Hitchhiker's Guide *to Biomorphic Software*

The natural world may be the
inspiration we need for solving
our computing problems.

KENNETH N. LODDING, NASA

While it is certainly true that “the map is not the territory,” most visitors to a foreign country do prefer to take with them at least a guidebook to help locate themselves as they begin their explorations. That is the intent of this article. Although there will not be enough time to visit all the major tourist sites, with a little effort and using the information in the article as signposts, the intrepid explorer can easily find numerous other, interesting paths to explore.

To describe the biologically inspired software approach I have elected to use the term *biomorphic*, first coined by British zoologist Desmond Morris and popularized by the Darwinist Richard Dawkins in his book *The Blind Watchmaker*.¹ Another term that is sometimes seen is *biomimetic*, but that is more generally ascribed to something which mimics a specific biological behavior, as opposed to being more of a metaphor.

WHY BIOLOGY?

Perhaps the first question that comes to mind is, why a biomorphic model? Isn't software engineering more a matter of mathematics? Of logic and algorithms? Of direct cause and effect: if A, then B? Aren't biological organisms parts of the *messy* real world, a world in which behaviors emerge from the interactions of parts, rather than from being explicitly programmed into the individuals? A world where individuals following simple rules seem to build complex patterns and structures? A decentralized world frequently lacking leaders, and apparently not having blueprints, recipes, or templates to control pattern formation? The answer to all these questions is a resounding yes! It is exactly the messiness—the looseness of the distributed, decentralized behavior, pattern formation, and intelligence of the biological models—that makes biomorphic architecture applicable to many com-

Hitchhiker's Guide to Biomorphic Software

puting problems.

Consider the following facts. In nature, swarms of wasps work together without central control to build complex nest structures. Individual ants forage for food, and by laying down a chemical signal promote the emergence of an optimal path to the food source for other colony members to follow. As a survival mechanism, individual slime mold cells join together and act as a single, multicell organism to weather periods of famine. In each example, self-organizing groups of simple, cooperating individuals join together to perform complex operations in a distributed, parallel manner.

Furthermore, the behavior exhibited by the group is not directly encoded within the individual members, but emerges from the interactions of the members. Consider that the membership is not static but is an ever-changing group of individuals, as members are added and removed in an unpredictable manner. To make it even more interesting, there is no central controller explicitly orchestrating the action of the individual parts. And in the midst of all this seemingly chaotic interaction, still the organism forms, grows, lives, repairs itself, adapts, and survives. If only my computer environment were as successful. Perhaps it can be.

A LITTLE HISTORY

Biologically inspired computing is not a recent idea. Alan Turing, in a 1948 paper, "Intelligent Machinery,"² wrote about neural network-like architectures he called *unorganized machines*. The paper was unkindly called a "schoolboy essay" by his manager. In a 1950 paper, "Computing Machinery and Intelligence,"³ Turing suggested that natural selection and evolution could be mechanisms in the effort to construct intelligent machines. In 1975 John Holland published his pioneering book, *Adaptation in Natural and Artificial Systems*,⁴ and described the idea of the genetic algorithm.

Neural networks have a spotted history beginning around 1962 with Rosenblatt's perceptrons,⁵ and after initial setbacks (Marvin Minsky and Seymour Papert showed that neural networks had certain shortcomings⁶), reentered mainstream research in 1982 with John

Hopfield's Association Network.⁷ In 1986 Craig Reynolds⁸ became interested in the coordinated manner in which birds flocked and identified three simple rules from which flocking behavior automatically emerged from large groups of independent simulated birds, which he termed *boids*. These rules provided the basis for software developed to provide the computer-simulated bat swarms and penguins in the 1992 Tim Burton film, *Batman Returns*. In his 1992 Ph.D. thesis, Marco Dorigo⁹ introduced the idea of ant colony optimization to solve discrete optimization problems, a system inspired by the foraging behaviors of real ants.

