IEEE DISTRIBUTED SYSTEMS ONLINE 1541-4922 © 2005 Published by the IEEE Computer Society
Vol. 6, No. 1; January 2005

Self Management: The Solution to Complexity or Just Another Problem?

Klaus Herrmann, *Berlin University of Technology*
Gero Mühl, *Berlin University of Technology*
Kurt Geihs, *Berlin University of Technology*

Traditionally, network and system management are manually controlled processes. It usually takes one or more human operators to manage all aspects of a dynamically evolving computing system. The operator is tightly integrated in this management process, and his or her tasks range from defining high-level policies to executing low-level system commands for immediate problem resolution. Although this form of human-in-the-loop management was appropriate in the past, it has become increasingly unsuitable for modern networked computing systems.

Several trends shaping the development of IT infrastructures have aggravated network and system management:

- The rapidly increasing size of individual networks and the Internet as a whole
- Hardware and software components' growing degree of heterogeneity
- The emergence of new networking technologies, such as ad hoc networking, personal area networks, and wearable computing, which are being combined with established technologies such as the Internet and cellular-phone networks

- Employees' growing need for mobile access to enterprise data

- The increased interdependency between business processes and the corresponding software products

- The idea of *disappearing computing*, which states that the growing number of computing devices pervading our everyday lives should be invisible to us

- The accelerated development of new technologies, which forces companies to restructure their IT systems more frequently

These trends indicate that IT infrastructures in large companies will grow even more complex in the future. How can we administer systems of this complexity adequately?

This question has stimulated a large interest in self-management technologies —technologies that help systems autonomously control themselves. It's not realistic for human operators to maintain control over a system that consists of thousands of networked computers, mobile clients, and numerous application servers and databases. We must redefine human operators' roles so that instead of being involved in the decision process in an interactive and tightly coupled fashion, operators define general goals and policies for system control.[1] But is self-management the solution? Only if we can first solve several open problems.

Current challenges

The Autonomic Computing Initiative (see the related sidebar) divides self-management into four functional areas:[2]

- *Self-configuration:* Automatically configure components to adapt them to different environments.

- *Self-healing:* Automatically discover, diagnose, and correct faults.

- *Self-optimization:* Automatically monitor and adapt resources to ensure optimal functioning regarding the defined requirements.

- *Self-protection:* Anticipate, identify, and protect against arbitrary attacks.

Because we can't realize these goals in one step, the ACI also defines five evolutionary levels: Level 1 (basic) is the current state of the art in network management, where human operators manage individual components more or less manually. Levels 2–4 define intermediate stages in the evolution of IT systems toward completely autonomous systems, which constitute Level 5 (autonomic).

Autonomic elements are an autonomic system's building blocks. An autonomic element consists of a *managed element* and an *autonomic manager* controlling the managed element. The managed element can be a single resource (hardware or software) or a combination of different resources. Essentially, an autonomic element consists of a *closed control loop* (Figure 1). Theoretically, closed control loops can control a system without external intervention and can keep it in a specified target state. This concept is vital for the ACI because it introduces the desired autonomy.

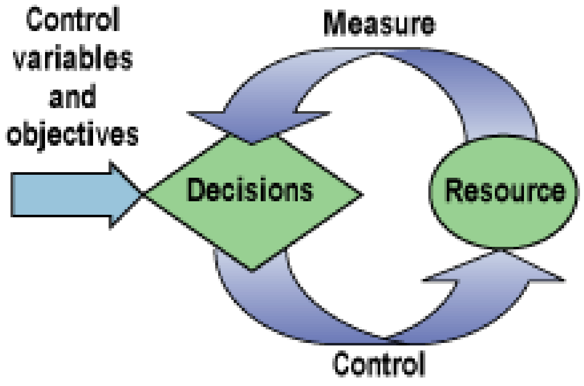

Figure 1. An autonomic element's closed control loop.

The vision of self-management at the autonomic level is far-reaching and presents many challenges. Researchers have already solved some problems by introducing techniques that help instrument and monitor resources[3] and that use autonomous mobile agents for proactive management.[4] However, many issues involved with rendering management tasks autonomous are still largely unresolved.

