindhoven to study this system in detail. He organised a demonstration of the system in operation under stress. He constructed a paper tape loop, whose input would fill the spooling buffers. He wrote an infinitely recursive program, whose stack would fill the drum supporting virtual memory. On occasion, the paper tape reader would stop, with a message on the console that warned of full buffers. The operator could override the warning. On occasion, another warning was displayed, identifying one of the four programs in the mix that was exceeding its storage bounds. This message too could be over-ridden. Eventually, the whole system ground to a halt, and printed out a brief message that the store was full, and the operating system should be reloaded. Edsger confessed that this was the first time that the final error message had occurred: it had not even been tested. Nevertheless, the text of the message had been stored permanently in the extremely precious main memory; it would only ever be needed when there was no room for it to be retrieved from drum! The story illustrates the concern of the good engineer to cater adequately for all eventualities, including those that the good scientist would much sooner, and very properly, choose to abstract from.

Nevertheless, I believe that Edsger's subsequent success as a scientist owes much to his early experience as a software engineer. It was as an engineer that he discovered deadlocks, race conditions, non-determinacy and infinite overtaking. It was as an engineer that he invented semaphores, producers and consumers and mutual exclusion. But it was as a scientist that he was able to extract the essence of each of these phenomena, and teach us all how to recognise and control the associated problems by the application of reason.

Edsger repeatedly advised that all researchers should always know their own “Buxton Index”, and that of their colleagues and their rivals. It was named after the British Computer Scientist John Buxton. It is “…the most relevant one-dimensional scale along which to place and compare individuals, organizations, industries, and movements. [It measures (in years)] how far in the future their planning extends…Cooperation between persons or groups with very different Buxton Indices leads to mutual (moral) reproaches as long as the partners are unaware of the difference: the partner with the small Buxton Index is accused of short-sightedness and opportunism, the one with the large Buxton Index is accused of hobbyism and a lacking sense of responsibility.(EWD988, 1986).”

I believe that differences between scientists and engineers can be recognised not just on the single numerical value of the Buxton index. It is necessary to consider in all at least seven criteria, including the Buxton index, under which the culture and practices of scientists and engineers can be differentiated. I would like each of you in my audience to think about each criterion, whether you have more affinity to the culture of science or of engineering; or whether you would like to move from your current position in one direction or another. For brevity, I will describe the case when the scientist is a man and the engineer is a woman. I will later suggest that they will both work better in close but not necessarily intimate partnership.

Timescale. The first major difference between scientists and engineers lies in the time-scales of their plans and ambitions. The scientist works for posterity. He wants to discover the absolute truth about the natural world, however long it takes, and however much it costs. Truth is permanent, and the knowledge of truth will never lose its literally inestimable advantage over the alternative. Even knowledge that is nowhere applied has more value than ignorance that is applied widely.

The engineer works for a known client, or for a known or speculative niche in an existing market. She is committed to a fixed budget and a fixed delivery date for the product of her labours. The service life of an engineering product is known to be limited, and so is the market life of a series of products, improving incrementally over time. In compensation for the shorter timescale, the engineer gains recognition sooner than the scientist, in the form of a grateful client or a commercially profitable product. And the recognition may well be financial as well.

Idealism. The scientist pursues absolute and objective scientific ideals. For example, the physicist pursues an ideal of accuracy of measurement; the

chemist constantly seeks to improve the purity of materials. If something can be measured to 99.99 percent accuracy, the physicist wants to add another nine to the fraction. If materials of 99.99 percent purity are obtainable, the chemist also wants to add another 9.

The scientist pursues his ideals for their own sake. He pursues them far beyond the current needs of the market place, or even its future projected needs. Scientists are not discouraged when the fundamental scientific laws of their own science restrict the ultimate limits of their achievement of an ideal: they just try to get ever closer and closer to that theoretical limit.

In computer science, Edsger was the first to formulate and pursue the ideal of total correctness of computer programs, achieved by mathematical proofs conducted during their design and implementation. Results of incompatibility and undecidability place a limit on what can be achieved: but there is still a lot of progress to be made before we get close to that limit.

The engineer has no motivation to pursue ideals. Her main duty is to fully understand the full requirements of her clients and customers. She has to understand their many conflicting needs, and seek an acceptable and feasible compromise between them. She develops her ingenuity to deliver on time and within budget a product that is only just good enough for present (or at least clearly foreseeable) requirements.

Certainty. The scientist seeks the highest degree of certainty, backed up by an overwhelming mass of convincing experimental evidence. For example, physicists will continue to devise new ways of testing Einstein's theory of relativity, long after its correctness has been put beyond doubt—even unreasonable doubt.