The observation that many social insects work together to solve a problem, such as building a hive or finding food, has led to the swarm style of biomorphic software. Kennedy and Eberhart in 1995 developed particle swarm optimization,¹⁰ where large numbers of particles search through a problem space to swarm to an optimal solution. Research into social insect behavior, such as the termites (Pierre-Paul Grassé, 1959¹¹) and ants (Jean-Louis Deneubourg, 1989¹²) influenced a number of researchers, including Bonabeau, Dorigo, and Theraulaz at the Santa Fe Institute. In 1999 they published *Swarm Intelligence: From Natural to Artificial Systems*,¹³ describing the emergent problem-solving capabilities of large groups of simple, interacting agents, such as ants, bees, termites, and wasps.

The range of biomorphic software styles and the span of applications that employ these styles continue to grow on a daily basis. Example applications include physically distributed sensor networks that monitor their environment,¹⁴ swarms of pattern-searching agents detecting and classifying information in distributed databases,¹⁵ and mobile ants managing routing in telecommunications networking.¹⁶

SO WHAT MAKES BIOMORPHIC DESIGN "BIOMORPHIC"?

In a nutshell, biomorphic software is simply algorithmic design concepts distilled from biological systems, or processes. It is biologically inspired computing. The common thread that runs through all current biomorphic architectures is that they describe dynamic, self-organizing systems. In biomorphic architectures, problem solutions emerge as the result of the interaction of numerous, individual elements within the system, rather than from the application of some external mechanism or algorithm. Biomorphic programs are not coordinated from start to finish by a controlling, global plan. Instead, control arises from the interaction of the individual elements, each viewing its local part of the world at any given moment. Control within biomorphic programs is decentralized.

Program execution is distributed and parallel.

Having said that, we must note that in most instances there are no easily identifiable structural characteristics that will consistently define a particular design or architecture as having been biologically inspired. There are no biomorphic equivalents to the high vaulted ceilings, pointed windows, and buttressed walls of Gothic architecture that will immediately and definitively categorize a software architecture as biomorphic. There is, however, an alternative to classifying a computer system by its (internal) architecture or structure: examine the system's behavior.

An architecture reflects the designer's view of how best to implement a system, while system behavior reflects the designer's view of how the software should act or react to its environment. In general, we cannot know the behavior of a system by simply examining its structure, nor can we derive the structure of an application from its behavior. If we separate structure out and focus on the system behavior—a separation of form and function—we can ignore the implementation details and accept the potential existence of multiple means of accomplishing the same behavior.

A number of mechanisms have been identified¹⁷ that describe behaviors that appear to be indicative of biomorphic systems. These mechanisms include:

- **Collective interaction.** Behavior results from the collective interaction of similar, multiple, independent units, such as in a swarm.
- **Autonomous action.** Individuals act autonomously; there is no one "master" individual controlling the behavior of the others.
- **Emergence.** Behavior results—emerges—from the interaction of members, rather than being explicitly designed into the individuals.
- **Local information and interaction.** Individuals tend to operate from only local information and interactions. Their scope of view is spatially local, rather than global.
- **Birth and death.** The addition and removal of individuals into the group (i.e., birth and death) are expected events.
- **Adaptation.** Individuals have the ability to adapt to changing goals, information, or environmental conditions.
- **Evolution.** Individuals have the ability to evolve over time.

A MULTICELLULAR ORGANISM FOR PERSISTENT COMPUTING

Computing environments are notorious for their inflexibility when faced with failure or change. Specialized

design efforts, essentially add-on capabilities, must be made to provide an adaptable, fault-tolerant computing environment. As computer systems grow in size and complexity, so do their chances for failure. If your computer is onboard a remote exploratory probe, repair is highly unlikely.

In contrast, natural (biological) systems are generally characterized as being robust, fault-tolerant, and adaptive, all attributes that make them valuable as metaphors for computer system design. A recent project at NASA Langley Research Center focused on the development of a fault-tolerant, adaptable computing environment modeled after a multicellular organism. The biologically inspired model had the goal of defining a hardware-software computing *environment* that allows systems to continue to operate at *reasonable* levels of service in spite of (un)expected failures.

Before I describe the software multicell organism we prototyped for the persistent computing project, it will help to take a short side trip to describe the necessary biological details that define our model. Earlier I spoke of the difference between the architectural structure of an application and its behavior: form and function. In biology the same split is maintained in the genotype/phenotype distinction. The *genotype* is the set of genetic instructions encoded in the organism's DNA—the behavior, or function. The *phenotype* is the physical organism itself—the structure, or form. The genotype controls the development of the phenotype in a process called *morphogenesis*.

