The problem of computer code: Leviathan or common power?

Adrian Mackenzie

Institute for Cultural Research, Lancaster University

March 2003

The problem of computer code: Leviathan or common power?

> [I]t is the solution that counts, but the problem always has the solution it deserves, in terms of the way in which it is stated (i.e., the conditions under which it is determined as a problem), and of the means and terms at our disposal for stating it (Deleuze, 1988, 16)

Is code important? Over the last decade, studies of digital culture, information culture, information society and cybercultures have focused on the content of new media and how new media are used. These studies identify changes in the communities, representations, interactions, ideologies, subjectivities and even corporealities associated with information and communication. This paper shifts focus and asks whether *code* - software or programs - is important.

Why code?

Code objects have very uneven visibility and significance within collective life. Some code objects are much more important and singular than others. Many are lost to sight, foundering in banal rationalisations, trivial applications and seemingly pointless fantasies of informatic order.[i] Some form critical components of key infrastructures. Ellen Ullman, a programmer from the West Coast of the USA, writes of her astonishment at finding that the entire transaction processing system of a global credit card provider relied on several dozen lines of antique assembler code that very few people could understand (Ullman, 1997). Neither the intrinsic properties of code as an object (Manovich, 2000), nor contests over its commodity value or property status (Lessig, 1999; Moody 2001), nor the particular kinds of labour, identity or subcultural life associated with it ('programmer', 'hacker', 'knowledge worker': Castells, 2000) guarantee the relevance or potency of code.

Given this, why analyse code rather than more visible, accessible, directly interactive and affective facets of contemporary culture such as websites, a chatroom, computer games or a database? Despite the existence of myriad books on how to program, despite

the efforts of whole academic disciplines devoted to code objects (computer science, software engineering, information systems), and in the face of millions of people scattered throughout Europe, America, Asia, and Africa who strongly identify themselves as programmers, software engineers, hackers, system analysts and software architects, we cannot be sure what code is and what it does. There are several reasons for this uncertainty.

Usually, we understand code as a set of instructions that control the operations of a computing machine. Popular and academic accounts consistently reiterate this understanding. It reduces code to a program for a mechanism. We should resist that reduction for a number of reasons. Many different kinds of practices, interactions, instabilities, contests, disputes and events populate the zone between the written instructions and the operations of computing machines. Even the starting and end points – the written text and the actual operations of computing devices – are frozen in place by this idealised understanding of code, rather than being seen as the outcome of complex interactions involving commodity production, organisational life, technoscientific knowledges and enterprises, the organisation of work, manifold identities and geopolitical-technological zones of contest. Code as written text, even its most fetishized form as proprietary *source code,* the code that programmers actually read and write (which Microsoft has recently agreed to sell for the first time to democratic governments in the interests of anti-terrorist security (Lettice, 2003; Lohr 2002)), rarely stands still or means one thing, as we will see.

Moreover, code runs deep in the increasingly informatically regulated infrastructures and built environments we move through and inhabit. Code participates heavily in contemporary economic, cultural, political, military and govermental domains. It modulates relations within collective life. It organises and disrupts relations of power. It alters the conditions of perception, commodification and representation. As Geoffrey Bowker and Susan Leigh Star write: '[i]t is politically and ethically crucial to recognize the vital role of infrastructure in the "built moral environment." Seemingly purely technical issues like how to name things and how to store data in fact constitute much of what we have come to know as natural' (Bowker & Star, 1999, 326). Code has been

naturalised through its direct participation in the ordinary and largely invisible infrastructures of moral, political, global, military, aesthetic, and entertainment environments. It potentiates new forms-of-life, lives which cannot be separated from their forms (Agamben, 1996).

Code is a 'solution', code is a problem

These two reasons to analyse code – to rescue it from the idea of a program, and to show how code is invisibly naturalised – raise more questions than they answer. In response to these questions, I will argue that mode of existence of code is *problematic*. In commercial software production, code is often called a 'solution'. Software producers provide solutions. To what kinds of problem is code a solution? Can we also see code as a problem, as a means of stating collective problems? A problem is something whose mode of existence is troublesome, contested, unstable and somewhat contradictory. We seek solutions to problems, but solutions are often partial or provisional. Moreover, a solution depends, as the epigraph from Gilles Deleuze indicates, on how the problem is posed or stated, and on 'the conditions under which it is determined as a problem'. Different ways of stating a problem suggest different ways of solving it.

