When the idea to write about the next fifty years of computing first entered my head, I wrote it off as utterly preposterous: what sane scientist purports to be able to see so far into the future? But then I realized that in a way that is precisely what educators do all the time: when designing our courses, we do dare to decide what to teach and what to ignore, and we do this for the benefit of students, many of whom will still be active forty to fifty years from now. Clearly some vision of the next half century of computing science is operational. To this I should add that it is all right if the crystal ball is too foggy to show much detail. Thirty-five years ago, for instance, I had no inkling of how closely program design and proof design would come together, and in such a detailed sense my life has been full of surprises. At the same time these surprises were developments I had been waiting for, because I knew that programming had to be made amenable to some sort of mathematical treatment long before I knew what type of mathematics that would turn out to be. In other words, when building sand castles on the beach, we can ignore the waves but should watch the tide.

* * *

Fortunately, there are a few things that we can learn from the past, for instance that the rate at which society can absorb progress is strictly limited, a circumstance that makes long-range predictions a lot easier. Earlier this decade I was told of a great invention called "the register window"; my informant was very young, but in my ears the invention sounded very familiar because I remembered the

Burroughs B5000 of 30 years before. So, if you have a bright and sound idea now, you can expect it to be hailed as a novelty around the year 2025.

Another thing we can learn from the past is the failure of characterizations like "Computing Science is really nothing but X", where for "X" you may substitute your favourite discipline, such as numerical analysis, electrical engineering, automata theory, queuing theory, lambda calculus, discrete mathematics or proof theory. I mention this because of the current trend to equate computing science with constructive type theory or with category theory.

Computing's core challenge is how not to make a mess of it. If people object that any science has to meet that challenge, we should give a double rebuttal. Firstly, machines are so fast and storage capacities are so huge that we face orders of magnitude more room for confusion, the propagation and diffusion of which are easily inadvertently mechanized. Secondly, because we are dealing with artefacts, all unmastered complexity is of our own making; there is no one else to blame and so we had better learn how not to introduce the complexity in the first place.

In this connection the history of the real-time interrupt is illuminating. This was an invention from the second half of the 50s, which enabled the completion of a communication with the external world to interrupt the execution of one program in favour of another. Its advantage was that it enabled the implementation of rapid reaction to changed external circumstances without paying the price of a lot of processor time lost in unproductive waiting. The disadvantage was that the operating system had to ensure correct execution of the various computations despite the unpredictability of the moments at which the interrupts would take place and the central processor would be switched from one computation to another; the nondeterminism implied by this unpredictability has caused endless headaches for those operating system designers that did not know how to cope with it. We have seen two reactions to the challenge of this added complexity.

The one reaction was to enhance the debugging facilities, as IBM did for the design of OS/360. (This was the operating system IBM tried to design for its 360-Series of machines, which were introduced in the first half of the 60s; IBM's problems with this design facilitated in 1968 the recognition of the world-wide phenomenon that became known as "the software crisis".) IBM built, in fact, special-purpose monitors that exactly recorded when the central processor honoured which interrupt; when something had gone wrong, the monitor could be turned into a controller, thus forcing a replay of the suspect history and making the "experiment" repeatable.

The other reaction could be observed at the THE (Technological University Eindhoven), viz. to determine the conditions under which one could feasibly and safely reason about such nondeterministic programs and subsequently to see to it that these conditions were met by hardware and software.

The difference was striking, showing once more that debugging is no alternative for intellectual control. While OS/360 remained a mess forever after, the Multiprogramming System designed at the THE was so robust that no system malfunction ever gave rise to a spurious call for hardware maintenance. Needless to say, the whole episode has made a lasting impression on me.

One moral is that the real-time interrupt was only the wave, whereas the tide was the introduction of nondeterminism and the development of the mathematical techniques to cope with it. A wider moral is the constructive approach to the problem of program correctness, to which we can now add the problem of system performance as well. It is only too easy to design resource-sharing systems with such intertwined allocation strategies that no amount of applied queuing theory will prevent most unpleasant performance surprises from emerging. The designer who counts performance predictability among his responsibilities tends to come up with designs that need no queuing theory at all. A last, and this time fairly recent, example is the design of delay-insensitive circuitry, which delegates all timing difficulties in clocked systems to the class of problems better avoided than solved. The moral is clear: prevention is better than cure, in particular if the illness is unmastered complexity, for which no cure exists.

The above examples point to a very general opportunity, in broad terms to be described as designs such that both the final product and the design process reflect a theory that suffices to prevent a combinatorial explosion of complexity from creeping in. There are many reasons to suppose that this opportunity will stay with us for a very long time, and that is great for the future of computing science because, all through history, simplifications have had a much greater long-range scientific impact than individual feats of ingenuity.