The biggest challenge is building closed control loops—the most important concept of self-management. The basic idea of control loops is well known from a wide variety of technical applications—a thermostat (consisting of a temperature sensor and a coupled flow control valve) and a car's antilock breaking system are just two examples. Closed control loops can control a system parameter on the basis of some predefined set point and constant observation of the parameter's current value (feedback). So, for a thermostat, a human manually defines the set point (desired temperature), and the thermostat measures the temperature and reacts by controlling the flow of heat (using the valve). An ABS detects if a wheel locks and reacts by reducing the breaking power on that wheel until it's spinning again. The fact that automotive companies spend considerable resources developing and calibrating their ABS systems for each car model shows that this is nontrivial in a more complex environment with numerous external influences.

However, modern distributed computing systems are several orders of magnitude more complex than our two examples. An ABS is tailored for a specific task for which all possible states and external influences are completely specified. In general, this isn't the case in computer systems because they usually comprise numerous heterogeneous components.

In fact, the name autonomic computing is inspired by the human body's autonomic nervous system, which controls bodily functions such as respiration and heart rate without requiring conscious action by the human being. This is exactly the aspiration behind the ACI with respect to controlling a distributed computing system's numerous software and hardware components. Application servers, databases, and communication infrastructures should control themselves without any manual, external intervention. The core problems in this respect are

- The lack of a widely accepted and concise definition of what self-management actually means
- The lack of appropriate standards for unifying the process of self-management, describing self-managing systems, and achieving interoperability in open distributed systems
- The absence of mechanisms for rendering self-managing systems adaptive and for enabling them to learn
- The fact that modern distributed systems tend to be inherently interwoven, which potentially creates an overwhelming complexity

Standardized definitions

The ACI's current definitions and notions lack the necessary clarity. Although the four functional areas represent a useful categorization of self-management subareas, the differences between them are somewhat fuzzy. For example, distinguishing between a faulty system and a system that's in a suboptimal state isn't always easy. While a faulty system would be subject to self-healing, a suboptimal system should self-optimize. What exactly distinguishes self-healing from self-optimization? And how do they relate to self-configuration? It's vital to answer such questions before comparing and categorizing the plethora of existing systems and approaches.

Other concepts that exist in the research community are self-adaptation and self-organization. Although we might intuitively regard them as being more general than the concept of self-management, finding a clear definition of these terms is extremely difficult if not impossible because the concepts behind them are still only partially understood.

Furthermore, we should compare the newly shaped concepts of self-management with the classic notions of dependable and fault-tolerant systems[5] as well as self-stabilizing systems.[6] These areas have already produced concepts and technologies that fall within the self-management domain. For example, leasing resources is a simple yet powerful and widely used mechanism for garbage collection in systems subject to partial failure. The system automatically frees a leased resource (such as a memory block) when the leaseholder (such as a process) doesn't actively renew it in time. So, the resource will always eventually be freed even if its holder crashes. Are lease-based systems self-healing or self-optimizing?

System description and interoperability

Autonomously managing a complex distributed computing system requires adequate means for describing the system as well as its current and desired states. This requires describing, for example,

- The system architecture
- Management policies
- Some kind of rule base for inferring concrete controlling measures from the currently perceived system state
- Service-level agreements and quality-of-service contracts that specify which requirements the system must fulfill

Formalisms and tools exist for each of these areas. However, we still need global standards to ensure interoperability, at least on a syntactic level. Some approaches exist in the research community to describe an adaptive system on the basis of Architecture Description Languages.[7–9] In the commercial sector, Microsoft promotes its Dynamic Systems Initiative,[10] which builds on the System Description Model. However, static descriptions aren't adequate for autonomous adaptive systems. Self-adaptation requires the system to also adapt the rules underlying the adaptation. For a system to be able to adapt to potentially unknown, dynamic environments, it must be able to learn while being applied in a productive environment. This, of course, also has consequences for potential description techniques.

