There Is No Software

The present explosion of the signifying scene, which, as we know from Barry McGuire and A. F. N. Dahran, coincides with the so-called Western world, is instead an implosion. The bulk of written texts—including the paper I am actually reading to you—no longer exist in perceivable time and space, but in a computer memory's transistor cells. And since these cells, in the last three decades of Silicon Valley exploits, have shrunk to spatial extensions of less than one micrometer, our writing scene may well be defined by a self-similarity of letters over some six orders of decimal magnitude. This state of affairs does not only make a difference to history, in which, at its alphabetical beginning, a camel and its hebraic letter gamel were just two and a half orders of decimal magnitude apart. It also seems to hide the very act of writing.

As one knows without saying, we do not write anymore. The crazy kind of software engineering that was writing suffered from an incurable *confusion between use and mention*. Up to Hölderlin's time, a mere mention of lightning seems to have been sufficient evidence of its possible poetic use. Nowadays, after this lightning's metamorphosis into electricity, manmade writing passes instead through microscopically written inscriptions, which, in contrast to all historical writing tools, are able to read and write by themselves. The last historical act of writing may well have been the moment when, in the early seventies, the Intel engineers laid out some dozen square meters of blueprint paper (64 square meters in the case of the later 8086) in order to design the hardware architecture of their first integrated microprocessor. This manual layout of two thousand transistors and their interconnections was then miniaturized to the size of an actual chip and, by electro-optical machines, written into silicon

layers. Finally, this 4004-microprocessor found its place in the new desk calculators of Intel's Japanese customer,[1] and our postmodern writing scene could begin. Actually, the hardware complexity of microprocessors simply discards such manual design techniques. In order to lay out the next computer generation, the engineers, instead of filling countless meters of blueprint paper, have recourse to Computer Aided Design, that is, to the geometrical or autorouting powers of the actual generation.

In constructing the first integrated microprocessor, however, Intel's Marcian E. Hoff had given an almost perfect demonstration of a Turing machine. After 1937, computing, whether done by men or by machines, can be formalized as a countable set of instructions operating on an infinitely long paper band and the discrete signs thereon. Turing's concept of such a paper machine,[2] whose operations consist only of writing and reading, proceeding and receding, has proven to be the mathematical equivalent of any computable function. Universal Turing machines, when fed the instructions of any other machine, can imitate it effectively. Thus, precisely because eventual differences between hardware implementations do not count anymore, the so-called Church-Turing hypothesis in its strongest or physical form is tantamount to declaring nature itself a universal Turing machine.

This claim in itself has had the effect of duplicating the implosion of hardware by an explosion of software. Programming languages have eroded the monopoly of ordinary language and grown into a new hierarchy of their own. This postmodern Tower of Babel reaches from simple operation codes whose linguistic extension is still a hardware configuration, passing through an assembler whose extension is this very opcode, up to high-level programming languages whose extension is that very assembler. In consequence, far-reaching chains of self-similarities in the sense defined by fractal theory organize the software as well as the hardware of every writing. What remains a problem is only recognizing these layers which, like modern media technologies in general, have been explicitly contrived to evade perception. We simply do not know what our writing does.

To wordprocess a text, that is, to become oneself a paper machine working on an IBM AT under Microsoft DOS, one must

first of all buy some commercial files. Unless these files show the file extension names of EXE or of COM, wordprocessing under DOS could never start. The reason is that only COM- and EXE-files entertain a peculiar relation to their proper names. On the one hand, they bear grandiloquent names like WordPerfect, on the other hand, more or less cryptic, because nonvocalized, acronyms like WP. The full name, alas, serves only the advertising strategies of software manufacturers, since DOS as a microprocessor operating system cannot read file names longer than eight letters. That is why the unpronounceable acronym WP, this posthistoric revocation of a fundamental Greek innovation, is not only necessary, but amply sufficient for postmodern wordprocessing. In fact, it seems to bring back truly magical power. WP does what it says. Executable computer files encompass, by contrast not only to WordPerfect but also to big but empty Old European words such as the Mind or the Word, all the routines and data necessary to their self-constitution. Surely, tapping the letter sequence WP and Enter on an AT keyboard does not make the Word perfect, but this simple writing act starts the actual execution of WordPerfect. Such are the triumphs of software.