The engineer has no time to pursue certainty as an end in itself. She develops confidence to live with the innumerable uncertainties of the real world, and finds ways of managing the inevitable risks involved.

Generality. The scientist seeks the most general theories, those that apply, possibly with different parameters, to the widest possible range of phenomena, both natural and artificial. This often requires introduction of new and abstract mathematical concepts, in order to show that a range of existing scientific laws can be seen as special cases of some more general unifying law. But the reward is that the unified law inherits all the evidence that has been accumulated in support of the more specific theories which it subsumes. And because they are special cases, each of the existing laws benefits from all the increased credibility of the general law.

Because of the high level of abstraction, a unified theory is often of less practical use than the specific theories which it subsumes. So the engineer has little truck with generality. Her duty is to study all the particularities of a specific environment, a specific customer, or a specific market place. She exploits specific solutions to all those particular and peculiar problems, for which the generality of science can offer no solution. She understands in their unavoidable diversity the whole range of specific scientific theories that are directly relevant to different aspects of her current project.

Separation of concerns. The scientist separates concerns. He designs his laboratory experiments in an idealised environment, to isolate them as far as possible from noise and dust and disturbances of the real world. He isolates all extraneous factors and carefully controls the relevant ones.

The engineer has to bring together an almost unbounded collection of relevant concerns, and tries to resolve them all simultaneously; none of them can be controlled, none can be ignored, and only rarely can they be elegantly simplified. She must not be bedevilled by the excruciating details of incompatible software platforms, user input errors, and standards for security. She therefore learns to live with the imperfections of the real world, and develops her ingenuity to avoid the problems, to work around them, and in the last resort, to persuade her customer to live with them. For the engineer, ‘good enough’ must always be good enough.

Originality. The scientist insists on originality in pursuit of new results, or corrections and refinements to established theories. In writing up the results, all sources must be scrupulously acknowledged, and plagiarism is punished by exclusion from the scientific community.

For the engineer, originality is to be avoided wherever possible. She is well aware of all current best practices and standards, and she relies on tried and true methods wherever applicable. If her project fails through an unnecessary attempt at innovation, she is legally liable to a civil suit for damages, and she may even be excluded from practice of her profession.

Formalisation. The scientist expresses scientific knowledge in precise mathematical formulae, often surprisingly elegant—“well worth a thousand pictures” he would say. The consequences of a collection of formulae can then be deduced by mathematical calculations or even proof. Calculations and proofs are often messy and laborious, but fortunately they are nowadays performed by scientific software, based on the relevant scientific theories. I will return to this point later.

The engineer has to deal with a multitude of specific problems, many of which are not formalisable, or not sufficiently repeatable to permit the experimentation necessary to the progress of science. To solve these problems, the engineer has to rely on common sense, backed up by good judgment and engineering intuition. Intuition is based on her long experience of the relevant technologies, and of the real world environments in which the technologies are to be applied. Her experience is developed and re-inforced by case studies and stories recounted by her teachers and colleagues.

3. Synthesis.

That is the end of my long discourse on the wide and multi-dimensional gap which separates the culture of science and engineering. I have been guilty of gross exaggeration of the antithesis between them. I now want to describe synthesis of the two extremes, and show how their complementary qualities can be exploited to their mutual benefit. The agent of this synthesis in the last twenty years has been the development and use of powerful software tools by both engineers and scientists.

Scientists now write computer programs which embody established or conjectured scientific theories; the software checks the plausibility of the theory by simulation of the relevant phenomena, and predicts in detail the results of future experiments. When the necessary real-world experiments have produced their results, software is essential to check conformity with a highly complex theory to the desired degree of accuracy. Finally, software is needed to manage the staggering volumes of experimental data that are produced by the larger-scale experimental equipment like telescopes, satellites, and particle accelerators. This data is made available for exploitation by other scientists throughout the world.

Engineers need software for similar purposes: to capture their earliest designs for a product, to simulate and predict its properties and behaviour when manufactured, and to optimize and check the relevant parameters against established scientific theory. Modern cars and airplanes (and even computers) are designed almost wholly inside a computer, and the role of physical prototyping and testing has been radically reduced.

Large and highly evolved suites of design automation software are now a major channel for technology transfer between scientists and engineers. Each new advance in science is incorporated as an improvement to an existing tool, which is instantly propagated to every engineer who uses the tool. The failure of any product produced or checked by the tool will lead to correction of the

tool, and thereby inhibit repetition of the failure by others. In some cases, the failure will be treated as a refutation of the theory embodied in the tool, and lead to its abandonment. Because of this possibility of refutation, the success of the engineering product, like success of a scientific experiment, will confirm confidence in the validity of the scientific theories embodied in the tool.