Additionally, we need standards for capturing relevant data from a running system. This involves issues such as instrumentation, interface formats, and data exchange formats. Fortunately, existing standards can help, such as the Open Group's Application Response Measurement API[11] or the Distributed Management Task Force's Common Information Model.[12]

Learning systems

Human operators add a quality to management systems that current artificial systems can't match—humans can handle unknown situations and learn from their experiences. They can categorize newly emerging faults and integrate this new knowledge into their repertoire for the future. If we eventually remove the human operator from the management control loop, then the management system must be able to similarly learn from its experiences and improve its capabilities. The increasing dynamics of distributed computing systems and the growing tendency to regularly restructure them require a management system that's highly flexible and adaptable. The system must not only adapt quickly to an existing static system but also autonomously customize itself to frequently occurring changes.

Artificial systems have yet to achieve the degree of dynamic adaptation and learning required for self-management. As a first step, Yixin Diao and colleagues have shown that you can optimize a running database system for e-commerce applications with respect to the system's response time.[13] They describe the dependencies of configuration parameters using a rule base. However, if the rules remain static, the system can't adapt to changes in its environment. Such a system doesn't exist in isolation. The complex interactions with other software and hardware components and dynamically changing usage patterns can produce a plethora of different working conditions. How does the system react if we add other applications besides the e-commerce system? How does the self-optimization module react if, for example, we need to optimize both memory usage and response time? Optimizing both of these parameters together probably isn't equivalent to optimizing both in isolation, so a new configuration and optimization problem emerges concerning the rule base.

Interwoven systems

In many respects, the comparison to the human body's autonomic nervous system is quite fitting. A healthy human body preserves an equilibrium involving all subsystems. The common goal of all the processes in our body is survival. To achieve this, nature has developed an enormous complexity that lets our body react appropriately to a wide variety of different external stimuli. Our immune system, for example, can successfully detect and eliminate even unknown, malicious substances. Moreover, in doing so, it strengthens its resistance to future infections. This system is a complex closed control loop with numerous subordinate control loops including temperature regulation, respiration, and metabolic functions. Many of those systems interact in ways that we still don't understand. An attempt to isolate one subsystem to study and understand it better normally fails because in doing so we ignore important interactions with other subsystems. This is the nature of a complex system: The whole is more than the sum of its parts. A classic reductionist approach isn't guaranteed to lead to a better understanding of the overall system.

Modern distributed computing systems also exhibit an interwoven structure. Different subsystems such as operating systems, communication networks, and application components depend on each other in many ways. A large portion of these dependencies are indirect and neither explicitly planned nor obvious. It's a commonly perceived problem that the overall system tends to behave in ways that we don't expect when analyzing the subsystems in isolation.

This leads to a fundamental problem for self-management. According to IBM's vision, an autonomic system might comprise thousands of closed control loops in different subsystems.[14] Separate control loops might control different aspects of the same subsystem. For the purpose of a modular design and the separation of concerns, it seems sensible that each manufacturer of a hardware or software component is in charge of developing the control loops for its own components. However, owing to the overall system's interwoven structure, these control loops will influence each other. Changes to one subsystem imply direct or indirect changes to other subsystems because they change the current system state that's perceived by numerous separate control loops.

For example, let's assume that Diao's system[13] improves the database's response time by employing a control loop that increases the size of the in-memory buffer for caching data if necessary. Let's further assume that, concurrently, the operating system runs a control loop for optimizing memory usage. This control loop monitors the amount of free memory, and if it detects a shortage, it uses a standard application-side interface for telling applications to reduce their memory usage. This system, consisting of the database, operating system, response-time control loop, and memory control loop (depicted in Figure 2), can produce an undesirable oscillation. The two optimization criteria directly conflict with each other: Caching large amounts of data in memory achieves a good response time, while keeping memory usage low results in higher response times. One action triggers the other, so they'll repeatedly increase and decrease the size of the in-memory database cache. This thrashing will likely decrease overall system performance considerably and might increase response time. So, the response-time control loop triggers an even greater increase of the cache size to react to this condition. This feedback loop will eventually lead to a new emergent behavior of the system: complete failure.