The opportunity for simplification is very encouraging, because in all examples that come to mind the simple and elegant systems tend to be easier and faster to design and get right, more efficient in execution and much more reliable than the more contrived contraptions that have to be debugged into some degree of acceptability. (One of the reasons why the expression "software industry" can be so misleading is that a major analogy with manufacturing fails to hold: in software, it is often the poor quality of the "product" that makes it so expensive to make! In programming, nothing is cheaper than not introducing the bugs in the first place.) The world being what it is, I also expect this opportunity to stay with us for decades to come. Firstly, simplicity and elegance are unpopular because they require hard work and discipline to achieve and education to be appreciated. Secondly we observe massive investments in efforts that are heading in the opposite direction. I am thinking about so-called design aids such as circuit simulators, protocol verifiers, algorithm animators, graphical aids for the hardware designers, and elaborate systems for version control: by their suggestion of power, they rather invite than discourage complexity. You cannot expect the hordes of people that have devoted a major part of their professional lives to such efforts to react kindly to the suggestion that most of these efforts have been misguided, and we can hardly expect a more sympathetic ear from the granting agencies that have funded these efforts: too many people have been involved and we know from past experience that what has been sufficiently expensive is automatically declared to have been a great success. Thirdly, the vision that automatic computing should not be such a mess is obscured, over and over again, by the advent of a monstrum that is subsequently forced upon the computing community as a de facto standard (COBOL, FORTRAN, ADA, C++, software for desktop publishing, you name it).

In short, the opportunity to simplify will remain with us for many years, and I propose, in order to maintain our sanity and our enthusiasm, that we welcome the long duration of that opportunity, rather than to suffer from impatience each time the practitioners deride and discard our next successful pilot project as a toy problem: they will do so, even if you have achieved what, shortly before, they had confidently predicted to be impossible.

* * *

By now we all know that programming is as hard or as easy as proving, and that if programming a procedure corresponds to proving a theorem, designing a digital system corresponds to building a mathematical theory. The tasks are isomorphic. We also know that, while from an operational point of view a program can be nothing but an abstract symbol manipulator, the designer had better regard the program as a sophisticated formula. And we also know that there is only one trustworthy way for the design of sophisticated formulae, viz. derivation by means of symbol manipulation. We have to let the symbols do the work, for that is the only known technique that scales up. Computing and Computing Science unavoidably emerge as an exercise in formal mathematics or, if you wish an acronym, as an exercise in VLSAL (Very large Scale Application of Logic).

Because of the very close connection between program design and proof design, any advance in program design has a direct potential impact on how general mathematics is done. Since the time computing scientists have built compilers, they are very used to the idea of mechanical manipulation of uninterpreted formulae, and I am sure that they will significantly contribute to a further realization of Leibniz's Dream of presenting calculation, i.e. the manipulation of uninterpreted formulae, as an alternative to human reasoning. The challenge of turning that Dream into reality, however, will certainly keep us busy for at least five decades.

It is not only that the design of an appropriate formal, notational and conceptual practice is a formidable challenge that still has to be met; the challenge is worse because current traditions are hardly helpful. For instance, we know that the transition from verbal reasoning to formal manipulation can be appreciated as narrowing the bandwidth of communication and documentation, whereas in the name of "ease of use" a lot of effort of the computing community is aimed at widening that bandwidth. Also, we know that we can only use a system by virtue of our knowledge of its properties, and, similarly, pay the greatest possible care to the choice of concepts in terms of which we build up our theories: we know we have to keep it crisp, disentangled, and simple if we refuse to be crushed by the complexities of our own making. But complexity sells better and the market pulls in the opposite direction. I still remember finding a book on how to use "Wordperfect 5.0" of more than 850 pages, in fact a dozen pages more than my 1951 edition of Georg Joos, "Theoretical Physics"! It is time to unmask the computing community as a Secret Society for the Creation and Preservation of Artificial Complexity. And then we have the software engineers, who only mention formal methods in order to throw suspicion on them. In short, we should not expect too much support from the computing community at large. And from the mathematical community I have learned not to expect too much support either, as informality is the hallmark of the Mathematical Guild, whose members --like poor programmers-- derive their intellectual excitement from not quite knowing what they are doing and prefer to be thrilled by the marvel of the human mind (in particular their own ones). For them, the Dream of Leibniz is a Nightmare. In summary, we are on our own.

But that does not matter. In the next fifty years, Mathematics will emerge as The Art and Science of Effective Formal Reasoning, and we shall derive our intellectual excitement from learning How to Let the Symbols Do the Work.

Calculemus!

Edsger W.Dijkstra

prof.dr Edsger W.Dijkstra
Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712-1188
USA