DOI:10.1145/1897852.1897868

Article development led by acmqueue
queue.acm.org

Models of determinism are changing IT management.

BY MARK BURGESS

Testable System Administration

THE METHODS OF system administration have changed little in the past 20 years. While core IT technologies have improved in a multitude of ways, for many if not most organizations system administration is still based on production-line build logistics (aka provisioning) and reactive incident handling—an industrial-age method using brute-force mechanization to amplify a manual process. As we progress into an information age, humans will need to work less like the machines they use and embrace knowledge-based approaches. That means exploiting simple (hands-free) automation that leaves us unencumbered to discover patterns and make decisions. This goal is reachable if IT itself opens up to a core challenge of automation that is long overdue—namely, how to abandon the myth of determinism and expect the unexpected.

We don't have to scratch the surface very hard to find cracks in the belief system of deterministic management. Experienced system practitioners know deep down that they cannot think of system administration as a simple process of reversible transactions to be administered by hand; yet it is easy to see how the belief stems from classical teachings. At least half of computer science stems from the culture of discrete modeling, which deals with absolutes as in database theory, where idealized conditions can still be simulated to an excellent approximation. By contrast, the stochastic models that originate from physics and engineering, such as queueing and error correction, are often considered too difficult for most basic CS courses. The result is that system designers and maintainers are ill prepared for the reality of the Unexpected Event. To put it quaintly, "systems" are raised in laboratory captivity under ideal conditions, and released into a wild of diverse and challenging circumstances. Today, system administration still assumes, for the most part, that the world is simple and deterministic, but that could not be further from the truth.

In the mid-1990s, several research practitioners, myself included, argued for a different model of system administration, embracing automation for consistency of implementation and using policy to describe an ideal state. The central pillar of this approach was stability.[2,4] We proposed that by placing stability center stage, one would achieve better reliability (or at the very least predictability). A tool such as IT is, after all, useful only if it leads to consistently predictable outcomes. This is an evolutionary approach to management: only that which survives can be successful.

As a physicist by training, I was surprised by the lack of a viable model for explaining actual computer behavior. It seemed that, instead of treating behavior as an empirical phenomenon full of inherent uncertainties, there was an implicit expectation that com-

