Real artificial life: Where we may be[1]

David H. Ackley

Department of Computer Science
University of New Mexico

Abstract

Artificial life research typically employs digital computers to *implement models* of living systems. However, there is now a growing if pre-theoretical feeling that computers, or perhaps the software running on them, *are themselves* some kind of living systems. Such a possibility can impact artificial life research in at least two ways: By highlighting that computers and communications networks can be *subjects*, as well as tools, for artificial life modelling, and by highlighting that insights, tools, and models from the life sciences can have explanatory, predictive, and design consequences for the construction of future computation and communications systems. This paper seeks perspective on such 'real artificial life', looking backwards and forwards at the rise of living systems in manufactured computer and communications systems.

Real artificial life

Strong notions of artificial life, as discussed in (Sober, 1992) for example, claim that computers and computer programs may not merely *simulate*, but actually *instantiate* living systems. When considering the small, closed-world models typical of most of our work in artificial life, the strong claim is often dismissed with variants of the 'confusion-of-levels' objection, pointing out that 'a model of a hurricane won't get you wet' and so forth. Supporters of the strong claim might argue that life and hurricanes are qualitatively different, or that a hurricane model *will* drench a *model of you*, or that some simulations might be different from but every bit as useful as "the real thing" (Dennett, 1978), and the discussions continue.

Here I want only to recognize that such unsettled issues exist, then set them aside and veer erratically towards the real and the concrete, accepting uncertain philosophical footings below. I suspect that if you asked the millions of computer-using people today to give an example of 'artificial life', a common answer would be 'computer viruses'. Whatever the ontological status of a simulation of a forest fire, a computer virus is as real as the computer programs and data files it infects. The Melissa virus invaded over 80,000 reported computers in under two weeks in 1999 (Vatis, 1999). Even with no 'viral payload' whatsoever, Melissa clogged networks and incapacitated servers around the world; it had direct effects on the real people in the real world, it is not 'merely' a model. Much as we may want to distance ourselves from the ethical and moral questions, do we really want to argue that the Melissa virus is *not* artificial life?

We might dismiss Melissa on the grounds that viruses shouldn't be considered alive anyway. The natural world supports some reasonably clear distinctions between hardy 'living' systems that flourish in wide-ranging environments, versus mere 'parasitic replicators' that require a 'virus-friendly' environment, one willing not only to copy nearly any information at hand but then also to interpret the copied program regardless of what it does (Dawkins, 1991). But in that sense there is a strong kinship between living cells and manufactured computers in that both cell and computer interiors present tremendously virus-friendly environments. Moreover, the actual physicality of a computer itself may support richer notions of life, compared to the apparent insubstantiality of a computer program and the resulting sometimes anemic quality of purely software candidates for artificial life.

Though we may prefer to work entirely with small, manageable programs—executable models that are 'close to theory' in some sense—do we really want to argue that the rapidly expanding world of internetworked computer systems is *not* artificial life? Here, my aim is to run with the naive view to see what insights it may afford, beginning with the position that the connections between computation and life are genuine and the ramifications of that linkage are manifest not merely in small-scale closed-world computer models of natural world phenomena, but in the past, present, and likely future of manufactured computing devices on Earth.

There have been approaches to 'real world' computing issues from explicitly alife perspectives—the enterprise I am here calling 'real artificial life'—for example (Cohen, 1987; Spafford, 1992; Kephart, 1994;

[1]Ongoing developmental version of paper in *The Proceedings of Artificial Life VII*, Portland, Oregon, August 2000

Ackley, 1996). Either from the outset or over the course of their development, all of those efforts had a pronounced emphasis on computer security. Indeed, one of the largest areas of impact of the artificial life mindset upon current and emerging computing practices is in the area of 'computer immune systems' for improving security and robustness, e.g. (Forrest et al., 1996; Kephart et al., 1995; Forrest et al., 1998). Later we suggest why this has been so.

In this exploration we seek understanding of what is happening around us as the internet grows. We seek leverage from the idea that the stunning explosion of computing power on earth over the last few decades is not unprecedented, but has antecedents in the long development of life of earth. We seek better approaches to the design of manufactured computing, possibilities of a different relationship between us and our computers, and between computers and each other.

This endeavor turns the original charter of artificial life almost precisely on its head. Rather than seeking to understand natural life-as-it-is through the computational lens of artificial life-as-it-could-be (Langton, 1989), we seek to understand artificial computation-as-it-could-be through the living lens of natural computation-as-it-is. The endeavor can fail; there is no *a priori* assurance the connections between life and computation are bidirectional, or that any identified points of contact will be substantial and specific enough to be usefully predictive.

Still, given the tremendous current and future impacts of the computer and communications hardware and software that we choose to design, and the current paucity of a systematic basis underlying computational design for robust security and privacy, and the relentlessly myopic market-driven approach that most often dominates deployment decisions, it seems worth some struggle to uncover new perspectives.

Outline

In keeping with the 'Looking backwards, looking forwards' theme of this Artificial Life conference, this paper contains takes on the past, present, and future of life in manufactured computing. The next section sets the stage with a familiar tale carried into these new circumstances, bringing us from the more or less the beginning up to more or less the present. Following that, we draw out of the present state of affairs some basic 'living systems' principles and guidelines as they increasingly seem to apply to networked computer systems, and present a few instantiations of such principles in research software development. We characterize computer source code as a principal genotypic basis for living computation, and consider methods of applying tools and techniques from biology to the understanding of computer systems. Finally we speculate briefly on possible futures for living computation and consider some possible implications of artificial life 'in the real'.