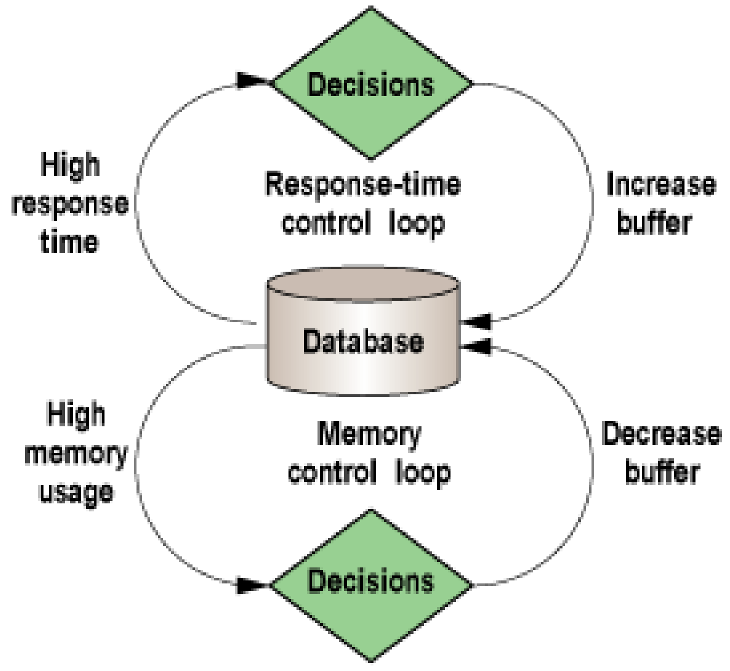

Figure 2. Indirect interaction between two separate control loops, causing undesirable oscillation.

The two control loops of our fictional example must be able to reach an agreement. We need to combine the optimization criteria that were previously independent into a compromise that's suboptimal with respect to the individual criteria. How can we achieve this? Do the two control loops have to "know each other" in advance, or is a direct or indirect mutual adaptation possible? Does this mutual adaptation scale with the number of control loops in the system? How can the system detect and handle hidden dependencies that extend over several indirections? How can we ensure that the "collective optimization criteria" of all control loops lead to a desirable system state? What effects do faulty control loops have in the presence of feedback effects that are possibly nonlinear (such as the oscillation build-up in the example)?

The interwoven structure of modern distributed computing systems is definitely the most important yet least-studied problem of self-management. It's unclear if real-world systems can be partitioned into completely independent subsystems to handle their interwoven structure adequately. This would require a complete analysis of the running system in the face of frequent changes. Even if such a separation were possible, the resulting independent subsystems might still be too large and their internal dependencies too complex. Static dependency checks based on architectural descriptions (similar to those found in modern compilers) are insufficient because computer systems evolve dynamically over time and their structure is subject to frequent changes.

Current initiatives and new approaches

Many current initiatives related to self-management take a practical approach, hoping to produce results fast. For example, several approaches to building closed control loops for online optimization exist. Diao and colleagues have demonstrated that a database system can self-optimize by describing parameter interdependencies in a rule base and by running an online optimization algorithm.[13] The simplex-based online optimization finds new parameter settings and applies them to the running system. The optimization algorithm uses the system reaction to evaluate the new parameter settings' quality. Sandeep Uttamchandani and his colleagues also use a rule base for deriving management actions.[15] The management system records and uses system behavior implications to select appropriate rules for the following actions. Boudewijn Haverkort employs a mix of online and offline optimization by testing possible adaptations against a system model before actually applying it to the running system.[16]

These systems demonstrate that self-optimization and self-configuration for isolated subsystems, with respect to specific properties, is possible. Of course, they rely on accurate system descriptions and powerful, static rule sets. Adaptive rule sets are still an open issue. Moreover, the case studies presented involve a single isolated subsystem, such as a database or proxy server.

Rejuvenation[17] is a more radical approach that builds on controlled, possibly partial reboot or reset of a system. It uses Markov-chain-based fault prediction to select the right time for rejuvenation and thus for self-repair.

It's impossible to present a complete overview of self-management-related systems, but these approaches represent the mainstream. Applying prediction algorithms and online optimization techniques to achieve closed-loop management is possible with current technologies, although this application has yet to prove its success in large-scale interwoven systems.

To create truly autonomous systems, scientists must consider much more radical approaches and shift to unconventional paradigms. Here, we briefly review some of these approaches for more intelligent IT solutions. Some are quite visionary and, at first glance, not in line with the goals we've presented. However, they point to valuable research fields that could lead to a better understanding of and eventually new approaches to self-management.