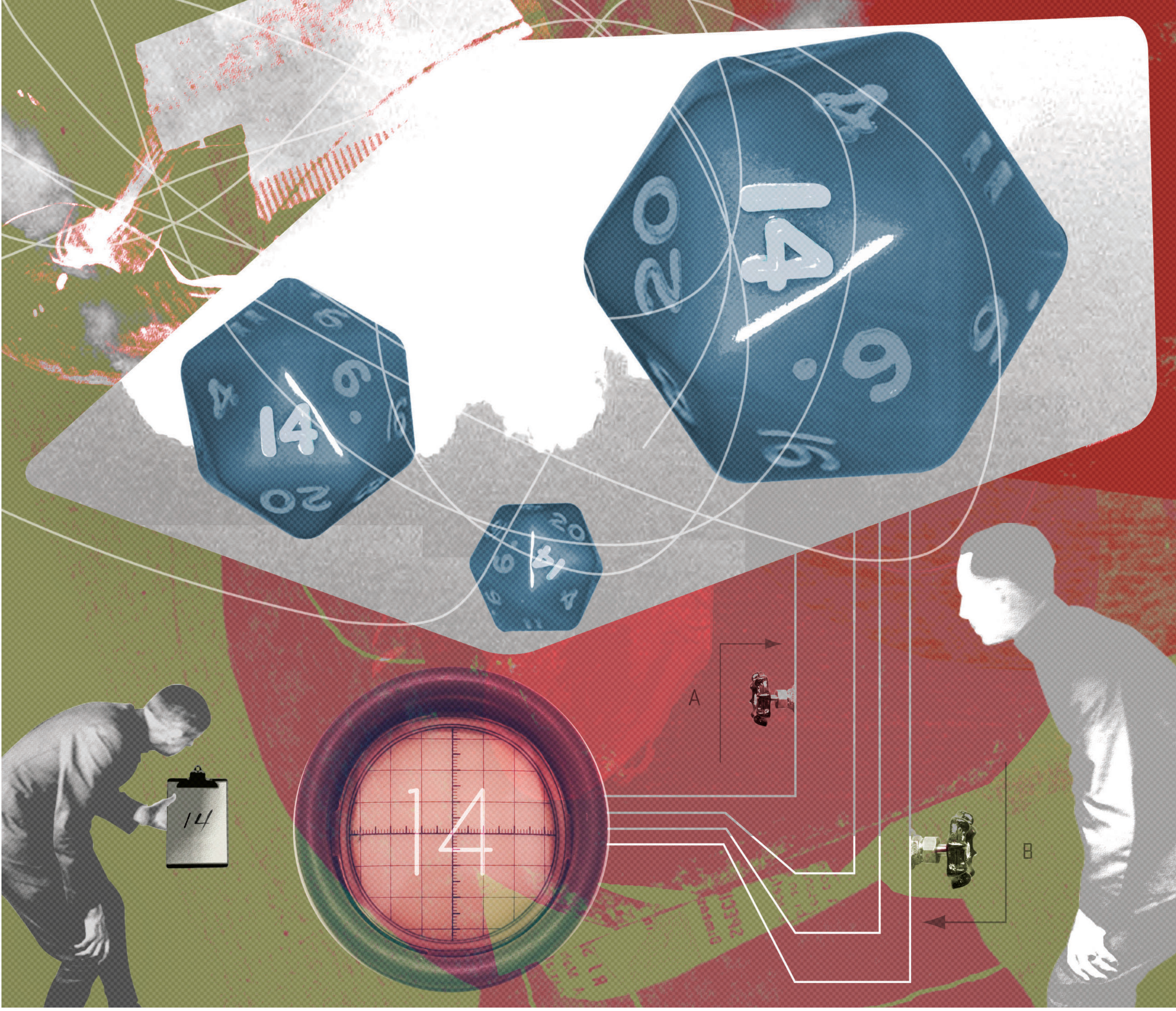

ILLUSTRATION BY STUART BRADFORD

puters would behave as programmed. Everyone knows this to be simplistic; yet still, a system administrator would worry about behavior only when an incident reported something to the contrary.

From Demolition to Maintenance

When a problem occurs, many organizations take affected systems out of service, wipe them, and restore them from backup or reinstall from scratch. This is the only way they know to assure the state of the system because they know no simple way of discovering what changed without an arduous manual investigation. The process is crude, like tearing down a building to change a lightbulb. But the reason is understandable. Current tools are geared for building, not repairing—and if you've only got a sledgehammer...then you rebuild.

There is growing acceptance of a test-driven or diagnostic approach to the problem. This was originally ushered in by Cfengine,[5] and then partially adopted in other software such as Puppet.[11] In a test-driven approach, system state is regulated by continual reappraisal at a microscopic level, like having a groundskeeper watch continuously over an estate, plucking the weeds or applying a lick of paint where needed. Such an approach required the conceptual leap to a computable notion of *maintenance*. Maintenance can be defined by referring to a *policy* or *model* for an ideal system state. If such a model could somehow be described in terms of predictable, actionable repairs, in spite of environmental indeterminism, then automating maintenance would become a simple reality. This, in essence, is how Cfengine changed the landscape of IT management.

The term *compliance* is often used today for correctness of state with respect to a model. If a system deviates from its model, then with proper automation it *self*-repairs,[2,4] somewhat like an autopilot that brings systems back on course. What is interesting is that, when you can repair system state (both static configuration and runtime state), then the initial condition of the system becomes unimportant, and you may focus entirely on the desired outcome. This is the way businesses want to think about IT—in terms of goals rather than "building projects"—thus also bringing us closer to a modern IT industry.

