Building Biologically-Inspired Self-Adapting Systems Extended Abstract

Yuriy Brun
Computer Science Department
University of Southern California
Los Angeles, California 90089, USA
ybrun@usc.edu

Biological systems are far more complex than systems we design and build today. The human body alone has orders of magnitude more complexity than our most-intricate designed systems. Further, biological systems are decentralized in such a way that allows them to benefit from built-in error-correction, fault tolerance, and scalability. Despite added complexity, human beings are more resilient to failures of individual components and injections of malicious bacteria and viruses than engineered software systems are to component failure and computer virus infection. Other biological systems, for example worms and sea stars, are capable of recovering from such serious hardware failures as being cut in half (both worms and sea stars are capable of regrowing the missing pieces to form two nearly identical organisms), yet we envision neither a functioning desktop, half of which was crushed by a car, nor a machine that can recover from being installed with only half of an operating system. It follows that if we can extract certain properties of biological systems and inject them into our software design process, we may be able to build complex self-adaptive software systems.

Biological systems' complexity makes them not only desirable to guide software design, but also difficult to fully understand. Thus one approach to building software similar to biological systems is by first building models of biology that we can understand. Then these models can guide the high-level design, or architecture of the software systems, resulting in systems that retain the model's fault tolerance, scalability, and other properties. Figure 1 shows the outline of the process of using a biological system to create a software design tool or technique.

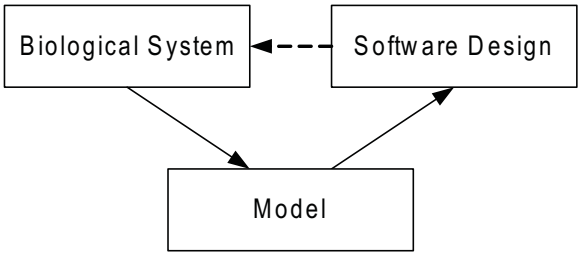

Figure 1. Outline of the process of using a biological system to create a software design tool.

Given a biological system, the designer must first create a model of that system. Some effort must go into studying and fully understanding the model. The purpose of the model is to prevent unwanted and poorly-understood properties of the biological system from affecting the software system. Once the designer has a solid understanding of the model, the process of creating the software design tool or technique can begin. The kind of design tools that may emerge from these models are likely to be high-level design paradigms that yield qualities of service such as fault tolerance, robustness, security, etc. Such design is most effectively approached from a software architectural perspective [5]. In particular, architectural *styles* [6] present generic design solutions that can be applied to problems with shared characteristics. Thus a likely result of the procedure represented in Figure 1 is an architectural style. Note that since the goal is to design some particular software system, or systems

Dagstuhl Seminar Proceedings 08031
Software Engineering for Self-Adaptive Systems
http://drops.dagstuhl.de/opus/volltexte/2008/1499

with particular qualities, that system or those qualities should play a role in selecting the underlying biological system and in creating the model. The most fruitful approach is most likely to iterate through the diagram several times, starting with a small model and building on its complexity.

I have gone through the exercise depicted in Figure 1 to create an architectural style called the tile architectural style. The tile style solves a particular software engineering problem, the discreet distribution problem. The internet's growth has created networks with great computing potential without a clear way to harness that potential to solve memory-intensive and processor time-intensive problems. Networks, such as the internet, can in theory solve NP-complete problems (and other problems for which we do not know polynomial time solutions) quickly, but as their individual nodes may be unreliable or malicious, users may desire guarantees that their computations are correct and are kept confidential. Several attempts at distributing computation over the internet have been successful (e.g., [3, 4]); however, these systems do not distribute the computation discreetly. The discreet distribution problem is the problem of distributing a computation on a large network without telling any small group of nodes on the network the problem it is helping to solve. I use a sample scenario to illustrate the problem. An espionage agency is attempting to break an RSA code sent by an enemy. The agency wishes to use a large network to factor the enemy's public key; however, it cannot allow anyone to know the key's factors or even whose key it is factoring. Since the agency has access to the internet, an incredibly large network of computers, it should be feasible to factor nondeterministically, or through brute force. However, the problem is to do so discreetly, without the nodes on the network learning the problem or the input.

The tile style leverages the nature's process of self-assembly. Thus it results in software systems that inherit nature's fault tolerance and robustness. Figure 2 outlines the result of applying the process from Figure 1 to this particular problem.

Instead of focusing on the details of the tile architectural style here, I wish only to present the process of how to create software design tools and techniques from biological systems. Details of the tile architectural style can be found here [2, 1]. Note

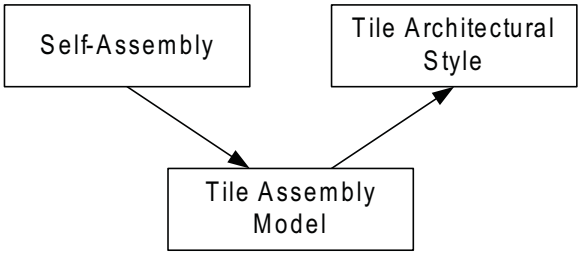

Figure 2. The tile architectural style is based on a formal mathematical model of self-assembly, the tile assembly model.

that the resulting systems may vary drastically from traditionally-designed software systems and may take a novel approach to achieving qualities of service, such as fault tolerance. These types of systems are in some ways different from the systems we are used to building, and thus we must develop tools not only to design and build them but also to compare and analyze them. While this is likely to require a significant amount of effort, the potential of software systems resembling nature's systems in complexity, self-management, and dependability makes the effort a worthy investement.

References

[1] Y. Brun and N. Medvidovic. An architectural style for solving computationally intensive problems on large networks. In *Proceedings of Software Engineering for Adaptive and Self-Managing Systems (SEAMS07)*, Minneapolis, MN, USA, May 2007.

[2] Y. Brun and N. Medvidovic. Discreetly distributing computation via self-assembly. Technical Report USC-CSSE-2007-714, Center for Software Engineering, University of Southern California, 2007.

[3] E. Korpela, D. Werthimer, D. Anderson, J. Cobb, and M. Lebofsky. SETI@home-massively distributed computing for SETI. *IEEE MultiMedia*, 3(1):78–83, 1996.

[4] S. M. Larson, C. D. Snow, M. R. Shirts, and V. S. Pande. *Folding@Home and Genome@Home: Using distributed computing to tackle previously intractable problems in computational biology.* Horizon Press, 2002.

[5] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, 1992.

[6] M. Shaw and D. Garlan. *Software architecture: perspectives on an emerging discipline*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.