In the example that follows, code presents at least four problematic facets. The example is a short piece of computer code called '*forkbomb.pl*' that won the Transmediale Art Prize, Berlin 2002. Within contemporary global technological cultures, where mediatic infrastructure and information platforms permeate domains of meaning and signification in complex, manifold ways (Lash, 2002), code has started to become an aesthetic object. The Transmediale jury wrote: '*forkbomb.pl* was awarded [the prize] as a crafted code which is transformed into an art of technical transgression' (Transmediale, 2002). The work is not an isolated example of artistic practice. Computer artists have been working with computer code since the 1960s. (For contemporary examples of software art, see (Software Art Repository 2003)). *forkbomb.pl* however is striking because the code itself, as well as any effect it might produce in execution, is key to the work. As a contemporary art object, code here becomes something that overflows ordinary work, use or exchange value and explicitly seeks to pose itself as a problem, something that figures a difference, a contradiction, a

potential for invention, generation or metamorphosis.

forkbomb.pl was written by British artist, Alex McLean. The entire source code (the code written by the programmer) for *forkbomb.pl* is listed below:

```
die "Please do not run this script without reading the documentation"
  if not @ARGV;

my $strength = $ARGV[0] + 1;

while (not fork) {
 exit unless --$strength;
 print 0;
 twist: while (fork) {
  exit unless --$strength;
  print 1;
 }
}
goto 'twist' if --$strength;
```

Along with some text commentary (approximately half a screen), this mundane listing, in all its rule-governed rationality, constitutes *almost* the entirety of the work.[ii] While not very complex by contrast with the millions of lines comprising a typical commercial operating system or major software installation, these lines economically condense some crucial dimensions of code objects. (Actually, an even more concise expression of the core of the program would be:' while 1 fork;'.) A substantial amount of tacit knowledge connects these lines of code to various other practices.

Two senses of code: text and process

What kind of object or thing is *forkbomb.pl?* It raises the question of what we mean by *code*. The word 'code' used by the Jury when they referred to *forkbomb.pl* oscillates between two different but closely, perhaps even undecidably related senses, *text* and *process*. This undecidable oscillation lies at the root of the problematic mode of existence of code.

In its first sense, code as a *text* is the product of the practice of programming or coding.

It usually takes the form of a set of text files containing *source* code or *script* such as those listed above. Hence the Jury speaks of 'crafted' code. Coding, programming or hacking produces code. Code in this sense is always written in some programming language – C, Perl, Java, assembly language, Lisp, Cobol, Basic, etc. The code shown above is written in the scripting language, Perl. The reading and writing practices which generate code range from highly creative and exploratory to mundane and repetitive. Code craft work over the last 50 years has accrued many sub-cultures, languages, idioms, dialects, design, engineering and architectural approaches. Internecine struggles and factional disputes constantly shift the meaning and value of code work. Sometimes these struggles concern coding 'style' and craft-aesthetic judgements over how code should be laid out on screen, how variables should be named, etc. Sometimes the struggles have wider ideological dimensions. Programming languages are platforms for corporate marketing strategies. Attracting software developers to a particular programming language opens the possibility of selling other software and hardware designed to work with that language. As a keys means of production of new code objects, programming languages are vital strategic resources to computer and operating system producers such as Microsoft, IBM, Apple and Sun. Microsoft's C# language, released in 2001, competes with Sun Corporation's highly successful and widespread Java language. Both languages are linked to the sale of other services, upgrades, maintainance and hardware. The relative economic value and geographical distribution of coding work shifts constantly as software production is automated, industrialised, shifted to cheaper labour markets (e.g. out-sourcing of programming work to India) or becomes 'free' (in the form of open source software). Much routine coding has been partially automated (for instance, newer web authoring tools obviate much of the need for hand-coding web pages). Code as a text-producing practice is variable and dynamic.

In its second sense code means something executable (depending on bugs and other unpredictable factors) within a computing environment or on some computing platform(s). The purpose of code, after all, is to work or operate. During execution, a code object becomes a *process*. The environments in which a process executes are diverse. Code objects might be something that a person double-clicks on and runs, they

might be running in an electronic toy or appliance, or they might be automatically downloaded and executed by a device such as a mobile phone. It is something that a particular hardware platform (Intel, AMD, Motorola, a PlayStation or Xbox, etc), an operating system (Windows, MacOS, Unix, PalmOS, Linux, etc), or perhaps a virtual machine such as the Java Virtual Machine can run. Code in this sense, something produced by the practices of coding, becomes software. Detached from the work of coding, it can be packaged, shrinkwrapped and circulated. (In the case of *forkbomb.pl*, a piece of supporting software (the Perl interpreter) translates the code into calls on lower level operating system functions. Perl provides a middle layer between the operating system and end-user applications.)