Convergence to a Desired State

Setting up a reference model for repair sounds like a simple matter, but it requires a language with the right properties. Common languages used in software engineering are not well suited for the task, as they describe sequential steps from a fixed beginning rather than end goals. Generally, we don't know the starting state of a machine when it fails. Moreover, a lot of redundant computation is required to track a model, and that would intrude on clarity. The way around this has been to construct declarative DSLs (domain-specific languages) that hide the details and offer predictable semantics. Although Cfengine was the first attempt to handle indeterminism, special languages had been proposed even earlier.[9]

Many software engineers are not convinced by the declarative DSL argument: they want to use the familiar tools and methods of traditional programming. For a mathematician or even a carpet fitter, however, this makes perfect sense. If you are trying to fit a solution to a known edge state, it is cumbersome to start at the opposite end with a list of directions that assume the world is fixed. When you program a GPS, for example, you enter the desired destination, not the start of the journey, because you often need to recompute the path when the unexpected occurs, such as a closed road. This GPS approach was taken by Cfengine[5] in the mid-1990s. It says: work relative to the desired end-state of your model, not an initial baseline configuration, because the smallest unexpected change breaks a recipe based on an initial state. This has been likened to Prolog.[7]

In simple terms, the approach works by making every change satisfy a simple algorithm:

$$\text{Change (arbitrary\_state)} \rightarrow \text{desired\_state} \quad (1)$$
$$\text{Change (desired\_state)} \rightarrow \text{desired\_state} \quad (2)$$

This construction is an expression of "dumb" stability, because if you perturb the desired state into some arbitrary state, it just gets pushed back into the desired state again, like an automated course correction. It represents a system that will recover from accidental or incidental error, just by repeating a dumb mantra—without the need for intelligent reasoning.

For example: suppose you want to reconfigure a Web server to support PHP and close a security hole. The server and all of its files are typically part of a software package and is configured by a complex file with many settings:

```
# >10kB of complex stuff
MODULES = SECURITY _ HOLE JAVA
   OTHERS
# >10kb of complex stuff
```

To fix both problems, it is sufficient to alter only this list (for example, a desired outcome):

```
# >10kB of complex stuff
MODULES = JAVA OTHERS PHP
# >10kB of complex stuff
```

Traditionally, one replaces the whole file with a hand-managed template or even reinstalls a new package, forcing the end user to handle everything from the ground up. Using a desired state approach, we can simple say: in the context of file `webserver.config`, make sure that any line matching "`MODULES = something`" is such that "`something`" contains "`PHP`" and does not contain "`SECURITY HOLE`." Figure 1 illustrates how this might look in Cfengine.

Thus, the code defines two internal list variables for convenience and passes these to the specially defined method `edit_listvar`, which is constructed from convergent primitives. For each item in the list, Cfengine will assure the presence or absence of the listed atoms without touching anything else. With this approach, you don't need to reconstruct the whole Web server or know anything about how it is otherwise configured (for example, what is in "`complex stuff`") or even who is managing it: a desired end-state relative to an unknown start-state has been specified. It is a highly compressed form of information.

I referred to this approach as *convergent maintenance* (also likening the behavior to a human immune system[2]), as all changes converge on a destination or healthy state for the system in the frame of reference of the policy. Later, several authors adopted the mathematical term idempotence (meaning invariance under repetition), focusing on the fact that you can apply these rules any number of times and the system will only get better.

Guarded Policy

In the most simplistic terms, this approach amounts to something like Dijkstra's scheme of guarded commands.[8] Indeed, Cfengine's language implementation has as much in common with Guarded Command Lan-

Figure 1. Reconfiguring a Web server in Cfengine.

```
bundle agent webserver_config
{
vars:
 "add" slist => { "PHP", "php5" };
 "del" slist => { "SECURITY_HOLE", "otherstuff" };

column_edits:

   "APACHE_MODULES=.*"
      edit_column => edit_listvar("$(add_modules)","append");
   "APACHE_MODULES=.*"
      edit_column => edit_listvar("$(del_modules)","delete");
}

[Note: The syntax (which incorporates implicit guards and iteration)
has the form:

 type_of_promise:

   "Atom"

       property_type => desired_end_state;
]
```

guage as it does with Prolog.[7] The assertion of X as a statement may be interpreted as:

```
If not model(X), set model(X)
```