Living Computation: The Past

Depending on how inclusive one chooses to be, the history of manufactured computing can stretch back hundreds and even thousands of years. To keep things manageable, we open in the recent past, with machines that are in some sense directly traceable ancestors of the machines surrounding us in the world today.

An origins story

In five decades, manufactured computer technology has come from nowhere to account for 10% of the world's industrial economies (Dertouzos, 1997), a spectacular growth process that, along the way, has been playing out a tale as old as storytelling:

1940's-1950's: The age of innocence Though there are many plausible candidates for the title of 'first manufactured computer', it's fair enough to call the Colossus machine built at Bletchley Park an 'early computer'. In January of 1944, the Colossus Mk 1 began breaking messages encoded with the Nazi's Enigma cipher within hours of their interception. By the end of the war, 63 million characters of German messages had been recovered by ten Colossus machines (Sale, 1998).

This war-torn period is the age of innocence in this story because the computer itself knew nothing of the conflict; it was unaware of allies or axis, of friends, enemies, or spies. It was completely open and at the mercy of whoever could reach the main plug panel next to the paper tape reader. Of course, buried deep in F Block on the ground of the top-secret Bletchley Park operation, such naive trust was both reasonable and efficient.

1950's-1970's: The knowledge of good and evil By the debut of the Digital Equipment Corporation PDP-10 in 1967, computers had found many viable niches and were spreading, but they were still major capital investments. The rise of time-sharing operating systems such as TOPS-10—with the ability to provide useful services to many users at once—sent communications links snaking across campuses and through office buildings. Mere physical control over the central hardware was no longer sufficient for system security.

User accounts, passwords, the distinction between ordinary 'user mode' and privileged 'supervisor mode' operation—all sorts of mechanisms for creating fences, enforcing separations, and permitting limited sharing—date from this period. Now for the first time, the internal design and operation of the computer begins to reflect the divisions and separations of the world outside it. Trust is no longer implicit and automatic; now it is explicit and conditional. Now for the first time, both in hardware and software the computer itself manifests a distinction between *self* and *other*, and the system administrator appears explicitly in the design, playing a third role, that of the 'trusted other'.

1970's-1980's: You can't go home again In August of 1981, the IBM 'personal computer' was launched. At an astoundingly low price, compared to mainframe and minicomputers, the PC offered computing power, inexpensive and convenient floppy disks for bulk storage, a monitor and keyboard for interactive use, printers and peripherals. What it didn't offer was any of the complex and resource-consuming trust management mechanisms of the time-sharing systems. Gone were the user ids, gone the passwords, gone the protected memory, gone the distinction between user and supervisor modes of operation. "This is not a time-sharing machine!" we can imagine somebody arguing, "This is a computer for *one* person! Who's to protect *from*?"

And then of course, the PC had scarcely hit the market when the first PC computer viruses appeared and began to spread via those same convenient floppy disks, without the least bit of immunological defense by the PC.

1980's-2000's: The big big world Belatedly comes the realization that 'personal computing' does not imply 'isolated computing'—if anything just the reverse, compared to the dedicated-function mainframes of old—but the genie is out of the bottle. Now we find ourselves in the odd situation of having myriads of these so-called personal computers connecting to the internet with essentially no systematic defenses, no immune system, and precious little sense of self. The scale of it is quite staggering, with the internet growing from thousands to tens of millions of hosts just in the last decade. Epidemic waves of infections flash through the networked population faster and farther than any natural pathogen—constrained to transmission vectors involving slow-poke matter—ever did, even as the exploding size and thus value of the global network makes remaining unconnected increasingly untenable for many purposes.

It is no wonder that the computer security industry is booming. Widely underappreciated is the fact that current commercial security systems, constrained to deal with the deployed base of hardware and software, are for the most part exactly as ad hoc, awkward, and unreliable as a plastic bubble is as substitute for an immune system—it works better than nothing, but surely you wouldn't trust your life to it if there was a more integrated and robust alternative.

The missing element

The story so far is one of machines designed in the image of the *conscious minds* of their creators—a single strictly serial process proceeding step by deliberate step, with nothing changing except as directed by the processor, with no need for coordination or communication, no peripheral or preconscious awareness, and so forth. Boundary conditions could be applied only before the computation began; when started it simply ran until finished. The machine had no stimulus-response ability, no interrupts, no hardware monitoring; as far as its functional repertoire was concerned, the machine not only could not control its 'body', it wasn't even aware it had one.

Living computation: The present

Thus has the evolution of manufactured computing systems to the present been backward to the evolution of natural computing systems such as the brain. The brain appeared only recently in the scope of the history of life on earth—and is no great shakes as a general-purpose algorithmic computing device—but from the beginning it has been richly interconnected to a body possessing sensory-motor apparatus that beggars anything we are currently able to manufacture at any price. The body has an extensive array of active and passive defense mechanisms, and the brain has extensive hardware support for threat assessment, triage, extrapolation, and rapid response.

The living computation perspective predicts that we are nearing childhood's end for computers, that current designs—still steeped in their innocent, safe, and externally-protected origins—will give way to designs possessing significant kinesthetic senses, defense and security at many levels, and with rich and persistently paranoid internal models representing the body and the stance of the body within the larger computational and physical environment.[1]

Manufactured computers began as 'pure mind designs' but were not thereby excused from the demands of existing in the physical world. With the rise of networked computing, and the rise of computers intended to survive in consumer environments and without the benefit of an expert human system administrator, the '*IOU: A body*' notes issued fifty years ago and more are now rapidly coming due.

