Biologically-Inspired Design: Getting It Wrong and Getting It Right

Steve R. White
IBM Thomas J. Watson Research Center, P.O. Box 704,
Yorktown Heights, NY 10598 srwhite@watson.ibm.com

Abstract. Large, complex computing systems have many similarities to biological systems, at least at a high level. They consist of a very large number of components, the interactions between which are complex and dynamic, and the overall behavior of the system is not always predictable even if the components are well understood. These similarities have led the computing community to look to biology for design inspiration. But computing systems are not biological systems. Care must be taken when applying biological designs to computing systems, and we need to avoid applying them when they are not appropriate. We review three areas in which we have used biology as an inspiration to understand and construct computing systems. The first is the epidemiology of computer viruses, in which biological models are used to predict the speed and scope of global virus spread. The second is global defenses against computer viruses, in which the mammalian immune system is the starting point for design. The third is self-assembling autonomic systems, in which the components of a system connect locally, without global control, to provide a desired global function. In each area, we look at an approach that seems very biologically motivated, but that turns out to yield poor results. Then, we look at an approach that works well, and contrast it with the prior misstep. Perhaps unsurprisingly, attempting to reason by analogy is fraught with dangers. Rather, it is critical to have a detailed, rigorous understanding of the system being constructed and the technologies being used, and to understand the differences between the biological system and the computing system, as well as their similarities.

1 Introduction

There is no doubt that computing systems are complex. They are arguably the most complex artifacts ever produced by humans. As computing systems become ever more complex, we naturally look to other fields to understand what tools and techniques we might bring to bear on the problems that we encounter. Computer scientists have long looked to mathematics, and even physics, for algorithms and methodologies. We have also, though perhaps less often, looked to biology.

But biological systems are quite different from computing systems, often radically so. We would not want to build a computer that counts on its fingers, or types on a keyboard. Why, even the term “biologically-inspired design” should make us a little nervous. We do not refer to the “mathematically-inspired design” of computing systems, or even to “physics-inspired design.” We refer to mathematical algorithms, or techniques borrowed from physics, that help us design better computing systems. What, then, is the role of biological inspiration?

In the remainder of this paper, we turn our attention to three problems in computing systems in which people have used biology as an inspiration to understand and construct computing systems. The first is the epidemiology of computer viruses, in which biological models are used to predict the speed and scope of global virus spread. The second is global defenses against computer viruses, in which the mammalian immune system is the starting point for design. The third is self-assembling autonomic systems, in which the components of a system connect locally, without global control, to provide a desired global function. In each problem area, we look at an approach that seems very biologically motivated, but that turns out to yield poor results. Then, we look at an approach that works well, and contrast it with the prior misstep. Finally, we summarize the reasons why one biologically-motivated approach fails, while another succeeds. We conclude that reasoning by analogy is dangerous, but that a deeper understanding of the differences, as well as the similarities, between biological and computing systems can help us avoid the pitfalls of biologically-inspired design.

This is a cautionary tale.

2 Computer Virus Epidemiology

Ever since Len Adleman coined the term “computer virus” to describe a self-replicating program [1], the temptation to use biological analogies for them has been overwhelming. Computer viruses authors have used techniques such a “polymorphism,” in which a virus changes its form with each succeeding generation in an attempt to evade detection, in much the same way as certain biological viruses mutate rapidly to evade the body’s defenses. Anti-virus programmers developed techniques such as looking in files for bit strings that were found in known viruses but not in normal programs, much like the mammalian immune system produces cells that bind to viruses but not to cells in the body.

There has also been an overwhelming temptation to use models of biological virus spread to model computer virus spread. This temptation is understandable. Both kinds of viruses infect individuals, whether they are mammals or computers. Both spread from one individual to another via infection vectors, whether it is sneezing or sending files via email.

In the late 1980’s, when computer viruses first became a serious problem, very little was known. Viruses spread on diskettes, which became infected when used on an infected computer and which could spread the infection when used on other computers. But little was understood about their global spread. In 1998, Peter Norton, later of Norton AntiVirus fame, was alleged to have said that computer viruses were an urban myth, "[…] like the story of alligators in the sewers of New York. Everyone knows about them, but no one's ever seen them."