A cellular program consists of genetic instructions—

Hitchhiker's Guide to Biomorphic Software

genes—which are short pieces of the DNA. The cell executes the instructions, or in biological terms, *expresses* the gene sequence while replicating. The genome, consisting of the full set of genes of an organism, describes what the cell could possibly do. It is an unordered collection of instructions fully describing the final organism. What the cell actually does—its *fate*—depends on which genes are expressed, and that depends on which genes are activated or inhibited. Each gene in the DNA is protected by a lock mechanism called a *regulatory region*. For the cell to express the gene associated with a particular regulatory region, a protein key must unlock the gene. A gene sequence can have multiple locks, requiring a set of keys to open it up for expression. The pattern of gene inhibition and excitation defines a biological network whose state is determined as a result of a complex chemical cocktail that originates both from within the cell itself and as signals from other cells. In the case of the human, the initial parent cell undergoes approximately 50 cell divisions, creating 10^{15} cells in your body, of which there are about 256 different types, ranging from blood to bone cells and muscle to neural cells. With minor exceptions, each cell contains the information to become any one of the 256 or so types. The process of becoming a specific type of cell is called *cellular differentiation*.

As a cell differentiates toward its final fate, it is sometimes prevented from taking on particular roles by its immediate neighbors. In this process, known as *lateral inhibition*, a cell that takes on a certain function immediately places an inhibiting signal into the environment as part of the chemical cocktail to stop other cells from taking on the same function. When the signaling cell dies and its inhibiting signal disappears, the first

inhibited neighbor cell to detect this loss of signal can continue its previously halted differentiation process, replacing the dead cell, and then inhibit the remaining neighbors.

These biological details, captured in the following list, form the conceptual kernel of our persistent computing architecture:

- The genome defines all possible tasks any cell can perform.
- All cells have the identical genome.
- The pattern of gene activation and inhibition within the genome determines the particular cell task.
- Gene state (activated/inhibited) is determined by signals present in the environment.
- Cells communicate by diffusing signals into the environment.

The cell is the basic hardware component in our persistent computing architecture. Each cell is a fully autonomous general processor with local and global communications capability. The cells of our organism are not mobile, maintaining a fixed position when placed into the architecture. There is neither a requirement for the cells to be spatially adjacent, nor for a predefined topological layout. As would be expected in a networked architecture, however, some configurations are *better* than others, and this can influence where hardware cells with specialized capabilities are positioned in the organism.

In our software simulation, the local communications are modeled as line-of-sight infrared links, while radio fulfills the global communications role. The multicellular organism has no global clock, no shared (global) memory, no pre-assigned cell IDs, and no central brain. Simply, it is a highly parallel, distributed computing environment

Meteorological Data-Flow Representation Mapped to a Possible Organism

with an ad-hoc network configuration possessing the ability to evolve to its final form (phenotype) at runtime from its genotype information.

To code the genotype information (the behavior of the organism), we chose for our initial experiments to use a simple data-flow graph model. Thus, programming the organism consists of building a data-flow graph that expresses the behavior of the multicell organism in terms of processes and the flow of information between them. Figure 1 shows such a data-flow graph. In this example the desired behavior of the organism is that of a simple meteorological station. To accomplish this behavior, individual cells must take on the function of wind-speed, wind-direction, and air-temperature sensors; a meteorological data processor; and a radio transmitter.

The data-flow graph that describes this organism is compiled and converted into a software genome, which will be entered into each cell of the final organism. For performance considerations, we assume a minimum hardware configuration consisting of five cells, one for each possible gene that will be expressed. To provide a

reasonably robust computing environment, more than the minimum number of cells is provided. In our model, cells that are not expressing a specific gene sequence act as undifferentiated stem cells, executing a generic null task until needed. Figure 2 shows a hypothetical mapping of the meteorological application onto a multicell organism built of seven cells.

There are four major phases in the “life” of our cells (See table 1). The first phase is *bootstrapping* and occurs immediately upon the cell being powered up. In this phase, all cells execute a standard, default software genome, which basically consists of becoming a unique *self*, finding the extent of the organism, and waiting to differentiate.