Process-control programming

In 1995, Mary Shaw introduced an alternative programming paradigm inspired by process control loops.[18] In this paradigm, the programmer doesn't divide a program into components in the sense of object-oriented programming. Instead, he or she uses abstractions known from control theory. A program is divided into a *process* and a *controller*. The process's interface provides access to the input and controlled variables and encapsulates the sensors for capturing relevant process variables. The controller contains the control algorithm and provides access to the set point. In Figure 1, the Decision module is the controller and the Resource module is the process. The controller and the process are connected and represent a dataflow architecture with feedback (controlled variable). Shaw shows that using these abstractions, we can more intuitively model a classical control process, such as a car's cruise control. Furthermore, this approach separates the main process's basic operation from the compensation for external disturbances. The task of building control loops in self-managing systems might require similar paradigms.

Self-stabilizing systems

Self-stabilization is an established area that shares some concepts with self-management.

Edsger Dijkstra first introduced the notion in 1974, defining a system as self-stabilizing if "regardless of its initial state, it is guaranteed to arrive at a legitimate state in a finite number of steps."[19]A self-stabilizing system can recover from arbitrary transient faults within a finite time, provided that no further faults occur before the system is stable again. In contrast to that, systems that aren't self-stabilizing might stay in illegitimate states forever, even if no further faults occur. Depending on the definition of "fault," classical self-stabilizing systems can be associated with self-healing or self-optimization. Self-stabilizing systems exhibit striking properties:

- They don't need any initialization because they stabilize from any (even illegitimate) state.
- They don't need to detect a fault to recover from it. Instead, self-stabilizing systems constantly push the system toward a correct state.
- They recover from arbitrary transient faults with a uniform mechanism.

Furthermore, general approaches exist to make algorithms self-stabilizing.[20] However, running self-stabilizing algorithms in parallel can raise problems similar to those that arise in the case of control loops. Shlomi Dolev has introduced sufficient preconditions for the composition of self-stabilizing algorithms:[20]

- The algorithms that are subject to composition must periodically have the chance to execute.
- There must not be a cyclic dependency between the algorithms' states.

In practice, guaranteeing the second condition is difficult owing to the interwoven nature of distributed systems. Recently, William Leal and Anish Arora[21] published some new ideas on how to achieve scalable self-stabilization via composition. Their approach is based on correlation and corruption relations that make the dependencies explicit.

Soft systems and homeostasis

Shaw also points to the fact that it's necessary to design software to be "soft" and flexible instead of "brittle" and rigid.[22] Traditionally, the system states upon which an application should act are completely specified and thus rigidly defined. Programs must be verifiable and are viewed as either correct or incorrect. In modern distributed systems, however, knowledge about the conditions under which a software component must run is inevitably incomplete. Programs that work under rigidly defined conditions are brittle and tend to be inflexible regarding acceptable working conditions. Slightly different conditions cause such programs to fail. Shaw thus introduces the notion of *sufficient correctness*—the definition of the current and desired state and of normal and faulty behavior should be fuzzy rather than precise. We need a region of degraded but still acceptable behavior to resolve the sudden transition between the healthy and faulty state that causes brittleness.

In addition, Shaw proposes putting *homeostasis* at the center of software design. A homeostatic system doesn't detect and explicitly repair faults. Instead, it constantly drives a system toward an acceptable state by continuously executing controlling actions in the background rather than triggering them only to repair a fault. This idea resembles that of self-stabilizing systems. Even though Shaw's idea isn't directly applicable in practice, it represents a new way of thinking about software. While the ACI leaves out any concrete ideas about how we should internally shape self-managing software, the principles of soft and homeostatic systems present a more tangible paradigm to building autonomic systems. Running homeostatic processes in the background becomes more feasible as the hardware becomes more powerful. For example, because everyday office applications don't require all the resources of modern desktop PCs, other operations (such as hard-disk indexing or cleanup operations) can use spare CPU cycles without interfering with the computer's normal operation.

