Articles: a032
Date Published: 10/18/1995
www.ctheory.net/articles.aspx?id=74
Arthur and Marilouise Kroker, Editors

There is No Software

Friedrich Kittler

Grammatologies of the present time have to start with a rather sad statement. The bulk of written texts - including this text - do not exist anymore in perceivable time and space but in a computer memory's transistor cells. And since these cells in the last three decades have shrunk to spatial extensions below one micrometer, our writing may well be defined by a self-similarity of letters over some six decades. This state of affairs makes not only a difference to history when, at its alphabetical beginning, a camel and its Hebraic letter gemel were just 2.5 decades apart, it also seems to hide the very act of writing: we do not write anymore. This crazy kind of software engineering suffered from an incurable confusion between use and mention. Up to Holderlin's time, a mere mention of lightning seemed to have been sufficient evidence of its possible poetic use. After this lightning's metamorphosis into electricity, human-made writing passes 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, 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 began. For the hardware complexity of such microprocessors simply discards manual design techniques; in order to lay out the next computer generation, the engineers, instead of filling out uncountable meters of blueprint paper, have recourse to Computer Aided Design, that is, to the geometrical or autorouting powers of the actual generation.

However, in constructing the first integrated microprocessor, Intel's Marcian E. Hoff had given an almost perfect demonstration of a Turing machine. Since 1937, computing, whether done by people or by machines, has been formalized as a countable set of instructions operating on an infinitely long paper band and its discrete signs. 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 with the instructions of any other machine, can effectively imitate it. 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 that very assembler. As a 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 the realization of these layers which, just as modern media technologies in general, have been explicitly contrived in order to evade all perception. We simply do not know what our writing does.

For an illustration of this problem, a simple text program like the one that has produced my very paper will do. May the users of Windows or UNIX forgive when I, as a subject of Microsoft DOS, limit the discussion to this most stupid of operating systems.

In order to wordprocess a text and, that is, to become yourself a paper machine working on an IBM AT under Microsoft DOS, you need first of all to buy some commercial programs. Unless these have the file extension of .EXE or of .COM, wordprocessing under DOS could never begin. The reason is that only .COM and .EXE files entertain a strange relation to their proper name. At the one hand, they bear grandiloquent names such as WordPerfect, on the other hand, they bear a more or less cryptic (because non-vocalized) acronym such as WP. The full name, alas, serves only the advertising strategies of software manufacturers, since DOS as a microprocessor operating system could never 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 the big, empty old European words such as "Mind" or "Word", all the old routines and data necessary to their self-constitution. Surely, tapping the letter sequence of "W", "P" 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 magical powers, written as they are in order to bridge the gap between formal and everyday languages. Electronics as literature; the 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, *et cetera ad infinitum*.

In fact, however, these actions of agent WP are virtual ones, since each of them 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 op code lines of a slave named WordPerfect.

The same argument would hold against DOS which, in the final analysis, resolves into an extension of the basic input and output system called BIOS. Not only no program, but no underlying microprocessor system could ever start without the rather incredible autobooting faculty of some elementary functions that, for safety's sake, are burned 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 decades. All code operations, despite their metaphoric faculties such as "call" or "return", come down to absolutely local string manipulations and 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 any longer by an environment of everyday languages. This environment, however, since a famous and twofold Greek invention, consists 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 output in hexadecimal numbers rather than in decimal ones.[5] On the contrary, the so-called philosophy of the computer community tends to systematically obscure hardware by software, electronic signifiers by 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 "hid[ing] 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, in the very last years, two fundamental changes in computer design (or DoD politics) have brought this secrecy system to its closure.

Firstly, on an intentionally superficial level, perfect graphic user interfaces, since they dispense with writing itself, hide a whole machine from its users. Secondly, on the microscopic level of hardware itself, 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 would not solve. Instead, the physical Church-Turing hypothesis, by identifying physical hardware with the algorithm forged for its computation has finally got rid of hardware itself. As an effect, software 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.

These kinds of functions, when used in their straightforward form, can be computed in reasonable time, 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 functions' output its presupposed input; would grow at an exponential and therefore unviable rates. One-way functions, in other words, hide an algorithm from its very 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 all the more. Every license, every dongle, every trademark registered for WP as well as for WordPerfect prove the functionality of one-way functions. In the USA, notwithstanding all mathematical tradition, even a copyright claim for algorithms has recently succeeded. At most, finally, there has been, on the part of IBM, research on a mathematical formula that would enable them to measure 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 of 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, by only as 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, the criminal law, at least in Germany, has recently abandoned the very concept of software as a mental property; instead, it defined software as a necessarily material thing. The high court's reasoning, according to which without the correspondent electrical charges in silicon circuitry no computer program would ever run[12], can already illustrate the fact that the virtual undecidability between software and hardware follows by no means, as system theorists would probably like to believe, from a simple variation of observation 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* there existed a machine with unbounded resources in space and time, with 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 the ever-growing incompatibilities between the different generations of eight-bit, sixteen-bit and thirty-two-bit microprocessors, but also a near impossibility of digitizing the body of real numbers[13] formerly known as nature.