2.1 Getting It Wrong

In 1991, Tippett asserted that the spread of computer viruses was like that of bacteria in a Petri dish – that without outside intervention their growth would tend to be exponential [2]. Though there was not a rigorous model behind these statements, they were based on the well-known fact that many population models exhibit exponential growth in their early phases. The reason for this is easy to see. The first infected individual might spread the infection to two other individuals, who in turn spread it to four more, and so on. The spread will be approximately exponential until a large fraction of the population is infected, at which time the infection will continue to spread, but more slowly due to the lack of uninfected targets. Assuming that everyone in the population is susceptible to the infection, the virus will ultimately infect 100% of the population.

Armed with this alarming prediction, Tippett and others called for emergency action, fearing that a worldwide pandemic was just months away. But as early data on worldwide virus infections became available, it became clear that there were problems with this model. Virus spread was nowhere close to exponential. In fact, it was surprisingly slow. Few viruses that were collected by anti-virus companies were ever seen in real-world infections, and even those that were took up to a year to become worldwide problems. Viruses never reached 100% of the population, even after a fairly long time. In fact, their prevalence would reach a peak of at most a few percent of the population, and then it would *decrease* [3].

2.2 Getting It Right

Two features of this simple model of infection are easily seen to be problematic. First, there is an assumption that infected individuals remain in the population indefinitely and continue to spread the infection. But infected computers do not stay infected forever. If the virus causes system problems, and most viruses did, users would be highly motivated to get rid of the virus. They might use anti-virus software, if it was clear that it was a virus. They might replace their boot records, which would have gotten rid of most boot viruses. They might have reformatted their hard drives and started over. Ultimately, users would have gotten rid of their computers and moved to new computers. Few of us are still using the computers that we used in 1990!

Second, there is an assumption that every individual is susceptible to infection. But, as the computer virus problem became worse, and more people started using anti-virus software regularly, this was no longer the case. Indeed, as anti-virus software gained the ability to actively prevent computer viruses from running on a system, and to stay up to date with the latest threats, many computers became immune to a virus before the virus could ever reach them.

A third feature of this simple model is perhaps more subtle. The model assumes that any infected individual has an equal chance of infecting any other individual. The model essentially assumes that the population is trapped in an elevator for several months, and that anyone sneezing has as much chance of infecting one of the individuals as another. In biological epidemiology, this is known as the "homogeneous mixing" assumption [4].

For rapidly-spreading diseases, such as influenza, in populations with a high degree of contact, such as cities, this is still a pretty good model. But computer users did not exchange diskettes in this pattern. They exchanged diskettes relatively infrequently, and often only within a group of close co-workers. Diskette exchanges between random people in the world occurred very infrequently. It turned out that the topology of how an infection may spread was a critical, and previously overlooked, feature of a successful model.

Kephart *et al.* described models of the epidemiology of computer viruses that had a rigorous basis and took these features into account [5, 6]. In these models, standard biological epidemiology was used to describe individuals who were susceptible, those who were infected and contagious, and those who were both cured and immune. Individuals who were susceptible could become infected, and could later become both cured and immune. The models supported standard epidemiological results such as epidemic thresholds: if the virus is killed off faster than it spreads, there is no epidemic. This was a likely explanation for the observation that most viruses were never seen in real-world infections. They were too inept at spreading or never got the chance. Similarly, epidemics in the model never reached 100% of the population. They were killed off by disinfecting infected computers or by preventative measures.

Instead of assuming homogeneous mixing, Kephart *et al.* modeled infections as spreading on a directed graph, in which the nodes were computers and the arcs denoted a pathway by which a particular computer could infect another. In very sparse graphs, which are likely the correct model for diskette-based virus spread, it was harder for an epidemic to start and easier for it to die off. In highly clustered graphs, representing more diskette sharing inside workgroups and less between them, viruses that were rampant in one part of the graph seldom leaked out to other parts of the graph, explaining the observation that some university computer labs had rampant, ongoing infections while more controlled environments rarely did.