Swarm intelligence

Many self-organizing biological systems[23] are based on the principle of *swarm intelligence*. In an SI system, intelligent behavior emerges from the numerous interactions of simple subcomponents. Prominent examples include ants, bees, fish, and birds. The idea that subcomponents are extremely simple but the overall system manages itself adaptively is tempting from the computer science perspective. However, engineering such a system is extremely difficult[24] because we don't fully understand the mechanisms responsible for the complex behavior of swarms. Such systems' high-level properties (such as foraging and nest building) are called emergent because they're not a direct consequence of the individuals' properties. So, specifying a desirable high-level behavior and mapping it to the lower-level components' behavior isn't straightforward.[25]

One of the many mechanisms at work in natural swarms is the use of pheromone trails to select the shortest path between two locations. Ant colonies apply this principle, which has inspired a new research direction called *ant colony optimization*.[26] Network routing[27] and fault management[28] apply ACO.

ACO and similar SI principles could become a vital ingredient in self-management, but first we need to solve the basic problem of purposefully engineering SI systems. The following sections present some encouraging steps toward this goal.

Self-structuring

Redhika Nagpal, Attila Kondacs, and Catherine Chang show how numerous low-level actors can form 2D shapes[29] on the basis of a high-level description and without any central control. This shows that we can describe the collective, complex behavior of numerous interacting entities on a high level. The system can then map the description to low-level instructions for the interacting entities. Although this is a rather theoretical result, it represents an important step toward appropriate mechanisms for describing the collective behavior of the subcomponents in a SI-based distributed computing system. Following that same path might eventually lead to high-level design mechanisms that make the interwoven structures described earlier manageable for the designers of self-managing systems.

Cell-based programming models

Selvin George, David Evans, and Lance Davidson present a basic approach to designing self-healing structures.[30] They create structures from artificial cells that follow simple programs. These structures can heal externally imposed defects by "killing" individual cells. The authors achieve this using a mechanism that's based on the diffusion of artificial chemicals: Cells emit these chemicals and detect their concentration to react upon it. Cells are only loosely coupled, and the interactions are local and indirect. This approach presents one possible biologically inspired structure of self-healing, distributed computing systems.

Learning from the immune system

Stephanie Forrest, Steven Hofmeyr, and Anil Somayaji describe a new approach for developing security systems[31] based on principles resembling those found in biological immune systems. They use the body's ability to distinguish its own substances from external and potentially threatening substances as an inspiration for their system. Their *computer immune system* identifies malicious sequences of system calls. Using a database with known friendly and hostile sequences of system calls, the security system can identify similar patterns, categorize them, and eventually permit or prohibit them. The security system inaccurately categorizes between 1 and 20 percent of the calls. Owing to this wide range in accuracy, the system isn't ready to be applied in practice. However, it demonstrates one possible realization of a self-protecting system.

Conclusion

Despite growing efforts related to self-management and the availability of well-known theories in related fields, we've only scratched the surface of autonomic systems. The concepts we've discussed lay out a possible agenda:

When developing new services and applications subject to management, we need to tailor them for self-management. Inherently designing systems as control processes might offer a useful abstraction to achieve this. Rendering the applications homeostatic and soft enables them to tolerate a much wider range of operating conditions. Researchers might take this initial step in the near future, because some of the necessary concepts are already present.

The next evolutionary step might occur as SI mechanisms and biologically inspired technologies mature. If we gain a better understanding of the principles applied by nature, computing systems will become increasingly bionic. However, the paradigm shift in this step will be drastic because natural systems have a completely different structure from contemporary computing systems. They rely on massively redundant subcomponents and probabilistic behavior (for example, ACO).

Clearly, this agenda for the evolution of self-managing systems requires an interdisciplinary effort. We expect scientific disciplines such as biology, complex systems research, physics, and sociology to contribute vital concepts that will enable computer science to overcome the inherent problems of self-management.

References

1. D.L. Tennenhouse, "Proactive Computing,"*Comm. ACM,* vol. 43, no. 5, 2000, pp. 43-50.
2. A.G. Ganek and T.A. Corbi, "The Dawning of the Autonomic Computing Era," *IBM Systems J.,* vol. 42, no. 1, 2003, pp. 5-18.
3. M. Debusmann and K. Geihs, "Efficient and Transparent Instrumentation of Application Components Using an Aspect-oriented Approach," *Proc. 14th*

IFIP/IEEE Workshop Distributed Systems: Operations and Management (DSOM 03), LNCS 2867, Springer-Verlag, 2003, pp. 209-220.