Like the practices of reading and writing code, code as process is not a simple, homogeneous or undifferentiated. On the contrary, the environment in which code executes, the *platform,* is a contested site, a 'lifted-out space' (Lash 2002, 24). There are the well known commercial struggles between platforms (such as Apple versus Microsoft) which feed the emergence of different sub-cultures of computer users. There are also struggles that concern the commodity status of a given platform. The 'open source' operating system Linux, for example, can be seen as springing from a desire to extract the generic x86 Intel family of CPU's manufactured by Intel and other companies such as AMD from the territorial dominance of Microsoft Corporation. By running a different operating system from the Microsoft operating systems that usually run on such commodity hardware, the Linux subculture alters the hegemonic status of the operating system in certain ways. The link between a particular operating system and a commodity hardware platform becomes more tenuous and open to variation.

Code integrates and disintegrates

We have already seen the code of *forkbomb.pl* as a written text. What happens when we see *forkbomb.pl* as a process? When the Transmedia jury refers to *forkbomb.pl* as 'transgressive', this other sense of code as executing process is in play. As a process, *forkbomb.pl* interferes with other processes in execution. A 'user' 'runs' the work by typing 'perl *forkbomb.pl* 6', and something like the following output appears:

```
01110111110001111111000111010111111111111000001111001111000011111111001111
01101011111101011000011011001010001011011010000010100111101100111111111001
11010100101000101100111111101001000001111111111111100001100010111111000000
01110011111111111111111111111111111000101111000000000100000000000000000010111
11110111111111111110000000000000000000000001101111101111101100000000000000111
11111001110001001110110011111111111110001111000100000000000000000000000010
```

Judging by this modest output, the perceptual dimensions of the work and the aesthetic sensations it immediately elicits are minimal, to say the least. We are not spectators of a conventional aesthetic object here. It hardly rates against the iconic density, colour and carefully designed layout of the average computer interface or the spectular visual detail seen in the latest generation of console computer games. Even within the realm of software art, other works have more explicit political and cultural significance. For instance, the well-known software artwork *CarnivorePE* (CarnivorePE, 2003), allows the movement of data on networks to be tracked and visualised. It mimics software used by the FBI to conduct electronic wiretaps, thereby making the operations of the US government visible and perhaps more contestable. In the case of *forkbomb.pl*, the executing process has no clear political or aesthetic significance. The output shows that an executing process generates ordered sets of marks. Here the set is composed simply of the figures 0 and 1. Just as coding involves editing text according to rules expressed syntactically by a programming language, the work done by executing code is mundane not mysterious: it generates sequences of marks over time.

However, just as the mundane coding work is animated by many other dynamics, the significance of a sequence of marks depends on how these marks are connected to other contexts. The artist Alex McLean suggests that the work be seen as producing a 'watermark' of the functional coupling of an operating system and hardware platform. Every operating system and platform will produce a different output. Even the same computer will produce different outputs depending on what other processes are running at that time (Cox, McLean & Ward, 2002). (Running it a few times bears this out, but it would need an extended description of how *forkbomb.pl* works to show why this is this case).

In view of the source code of *forkbomb.pl* and its output of 1s and 0s, code shows itself under two facets, one oriented towards the practices of producers such as artists, programmers and hackers reading-writing programs, and another facing towards the environmental particularities of machines, networks and operating systems which contain executing processes and users. Code is both written and read by people, and executed within a computing environment by users. Code has a complex or *forked* mode of existence. It exists both as highly rule-governed practices of written expression with particular traits that we could look at more closely (for instance, this code is written in Perl rather than Java: what difference does that make?), and as something 'prehended' or articulated within the specificity of an environment that computes, a *computing environment* or *platform* within which the code executes. *forkbomb.pl* addresses the problem of relating different times and places, that of reading-writing and that of executing.

forkbomb.pl can do more than 'watermark' an operating system. To *fork* in Unix operating system terminology means to start a new process alongside existing processes:

> The system call **fork** creates two nearly identical copies of a process. One copy is called the parent and the other the child. In the *parent* process, **fork** returns the process number of the child. The value returned in the child is zero. If **fork** cannot create a new process the a -1 is returned. This can happen, for example, when the system process table is full, or if the **fork** call is interrrupted (Bourne, 1987, 135)

A *forkbomb* occurs when an operating system attempts to **fork** too often. As *The New Hacker's Dictionary* writes, 'a fork bomb process "explodes" by recursively spawning copies of itself. ... Eventually it ... wedges the system' (Raymond, 1996, 203). The work takes its name from what happens if a user types something like 'perl *forkbomb.pl* 20'. Depending on the platform, increasing the numerical value at the end of the command line from 6 to 20 will probably progressively bring a computer to a hung or 'bombed' state where it produces no further perceptible output. *forkbomb.pl* 'bombs' a computer by colonising the operating system with an exponentially growing number of clones of the *forkbomb.pl* process, each recursively generating copies of itself (as do viruses such as LoveBug).