It turned out that virus epidemiology in computer systems bore deep and striking similarities to the biological world. The same models could represent viruses in both worlds. The thing that distinguished the directed graph models from the "exponential growth" model was that it was not a case of reasoning by analogy. It had a rigorous mathematical basis and an explicit set of assumptions that could be validated in the real world. It was inspired by biology, but grounded in the actual system at hand.

3 A Digital Immune System

The mammalian immune system is very complex, and has evolved over millions of years to protect individuals against a very large and ever changing array of threats. It has a number of mechanisms, both innate and adaptive, to find and destroy foreign organic material that may pose a threat to the body.

The immune system is an obvious place to look for inspiration in combating cyberspace threats such as computer viruses. Before we do so, let us look at some of the mechanisms that it uses.

When viruses enter the bloodstream, some of them are engulfed and destroyed by macrophages (white blood cells), which then present antigens (proteins from the bacterium or virus) on their surface. Cells called T cells are capable of recognizing particular antigens by binding to them chemically. There are a vast number of T cells in the bloodstream, and collectively they are capable of recognizing a vast number of different antigens. When a particular T cell recognizes an antigen, it is stimulated to reproduce, so there are more T cells to find instances of that virus. It is also stimulated to produce antibodies that bind to the antigens on the surfaces of the virus. Viruses that are coated with antibodies are easier for macrophages to ingest. So, in response to an invading virus, the immune system produces a huge number of antibodies that help kill off that particular virus.

T cells that happen to recognize proteins found in the body are weeded out at an early stage in their lives, so the body does not (usually) produce antibodies against itself. Only T cells that might recognize viruses and do not recognize the body's cells (the "self") are allowed to circulate [7].

3.1 Getting It Wrong

We begin with the problem of detecting a computer virus in the first place.

Forrest *et al.* suggest a method for detecting computer viruses that is very strongly rooted in the mammalian immune system [8]. Given a set of files that they want to protect on a PC, they divide the files into a collection of bit strings of a fixed length, say 32 bits long. They then generate 32-bit "detector" bit strings at random and discard those that have a match to the strings that make up existing files on the PC. This is very much like the immune system creating T cells, and weeding out those that attach to proteins in the "self." Forrest *et al.* calculate the number of non-self detector strings that will be needed in order to detect new or changed files (i.e. non-self files) with a given probability.

A sufficient number of non-self detector strings is then generated, and the PC is scanned periodically to determine if any of the non-self strings are found within the files. This is very much like the immune system spreading T cells throughout the body, and reacting to any of them binding to non-self proteins.

So far, this is very plausible. It is a general method for detecting changes in the system, that is, files that have come to look different than they were to begin with. This could well indicate the presence of a computer virus.

Let us examine what it would take to implement this on a typical PC today. In doing so, we will make assumptions that more strictly parallel biology than Forrest *et al.* might advocate. The model that they report allows, for instance, only approximate matching of detector strings to strings in the files of the PC as a way of increasing its efficiency. Here we will assume exact matching.

In the experiments involving this method, the authors typically assume a rather high probability of failing to detecting a change. 0.02 is a typical probability that is used. We will set a stricter standard, as biology does. Let us suppose that we want to detect changes on a typical PC and we want the probability of failing to detect a single-bit change in one of the 32-bit strings in the files to be less than 2^{-32}. This is not an unreasonable bound, given that a typical PC with 100GB of storage has $\sim 2^{40}$ bits on it which, if we separate these into 32-bit strings for this method, yields 2^{35} such strings. If we randomly change all 2^{35}, we would only fail to detect 1000 of the changed strings.

Using the equations developed in [8], we estimate that we will need nearly 10^{11} detection strings that are each 32 bits long to achieve the required detection probability. If this were the mammalian immune system, that would be a small number of T cells. In a computer, however, that many detection strings would require nearly 400GB of storage, which is more than a typical PC has these days. Plus, scanning for the presence of 10^{11} detection strings would take quite some time!

