The next forty years

Last fall, I was invited to Zürich to assist the ETH in all sorts of celebrations: a new building, the 20th anniversary of their Institute of Informatics, and the 40th anniversary of their Institute of Applied Mathematics. Moreover it was 40 years ago when the ETH acquired a Z4 as its first computing engine. The combination of the occasion and my long involvement in the topic seduced me to present my biased version of what really had happened during the last four decades of automatic computing. This was at the beginning of the Academic Year; wouldn't it be nice to end this year with a talk about the next four decades of automatic computing?

When the idea first entered my head, I wrote it off as utterly preposterous: which sane scientist purports to be able to see so far into the future? But, preposterous or not, the idea did not leave me, and, as time went by, I caught myself inventing all sorts of reasons why it should be possible to say something of such long-range relevance. Instead of telling you all the ways in which I deluded myself, I shall confine myself to one of them, addressed to all of you that are engaged in education. When designing our courses, we do dare to decide what to teach

and what to ignore, and we do this for the benefit of people, many of whom will not have retired yet by the year 2029. Clearly, some vision of the next 40 years 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 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 turned into an endeavour amenable to some sort of mathematical treatment, long before I knew what kind 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 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 prediction a lot easier. The other month I was told of a great invention called "the register window". My spokesman was young but in my ears it sounded very familiar because I remem-

bered the Burroughs B5000 of 30 years ago. So, if you have a bright and sound idea now, you can expect it to be hailed as novelty around the year 2015.

Another thing we can learn from history is the failure of characterizations like "Computing Science is really nothing but X", where for X you may substitute your favorite 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.

Computing's core challenge is how not to make a mess of it. If people object that any science meets that challenge, we should give a double rebuttal. Firstly, machines are so fast and store 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.

The history of the real-time interrupt is in this connection illuminating. It was invented for the purpose of facilitating processor sharing; its effect was the introduction of nondeterminism and endless headaches for many an operating systems designer. We have seen two reactions to it. For the purpose of debugging OS/360, IBM built 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 was to determine the conditions under which one could feasibly and safely reason about nondeterministic programs, and subsequently to see to it that those conditions were met by both hardware and software. OS/360 was a mess forever after; the THE Multiprogramming System, in contrast, was so robust that no system malfunctioning ever gave rise to a spurious call for hardware maintenance. Needless to say, the whole episode has made a lasting impression on me.

On moral is that the real-time interrupt was only a wave, whereas the tide was the introduction of nondeterminism and of system invariants as a means of coping 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 that counts performance predictability among his responsibilities tends to come up with designs that need no queuing theory at all. At last, and this time recent, example is the design of delay-insensitive circuitry, which delegates a whole class of timing difficulties 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 the examples that come to mind the design process cost much less labour and led to a much better final product than its intuitively conceived alternatives. 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 designer, 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 all 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. Ten years ago it was Ada; now it seems to be the software for desktop publishing. Whether the academic community will resist and ignore the latter as successfully as it did the former, remains to be seen; I have my doubts.

In short, the opportunity to simplify will remain with us for many years, and I propose, in order to maintain our sanity and 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 of the program 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. Computing and computing science unavoidably emerge as an exercise in formal mathematics or, if you wish an acronym, as VLSAL (= Very Large Scale Application of Logic).

The extremely close connection between program design and proof design has revived interest in automatic programming and mechanical theorem proving. We now understand that both goals embody the same dream. But this insight does not make the dream more realistic than before. I have no first-hand experience with automatic theorem proving, but have an impression from looking at it from some distance, and am willing to share that impression for what it is worth.

Firstly, mechanical proof verification seems a more apt description than mechanical theorem proving. Secondly, the use of a mechanical proof verifier requires a great amount of human symbol manipulation of a usually painful kind.

Thirdly, mechanical proof verifiers are primarily praised for being totally unforgiving: their users tell me that, over and over again, the purported proofs they offer to the verifier turn out to be incomplete. Fourthly, the formalisms their user communities have adopted invariably seem to be notationally atrocious.

I am sometimes wondering how these efforts will be evaluated one or two decades from now. You see, right from the start, the opinion that formal mathematics is by definition too laborious, too tedious, too lengthy, and hence too error-prone to be done by hand, has been raised to a dogma, but as time goes by, I find this dogma becoming more and more suspect. Firstly, it was politically too convenient: the AI community has used it extensively to justify these efforts at mechanized symbol manipulation, thus losing all interest in the simplification of doing formal mathematics. I have, indeed, seen the mechanical production of pages of verification conditions, much longer than anyone would care to produce by hand; very impressive until you realized that they should not have been produced in the first place. Secondly, it is quite likely that the doing of formal mathematics as commonly perceived is needlessly cumbersome. Over the last decade I have gained considerable experience in

the design of purely calculational proofs. It may, of course, be due to the topics I have dealt with, but I find the dogma not confirmed; on the contrary, I find my calculational proofs shorter than any alternative I can think of. To which I should add that, in order to reach this goal, I had to deviate, over and over again, from established notational practice and had to learn a bunch of little general theorems about classical logic that are by no means common knowledge. From this experience I can draw only one conclusion: thus far, formal mathematics has never been given a fair chance. Inescapably I must conclude that, by and large, the art and science of letting the symbols do the work is still in its infancy.

The above conclusion has all sorts of consequences. A few decades from now, current efforts at mechanical theorem proving are likely to be characterized as premature in the sense that they attempted to mechanize what we did not know yet ourselves how to do well.

Much further reaching are its consequences for our conception of doing mathematics and, eventually, for our practice of designing proofs and programs. Seventeen years ago I have pointed out that computers would have their greatest impact on our culture, not in their capacity

of tools, but in their capacity of intellectual challenge; at the time, my vision of that challenge was focussed on the programming task, but now I can be at the same time more specific and more general.

By their mere existence, computers are a daily reminder of their healthy limitations and drive home the full implication of the realization of Leibniz's Dream of presenting 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 four decades.

It is not only that the design of an appropriate formal, notational, and conceptional practice is a formidable challenge that still has to be met; it 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, obviously, the market pulls in the opposite direction. The other day I found in the University bookstore in Eindhoven 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. For them, the Dream of Leibniz is a Nightmare. In summary, we are on our own.

But that does not matter. In the next forty

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.

I thank you for your attention.

Nuenen, 14 June 1989

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