The different possible outputs from *forkbomb.pl* – watermark of the operating system or a frozen, 'wedged' system in which nothing appears to be happening - suggest two different critical evaluations of code. The watermark outcome reflects order, co-ordination, sharing of resources, distribution and division of labour. The 'wedged' outcome suggests disintegrating, scattering, dispersal and disaggregating loss of working functionality. With this outcome in mind, newsgroup and listserver responses amongst Perl programmers to *forkbomb.pl's* prize tended to be wrathful. For instance, one irked response ran:

> 'Do hack attacks now count as art? Viruses? Wouldn't an obfuscated program to completely erase the disk of the subject system be even more "crafted" and an even greater "technical transgression", and therefore, even better art? Will be seeing [it as] art a defense in computer crime trials now?' (Permungkah, 2002).

As the *Hacker's Dictionary* goes on to suggest, 'creating [a forkbomb] seldom accomplishes more than to bring the just wrath of the gods down upon the perpetrator' (Raymond, 1996, 203).

Code evades visibility

The just wrath of the gods can take on a more fundamentalist tone. In an article entitled 'There is No Software', the literary new media theorist Friedrich Kittler characterises the cultural problem of code in terms that go beyond irked responses to transgressive artworks such as *forkbomb.pl*:

> 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. ... 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. (Kittler, 1997, 148)

In this passage, the problem of code hinges on the way it evades perception. As happens

when *forkbomb.pl* runs at the trangressive strength of 20, 'we do simply do not know what what our writing does'. The general problem of code is that 'it seems to hide the very act of writing' (Kittler, 1997, 147). Furthermore, the inscriptions that result from this writing 'are able to read and write by themselves' (147). At the same time, hardware contracts down into ever more densely interconnected circuits in the computer hardware innovation race. As inscription shrink down into chips, code itself piles up into a postmodern Tower of Babel, or into a cacophony of different coding languages, sometimes hierarchically related, sometimes not.

Moving upwards and downwards, inwards and outwards, code appears to write itself. As an effect of the scale and complication of layers, code evades cognition by any one observer. In contrast to printed texts, code writes itself hermetically. Kittler's response is radical and in some ways hard to contest: the constitutive limit for code, one that it will inevitably encounter, is 'sheer hardware' or non-programmable systems whose complexity 'may well be the only way to enter that body of real numbers originally known as chaos' (Kittler, 1997, 155). Nonprogrammable systems are indeed the object of ongoing technoscientific speculation. In popular scientific and technology magazines such as *New Scientist* or *Technology Review* quantum or DNA computing systems of exceptional but almost uncontrollable computing power frequently come up for discussion.

Kittler's response points to a third facet of the problematic mode of existence of code. Code evades perception. As it covers over the non-programmable complexity of the real or 'chaos' by seeking to name and order it, it generates opacity and unintelligible complexity as byproducts. The spawning, mutating and cloning of different idioms of code, and indeed of different versions of similar software applications or 'solutions' generates code babble. There are many banal or ordinary manifestations of this disintegration. The need to constantly upgrade software to fix bugs and ensure compatability, the struggles over standardisation of different protocols, and proliferation of clones and variants of the same application or solution all attest to this code babble and to a multiplication of tongues, so to speak.

Where is the object? Where is the subject?

Compare Kittler's treatment of code as writing that attempts to intelligbly order the irreducible contingencies of matter and ends up rendering order almost unintelligible, with the standpoint of Bruno Latour. Writing about the complexity of contemporary technical objects, Latour observes:

> The paradox of a technological object with millions of instructions is that it is, from the standpoint of the division of labour, a fractal object that is equally simple at every point, and whole looks nevertheless like a Leviathan that infinitely surpasses human measure. (Latour, 1996, 217)

The 'Leviathan' of code ' has affinities to Kittler's 'we do not write anymore' (Kittler, 1997, 147). Leviathans often seem to be montrous and out of control. But Latour writes '[t]he entire technological wizardry lies in the impenetrable partitions and in the pegs that make it possible to hook one’s task to a neighbour’s' (217). Here the effects of Babelisation described by Kittler are evaluated differently: a social organisation of boundaries, detours, of folds and orderings permits 'a piling-up of Russian dolls' (217).