Functionally, this is a method of determining if new files have been added to the PC, or if existing files have changed. Let us consider another method of accomplishing this goal. Suppose we calculate a 32-bit hash, or checksum, for each file on the computer, and store it away along with the path and filename of the file. This will allow us to detect changes in files with a failure rate of 2^{-32} per file. To be fair, the two detection methods are not functionally identical. The hash method can detect deleted files, whereas the detector method cannot. The detector method has a higher probability of detecting multiple-bit changes. Nonetheless, it is an instructive comparison.

If we keep 4 Bytes (32 bits) of hash per file, and ~32 Bytes of path and filename information, we need ~36 Bytes per file. A typical PC might have $\sim 10^5$ files on it, so we need less than 4MB of storage for our hash database. If all we want to do is detect changes to files on our PC, this is a much more economical way to do it.

This surprising economy is not available to the mammalian immune system. While it is easy to implement hash functions for files on a computer, it is difficult to think of a way that evolution could have provided a hash function for protein sequences, or even what such a hash function would look like. Biology has vast numbers on its side, so producing billions of T cells is a natural approach. Computing has much more strict limits on its resources, but much more flexibility in its computations.

A closer examination of the assumptions made by the detector model reveals a curiousity. It assumes that the makeup of the "self" that the method defends is constant. That is, it assumes that strings that initially matched strings in the "self" will match them in the future, and that strings that did not match strings in the "self" will not match them in the future. This is a good assumption in mammals, where the proteins that are expressed on the surfaces of cells are determined by the organism's genetic makeup, and do not vary over time.

It is not a good assumption, however, in computer files, which change all the time for valid, benign reasons. New files are created, existing files are updated, and old files are deleted. There is no static "self" in computers. Just because a file changes does not indicate the presence of a virus. Quite the contrary, the number of files that change due to viruses is much, *much* smaller than the number that change for benign reasons. In this case, the computer world is very different from the biological world.

In a subsequent paper based on this same approach [9], Somayaji *et al.* state:

> "Although we believe it is fruitful to translate the structure of the human immune system into our computers, ultimately we are not interested in imitating biology. Not only might biological solutions not be directly applicable to our computer systems, we also risk ignoring non-biological solutions that are more appropriate. A more subtle risk, however, is that through imitation we might inherit inappropriate 'assumptions' of the immune system."

This is the ongoing risk of biologically-inspired design.

3.2 Getting It Right

If we cannot rely on a distinction between self and non-self to recognize computer viruses, how can we recognize them? Perfect recognition of computer viruses – determining that an arbitrary program is a virus and never making a mistake – is equivalent to the halting problem [1, 10]. Nevertheless, there are a variety of heuristics that, in practice, turn out to be remarkably effective. Many viruses are variants of older, known viruses, and can often be found by scanning for strings that are found in known viruses but that are unlikely to be found in normal programs. Many viruses use a few common tricks, like self-encryption to attempt to hide from scanners, so noticing that a program uses one of these tricks may lead us to suspect it of being a virus.

Unfortunately, all of these heuristics have false positives – they occasionally accuse a perfectly normal program of being a virus. It would be bad if the system acted on this accusation without being sure, erasing the accused file or, worse, attempting to remove the "virus" from the file.

In a system described by Kephart *et al.* [11, 12], heuristics were used to identify files that might contain a virus, and a copy of these files was sent to a central virus analysis lab. Here, an important difference between biological and computing systems was exploited. In biological systems, lots of things replicate themselves: DNA, viruses, our body's cells and entire organisms. Self-replication is one of the most important capabilities of all life. In computing systems, however, almost nothing that is really useful undergoes self-replication. Almost without exception, if it self-replicates, it is a computer viruses, and hence it is undesirable. So the virus analysis lab isolated the suspect virus in a virtual machine and tried to make the virus self-replicate. If it did, it was indeed a virus.