Life principles for computation design

Many aspects, problems, and developments in current and emerging computing can be understood in this context. Here, we highlight several computational aspects of living systems that stand in contrast to traditional approaches to computing, then offer a few examples from current work illustrating some ways of that such strategies can apply in current and near-future systems.

Termination considered harmful In the fundamentals of computer science, an *algorithm* is typically defined as a *finite effective procedure* (Horowitz et al., 1997). A 'procedure'—a description or plan of action to accomplish something—that is 'effective'—so it can actually be

[1] I would have said this prediction was too obvious to be worth making, except that so much computer research, development, and deployment continues to ignore even the rudiments of robust system design.

performed, step-by-step—and is 'finite'—so it will definitely stop eventually. What are we to make of, say, an operating system, whose number one mission in life is to never ever stop?

(Horowitz et al., 1997) acknowledge the importance of such non-algorithmic 'computational procedures', but only in the process of excluding them from further consideration. Here, in contrast, they are our central focus—so much so that calling a system 'living' may in some sense *mean* the system is running an *infinite effective procedure*. Such an approach, though certainly unconventional, is compatible with the 'Computation as interaction' approach to redefining introductory Computer Science (Stein, 1998), and more generally with the ongoing shift from algorithmic and procedural computation to object-based open-ended computation. Living systems potentially offer an epistemological framework surrounding and motivating these newer characterizations of computation.

Programs are physical Given the mathematical and algorithmic emphasis underlying computers and computation, it is unsurprising that many computer scientists and other computing professionals tend to think of a computer program fundamentally in terms of the algorithm it implements, and to carry that sense of abstraction over to the computer program itself. While that is often a helpful, or at least harmless, way of thinking, it is at root a categorical error.

An actual, functioning computer program is literally a physical entity. It occupies actual physical space, in RAM, disk, or other media; while one computer program occupies some particular space, nothing else can be there. A functioning computer program consumes actual energy as it executes, producing waste heat that must be dissipated by a cooling system. It doesn't matter if the same amount of waste heat is produced by an operating CPU regardless of what program happens to be running, what is essential is that when some particular program is running on some particular CPU, the energy that is consumed is consumed at the behest of that program.

What distinguishes digital software from most other organizations of matter is that in a computer it can be copied so quickly and easily at high fidelity; DNA molecules in a cell of course have the same property. The flip side in both cases is that either can also be easily erased. Computers provide an additional feature for software that cells at least in principle could provide for DNA but to my knowledge do not: The ability to transduce losslessly between the relatively stable matter-based representation and ephemeral, fast-moving wave forms.

For internetworked computers only the physical boundaries between separately owned and administered systems are truly fundamental. Despite the much-touted *non*-spatiality of 'cyberspace', driven by precisely that lossless transduction, in fact each piece of hardware—each computer and disk, each wireline and switch—is locallized in space, and each piece of hardware has an owner. For small personal computers no less than huge corporate computing facilities, physical access and legal ownership are the two key elements defining a player in the game.

The evolutionary mess Establishing and maintaining those boundaries, today, is a disaster. Modern computer security amounts to a porous hodge-podge of corporate firewalls, third-party virus scanners, hastily written and sporadically applied operating system patches, and a myriad of woefully inadequate password authentication schemes. Dozens or hundreds of break-ins occur daily, computer virus infections are everyday life for millions of computer users, and new virus detections are booming, even as more and more businesses and business transactions move to the internet, and as software companies race with each other to deliver new ways of embedding code inside data.

Though this situation is far from what one would expect to find in a thoughtfully engineered, deployed, and maintained system-of-systems, it is precisely what one expects to find in systems produced by blindly reactive evolutionary processes. For example, it is too easy simply to lay blame for computer viruses on the early mass-market computer designers, even considering the body of knowledge available from the era of time-sharing. On the contrary, the tale of the PC and the virus is one of evolution in action: When the machine was designed, there were essentially no viruses in the wild—there was no wild to speak of—and code exchanges were either in large system-administrator-managed mainframe environments or in the tiny computer hobbyist community. Why would anybody waste design and manufacturing resources, increase costs greatly, and sacrifice time-to-market, just to defend against a non-existent problem?

Having humans in the loop, with all our marvelous cognitive and predictive abilities, with all our philosophical ability to frame intentions, does not necessarily change the qualitative nature of the evolutionary process in the least. Market forces are in effect regulated evolutionary forces; in any sufficiently large and distributed system, nobody is in charge, and evolutionary forces are constantly at work.

Living computation: Examples

This section provides a few examples drawn from the author's work, illustrating ways that living computation principles can be applied in the implementation of near-term software systems, and how some of the analytical tools of the life sciences can find applications in understanding the expanding software systems around us.

For	*Primary purpose*
End users	Peer-to-peer security-aware chat system / graphical MUD
'Agents' and artificial life	Scalable distributed environment with human interactions
Real artificial life	Investigate life-like computation and communications strategies

Select version dates		*v0.4+ algorithms*	
v0.4.0=	3/30/00	Authentication:	El Gamal
v0.2.6=	8/ 4/98	Encryption:	Twofish
v0.1.91=	9/15/97	PKI:	included
v0.1.80=	4/11/95	Transport:	TCP/UDP
v0.1.24=	10/26/92	Addressing:	IPv4+PKI

File counts and code size (by `wc`*)*

Content	*Files*	*KLines*	*%tot*	*%new*
C code	721	187.3	49.1	57.0
history	19	50.4	13.2	92.9
documentation	80	34.4	9.0	60.7
Perl code	73	33.0	8.6	100
development	153	24.0	6.3	41.5
Tcl/Tk code	37	18.1	4.7	100
assembly code	145	15.2	4.0	0.0
C++ code	43	10.4	2.7	93.4
uncategorized	24	6.7	1.7	1.2
ccrl code	19	2.3	0.6	100
empty	1	–	0.0	–
Total	1315	381.7	100.0	64.9

Table 1: Views of the **ccr** research prototype: Purpose, history, technology, and 'genetic' content. '%new' refers to code developed within the project rather than acquired from the environment.