Like Kittler's account, a process of piling up or layering of code accounts for the apparent disappearance of spirit into matter. In contrast to Kittler, no irreducible, nonprogrammable domain underwrites a quasi-transcendental judgment of what code can or cannot do. Instead of covering over irreducible complexity, the code object is 'equally simple at every point' . Latour writes:

> We thought there was a frontier beyond which one really moved into matter, that inert and cold stuff, functional and soulless, which earned the admiration of materialists and the scorn of the humanists. (Latour, 1996, 222)

In fact, according to Latour, one never encounters a barrier between sign and thing since at every level there are further forms, 'half imprint and half text' (222).[iii]

It is striking, and this points to a fourth problematic element in code, that the Leviathan seems to surpass 'human measure.' Latour's account sees code as concealed social order, and as the outcome of co-ordinated partitioning of behaviours and actions

between different actors, human and non-human. His account allows us to interpret *forkbomb.pl* as a mini-Leviathan, as an expression of co-ordination and reciprocal interaction between a multitude of agents within contemporary collectives. In Latour's terms, the fact that *forkbomb.pl* runs at all attests to a piling up of Russian dolls, a labrynthine folding together of different actions to which no ultimate limit is prescribed. Latour's use of the term 'fractal' conveys that sense of nested or recursive order. In this case, the ordering pertains to computing resources. *forkbomb.pl* highlights the finite allocation of computing resources available on any computer. The allocation of computing resources within computing environments underpins the effects of interactivity, responsiveness, navigability, retrievability and storability on which informatic forms of life depend. The immobilisation produced by *forkbomb.pl* only signals that the ordering process has not been carried sufficiently far enough to create the effect of a true Leviathan, a monster of 'infinite measure'. Another set of detours, translations, mediations or hybridisations is needed. They could be social- a system administrator might set a limit to the number of processes that can be 'spawned' by any single program – or they be technical – an operating system might start distributing the recursively generated processes onto other machines forming part of a computing cluster or a 'server farm'.

Theorematic code

Through *forkbomb.pl,* the problematic mode of existence of code appears in a number of different guises. Code oscillates between text and process. It integrates and disintegrates contexts. In attempting to bind contexts together within a common language, it re-distributes and divides tasks in ways that overflow any one standpoint and evade perception. Finally, code can constitute a compressed Leviathan, a collective where bodies, actions, rules and power are orchestrated. It can give the appear of an almost executive power, yet in reality it relies on a finitie allocation of resources.

The example of *forkbomb.pl* is a risky one in certain respects. When code becomes an aesthetic object, how can we judge its importance? The philosopher A.N. Whitehead writes:

> One characterization of importance is that it is that aspect of feeling whereby a

> perspective is imposed upon the universe of things felt. (Whitehead, 1956, 15)

In treating the execution of *forkbomb.pl* as an autonomous aesthetic phenomena, can we locateor invent different perspectives 'upon the universe of things felt'? Does it allow us to see how and under what conditions code becomes important?

There are competing interpretations of the importance of code. In the academic disciplines of computer science and software engineering, where a formalist or theorematic understanding of code prevails, the importance of code consists in abstraction. These engineering disciplines represent code as a system whose properties can be modelled and expressed rigorously, preferably in a formal grammar, if not as a mathematical proof. Computer scientists formulate theorematic interpretations of code using set theory:

> Almost all the entities arising in the study of computer software systems including computer programs, languages, data structures, and operating systems, can be abstractly (mathematically) defined as 'algebraic systems': i.e., as sets of objects together with certain 'relations' and 'operations' associated with the sets (Lew, 1985, 51).

From this perspective, the work code does within collective lifeultimately reduces to relations and operations (such as sorting, comparing, copying, removing) pertaining to information entities. The actual code that programmers read and write consist of practical expressions of abstract relations and formal operations between entities grouped in sets. The concrete domain in which these entities, relations and operations exist and the actual code that expresses them would be secondary. Such an interpretation allow a formalist reading of *forkbomb.pl*'s code. Each line of the code would either define an entity, a relation or an operation on relations between entities within the domain of an algebraic system. So for instance, the line:

```
my $strength = $ARGV[0] + 1;
```

would define an entity (in this case a variable), associate it with a name ('strength'), assign it a value (in this case, by referring to a value supplied by the user and contained in the variable $ARGV[0]), and then carry out the arithmetic operation of adding 1 to

the variable named 'strength.' The theorematic interpretation rests on a well-established separation between form and content which homogenises content and subordinates it to form.