Multiple replicas were gathered, so that the system could take into account any variation between them. The replicas were analyzed, and strings were extracted that detected all of the replicas but were very unlikely to be found in normal programs. The goal of this latter step was much the same as the goal of the immune system in producing T cells: create something that will recognize the virus but will not also recognize good cells/files. Because it would be infeasible to follow biology closely and test the string against every file that exists or will exist on the Earth, a statistical characterization of a large collection of normal programs was used to estimate the probability that the string would be found in any normal program. Only strings with extremely small probabilities were used. At the same time, an algorithm for disinfecting the file – for removing the virus and returning the file to its original state – was derived.

Once these detection strings and disinfection algorithms were extracted and tested, they were sent back to the infected system, which then used them as a highly specific way of finding and disinfecting that particular virus. At the same time, they were made available worldwide to protect computers that were not yet infected. In most cases, this was all done automatically, with quality that exceeded human analysis, and was complete from detection to cure in a few minutes.

While this process bears some resemblance to the way the mammalian immune system works, it is really very different. In fact, it bears more resemblance to an early 20th century theory of the mammalian immune system called "instruction theory," in which antigens themselves caused the formation of antibodies, but only after the antigen appeared and by somehow using parts of the antigen in antibody formation [13]. This theory was disproven shortly after it was proposed. But computers are not mammals, and mechanisms that work poorly in biology may be just the ticket in computing.

In computing, what constitutes "self" and "non-self" changes constantly, so observing a computer virus reproduce is one of the few sure ways to determine that it really is a virus and not just a normal program. Furthermore, crafting specific defenses for specific viruses works very well in computing system, where we cannot have billions of detectors for "non-self." Once again, we see how critical it is to understand the differences between biological and computing systems, as well as the similarities.

4 Self-Assembling Autonomic Systems

Since the first paper outlining the vision of autonomic computing [14], biology has been used as an analogy for how large computing systems should work. The autonomic nervous system plays an important role in regulating critical systems in the body – such as breathing, heartbeat, digestion, and eye focus – without involving our conscious minds. This lets our conscious mind focus on conceptual problems with fewer distractions. By analogy, autonomic computing seeks to create computing systems that are largely self-regulating, allowing system administrators to tell the systems what to do at a high level, and then have the systems themselves figure out how to do it.

The autonomic nervous system is one possible biological source of inspiration for autonomic computing. Let us examine another.

In the early stages of embryonic development in mammals, cells divide to form a blastocyst, a roughly spherical collection of cells that start out nearly identical. As development proceeds, these cells reproduce and differentiate to form structures, such as arms and a spine, based on their own genetic information and their local chemical environment. Remarkably, there is no central planning agent that tells the body how to develop.

Nevertheless, trillions of cells acting in their local environments manage to develop into extremely complex structures such as eyes, muscles and brains. Consider the circulatory system, which must carry blood to all parts of the body. How does the developing circulatory system know where to grow new capillaries? The answer, of course, is that it does not, at least in the sense that there is no centrally managed plan for where they should be. Rather, cells that are getting insufficient oxygen generate growth factors that stimulate nearby capillaries to grow. Thus cells in regions of the developing body that are not yet getting enough oxygen stimulate capillary development in that region until they are getting enough oxygen, at which time they stop [15].

The mammalian body has countless mechanisms that direct its resources to places and for purposes that most benefit the body. Mechanisms that enable distributed self-assembly of complex features are among the most powerful.

4.1 Getting It Wrong

Let us focus on one important aspect of self-assembly in computing systems: determining where in the system to put a new server that has become available. We have a large computing center, with many application environments. Each application environment is a collection of the computing resources needed for a particular application – for a web server, for instance, or a portfolio analysis application. We want to figure out into which application environment we should put our new server, and how best it can be used within that environment.

In the developing blastocyst, it does not matter where a new cell is placed. It develops according to what it senses of its local environment. Suppose we held slavishly to biology and did the same thing with our new server. There is no sense of “local environment” in our collection of application environments. Logically, they are all peers. So we let the server choose the first application environment that it finds in a directory of such things.