Living computation by design

For several years, we have been building a series of research prototypes to explore 'life-like' design strategies for networked computations. This **ccr** project overall has been introduced previously (Ackley, 1996); here we provide only the briefest overview, then draw a few examples from the current system design, and then use the software itself as a sample object of study.

At its core **ccr** is a code library for peer-to-peer networking with research emphases on security, robust operation, object persistence, and run-time extensibility. Built upon the core libraries are the graphical user interface world **ccrTk**, and the text-only world **ccrt**. Table provides views of the system along several dimensions: Its major functions, history, and technology, along with a breakdown of the current contents of the system software. Figure 1 visualizes the system in the linear 'tar file' format in which it normally reproduces. The 14Mbytes in the 'genome' are depicted in terms of 'coding' regions that are the actual source code; 'promoter' regions contain metadata guiding the migration of code segments during early development; 'garbage' regions are simply wasted by the tar file format. Successive 'zooms'

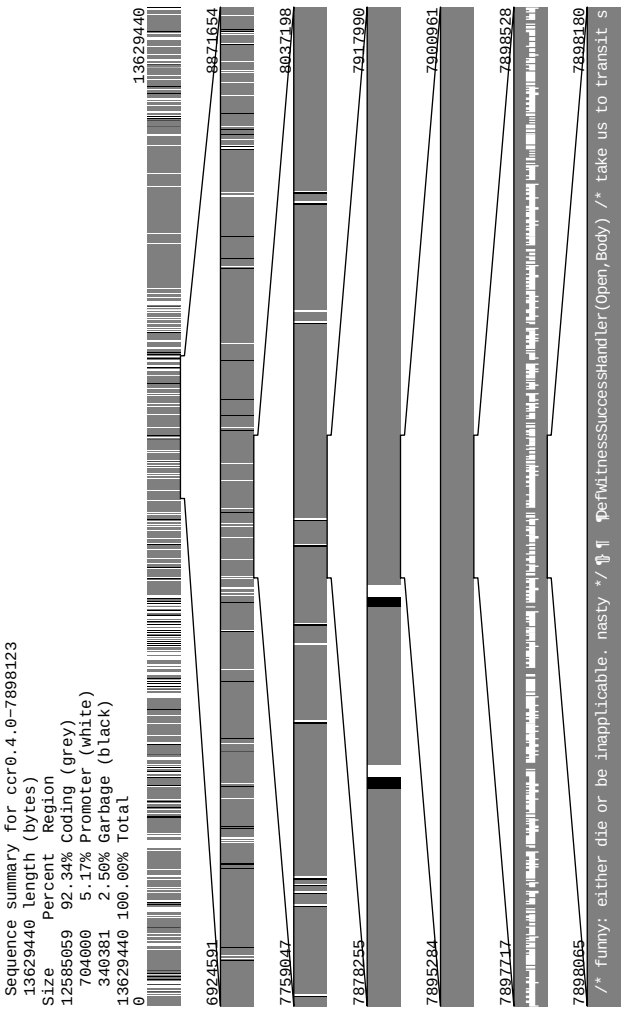

Figure 1: Views of the ccr v0.4.0= genome

in Figure 1 across some five orders of magnitude ultimately resolve individual bytes, illustrating the scale of a modest-sized system like **ccr**.

A fundamental design element of **ccr**, arguably a *sine qua non* for an independent living system, is its peer-to-peer communications architecture: A **ccr** 'world' *can* communicate with other **ccr**worlds, but it doesn't *require* any other worlds to function; each **ccr** world is both 'client' and 'server' in interactions with other worlds.

Self-reliance Taking seriously the independent living system approach implies that we must make as few assumptions about the 'outside world' as we reasonably can, and be as self-reliant as we can. This caution extends through all levels of the system and along multiple dimensions. One approach would simply avoid all external influences, but when it comes to communications

and communications risk, fundamentally we are damned if we do and damned if we don't: We might have had warning of a threat in time to avoid it, had we been allowing external factors to affect system behavior.

Most work in computer-based communication has focused on efficiency and power rather than safety. High-speed communications protocols, for example, can now supply data as fast or faster than most processes are ready to digest it, and even the ubiquitous TCP/IP internet protocol requires reacting to a connection attempt even just to ignore it, leading in part to the 'denial of service' attacks currently occurring on the internet. Similarly, nearly all software for personal computers focuses heavily on adding ever more programmed abilities and 'features,' even as automated network access is woven more deeply into the system rather than being more isolated, with the predictable results that your own computer's processing and data can be stolen out from underneath you by anyone simply by 'speaking' to your machine in particular ways.

Natural living systems have a large variety of mechanisms for evaluating interactions, assigning degrees of trust, and allowing only limited influences in proportion to estimated risk. Complex chemical signals and hard-to-duplicate bird songs, for example, increase confidence that messages are genuine; between **ccr** worlds cryptography serves that purpose, among others, helping on the one hand to establish identity and increasing (in particular) the sender's cost to generate a valid message on the other hand.