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 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 sere 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, that is of "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 prize 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 hardware, more or less suited as it is to house some notation system. When Claude Shannon, in 1937, proved in what is probably the most consequential MA thesis ever written[16] that simple telegraph switching relays can implement by means of their different interconnections the whole of Boolean algebra, such a physical notation system was established. And when the integrated circuit, developed in the 1970s out of Shockley's transistor, combined on one 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 just be a billion dollar deal based on the cheapest elements on earth. For, in their combination on chip, silicon and its oxide provide for perfect hardware architectures. That is to say that the 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 be easily illustrated. "A combination lock," for instance, "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 structurally 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 behaviours." On the contrary, "a digital computer used to simulate a combination lock is structurally programmable since the behaviour 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 price 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 those elements and, that is, of computing power as such, is proven to have a square root function as its upper bound. 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 which otherwise, "according to current force law" as well as to the basics of combinatorial logics, would be bounded only by a maximum equalling the square number of all involved elements.[20]

Precisely this maximal connectivity defines nonprogrammable systems, on the physical side, 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 subjected to the same bounded resources. Software in the usual sense of an ever-feasible abstraction would not exist any more. The procedures of these machines, though still open to an algorithmic notation, would have to work essentially on a material substrate whose very connectivity would allow for cellular reconfigurations. And even though the "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..."[21] programming them will have little to do anymore with approximated Turing machines.

Silicon hardware obeys many of the requisites for such highly connected, non-programmable systems. Between its millions of transistor cells, some million to the power of two interactions take place already; there is electronic diffusion, there is quantum mechanical tunneling all over the chip.[22] Yet, technically, 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 possible to eliminate is the prize paid 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.

Notes

1. Vgl. Klaus Schrodl, "Quantensprung." *DOS* 12/1990, p. 102f.

2. Cf. Alan M. Turing, "On Computable Numbers, with an application to the Entscheidungsproblem", *Proceedings of the London Mathematical Society*, Second Series, V. 42, 1937, p. 249.

3. Stephen C. Kleene, quoted by Robert Rosen, "Effective Processes and Natural Law", In: *The Universal Turing Machine, A Half Century Survey*, Rolf Herken, ed., Hamburg-Berlin-Oxford, 1988, p. 527.

4. Cf. Johannes Lohmann, "Die Geburt der Tagšdie aus dem Geiste der Muzik". Archiv fur Misikwissenschaft, 1980, p. 174.

5. Cf. Andrew Hodges, *Alan Turing: The Enigma*, New York 1983, p. 399.

6. Cf. "TOOL Praxis: Assembler-Programmierung auf dem PC", *Ausgabe 1*, WŸrzburg 1989, p.9.

7. Nabajyoti Barkalati, *The Waite Group's Macroassembler Bible*, Indiana, 1989, p. 528.

8. Cf. Friedrich Kittler, "Protected Mode", In *Computer, Macht und Gegenwehr*. InformatikerInnen fŸr eine andere Informatik, Ute Bernhardt and Ingo Ruhmann, ed. Bonn 1991, p. 34-44.

9. cf. Friedrich Kittler, *Signal-Rausch-Abstand*. In: *MaterialitŠt der Kommunikation*, ed. Hans-Ulrich Gumbrecht and K. Ludwig Pfeiffer, Frankfurt/M. 1988, p.343-345.

10. Charles H. Bennett, "Logical Depth and Physical Complexity", in Herken, ibid., p. 230.

11. With thanks to Oswald Wiener/Yukon.

12. Cf. M. Michael Kšnig, Sachlich sehen. Probleme bei der Uberlassung von Software. c't 3, 1990, p. 73 (Bundesgerichtshofentscheidung vom 2.5.1985, Az. I ZB 8/84, NJW-RR 1986, 219. Programs are defined as "Verkšrperungen der geistgen Leistung, damit aber Sachen".).

13. I am at a loss of understanding how Turing's famous paper could, in its first line, "briefly describe the 'computable numbers' as the real numbers whose expressions as a decimal are calculable by finite means" (On Computable Numbers, p. 230), then proceed to define the set of computable numbers as countable and finally call pi, taken as "the limit of a computably convergent sequence," a computable number (p. 256).

14. Brossl Hasslacher, "Algorithms in the World of Bounded Resources", in Herken, ibid., p. 421f.

15. Hasslacher, p. 420.

16. cf. Friedrich-Wilhelm Hagemeyer, *Die Entstehung von Informationskonzepten in der Nachrichtentechnik. Eine Fallstudia zur Theoriebildung in der Technik in Industrie- und Kriegsforschung*, Diss. phil. Berlin 1979, p.432.

17. cf. Alan Turing, *Intelligent Machinery, A Heretical Theory* in Sarah Turing, Alan M. Turing, Cambridge 1959, p. 134.

18. Michael Conrad, "The Prize of Programmability", in Herken, p. 289.

19. Ibid., p. 293.

20. Ibid., p. 290.

21. Ibid., p. 304.

22. Vgl. Conrad, Ibid., p. 303f.

Friedrich Kittler is one of Germany's leading media theorists. He is professor of philosophy at the University of Freiburg/Breisgau.