A different interpretation of code correlates much more closely with the day to day practices of programmers, and perhaps accords some generative potentials to coding. Deleuze and Guattari write: 'whereas the theorem belongs to the rational orders, the problem is affective and is inseparable from the metamorphoses, generations, and creations' (Deleuze & Guattari, 1987, 362). In their account of how 'collective bodies' can trigger novel compositions or metamorphoses, Deleuze and Guattari attribute potency to 'fringes or minorities' grappling with the problems posed by quite technical enterprises such as metallurgy, music, cathedral or bridge-building (1987, 366), or we could add, code. The practices of these fringe groups demonstrate a different relation between form and content than the usual imposition of form onto content/matter. A less theorematic and more problematic approach appears there:

> [M]atter, in nomad science, is never prepared and therefore homogenized matter, but is essentially laden with singularities (which constitute a form of content). And neither is expression formal; it is inseparable from pertinent traits (which constitute a matter of expression) (Deleuze & Guattari, 1987, 369)

If we take these ideas of material singularities and traits of expression seriously, code cannot be seen as a formal set of operations and relations concerning entities. The matter on which code works is never homogeneous or fully prepared. The 'matter' on which code operates includes collective arrangements of knowledge production, circulation and consumption, patterns of transaction and exchange, individual and mass communication-interaction, production, reception and audiencing of various audiovisual texts ranging across sound, image and texts, management and organisational structures, etc – is highly heterogeneous, dynamic and mutable.

Problematic code

What would these mean in terms of analysing *forkbomb.pl* as piece of code? As suggested above, *forkbomb.pl* is a somewhat self-referential work. All it does is

generate an exponentially increasing number of copies of itself. At the same time, the work relies on two particular distinctive features of contemporary multi-user interactive operating systems. These features are directly readable from the code itself. The key lines that point towards both material singularities within the computing environment and traits of expression within the code are:

```
while (not fork) {
 exit unless --$strength;
 print 0;
```

forkbomb.pl relies on the presence of a 'fork' operation within the programming language Perl. Not all programming languages have such a construct. The first point here is that *forkbomb.pl* could not be easily written in just any programming language. Perl displays some pertinent traits of expression.

The fork operation in Perl owes its existence to the fact that language was originally designed by a computer system administrator for use by other system administrators. Unlike most language designers, working in academic computing science departments or large corporate or institutional research laboratories, Larry Wall, the language designer, was working during the 1980s as a system administrator in a corporate environment on the West Coast of USA, somewhat removed from the theorematic agenda of computing science as an academic discipline (Moody, 2001, 132). The motivation for Perl, according to Wall, stemmed from an unoccupied 'ecological niche' in existing computing environments (Wall, 1999). Apart from desktop PCs, most computer systems at the time used an operating system called Unix, but system administrators had to either use very low level tools or high-level software applications. Dating from 1987, Perl attained almost unrivalled popularity in web programming and as a system administrator's tool because it lies midway between low level programming languages such as C and high level software tools and applications. Larry Wall describes his motivation as follows:

> When I started designing Perl, I explicitly set out to deconstruct all the computer languages I knew and recombine or reconstruct them in a different way, because there were many things I liked about other languages, and many things I disliked. I lovingly reused features from many languages. ... Whatever the verb you choose, I've

> done it over the course of the years from C, sh, csh, grep, sed, awk, Fortran, COBOL, PL/I, BASIC-PLUS, SNOBOL, Lisp, Ada, C++, and Python. To name a few. To the extent that Perl rules rather than sucks, it's because the various features of these languages ruled rather than sucked. (Wall, 1999)

Without going into the details, much of the *forkbomb.pl* code shown above could be analysed in terms of the way is 'laden with singularities' drawn from antecedent programming languages. The use of a 'while' statement, for instance, reflects the influence of the C programming language. The more important point here is that rather than being a formal description of relations between a set of entities, Perl effectively stated a problem that existed amidst the assemblage of large-scale computing networks developing during the late 1980s. Perl responds to the problem of, as Wall puts it, an 'unoccupied ecological niche.'

Wall combined features from many different programming languages, as well as drawing on linguistic theory. The language 'follows the connections between singularities of matter and traits of expression, and lodges on the level of these connections, whether they be natural or forced' (Deleuze & Guattari, 1987, 367). Combining singularities of matter and traits of expression that equally concern the architecture of hardware platforms, the existence of widely-shared practical knowledge of particular program constructs (such as loops and arrays), the commonalities of available operating systems, and existing ways of organising and dividing coding work, Perl changes how programmers work.