Ritualized interactions such as mating behaviors allow gradual and mutual stepping-up of trust and acceptable risk as confidence in identities and intentions grows. A **ccr** world wishing to communicate, likewise, engages in protocols designed to capture as much of the value of communication as possible while exposing the world to as little risk as possible. Here, we describe a few of those protocols and mechanisms, to provide concrete examples of living computation design strategies in artificial systems.

A cautious "Hello World" The protocol by which **ccr** worlds establish a communications link with each other moves through several stages, with a gradually increasing 'message size limit' allocated to the connection as the stages are successfully negotiated. Note that all of the strategies discussed here are in addition to the mechanisms provided by the TCP and IP version 4 transport mechanisms. Initially a **ccr** world will read only small messages, of no more than 128 bytes, from an incoming connection. Such messages are sufficient to exchange version information and establish a cryptographic 'session key' for the connection, which both establishes identities and insulates the channel from eavesdroppers and intruders. Any attempt to send a larger message causes the connection to be cut at the receiving end.

If this initial stage succeeds, more trust is warranted, and the incoming message size limit is raised to 1Kbyte, which is enough to complete the connection establishment protocol. At the successful conclusion of the 'greeting ritual', the message size limit is raised to 100Kbytes. Note that while that number is high enough for most typical channel uses, it is much less than it could be. Higher limits, if desired, can be set by deliberate act of the **ccr** world owner. This strategy is typical of **ccr**'s self-protection mechanisms. Even once a remote **ccr** world is identified and the communications channel secured, still only limited trust is granted to the channel because inconvenient or dangerous things still may happen, either due to a user's mistake, or to malicious intent, or to bugs in the code.

Friction as friend To help guard against such possibilities, in addition to the message size limit, a **ccr** world also places rate limiters on every established network connection, to reduce the risk of accepting and acting on network communications, whatever they may be. The *inbound bandwidth limiter* specifies the maximum average rate in bytes/second that one world is willing to read data from another world, with an effective default of 2Kb/s, which is enough to allow most normal channel usage to flow unimpeded. If more than 2Kb/s is supplied, the receiving world simply *doesn't read it* until enough time has passed that the overall bandwidth doesn't exceed the set bandwidth limit.

In addition to the communication bandwidth control, a **ccr** world also maintains a *processing bandwidth limiter* for each world that is in contact. For two worlds to communicate meaningfully, it is necessary that data sent from one world *somehow* affect the processing that occurs on the other world; therein also lies the risk. Although **ccr** uses several mechanisms to control what operations remote worlds can perform locally—for example, by controlling the language used to express the messages the receiving world will choose to read—here we focus only the processing time control. Each operation a **ccr** world can perform has an associated cost in terms of "work units". Each computation request received from another world is tagged locally with the identity of the requesting world, and as processing proceeds, work units are logged against the remote world. As with the bandwidth limiter, if processing on behalf of a remote world exceeds a specified rate, then the local world delays accepting further input from that world until the overall processing rate drops within established limits.

With a default of 20 work units/world/second, once again most normal inter-**ccr** operations are at most minimally impacted by the limiter. In pathological situations, however, the protection they afford can be significant. Figure 2 illustrates their effect via two simulated denial-of-service attacks launched by World 'B' against World 'A'. World 'A's data appears in the upper graph

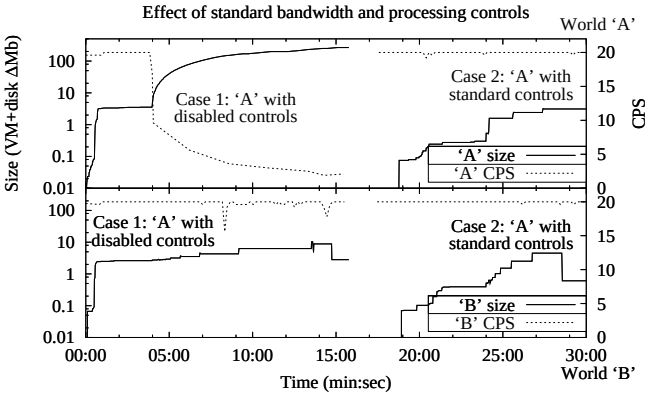

Figure 2: Using rate-limiters on bandwidth and processing time to mitigate denial-of-service events. *Case 1:* World 'B' floods world 'A'—with defenses disabled—starting at 00:00. *Case 2:* World 'B' floods world 'A'—with defenses in place—starting at 18:00. See text for details.

and World 'B's in the lower graph; in both cases the solid lines represent memory usage and are measured in megabytes of growth (the left side y-axis labels), and the dotted lines represent the currently main loop processing rate in cycles/second (the right side y-axis labels). A rate of 20CPS is the target 'heartbeat' rate of a **ccr** world; it will 'sleep' if it doesn't require the full 50ms to complete all scheduled tasks.

Before the first attack, which starts at zero minutes into the displayed Case 1 data, World 'A's limiters were effectively disabled by setting the acceptable rates to extremely large values. 'A' grows by several megabytes quickly and then stabilizes for several minutes, and then shortly before the 5:00 mark it begins growing exponentially and its cycle rate crashes. The simulated attack was stopped shortly after 15:00 minutes, at which point 'A's size had exceeded 200Mb and it had nearly exhausted the swap space on its machine. Both worlds were then restarted and the attack was repeated, this time with the normal values for the limiters. Now in Case 2 'A's growth rate is slower, and remains under 10Mb, and it turns a steady 20CPS throughout the event: The controls are performing effectively.