4. K. Herrmann and K. Geihs, "Integrating Mobile Agents and Neural Networks for Proactive Management," *Proc. IFIP Int'l Working Conf. Distributed Applications and Interoperable Systems* (DAIS 01), Chapman-Hall, 2001, pp. 203-216.
5. A. Avizienis, J.-C. Laprie, and B. Randell, *Fundamental Concepts of Dependability,* research report N01145, Laboratory for Analysis and Architecture of System of the Nat'l Center for Scientific Research (LAAS-CNRS), 2001.
6. M. Schneider, "Self-Stabilization," *ACM Computing Surveys* (CSUR), vol. 25, no. 1, 1993, pp. 45-67.
7. P. Oriezy, et al., "An Architecture-Based Approach to Self-Adaptive Software," *IEEE Intelligent Systems,* vol. 14, no. 3, 1999,pp. 54-62.
8. I. Georgiadis, J. Magee, and J. Kramer, "Self-Organising Software Architectures for Distributed Systems," *Proc. 1st ACM SIGSOFT Workshop Self-Healing Systems* (WOSS 02), ACM Press, 2002, pp. 33-38.
9. D. Garlan, S. Cheng, and B. Schmerl, "Increasing System Dependability through Architecture-Based Self-Repair," *Architecting Dependable Systems,* R. de Lemos, C. Gacek, and A. Romanovsky, eds., LNCS 2677, Springer-Verlag, 2003, pp. 33-38.
10. *Microsoft Dynamic Systems Initiative,* white paper, Microsoft, 2003.
11. *Open Group Technical Standard CO14, Application Response Measurement Issue 3.0 Java Binding,* The Open Group, 2001.
12. *Common Information Model (CIM) Specification Version 2.2.,* Distributed Management Task Force, 1999.
13. Y. Diao, et al., "Generic Online Optimization of Multiple Configuration Parameters with Application to a Database Server," *Proc. 14th IFIP/IEEE Workshop Distributed Systems: Operations and Management* (DSOM 03), LNCS 2867, Springer-Verlag, 2003, pp. 3-15.
14. *An Architectural Blueprint for Autonomic Computing,* white paper, IBM, 2003.
15. S. Uttamchandani, C. Talcott, and D. Pease, "Eos: An Approach of Using Behavior Implications for Policy-Based Self-Management," *Proc. 14th IFIP/IEEE Workshop Distributed Systems: Operations and Management* (DSOM 03), LNCS 2867, Springer-Verlag, 2003, pp. 16-27.
16. B.R. Haverkort, "Model-Based Self-Configuration for Quality of Service," *Proc. SELF-STAR: Int'l Workshop Self-* Properties in Complex Information Systems,* 2004, www.cs.unibo.it/self-star.
17. M. Malek, F. Salfner, and G.A. Hoffmann, "Self Rejuvenation: An Effective Way to High Availability," *Proc. SELF-STAR: Int'l Workshop Self-* Properties in Complex Information Systems,*2004, www.cs.unibo.it/self-star.
18. M. Shaw, "Beyond Objects: A Software Design Paradigm Based on Process Control," *ACM Software Eng. Notes,* vol. 20, no. 11, 1995, pp. 27-38.