Connecting singularities of matter and traits of expression also involves 'another organization of work and of the social field through work' (Deleuze & Guattari, 1987, 369). Beyond the small examples of particular programming constructs we have analysed, we could examine what Perl has done to code during the last two decades. It would be possible to map out the kinds of programming constructs that Perl supplies, the syntactical idioms and contractions it makes available, or the vast library of add-on Perl 'modules' (in areas such as cryptography, database access, network control, text processing and pattern matching) that have become freely available over the last 15 years. Perl has been important because it allows code to be written quickly, often using

highly compressed and idiomatic syntax. Such code has been particularly useful for system administrators and programmers who effectively stitched together the infrastructures of information and communication today. Perl made it possible to write code that links web servers into other information systems (for instance, in eCommerce where webservers need to be connected to databases and transaction processing systems).

The fact *forkbomb.pl* is written in Perl carries a certain connotation of being connected to the infrastructures of information and communication. Because Perl is a system administrator's tool, writing in Perl is associated with inhabiting the infrastructures.

Conclusion: a feeling for code

Problems occupy a strange space. A problem is a task, according to Manuel Delanda, defined by 'a distribution of the singular and the ordinary, the important and the unimportant, the relevant and the irrelevant' (Delanda, 2002, 127). I have been arguing that code has a problematic mode of existence in several ways: in the relation between text and process; in the contest between ordering and disordering; in its becoming-imperceptible; and in the collective human-nonhuman formations it institutes. Code exists as a problem in the way that a mathematical problem might be said to exist, as something that people try to pose or state, as something that attracts different solutions and something that leads to affective responses.

Is code important? Code provides means and terms for stating solutions to certain problems posed within collective life. A problem, writes Deleuze, 'always has the solution it deserves, in terms of the way in which it is stated ... and of the means and terms at our disposal for stating it' (Deleuze, 1988, 16). These means and terms are not indifferent, ahistorical, transcendental, apolitical or incontestable. We cannot think code if we reduce its specificity to a totalised understanding of technology or information society, or we inflate its status to a radically new kind of substance or ground.

Can code provoke inventive or critical thought? Is code good to think with politically or ontologically? Can code trouble the typologies and ontologies governing contemporary

understandings of information, new media, communication and collective life? *forkbomb.pl* gives a feeling of how code enables particular solutions, particular ways of connecting material singularities and traits of expression. Code is written and run within situated practices, with reference to particular domains, and within particular orderings and disorderings of collective life. Its forms and abstractions are attached to lives. Via Perl, *forkbomb.pl* taps into material singularities concerning the distribution and allocation of computing resources, a key site of contestation within contemporary technoscapes. These contests pervade commodity computing platforms. As unwanted browser windows appear, as email spam fill the mailboxes, as viruses corrupt hard-drives, and as denial of service attacks render Al-Jazeera's website invisible, resource allocation contests structure the everyday experience of interactive computing that we mostly take for granted as part of the built-environment. The traits of expression *forkbomb.pl* draws on are embedded in Perl. They pertain to an organisation of coding work developed on the margins of mainstream computing science, in the technician's domain of system administration. The once relatively minor technical work of computer system administration today has become an integral link in constructing networks and allowing movement between computing environments.

Under what conditions does a feeling for code impose a perspective upon the 'universe of things felt'? How does code become an object of affective response? In principle, feeling tags a shift in relations between bodies. In their account of the relation between feeling and collective life, the political philosophers Genevieve Lloyd and Moira Gatens suggest:

> The awareness of human collectivities - the awareness of bodies in relation - is not a merely cognitive awareness of bodily change. It is shot through with emotion - with the awareness of bodily transition to greater or lesser states of activity. Sociability is inherently affective. (Gatens & Lloyd, 1997, 77)

Any feeling, including a feeling for code, of code's importance, arises from 'bodies in relation', to use Gatens and Lloyd's Spinozist terminology.

How does a feeling for code or about code come about? In Gatens and Lloyd's terms,

code may involve transition to lesser or greater states of activity, it may potentiate or 'passivate' bodies in relation. I have been suggesting that code has a problematic (rather than theorematic) mode of existence. Certain solutions reduce that problematic aspect to ordinary operationality. In fact, much of what becomes invisible, natural, easy in information infrastructures attests to a reduction to the ordinary or the normal. Much of the everyday work of coding takes place as a set of ordinary ordering practices. Other solutions maintain a closer relation to the problem from which they spring. Occasionally, the solution, a piece of code, actually reformulates and renders visible a problematic nexus that previously had been badly formulated or posed in a confused way.