We will even credit the application environment with good sense about how to use the server. Perhaps it is asked to become a web server to handle more customer requests. Perhaps it becomes a host for that processor-intensive portfolio analysis to achieve more accurate results.

Of the biologically-inspired approaches that we have discussed so far, this one has the most obvious flaws. Clearly, choosing a random application environment into which to incorporate the server is unlikely to be the best choice. The chosen environment may be handling its traffic just fine, whereas another environment is starved for resources.

Servers are not cells. Cells reproduce, and their population can grow nearly limitlessly to serve the needs of the developing organism. Servers, on the other hand, are a highly constrained resource. Putting a server to work on one application often means that it is not available for another application.

This approach is *too* distributed. By not taking advantage of global information, in this case information about the value of an additional server to the various applications that might make use of it, we are stuck with a very suboptimal result.

4.2 Getting It Right

A traditional approach to allocating a new server is for system administrators to examine the various application environments in detail and plan out, quite a long time in advance, where that new server is most needed.

This can be made less burdensome for system administrators by allowing them to specify policies about how resources such as servers should be allocated. The web server application may be highly important for customer satisfaction, and the policy might be to allocate new servers to it until it is meeting its performance goals. Servers not needed for this application could be given to the less important but computationally intensive portfolio analysis application that can use as many servers as it can get.

Research is underway to imbue servers with the ability to incorporate themselves into an application environment, once the choice of environment is made. They can find the other resources that they need to operate and hook themselves up without the need for manual intervention by system administrators [16].

This idea can be extended to the dynamic operation of the system. Suppose, in our previous example, that the load on the web application varies, that it is high during the day and low at night. A global resource arbiter can be given a policy that instructs it to give as many servers as needed to the web application, but to move any servers that it does not need to the portfolio analysis application. We would see servers moved to the financial application in the morning, and then moved back to the computationally intensive application in the evening.

More generally, application environments can have a quantitative measure of the benefit that they could provide if given one or more additional servers. A global mechanism could then arbitrate between the environments to determine the best global allocation of all servers [17].

This keeps the best features of self-assembly while achieving globally optimal utilization of scarce resources. In large part, system administrators could be relieved of the burden of planning out, in detail, which environment should get which server at any given moment, and the burden of adding that server to that environment. Instead, administrators could set higher-level policies and let the system figure out how best to achieve them.

Again, this is not the way cells work in the body. Cells do not transform themselves from liver cells to brain cells when we are working on hard math problems, nor do kidney cells become muscles when we run. But principles like self-assembly from biological systems can be applicable to computing systems if we understand the differences between the systems.

5 Conclusions

In this paper we reviewed three areas in computing in which people have drawn inspiration from biology. In the first, computer virus epidemiology, we saw that simple analogies with biological virus spread do not capture essential features of computer virus spread, but that a rigorous and biologically-based model can. In the second, we saw that following the workings of the biological immune system too closely can result in an unwieldy and inaccurate technology for detecting computer viruses, whereas a deep understanding of how computers differ from biological organisms can lead us to a digital immune system that works extremely well. In the third, we saw that a simple analogy with self-assembling biological systems results in decisions about where to place a new server in a data center that are clearly wrong, while an understanding of how global information differs between biological systems and our data center helps us use the best features of biological self-assembly and avoid suboptimal solutions.

Biology does things for its own reasons. In the mammalian body, development must be consistent with evolution and the mechanisms available to it. We cannot grow a hand without growing an arm at the same time. And it must work with the materials available to it – cells but not electronic circuitry.

In engineering, we face different constraints. We are able to harness incredible computational power, but we do not get access to trillions of self-reproducing parts. Hence the solutions that we adopt in engineering will often be very different from the solutions adopted by biology.

Reasoning by analogy is dangerous. It tempts us to ignore the underlying assumptions that make a technique work in one field but fail in another. Instead, we must know the assumptions that are being made in both computing and biological systems. We must have a rigorous underlying model, preferably a mathematical model, of the systems that we are building. And we must know when a computing system does not behave like a biological system. In many cases, this knowledge can help us find a solution that is even better than those used in biological systems.