The growth behavior displayed by World 'B' through the two events was somewhat unexpected: It was much *less* different between the two cases than anticipated. The 'attack' was performed by instructing a character in world 'B' to 'speak' 10Kb strings of random numbers approximately 50 times per second. In both cases 'B's size gradually grows. In Case 2 'B' grows because 'A' is deliberately delaying reading from 'B', to protect itself, while 'B' continues to speak, causing 'B's 'pending output' buffers to expand. In Case 1 the same effect occurs, but there it is because 'A' is in severe distress from memory thrashing and processing overload, and its cycle rate has crashed, so there also it is not reading from 'B' as frequently. In both cases 'B's size eventually drops, when the communication channel was closed by yet another watchdog within the system. That mechanism injects 'Are you alive?' messages into communications streams at random intervals and times how long the response takes; if no response is received after several minutes the connection is killed.

Software genetics

The amount of code in the world is exploding, as is the amount of code in any given program. Today, essentially all application programs take advantage of prewritten libraries of code—at the very least the runtime library of the chosen programming language(s), and usually many other existing components as well, for graphical interfaces, database access, parsing data formats, and so forth. The analogy to natural genetic recombination is quite strong: Computer source code as genome; the software build process as embryological development; the resulting executable binary as phenotype. The unit of selection is generally at the phenotypic level, or sometimes at the level an entire operating system/applications environment.

A main place where the analogy breaks down is that in manufactured computers, but not in the natural world, there are two distinct routes to producing a phenotype. The extreme 'copy anything' ability of digital computers means that source code is not required for to produce a duplicate of a phenotype. Source code *is* a requirement, in practical terms, for significant evolution via mutation and recombination.

Commercial software is traditionally distributed by direct copying of precompiled binary programs while guarding access to the 'germ line' source code, largely to ensure that nobody else has the ability to evolve the line. In that context, the rapidly-growing corpus of 'open source' software is of particular interest. With source code always available and reusable by virtue of the free software licensing terms, an environment supporting much more rapid evolution is created. The traditional closed-source 'protect the germ line at all cost' model is reminiscent of, say, mammalian evolution; by contrast the free software movement is more like anything-goes bacterial evolution, with the possibility of acquiring code from the surrounding environment and in any event displaying a surprising range of 'gene mobility', as when genes for antibiotic drug resistance jump between species. There is therefore reason to expect open source code, on average, to evolve at a faster rate than closed source, at least up to some level of complexity depending on design where the chances of new code being useful rather than disruptive become negligible.

As software systems grow, and software components swallow each other and are in turn swallowed, and older 'legacy systems' are wrapped with new interface layers and kept in place, we are arriving at the situation where actually *reading* fragments of source code tells us less and less about how—if at all—that code ever affects the aggregate system behavior. As this trend accelerates, tools and techniques from biological analysis are likely to be increasingly useful.

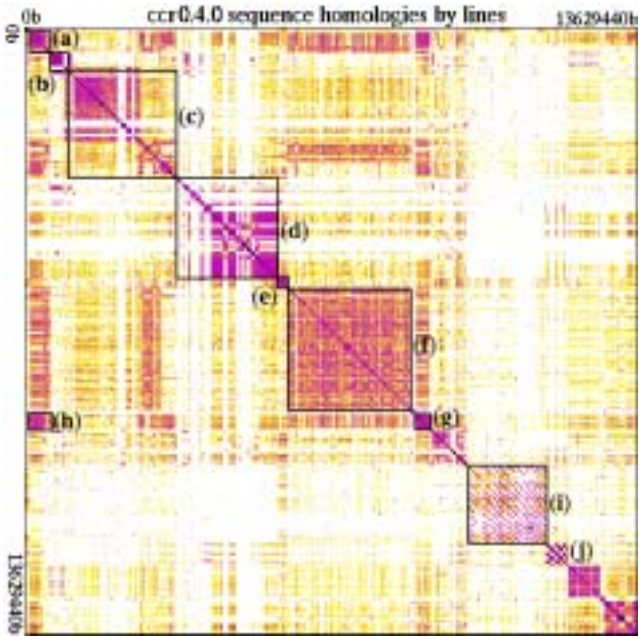

Figure 3: ccr genome vs itself. Darker represents greater homologies. See text for discussion.

Figure 3 presents another view of the **ccr** genome, using the 'dotplot' program (Helfman, 1996) for visualizing large data sets. This view shows the **ccr** genome plotted against itself, using lines of code as the fundamental unit of similarity; there is a black line representing perfect overlap down the main diagonal. Dark areas significantly off the main diagonal represent similarities between widely-separated code regions; squares on the diagonal represent 'cohesive' regions with more similarity within than without. 'Looking under the hood', we find that often such regions either are or are components of larger functional units—'genes'—within the genome. Several such genes have been highlighted with black outlines: Region *(a)* codes for **ccr**'s web server/client program; *(b)* is the configuration system that guides the overall system ontogeny; regions *(c)–(e)* are separately-evolved code segments (for JPEG images, long integer manipulation, and regular expressions, respectively) that have become incorporated, essentially unchanged, into **ccr**. The large region near the middle of the genome *(f)* contains the **ccr** core components themselves. Region *(g)* deals with processing animated images; interestingly,