The accompanying paperware cannot but multiply these magic powers. Written to bridge the gap between formal and everyday languages, electronics and literature, the usual software manuals introduce the program in question as a linguistic agent ruling with near omnipotence over the computer system's resources, address spaces, and other hardware parameters: WP, when called with command line argument X, would change the monitor screen from color A to B, start in mode C, return finally to D, etc. ad infinitum.

In fact, however, these actions of agent WP are virtual ones, since each of them (as the saying goes) has to run under DOS. It is the operating system and, more precisely, its command shell that scans the keyboard for eight-bit file names on the input line, transforms some relative addresses of an eventually retrieved file into absolute ones, loads this new version from external mass memory to the necessary random access space, and finally or temporarily passes execution to the opcode lines of a slave named WordPerfect.

The same argument would hold for DOS, which, in the final analysis, resolves into an extension of the basic input and output sys-

tem called BIOS. Not only no program, but also no underlying microprocessor system could ever start without the rather incredible autobooting faculty of some elementary functions that, for safety's sake, are burnt into silicon and thus form part of the hardware. Any transformation of matter from entropy to information, from a million sleeping transistors into differences between electronic potentials, necessarily presupposes a material event called reset.

In principle, this kind of descent from software to hardware, from higher to lower levels of observation, could be continued over more and more orders of magnitude. All code operations, despite such metaphoric faculties as call or return, come down to absolutely local string manipulations, that is, I am afraid, *to signifiers of voltage differences*. Formalization in Hilbert's sense does away with theory itself, insofar as "the theory is no longer a system of meaningful propositions, but one of sentences as sequences of words, which are in turn sequences of letters. We can tell [say] by reference to the form alone which combinations of the words are sentences, which sentences are axioms, and which sentences follow as immediate consequences of others."[3]

When meanings come down to sentences, sentences to words, and words to letters, there is no software at all. Rather, there would be no software if computer systems were not surrounded by an environment of everyday languages. This environment, however, ever since a famous and twofold Greek invention, has consisted of letters and coins, of books and bucks.[4] For these good economical reasons, nobody seems to have inherited the humility of Alan Turing, who, in the stone age of computing, preferred to read his machine's outprint in hexadecimal numbers rather than in decimal numbers.[5] On the contrary, the so-called philosophy of the so-called computer community tends systematically to obscure hardware with software, electronic signifiers with interfaces between formal and everyday languages. In all philanthropic sincerity, high-level programming manuals caution against the psychopathological risks of writing assembler code.[6] In all friendliness, "BIOS services" are currently defined as designed to "hide the details of controlling the underlying hardware from your program."[7] Consequently, in a perfect gradualism, DOS services would hide the BIOS, WordPerfect the operating system, and so on and so on until, very recently, two fundamental

changes in computer design (or DoD politics) have brought this system of secrecy to closure. First, on an intentionally superficial level, perfect graphic user interfaces, since they dispense with writing itself, hide a whole machine from its users. Second, on the microscopic level of hardware, so-called protection software has been implemented in order to prevent "untrusted programs" or "untrusted users" from any access to the operating system's kernel and input/output channels.[8]