The second phase consists of *inoculation and distribution* in which a single cell is inoculated with the software genome, which the organism is intended to execute. In our example this is the “meteorological station” genome. The cell that is inoculated with the new genome is tasked to distribute copies of it to all other cells.

After the new genome is distributed, the third phase occurs when each cell parses its copy of the genome and chooses a gene sequence to express: biologically this is cell differentiation, but we term it task differentiation. This is the most complex phase and can involve significant amounts of simulated chemical signaling as cells select gene sequences to express and attempt to inhibit other cells from the same selection. At this point the whole multicellular organism can seem to oscillate or ring as cells both sense and emit chemical signals to control the differentiation process and begin the execution portion of this phase. We have copied the DNA lock-and-key concept in our software genome. The genome “lock” describes the hardware/software capabilities required to execute the gene it protects. The “key” is cell-specific information describing the hardware and software capabilities of the particular cell. The task-differentiation process consists of parsing the genome and building a prioritized list of candidate genes for expression. Each cell decides which gene or genes from the prequalified list it will express based upon the chemical signals currently found in the environment. In the case in which there is no conflict, the cell simply places its inhibiting signal for this gene into the environment. If the particular gene is already inhibited, the cell selects the next candidate from the list. If two or more cells find themselves attempting to express the same gene simultaneously, the cells rest, wait a random length of time, and try again to gain ownership of the gene.

The fourth and final phase a cell may enter is *failure and recovery*. This phase occurs only when a cell dies and

TABLE 1 Phases of a Cell's Life

Bootstrap

- Generate a unique self-identifier.
- Establish the members of the local, line-of-sight neighborhood.
- Discover the current network topology.

Inoculation and Distribution

- Targeted cell receives new software genome.
- Cell ripples new genome through the organism to all cells.

Differentiation and Execution

- Cells parse the genome and evaluate current chemical signal values to determine which, if any, gene sequence to express. If necessary, all cells can default to a null gene sequence.
- Cells that have differentiated begin execution.

Failure and Recovery

- Cell death causes the loss of a unique chemical signal identifying the particular gene sequence that was being expressed.
- Remaining cells redifferentiate to pick up lost capability.

Hitchhiker's Guide to Biomorphic Software

the organism must effectively redifferentiate to continue functioning. Cell death is detected by the loss of a chemical inhibiting signal. Redifferentiation is accomplished by reparsing the software genome and results in either a previously inhibited cell coming online or an active cell picking up and adding the task to its workload. If a special capability is required, such as the ability to sense temperature, the system will reconfigure about a cell that can provide that capability. If there is no such cell, that gene expression will not occur. Conversely, in some failed configurations, a temperature-sensing cell may be required to express another gene, as well as the temperature task—that is, multitask. The organism still functions, although it might miss some data because of workload conflicts. This fundamental cycle is shown in figure 3.

LIFE IS HARD

Our software simulation has successfully demonstrated the idea of a multicellular organism as a design metaphor for providing a suitable architecture for a persistent computing environment. Although at this stage of the work, our application is rather toy-like, it has served to demonstrate the idea, as well as some of the problems a

more reasonable implementation will face. Some of the issues that need to be addressed in future work include the following:

1. **Scalability.** Moving the design upward several orders of magnitude will not be a simple exercise in program design. At this time, the size of the software genome directly reflects the complexity of the application being performed. In the prototype a simple list structure linked the individual genes together, and each cell had a complete copy of all executable code referenced by the full genome. This will not, and was not intended to, scale upward in any reasonable manner.
2. **Programming.** Programming the prototype multicellular organism was addressed using a simple data-flow approach and hand-compiling the information into the software genome. This will not be practical for real-world applications. What is required is a method to describe the high-level behavior of the desired organism that can then be compiled into a format suitable for inclusion in the software genome. A possible approach is to model a solution after the subsumption-style architectures implemented for robotic experiments. This breaks down a high-level behavior into a number of sub-behaviors, which themselves might be further recursively decomposed. In such an approach, the software genome might change from a data-flow list to a hierarchical tree of behaviors.
3. **Communications.** Related to scalability are communication issues. As both the complexity of the genome and the size of the multicell organism increase, there will be a need for more chemical signals to be dif-

fused into the organism's environment. Using a single continuous signal per inhibiting chemical is at best extravagant, and at worst undoable. A more appropriate mechanism will be needed for organisms composed of hundreds, if not thousands, of cells executing complex software genomes.