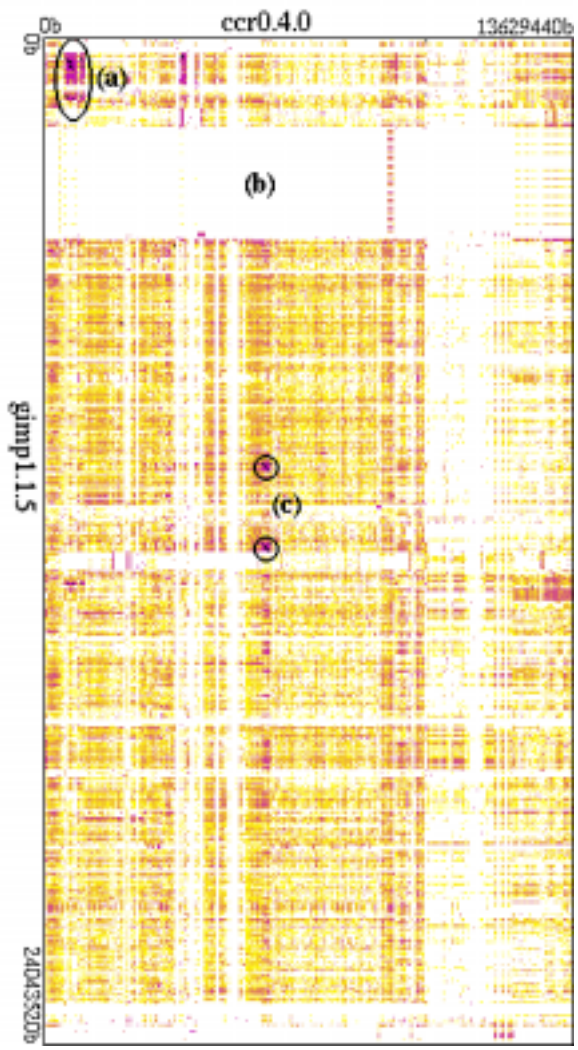

Figure 4: The **ccr** genome vs the 'gimp' graphics package genome. See text for discussion.

it shows a perceptible overlap *(h)* with the web code *(a)* even though their functions are very different and in fact they are expressed in different languages (C++ *(a)* vs C *(g)*)—but, it turns out, both were created by the same programmer. Semi-automated project historical notes *(i)* display a distinctive pattern, as does program-generated Postscript documentation *(j)*.

Figure 4 compares **ccr** to a different 'species'—the 'GNU Image Manipulation Program' (GIMP) (Mattis and Kimball, 2000). There are fewer dark regions, reflecting a generally lower degree of similarity between the code sequences. The dark region *(a)* is aligned with Figure 3*(b)* and reveals that both systems use evolved variants of the same 'autoconf' developmental control system—though the region is rectangular indicating that GIMP's instantiation of the code is bigger than **ccr**'s. A large 'internationalization' segment *(b)*—allowing GIMP to operate in some eleven natural languages—is strik-

ingly different than almost everything in **ccr**.

A surprising element in this comparison are two short lines *(c)*—black diagonals indicating identical sequences. The relevant sequence is the GNU regular expression package, which is used in both **ccr** (Figure 3*(e)*) and the GIMP, and is a good example of a highly useful gene incorporated into multiple different applications out of the free software environment. *Two* short black lines, aligned vertically, show that the GIMP contains two identical copies of the GNU regex gene. Rather than being wasteful—as traditional software practices might have construed it—such gene duplication reduces epistasis and increases evolvability; one copy is deep in the application core and the other in a relatively peripheral 'plug-in scripting' segment. Furthermore, from one point of view it's odd that this gene appears at all, because regular expression support is actually a required part of any POSIX-conforming operating system. Yet, like complex living systems, both **ccr** and the GIMP acquired captive regex 'organelles' instead, reducing their vulnerability to environmental variations and increasing their ecological range.

Real artificial life: The Future

I have argued that the similarities between living systems and actual computational systems are too overwhelming to dismiss. I've suggested that many of the differences between manufactured computation and natural living systems, both superficial and substantive, have arisen from the complementary circumstances surrounding the origins of the two technologies, but that both approaches must address the same imperatives and are therefore on converging evolutionary paths.

If these arguments even mostly hold up, then we can predict major changes in future architectures of manufactured computation. A recurring theme here has been that many of the defining claims for digital computation and communication—ranging from 'instant communication' to 'frictionless commerce' to 'location transparency', and possibly even the notion of 'general-purpose computing'—simply are too good to be true, having been purchased at the expense of utterly ignoring the basic tenets of self-versus-other and local self-reliance. It will not continue this way. Even though many parties would like to have control over individual hardware systems—ranging from software and hardware manufacturers to internet and application service providers to governments and regulatory agencies—in the end the geometry of physical space will assert itself over 'cyberspace' as computing systems become aware of themselves and their universe.

What we can do The hypothesis is that mass-market computer communications systems, at least, will become more and more like natural living systems. The scope and nature of that evolution is far from clear, and artificial life research and researchers can contribute significantly to the process, bringing fresh technical, biological and philosophical perspectives to the growth and development of the network, a process which only with great naiveté can be regarded simply as engineering. Alife models developed for natural systems can be and will increasingly need to be applied to the hardware, software and data of the internet. Biological principles, hypotheses, and scaling laws may find analogs in the growing computational ecosystem. There is much to be done.

Where we may be Over sixty years of development, computer programs have grown from a few bytes to hundreds of megabytes, from a few lines of assembler source code to tens of millions of lines of complex programming language code. We have been living with the 'software crisis'—which usually means rapidly increasing software development and maintenance time and cost, often with decreasing reliability—now for several decades, and a number of proposed solutions have come and gone.

Especially over the last fifteen years, 'object-oriented programming' (Meyer, 1988; Booch, 1994, and many others) has emerged in various forms as a durable programming methodology. There are debates over technical details, and factionalism surrounding specific object-oriented programming languages, but the overall approach continues to gain design wins for more and larger projects when significant new code is needed.