This ongoing triumph of software is a strange reversal of Turing's proof that there can be no mathematically computable problem a simple machine could not solve. Instead, the physical Church-Turing hypothesis, by identifying physical hardware with the algorithms forged for its computation, has finally gotten rid of hardware itself. As a result, software has successfully occupied the empty place and profited from its obscurity. The ever-growing hierarchy of high-level programming languages works exactly the same way as one-way functions in recent mathematical cryptography. Such functions, when used in their straightforward form, can be computed in reasonable time, for instance, in a time growing only in polynomial expressions with the function's complexity. The time needed for its inverse form, however (that is, for reconstructing from the function's output its presupposed input), would grow at exponential and therefore unviable rates. One-way functions, in other words, hide an algorithm from its result. For software, this cryptographic effect offers a convenient way to bypass the fact that by virtue of Turing's proof the concept of mental property as applied to algorithms has become meaningless. Precisely because software does not exist as a machine-independent faculty, software as a commercial or American medium insists on its status as property all the more. Every license, every dongle, every trademark registered for WP, as well as for WordPerfect, proves the functionality of one-way functions. In this country, notwithstanding all mathematical tradition, even a copyright claim for algorithms has recently succeeded. And, finally, IBM has done research on a mathematical formula for measuring the distance in complexity between an algorithm and its output. Whereas in the good old days of Shannon's mathematical theory of information, the maximum in information coincided strangely with maximal unpredictability, or noise,[9] the new IBM measure, called logical

depth, has been defined as follows: "The value of a message . . . appears to reside not in its information (its absolutely unpredictable parts), nor in its obvious redundancy (verbatim repetitions, unequal digit frequencies), but rather in what may be called its buried redundancy—parts predictable only with difficulty, things the receiver could in principle have figured out without being told, but only at considerable cost in money, time, or computation. In other words, the value of a message is the amount of mathematical or other work plausibly done by its originator, which the receiver is saved from having to repeat."[10] Thus, logical depth in its mathematical rigor could advantageously replace all the old, everyday language definitions of originality, authorship, and copyright in their necessary inexactness, were it not for the fact that precisely this algorithm intended to compute the cost of algorithms in general is Turing-uncomputable itself.[11]

Under these tragic conditions, criminal law, at least in Germany, has recently abandoned the very concept of software as mental property; instead, it defines software as necessarily a material thing. The high court's reasoning, according to which no computer program could ever run without the corresponding electrical charges in silicon circuitry,[12] can illustrate the fact that the virtual undecidability between software and hardware by no means follows, as systems theorists would probably like to believe, from a simple variation of observation on points. On the contrary, there are good grounds to assume the indispensability and, consequently, the priority of hardware in general.

Only in Turing's paper *On Computable Numbers with an Application to the Entscheidungsproblem* does there exist a machine with unbounded resources in space and time, with an infinite supply of raw paper and no constraints on computation speed. All physically feasible machines, in contrast, are limited by these parameters in their very code. The inability of Microsoft DOS to tell more than the first eight letters of a file name such as WordPerfect gives just a trivial or obsolete illustration of a problem that has provoked not only increasing incompatibilities between the different generations of eight-bit, sixteen-bit, and thirty-two-bit microprocessors, but also a near impossibility of digitalizing the body of real numbers formerly known as nature.[13]

According to Brosl Hasslacher of Los Alamos National Laboratory:

> This means [that] we use digital computers whose architecture is given to us in the form of a physical piece of machinery, with all its artificial constraints. We must reduce a continuous algorithmic description to one codable on a device whose fundamental operations are countable, and we do this by various forms of chopping up into pieces, usually called discretization . . . The compiler then further reduces this model to a binary form determined largely by machine constraints.
>
> The outcome is a discrete and synthetic microworld image of the original problem, whose structure is arbitrarily fixed by a differencing scheme and computational architecture chosen at random. The only remnant of the continuum is the use of radix arithmetic, which has the property of weighing bits unequally, and for nonlinear systems is the source of spurious singularities.
>
> This is what we actually do when we compute up a model of the physical world with physical devices. This is not the idealized and serene process that we imagine when usually arguing about the fundamental structures of computation, and very far from Turing machines.[14]