4. Synchronization.

Although our design postulates an inherently asynchronous operation, there are times when synchronous behavior of at least parts of the organism

will be required to provide a level of coordination.

Examples of required synchronization include multiple sensor cells simultaneously taking an environmental reading, or controlling complex behaviors such as walking. One approach to this problem could be the inclusion of pacemaker cells within the organism, similar to the autorhythmic cells that spontaneously generate time hacks to give the heart the ability to beat automatically. Nature provides an intriguing alternative approach in the near-perfect, self-organized, synchronized flashing found in some species of fireflies. Other chemical-signaling mechanisms also exist.

The obvious benefit of solving these problems is the ability to provide a persistent computing environment that is both robust and adaptable. It is robust in that it can handle multiple failures of component parts in any order. And it is adaptable because the software genome can be changed on the fly, altering the organism's behavior and goals.

PERSPECTIVES

Three simple rules controlling separation, alignment, and cohesion can produce a bird-like flocking behavior in a group of agents. Genetic algorithms mutate, crossbreed, and evolve until they are selected as the best of breed and permitted to live. Large numbers of ants follow trails of chemical smells and end up finding the shortest route as

a result of evaporation and a desire to follow their noses. Cells form multicellular organisms that are robust, adaptable, and highly fault tolerant. Pull a leg off a hydra, and it can grow back.

The common thread that runs through the fabric of biomorphic form is that behavior emerges from the interactions of generally large groups of cooperating, possibly simple, autonomous individuals. While a hierarchical nesting of groups may exist, there is a genuine lack of a single controlling part: no central brain; no global, shared memory; no master clock; and no single piece of code controlling the group behavior. Instead the group interacts and creates a collective behavior that cannot be produced or even found in any single individual. Behavior emerges from the interaction of the parts.

The desirable characteristics of the biologically inspired architectures are evident:

- They are robust. Failure of one or more individuals does not generally fault the group.
- They are adaptable. Biomorphic software can adapt to its environment in a number of ways, including evolving, learning, or swapping DNA.
- They can self-organize.
- They are distributed and parallel.
- They are built from simple units.

But there are problems in designing biomorphic architectures:

This short reading list provides a reasonable starting point for learning about biologically inspired computing. The list is neither complete nor definitive. It is highly varied in subject matter and presentation of information. A little searching will reveal many more titles that will add to your understanding and knowledge on these and related topics.

Camazine, S., Deneubourg, J., Franks, N., Sneyd, J., Theraulaz, G., and Bonabeau, E. *Self-Organization in Biological Systems*. Princeton University Press, Princeton: NJ, 2001.

Coen, E. *The Art of Genes: How Organisms Make Themselves*. Oxford University Press, New York: NY, 1999.

Bonabeau, E., Dorigo, M., and Theraulaz, G. *Swarm Intel-*

ligence: From Natural to Artificial Systems. Oxford University Press, New York: NY, 1999.

Kelly, K. *Out of Control*. Perseus Books, New York: NY, 1994.

Eberhart, R., Kennedy, J., and Shi, Y. *Swarm Intelligence*. Morgan Kaufmann Publishers, San Francisco: CA, 2001.

Kauffman, S., *At Home in the Universe*. Oxford University Press, New York: NY, 1995.

Kumar, S., and Bentley, P., eds. *On Growth, Form and Computers*. Elsevier Academic Press, San Diego: CA, 2003.

Solé, R., and Goodwin, B. *Signs of Life: How Complexity Pervades Biology*. Basic Books, New York: NY, 2000.

RESOURCES

Hitchhiker's Guide to Biomorph Software

- They can be difficult to scale.
- They can be difficult to engineer.
- They can be difficult to control.
- They can be difficult to comprehend. Approaches such as genetic algorithms produce solutions that can be so convoluted and obscure that we are forced to accept that "it works by magic."

Possibly the biggest problem facing the development of biomorphic architectures based upon emergent behavior is this: How do you program for appropriate emergent behavior? What language constructs, semantics, and syntax support emergent behavior? How are these tied to a particular application? How can we predict the behavior of the group based upon a program running on an individual? What techniques exist for describing the desired behavior of an organism and for placing constraints on acceptable behavior?