From the living computation perspective, one interpretation of that history is difficult to resist. In coarsest outline the arc of software development paralleling the evolution of living architectures: From early proteins and autocatalytic sets amounting to direct coding on bare hardware; to the emergence of higher level programming languages such as RNA and DNA, and associated interpreters; to single-celled organisms as complex applications running monolithic codes; to simple, largely undifferentiated multicellular creatures like SIMD parallel computers. Then, apparently, progress seems to stall for a billion years give or take—the software crisis.

Some half a billion years ago all that changed, with the 'Cambrian explosion' of differentiated multicellular organisms, giving rise to all the major groups of modern animals (Gould, 1989, for example). Living computation hypothesizes that it was primarily a programming breakthrough—combining what we might today view as object-oriented programming with plentiful MIMD parallel hardware—that enabled that epochal change.

Where we may be is in the leading edge of the Cambrian explosion for real artificial life. If so, there is of course no certainty, from our vantage point today, how or how quickly the process will play out. On the other hand, in this interpretation we are aligning perhaps three billion years of natural evolution with perhaps a century of artificial evolution.

We are living in interesting times.

Acknowledgments

Many of the ideas in this paper have been developed and elaborated in collaboration with Stephanie Forrest; but for exigencies of time and deadlines she would have been a co-author. The **ccr** development group at UNM has included Adam Messinger, Brian Clark, Neal Fachan, Nathan Wallwork, Suresh Madhu, Peter Neuss, Steve Coltrin, and Jeff Moser. This work was supported in part by an NSF research infrastructure grant, award number CDA-95-03064, and in part by the DARPA Intelligent Collaboration and Visualization program, contract number N66001-96-C-8509.

References

Ackley, D. H. (1996). ccr: A network of worlds for research. In Langton, C. and Shimohara, K., editors, *Artificial Life V. (Proceedings of the Fifth International Workshop on the Synthesis and Simulation of Living Systems)*, pages 116–123, Cambridge, MA. The MIT Press.

Booch, G. (1994). *Object-Oriented Analysis and Design With Applications*. Addison-Wesley Object Technology Series. Addison-Wesley, 2nd edition.

Cohen, F. (1987). Computer viruses. *Computers & Security*, 6:22–35.

Dawkins, R. (1991). Viruses of the mind. On the web 1/10/2000 at http://www.santafe.edu/~shalizi/Dawkins/viruses-of-the-mind.html.

Dennett, D. C. (1978). Why you can't make a computer that feels pain. *Synthese*, 38(3):415–456. Also in *Brainstorms: Philosophical Essays on Mind an Psychology*, Bradford Books.

Dertouzos, M. L. (1997). *What Will Be: How the New World of Information Will Change Our Lives*. Harper Edge, San Francisco, USA.

Forrest, S., Hofmeyr, S., Somayaji, A., and Longstaff, T. (1996). A sense of self for unix processes. In *Proceedings of the 1996 IEEE Symposium on Computer Security and Privacy*. IEEE Press.

Forrest, S., Somayaji, A., and Ackley, D. H. (1998). Building diverse computer systems. In *Sixth Workshop on Hot Topics in Operating Systems*.

Gould, S. (1989). *Wonderful life: The Burgess Shale and the nature of history*. W.W. Norton.

Helfman, J. (1996). Dotplot patterns: A literal look at pattern languages. *Theory and Practice of Object Systems*, 2(1):31–41.

Horowitz, E., Sahni, S., and Rajasekaran, S. (1997). *Computer algorithms/C++*. W. H. Freeman Press.

Kephart, J. O. (1994). A biologically inspired immune system for computers. In Brooks, R. and Maes, P., editors, *Proceedings of Artificial Life IV*, Cambridge, MA. MIT Press.

Kephart, J. O., Sorkin, G. B., Arnold, W. C., Chess, D. M., Tesauro, G. J., and White, S. R. (1995). Biologically inspired defenses against computer viruses. In *IJCAI '95*. International Joint Conference on Artificial Intelligence.

Langton, C. G. (1989). Artificial life. In Langton, C. G., editor, *Artificial Life (Santa Fe Institute Studies in the Sciences of Complexity, Vol VI)*, pages 1–47, Reading, MA. Addison-Wesley.

Mattis, P. and Kimball, S. (2000). The GNU Image Manipulation Program home page. At http://www.gimp.org/.

Meyer, B. (1988). *Object-oriented Software Construction*. Prentice Hall.

Sale, A. E. (1998). The colossus of Bletchley Park: The German cipher system. On the web 10/31/1999 at http://www.inf.fu-berlin.de/~widiger/ICHC/papers/Sale.html. Presented at the International Conference on the History of Computing, Paderborn, Germany.

Sober, E. (1992). Learning from functionalism: Prospects for strong artificial life. In Langton, C. G., Taylor, C., Farmer, J. D., and Rasmussen, S., editors, *Artificial Life II*, pages 749–765, Reading, MA. Addison-Wesley.

Spafford, E. H. (1992). Computer viruses—a form of artificial life? In Langton, C. G., Taylor, C., Farmer, J. D., and Rasmussen, S., editors, *Artificial Life II*, pages 727–745. Addison-Wesley, Redwood City, CA.

Stein, L. A. (1998). What we've swept under the rug: Radically rethinking CS1. *Computer Science Education*, 8(2):118–129.

Vatis, M. A. (1999). Statement for the record before the House Science Committee, Subcommitte on Technology. Comments by the Director of the National Infrastructure Protection Center. On the web at http://www.house.gov/science/vatis_041599.htm.