Thus, instead of pursuing the physical Church-Turing hypothesis and "injecting an algorithmic behavior into the behavior of the physical world for which there is no evidence,"[15] one has rather to compute what has been called "the price of programmability" itself. This all-important property of being programmable has, in all evidence, nothing to do with software; it is an exclusive feature of hardwares, more or less suited as they are to house some notation system. When Claude Shannon, in 1937, proved in what is probably the most consequential M.A. thesis ever written that simple telegraph switching relays can implement, by means of their different interconnections, the whole of Boolean algebra,[16] such a physical notation system was established. And when the integrated circuit, developed in the 1970s out of Shockley's transistor, combined on one and the same chip silicon as a controllable resistor with its own oxide as an almost perfect isolator, the programmability of matter could finally "take control," just as Turing had predicted.[17] Software, if it existed, would be just a billion-dollar deal based on the cheapest elements on earth. For in their combination on chip, silicon and its oxide provide perfect hardware architectures. That is to say, millions of basic elements work under almost the same physical conditions, especially as regards the most critical, namely, temperature-dependent degradations, and yet electrically all of them are highly isolated from each other. Only

this paradoxical relation between two physical parameters, thermal continuity and electrical discretization on chip, allows integrated circuits to be not only finite-state machines like so many other devices on earth, but to approximate that Universal Discrete Machine into which its inventor's name has long disappeared.

This structural difference can easily be illustrated. "A combination lock," for instace, "is a finite automaton, but it is not ordinarily decomposable into a base set of elementary-type components that can be reconfigured to simulate an arbitrary physical system. As a consequence it is not structually programmable, and in this case it is effectively programmable only in the limited sense that its state can be set for achieving a limited class of behaviors." On the contrary, "a digital computer used to simulate a combination lock is structurally programmable since the behavior is achieved by synthesizing it from a canonical set of primitive switching components."[18]

Switching components, however, be they telegraph relays, tubes, or, finally, microtransistor cells, pay a prize for their very composability. Confronted as they are with a continuous environment of weather, waves, and wars, digital computers can cope with this real number avalanche only by adding element to element. However, the growth rate of possible interconnections between these elements, that is, of the computing power as such, has proven to have as its upper bound a square root function. In other words, it cannot even "keep up with polynomial growth rates in problem size."[19] Thus, the very isolation between digital or discrete elements accounts for a drawback in connectivity that otherwise, "according to current force laws" as well as to the basics of combinatorial logics, would be bounded only by a maximum equalling the square number of all elements involved.[20]

Precisely this maximal connectivity, on the other, physical side, defines nonprogrammable systems, be they waves or beings. That is why these systems show polynomial growth rates in complexity and, consequently, why only computations done on nonprogrammable machines could keep up with them. In all evidence, this hypothetical, but all too necessary, type of machine would constitute sheer hardware, a physical device working amidst physical devices and subject to the same bounded resources. Software in the usual sense of an ever-feasible abstraction would not exist any longer. The procedures

of these machines, though still open to an algorithmic notation, should have to work essentially on a material substrate whose very connectivity would allow for cellular reconfigurations. And even though this "substrate can also be described in algorithmic terms, by means of simulation," its "characterization is of such immense importance for the effectiveness . . . and so closely connected with choice of hardware," that programming it will have little to do any longer with approximated Turing machines.[21]

In what I have tried to describe as badly needed machines that are probably not too far in the future (and drawing quite heavily on recent computer science), certain Dubrovnik observers' eyes might be tempted to recognize, under evolutionary disguises or not, the familiar face of man. Maybe. At the same time, however, our equally familiar silicon hardware obeys many of the requisites for such highly connected, nonprogrammable systems. Between its million transistor cells, some million to the power of two interactions always already take place. There is electron diffusion; there is quantum-mechanical tunneling all over the chip.[22] Technically, however, these interactions are still treated in terms of system limitations, physical side effects, and so on. To minimize all the noise that it would be impossible to eliminate is the price we pay for structurally programmable machines. The inverse strategy of maximizing noise would not only find the way back from IBM to Shannon, it may well be the only way to enter that body of real numbers originally known as chaos.