One approach to programming emergent behavior that is obviously consistent with the biological metaphor being used is to "evolve" the program. This approach is distinctly different from the more traditional and better-understood approach of "engineering" a solution. Evolving a program is more akin to tinkering. Engineers and tinkerers apply two very distinct and different approaches to get to a solution. Engineers plan and design, while evolution "as a tinkerer, works with odds and ends, assembling interactions until they are good enough to work."¹⁸ It is interesting that the results of evolving a solution share a number of good design principles with what are considered good engineering design principles, including modularity, robustness, and use of recurring elements. Perhaps we can evolve what might be termed "well-tinkered solutions."

Will the end user be aware that a biomorphic design approach has been used? Will it be obvious that the program was evolved? Or that the program is composed of large numbers of interacting entities? Will the user "see" a difference that would flag the fact that biologically inspired concepts were used? The answer is most likely no. We hope that the systems will perform in a way that meets or exceeds the user's expectations for performance and capability, but there should be no give away

as to how the magic is being performed. The individuals behind the green curtain should not be noticeable.

THE FUTURE

In trying to envision the future of biomorphic software architecture, it seems wise to heed the words of Dilbert's creator, Scott Adams, concerning prediction: "There are many methods for predicting the future. For example, you can read horoscopes, tea leaves, tarot cards, or crystal balls. Collectively, these methods are known as 'nutty methods.' Or you can put well-researched facts into sophisticated computer models, more commonly known as a 'complete waste of time.'"¹⁹

We are still at the learning stage in biomorphic software design. It is not clear at all how information technology will be affected by this new architectural style. That there is growing interest in the area is obvious.

A large number of conferences are focusing solely on biologically inspired computing concepts. They include: GECCO 2004 (Genetic and Evolutionary Computation Conference);²⁰ 2004 NASA/DoD Conference of Evolvable Hardware;²¹ ANTS 2004 International Workshop on Ant Colony Optimization and Swarm Intelligence;²² SIP'04 (Swarm Intelligence and Patterns) workshop session;²³ and ESOA'04 (Engineering Self-Organizing Applications).²⁴

A quick search on Google returns numerous hits for college courses teaching biologically inspired computing courses, such as evolutionary computation, swarm intelligence, evolvable hardware, neural computing, genetic algorithms, and the like.

Biomorphic concepts are entering the mainstream through both college-level and general readership publications (see Resources).

Learning, experimenting, and publishing are all parts of the initial development stages of an engineering concept. Some brave souls will rapidly move the basic ideas associated with biomorphic design into real-world applications. Picking the winners? See the Dilbert quote!

I hope that this brief look at biomorphic architectures will prompt others to become excited and involved in this work. From simple, single-cell slime molds working together for survival, through swarms of interacting social insects to the sophisticated multicellular organism with an immune system and ability to self-regenerate, nature provides us with many powerful models to emulate. We need only look. Q