In summary, software provides a synthesis for the antithetical cultures and practices of science and engineering; and it is to their immense mutual benefit. The advantages are multiplied in the presence of good feedback between the producers and users of a tool. Indeed, progress in both science and engineering, for a limited period, can be given an exponential boost, such as can be seen in amazing progress of modern genetics and biology around the turn of this century. Over the same period, similar rates of advance have been observed in more traditional branches of science—for example, in physics and in astronomy.

My prediction (or perhaps rather my hope) is that Computer Science itself will benefit from a similar boost, by the incorporation of verification technology in generally used software tools, and their widespread use both by scientists and engineers. What is the evidence for this prediction, or grounds for this hope? First, there is evidence provided by analogy of the recent adoption of proof technology, including model checking, within the electronics industry. I will take examples from developments which took place here in Austin Texas during and since my sabbatical visit in 1986/87. My other main source of examples in software comes from my experience (2000-2010) at Microsoft, where the software development divisions now routinely use productivity tools which incorporate logical calculation and proof. The tools were developed by the Research Division, on the basis of the scientific results of forty years of University research by Computer Scientists. And in its turn, this research was largely inspired by Edsger.

3.1 Hardware

My first example of the application of proof to computer chips was made at Computational Logic Inc, a spin-off of the University of Texas in Austin. It was part of the ambitious STACK project which used the previous version of the ACL2 proof tool to prove correctness of software: an assembler, a compiler, and a verification condition generator. But the base of the stack was the top-down design of a complete microprocessor, which was thereby proved correct by construction. An early industrial application of proof was provided by Inmos, the UK chip manufacturer, in collaboration with Bill Roscoe at Oxford University. His team built and used a simple tool to check the design of the

floating point hardware of the transputer, thereby obviating the need for soak-testing before first delivery. An early application of model checking technology, for which Alan Emerson shared a recent Turing Award, was made by Carl Pixley at the MCC research corporation in Austin. When he demonstrated his tool on a significant example, its speed excited the amazement of an engineer from Motorola, who expected him to go on to demonstrate it on a different set of input data. He was even more amazed when Carl assured him that this was unnecessary: the single run had checked all possible input data.

The technology transfer of verification technology into hardware design was triggered by a single event: the infamous Intel floating point division bug, which hit the Company where it hurts most—in its share price. Proof technology is now incorporated into the design automation toolset of major chip manufacturers like Intel. Many of them use ACL2, and other academically developed proof tools. This morning, Warren Hunt described to me the way that an ACL2 prover has been adapted for routine used in Centaur, a chip design Company located in Austin, Texas. ACL2 has proved correctness of the Company's recent design of a number of floating point, integer, and logical operations of the X86 architecture. Every night, the entire half million lines of their latest Verilog design is translated to logical form and fed into a new copy of ACL2, ready for use by designers the next morning. Each night they also re-run many of the verifications that have been done previously, to make sure that recent changes are safe. Each week, they attempt to re-run the entire regression suite of previously proven results. The Company is likely to continue to expand the use of this technology, because it is faster, cheaper and more thorough than non-exhaustive simulation.

3.2 Software

The gradual infiltration of proof technology into Microsoft programming practice has followed a course similar to that of hardware verification. The first success story was the PREfix program analysis tool, used first to analyse the complete code base of Windows Server in 2003. It detected around ten percent of the errors corrected in the final integration tests before release. A necessary condition of its adoption was that it did not require any annotation of the program. It therefore had to concentrate on generic errors like subscript overflow, uninitialized variables and null dereferences.

In the ideal, any such operation that cannot be proved correct should be brought to the programmer's attention as a possible error. However, the number of such warnings was initially far too high. The cost in time for

checking that a warning is unjustified (a so-called false positive) is actually greater than that of merely determining that the warning is true, and simply correcting it.

Heuristic filters were therefore developed by experimental application of the tool to the specific patterns of Microsoft software; and this brought the number of false positive reports down to an acceptable level, somewhere round fifty percent. However it introduced the danger of masking real errors; failure to report an error is known as a false negatives. The PREfix tool was shortly joined by a faster version (PREfast) adapted for unit testing; and it was extended by an assertion language SAL, to permit more precise analyses.