19. E.W. Dijkstra, "Self-Stabilizing Systems in Spite of Distributed Control," *Comm. ACM,* vol. 17, no. 11, 1974, pp. 643-644.
20. S. Dolev, *Self-Stabilization,* MIT Press, 2000.
21. W. Leal and A. Arora, "Scalable Self-Stabilization via Composition," *Proc. 24th Int'l Conf. Distributed Computing Systems,* IEEE CS Press, 2004, pp. 12-21.
22. M. Shaw," 'Self-Healing': Softening Precision to Avoid Brittleness," *Proc. 1st ACM SIGSOFT Workshop Self-Healing Systems* (WOSS 02), ACM Press, 2002, pp. 111-113.
23. S. Camazine, et al., *Self-Organization in Biological Systems,* Princeton Univ. Press, 2001.
24. J.M. Ottino, "Engineering Complex Systems,"*Nature,* vol. 427, Jan. 2004, p. 399.
25. A. Deutsch, "Self-Organization in Interacting Cell Networks: From Microscopic Rules to Emergent Behavior," *Proc. SELF-STAR: Int'l Workshop Self-* Properties in Complex Information Systems,*2004, www.cs.unibo.it/self-star.
26. V. Maniezzo, L.M. Gambardella, and F. De Luigi, "Ant Colony Optimization," *New Optimization Techniques in Engineering,* G.C. Onwubolu and B.V. Babu, eds., Springer-Verlag, 2004, pp. 101-117.
27. G.D. Caro and M. Dorigo, "Ant Colonies for Adaptive Routing in Packet-Switched Communications Networks," *Proc. Parallel Problem Solving from Nature,* LNCS 1498, Springer-Verlag, 1998, pp. 27-30.
28. T. White and B. Pagurek, "Distributed Fault Location in Networks Using Learning Mobile Agents," *Proc. Approaches to Intelligent Agents: 2nd Pacific Rim Int'l Workshop Multi-Agents* (PRIMA 99), LNCS 1733, Springer-Verlag, 1999, pp. 182-196.
29. R. Nagpal, A. Kondacs, and C. Chang, "Programming Methodology for Biologically-Inspired Self-Assembling Systems," *Computational Synthesis: From Basic Building Blocks to High-Level Functionality: Papers from the AAAI Spring Symp.,* AAAI Press, 2003, pp. 173-180.
30. S. George, D. Evans, and L. Davidson, "A Biologically Inspired Programming Model for Self-Healing Systems," *Proc. 1st ACM SIGSOFT Workshop Self-Healing Systems* (WOSS 02), ACM Press, 2002, pp. 102-104.
31. S. Forrest, S.A. Hofmeyr, and A. Somayaji, "Computer Immunology," *Comm. ACM,* vol. 40, no. 10, 1997, pp. 88-96.

Klaus Herrmann is a PhD student at the Berlin University of Technology. His research interests include self-organization principles in mobile networks, event-based middleware systems, and mobile agents. He received his MS (Diplom-Informatiker) in computer science from the Goethe University of Frankfurt. Contact him at Sekretariat EN6, Einsteinufer 17, 10587 Berlin, Germany;klaus.herrmann@acm.org.

Gero Mühl is a postdoctoral researcher at the Berlin University of Technology. His research interests are middleware, event-based systems, self-organization, and mobile computing. He received his PhD in computer science from the Darmstadt University of Technology. He is a member of the ACM. Contact him at Sekretariat EN6, Einsteinufer 17, 10587, Germany g_muehl@acm.org.

Kurt Geihs is a professor of distributed systems at the Berlin University of Technology. His research interests include distributed systems, operating systems, networks, and software technology. His current projects focus on quality-of-service management, component-based software, and middleware for mobile and ad hoc networking. He received his PhD in computer science from the Aachen University of Technology. Contact him at Sekretariat EN6, Einsteinufer 17, 10587 Berlin, Germany; geihs@ivs.tu-berlin.de.

The Autonomic Computing Initiative

Concepts for building dependable systems that tolerate errors and ensure robust and stable systems have existed for almost 40 years.[1] The IEEE and IFIP initiated corresponding task forces in 1970 and in 1980, respectively. Recent efforts, however, define far more ambitious goals by propagating self-management as their ultimate objective.

IBM started the Autonomic Computing Initiative[2] in 2001 and leads the trend toward self-managing systems. The ACI sketches a far-reaching vision with the ultimate goal of rendering IT systems completely self-managing. The basic concepts of the ACI aren't new, but they represent a good basis for comparing and categorizing different approaches. We thus use it as a reference model for discussing open issues in self-management.

References

1. A. Avizienis , J.-C. Laprie, and B. Randell, , *Fundamental Concepts of Dependability,* research report N01145, Laboratory for Analysis and Architecture of System of the Nat'l Center for Scientific Research (LAAS-CNRS), 2001.
2. A.G. Ganek and T.A. Corbi, , "The Dawning of the Autonomic Computing Era," *IBM Systems J.,* vol. 42, no. 1, 2003, pp. 5-18.

Cite this article: Klaus Herrmann, Gero Mühl, and Kurt Geihs, "Self-Management: The Solution to Complexity or Just Another Problem?" IEEE Distributed Systems Online, vol. 6, no. 1, 2005.