REFERENCES

1. Dawkins, R. *The Blind Watchmaker*. W.W. Norton &

- Company, New York: NY, 1996.
2. Turing, A. Intelligent Machinery. In *Machine Intelligence*. Meltzer, B., and Michie, D., eds. Edinburgh University Press, Edinburgh: Scotland (1969), 3–23.
 3. Turing, A. Computing Machinery and Intelligence. In *Readings in Cognitive Science: A Perspective from Psychology and Artificial Intelligence*. Collins, A., and Smith, E. E., eds. Morgan Kaufmann, San Mateo: CA (1988), 6v19.
 4. Holland, J. H. *Adaptation in Natural and Artificial Systems*. MIT Press, Cambridge: MA, 1992.
 5. Rosenblatt, F. The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain. *Psychological Review*, 65, 6 (1958) 386–408.
 6. Minsky, M., and Papert, S. *Perceptrons: An Introduction to Computational Geometry*. MIT Press, Cambridge: MA, 1969.
 7. Hopfield, J. J., Neural networks and physical systems with emergent collective computational properties. *Proceedings of the National Academy of Sciences of the USA* 79, 4 (1982), 2554–2588.
 8. Reynolds, C. W. Flocks, Herds, and Schools: A Distributed Behavioral Model. *Computer Graphics* 21,4 (1987), 25–34.
 9. Dorigo M. Optimization, Learning and Natural Algorithms. Ph.D.Thesis, Politecnico di Milano, Italy, 1992 (in Italian). See also Dorigo M., Maniezzo, V., and Colnari, A. The Ant System: Optimization by a Colony of Cooperating Agents. *IEEE Transactions on Systems, Man, and Cybernetics-Part B* 26,1 (1996), 29–41.
 10. Eberhart, R., Kennedy, J., and Shi, Y. *Swarm Intelligence*. Morgan Kaufmann Publishers, San Francisco: CA, 2001.
 11. Grasse, P. P. La reconstruction du nid et les coordinations inter-individuelles chez *Bellicositermes natalensis* et *Cubitermes* sp. La theorie de la stigmergie: Essai d'interpretation des termites constructeurs. *Insect Societies* 6 (1959), 41–83.
 12. Deneubourg, J. L., Goss, S., Franks, N. R., Sendova-Franks, A., Detrain, C., and Chretien, L. The Dynamics of Collective Sorting: Robot-like Ant and Ant-like Robots. In *Simulation of Adaptive Behavior: From Animals to Animats*. Meyers, J. A., and Wilson, S.W., eds. MIT Press/Bradford Books, Cambridge: MA (1990), 356–363.
 13. Bonabeau, E., Dorigo, M., and Theraulaz, G. *Swarm Intelligence: From Natural to Artificial Systems*. Oxford University Press, New York: NY, 1999.
 14. Wokoma, I., Sacks, L., and Marshall, I. *Biologically Inspired Models for Sensor Network Design*. London Communications Symposium 2002; <http://www.eleceng.ucl.ac.uk/lcs/papers2002/LCS119.pdf>.
 15. Brueckner, S., and Van Dyke Parunak, H. *Swarming Agents for Distributed Pattern Detection and Classification*. Autonomous Agents and Multiagent Systems (AAMAS), 2002; <http://autonomousagents.org/ubiquitousagents/papers/papers/18.pdf>.
 16. White, T., Pagurek, B., and Oppacher, F. Connection Management by Ants: An Application of Mobile Agents in Network Management; <http://dsp.jpl.nasa.gov/members/payman/swarm/white98-co.pdf>.
 17. Wang, M., and Suda, T. *The Bio-Networking Architecture: A Biologically Inspired Approach to the Design of Scalable, Adaptive, and Survivable*. Symposium on Applications and the Internet (2001); <http://csdl.computer.org/dl/proceedings/saint/2001/0942/00/09420043.pdf>. Available from Network Applications, Technical Report 00-03, Department of Information and Computer Science, University of California, Irvine.
 18. Alon, U., Biological Networks: The Tinkerer as an Engineer. *Science* 301: (Sept. 2003), 1866–1867.
 19. Adams, S. *The Dilbert Future*. HarperBusiness, New York: NY, 1997.
 20. GECCO; see <http://gal4.ge.uiuc.edu:8080/GECCO-2004/>.
 21. NASA/DoD Conference on Evolvable Hardware; see <http://ehw.jpl.nasa.gov/events/nasaeh04/>.
 22. ANTS 2004; see <http://iridia.ulb.ac.be/~ants/ants2004/>.
 23. SIP '04; see <http://alfa.ist.utl.pt/~cvrm/staff/vramos/SIP.html>.
 24. ESOA '04; see <http://esoa.unige.ch/esoa04-cfp.html>.

LOVE IT, HATE IT? LET US KNOW

feedback@acmqueue.com or www.acmqueue.com/forums

KENNETH N. LODDING has 29 years of software design experience and is employed by NASA in the Data Analysis and Imaging Branch of the Systems Engineering Competency at the Langley Research Center in Hampton, Virginia. His work in biologically inspired software design has included developing biomorphic architectures for pattern searching, data swarming, and fault-tolerant computing. He is currently supporting the development of swarm algorithms for groups of wind-driven, remote exploratory vehicles. Lodding received a B.S. in computer science from New York Institute of Technology and is currently working on an M.S. in computer science.

© 2004 ACM 1542-7730/04/0600 \$5.00