In the year 2000, a survey of enterprise customers revealed that a leading cause of server crashes (blue screens) were errors in device drivers, most of which are not written by Microsoft programmers, and are actually never seen by Microsoft. The SLAM project was undertaken to address the problem. It inferred many of the assertions needed for correctness checking by a new technique of counter-example-guided abstraction refinement. SLAM concentrated on detecting the violation of sequencing constraints on calls to the Microsoft Driver Kernel; and the availability of a checker motivated the documentation of many previously unrecorded rules of correct usage of the kernel API. For each constraint, SLAM was able to check whether a driver met the constraint or did not. These reports were reliable, containing essentially no false positives or false negatives. But there was a third report, less useful and fortunately rarer, which was given when the checker ran out of time. The SLAM technology is now incorporated as part of Microsoft's Static Driver Verifier SDV. It is distributed as a tool in Visual Studio, the software development toolkit; for use by all manufacturers of devices that can be attached to PCs.

The real trigger that motivated accelerated transfer of program proof technology within Microsoft was one that could never have been predicted by any of the researchers into the technology. It was the computer virus, worm, or bot, generally now classed as malware. For example, the infamous Code Red worm in 2001 brought the commercial and financial networks of the whole world to a standstill for several days. The estimated cost to the world economy was around four billion dollars.

The most easily exploitable vulnerability of PC software, whether written by Microsoft or by independent software vendors, was buffer overflow, occurring for example on call of the C standard input routines. It is a particular case of subscript error. Since the designers of malware have access to the whole

Microsoft code base, complete elimination of this vulnerability was adopted as target. False negatives had to be vanishingly unlikely for this particular error, so that even malware writers would stop looking for them. That was the goal for development and widespread use of a new tool called ESPX. which aimed to eliminate false negatives in the report of subscript overflow. To support the necessary proof a specialised assertion language was designed, and software developers, with the aid of an inference tool, were made responsible for including them.

A dramatic reduction in false positives is the goal of more recent tools like PEX and SAGE, developed by Microsoft Research. The method is to generate test cases that reach every point in the code which cannot be proved correct. So every error report would include a test case which revealed the error. For this, and many other fascinating tools, please consult

http://research.microsoft.com/en-us/groups/rise

My final example is an experimental project to prove correctness against a full and formal specification of an important item of Microsoft software, the separation kernel of the Microsoft Hypervisor. The kernel is a C program of some hundred thousand lines of code, with five thousand lines of assembly. It uses pointers and threads and volatile variables, together with ingenious data representations and algorithms, to achieve high efficiency of interpretation of guest operating systems on the X86 architecture. The project was conducted in collaboration with the University at Saarbrucken, and with significant participation by mathematical engineers from Russia. It used a new verifying compiler for Concurrent C, known as VCC. The proof strategy was designed by Ernie Cohen, a graduate from UT Austin whom I met in 1986. The project lasted three years, and produced a machine-checked verification of some thirty thousand lines of code, sufficient evidence for the feasibility of extending the proof to the complete program.

The project revealed a significant problem. It required more than one line of annotation for every line of code. So although the project met its original scientific goal as a feasibility study, its continuation was not justifiable on commercial grounds. A similar fully completed project in Australia to verify an L4 kernel suggests that proof of a shared-memory concurrent program generates ten times more verification conditions than a conventional sequential program. Clearly more research is needed into the science that underlies concurrency, together with improvements in the tool technology, if the machine-checked proof of operating systems is to become a standard precaution. The necessary breakthrough is more likely to result from

academic research, while its gradual scaling up and scaling out to widespread application will best be left to Industry.

Envoi

I predict that the simple commercial pressure of saving money and making profits will lead to yet further continuous advances in the application of proofs to programs in the Software Industry. In advancing the technology, Microsoft and other software suppliers have one great advantage over academic researchers: a vast supply of stable software to use in experiments, and to guide the improvements of the necessary tool in a direction most profitable to the Company. In most cases, this will be in the analysis of existing software, and the detection of errors introduced by modifying it. Academic researchers will be welcome to assist and advise in this enterprise, but the more idealistic researchers will avoid too much exposure to short-term industrial goals. As Edsger has taught us, the goal of making profits for a commercial Company is irrelevant to academic research. It is the more idealistic academic researchers, by concentrating on what they do best, who will make the breakthrough that leads to a step change in the rate of advance.

As I have described, Edsger's view is that the goal of academic research is the advancement of scientific knowledge: our understanding of what programs do, how they do it, why they work, and how to obtain convincing evidence for the answers to all the above questions. It still remains a major scientific challenge for academic research to demonstrate that the technology of correctness proofs will apply to the kind of programs which are widely used today; that they can cover a wider range of desirable properties, right up to total correctness; and above all, that they can be integrated into the design processes, environments and tools of the software engineer, and so get ever closer to Edsger's ideal of correctness of software by construction.

---

revised Thu, 25 Nov 2010