We can be inspired by biology. Indeed, we should be. Biology is very inspiring and can often lead to new ways of thinking about computing systems. But we must avoid the temptation of letting it dictate our designs.

6 Acknowledgements

The author would like to thank Jeffrey O. Kephart and David M. Chess for many insightful conversations.

7 References

1. F. Cohen, Computer Viruses, Ph.D. thesis, USC (1985)

2. P.S. Tippett, The Kinetics of Computer Virus Replication: A Theory and Preliminary Survey, Safe Computing: Proceedings of the Fourth Annual Computer Virus and Security Conference, New York, NY, March 14-15, 1991, p. 66-87
3. J.O. Kephart, S.R. White, Measuring and Modeling Computer Virus Prevalence, in Proceedings of the 1993 IEEE Computer Society Symposium on Research in Security and Privacy, Oakland, CA, May 24-25, 1993, p. 2-14
4. N.T.J. Bailey, The mathematical theory of infectious diseases and its applications, second edition, Oxford University Press, New York, 1975
5. J.O. Kephart, S.R. White, Directed-Graph Epidemiological Models of Computer Viruses, in Proceedings of the 1991 IEEE Computer Society Symposium on Research in Security and Privacy, Oakland, CA, May 20-22, 1991, p. 343-359
6. J.O. Kephart, S.R. White, D.M. Chess, Computers and epidemiology, Spectrum, IEEE, Vol. 30, Issue 5, May 1993, p. 20-26
7. R.A. Goldsby, T.J. Kindt, B.A. Osborne, Kuby Immunology, fourth edition, New York: W.H. Freeman, 2000
8. S. Forrest, A.S. Perelson, L. Allen, R. Cherukuri, Self-Nonself Discrimination in a Computer, in Proceedings of the 1994 IEEE Symposium on Research in Security and Privacy, Los Alamitos, CA: IEEE Computer Society Press, 1994
9. A. Somayaji, S. Hofmeyr, S. Forrest, Principles of a Computer Immune System, in Proceedings of the 1997 New Security Paradigms Workshop, Langdale, Cumbria, UK, 1997
10. F. Cohen, Computer Viruses: Theory and Experiments, in Minutes of the 7th Dept. of Defense / NBS Computer Security Conference, September 24-26, 1984
11. J.O. Kephart, G.B. Sorkin, M. Swimmer, S.R. White, Blueprint for a Computer Immune System, in Proceedings of the Seventh International Virus Bulletin Conference, San Francisco, CA, October 1-3, 1997
12. S.R. White, M. Swimmer, E.J. Pring, W.C. Arnold, D.M. Chess, J.F. Morar, Anatomy of a Commercial-Grade Immune System, in Proceedings of the Ninth International Virus Bulletin Conference, September/October 1999, p. 203-228
13. A.M. Silverstein, Darwinism and immunology: from Metchnikoff to Burnet, Nature Immunology, Vol. 4, No. 1, p. 3-6, 2003
14. P. Horn, Autonomic Computing: IBM's Perspective on the State of Information Technology, October 2001, http://www.research.ibm.com/autonomic/manifesto/ autonomic_computing.pdf
15. B. Alberts, A. Johnson, J. Lewis, M. Raff, K. Roberts, P. Walter, Molecular Biology of the Cell, fourth edition, Garland Science, New York, Chapter 22
16. D.M. Chess, A. Segal, I. Whalley, S.R. White, Unity: Experiences with a Prototype Autonomic Computing System, in Proceedings of the First International Conference on Autonomic Computing (ICAC'04), May 17-18, 2004, New York, NY, p. 140-147
17. G. Tesauro, R. Das, W.E. Walsh, J.O. Kephart, Utility-Function-Driven Resource Allocation in Autonomic Systems, Proceedings of the Second International Conference on Autonomic Computing (ICAC'05), June 13-16, 2005, Seattle, WA, p. 342-343