References

Agamben, Giorgio, 'Form-of-life' eds.Hardt, Michael & Virno, Paolo, *Radical Thought in Italy. A Potential Politics,* University of Minnesota Press, Minneapolis & London, 1996, 151-156

Appadurai, Arjun, *Modernity at large: cultural dimensions of globalization,* University of Minnesota Press, Minneapolis, 1996

Bowker, Geoffrey C. & Star, Susan Leigh *Sorting Things Out. Classification and Its Consequences*, MIT Press, Cambridge MA, 1999

Bourne, Stephen, R. *The Unix System V Environment,* Addison-Wesley Publishing Company, 1987

CarnivorePE, http://www.rhizome.org/carnivore/ (27 January 2003)

Castells, Manuel *The Internet galaxy : reflections on the Internet, business, and society,* Oxford U.P, 2001

Cox, Geoff, McLean, Alex & Ward, Adrian 'The Aesthetics of Generative Code', http://www.generative.net/papers/aesthetics/index.html (12 December, 2002)

Delanda, Manuel, *Intensive Science and Virtual Philosophy,* Continuum Books, London &

New York, 2002

Deleuze, Gilles, & Guattari, Felix, *A Thousand Plateaus,* tr. Brian Massumi, University of Minnesota Press, Minneapolis, 1987

Deleuze, Gilles, *Bergsonism,* Zone Books, New York, 1988

Gatens, Moira & Lloyd, Genevieve, *Collective Imaginings. Spinoza, past and present* Routledge, London & New York 1999

Kittler, Friedrich, 'There is no software', *Literature Media Information Systems,*ed Johnston, John, G&B Arts International, 1997

Lash, Scott *Critique of Information* Sage Publications Ltd 2002

Latour, Bruno *Aramis, or the love of technology,* trans. by Catherine Porter, Harvard U.P, 1996

Lessig, Lawrence *Code and other Laws of Cyberspace*, Basic Books, 1999

Lettice, John (2003), 'MS plays the security card in Gov shared source retread', *The Register,* http://212.100.234.54/content/4/28869.html 22 Jan 2003)

Lew, Art *Computer Science: A Mathematical Introduction,* Prentice Hall International, 1985

Lohr, Steve, 'Microsoft to Give Governments Access to Code' *New York Times, Business/Financial Desk,* 15 January , 2003, Section C , p.10

Manovich, Lev, *The language of new media* MIT Press, Cambridge, 2000

Moody, Glyn *Rebel Code. Linux and the Open Source Revolution,* (Penguin, 2001).

Permungkah 2002, 'Great quote!' http://use.perl.org/comments.pl?cid=4434&sid=3218 (15 Jan 2003)

Raymond, Eric, *The New Hackers Dictionary,* 3rd Edition, MIT Press, 1996

Siever, Ellen, Spainhour, Stephen, & Patwardhan, Nathan, *Perl in a Nutshell. A Desktop*

Quick Reference, O'Reilly & Associates, Sebastapol,1999

Software Art Repository http://runme.org (13 Jan 2003)

Transmediale 2002 Prizes, http://www.transmediale.de/en/02/pressreleases.php (15 Jan 2003)

Ullman, Ellen, *Close to the Machine. Technophilia and its Discontents,* City Lights Books, San Francisco, 1997

Wall, Larry, 'Perl, the first postmodern computer language', http://www.perl.com/lpt/a/1999/03/pm.html (3 July 2002)

Whitehead, Alfred North *Modes of Thought* Cambridge University Press, 1956

i A major online code repository such as SourceForge (SourceForge, 2002) hosts thousands of coding projects. In January 2003, SourceForge claimed to contain 55,112 projects and have around a half million registered users. Although the number of registered users only amounts to a small city in contemporary terms, around 55,000 different software projects are under development there.

ii This code fragment shows a couple of important things. The text basically consists of two 'while' loops. Typically, the work of coding involves making loops, splicing loops together, and nesting loops within loops. Loops often interact with each other during execution of the code. In this case, the loops, twisted through each other, constantly alter each others' bounding conditions to a degree regulated by the 'strength' variable provided the user.

iii In a sense then, Latour's position echoes the phenomenological description of code given above: code is something both read and executed, it describes and operates, it expresses something and does something, it represents and performs what it represents. This suggests that crucial questions about code will concern not whether or not it can capture the real, but how code can seem to do both things at once, both a text describing a narrowly defined context, and a train of operations ordering and regulating a context.