For example:

```
"/etc/passwd" create => "true";
```

Repeating an infinite number of times does not change the outcome. With hindsight, this seems like a trivial observation, and hardly a revolutionary technology, yet it is the simplest of insights that are often the hardest won. The implication of this statement is that X is not just what you want, but a model for what should be. The separation of the intended from the actual is the essence of the relativity.

There is one more piece to the puzzle: Knowing the desired state is not enough; we also have to know that it is achievable. We must add *reachability* of the desired state to the semantics.

Getting Stuck in Local Minima

It is well known from artificial intelligence and its modern applications that algorithms can get stuck while searching parameter landscapes for the optimum state. When you believe yourself at the bottom of the valley, how do you know there is not a lower valley just over the rise? To avoid the presence of false or local minima, you have to ensure that each independent dimension of the search space has only a single minimum, free of obstacles. Then there are two things at work: independence and convergence. Independence can be described by many names: atomicity, autonomy, orthogonality, and so on. The essence of them all is that the fundamental objects in a system should have no dependencies.

What we are talking about is a theory of policy atoms in an attribute space. If you choose vectors carefully (such as, file permissions, file contents, and processes) so that each change can be made without affecting another, no ordering of operations is required to reach a desired end-state, and there can be only one minimum. Indeed order-independence can be proven with periodic maintenance as long as the operators form irreducible groups.[3,6]

The discovery of such a simple solution suggests a panacea, ushering in a new and perfect world. Alas, the approach can be applied only partially to actual systems because no actual systems are built using these pure constructions. Usually, multiple change mechanisms tether such atoms together in unforeseeable ways (for example, packages that bundle up software and prevent access to details). The approximation has worked remarkably well in many cases, however, as evidenced by the millions of computers running this software today in the most exacting environments. Why? The answer is most likely because a language that embodies such principles encourages administrators to think in these terms and keep to sound practices.

If you are trying to fit a solution to a known edge state, it is cumbersome to start at the opposite end with a list of directions that assume the world is fixed.

Tangled by Dependency: The Downside of Packaging

The counterpoint to this free atomization of system parts is what software designers are increasingly doing today: bundling atoms and changes together into packages. In modern systems packaging is a response to the complexity of the software management process. By packaging data to solve one management problem, however, we lose the resolution needed to customize what goes on inside the packages and replace it with another. Where a high degree of customization is needed, unpacking a standard "package update" is like exploding a smart bomb in a managed environment—wiping out customization—and going back to the demolition school of management.

We don't know whether any operating system can be fully managed with convergent operations alone, nor whether it would even be a desirable goal. Any such system must be able to address the need of surgically precise customization to adapt to the environment. The truly massive data centers of today (Google and Facebook) are quite monolithic and often less complex than the most challenging environments. Institutions such as banks or the military are more representative, with growth and acquisition cultures driving diverse challenges to scale. What *is* known is that no present-day operating system makes this a completely workable proposition. At best one can approximate a subset of management operations, but even this leads to huge improvements in scalability and consistency of process—by allowing humans to be taken out of the process.

From Demanding Compliance To Offering Capability

What is the future of this test-driven approach to management? To understand the challenges, you need to be aware of a second culture that pervades computer science: the assumption of management by *obligation*. Obligations are modal statements: for example, X must comply with Y, A should do B, C is allowed to do D, and so on. The assumption is that you can force a system to bow down to a decision made externally. This viewpoint has been the backbone of policy-based systems for years,[12] and it suffers from a number of fundamental flaws.

The first flaw is that one cannot generally exert a mandatory influence on another part of a software or hardware system without its willing consent. Lack of authority, lack of proximity, lack of knowledge, and straightforward impossibility are all reasons why this is impractical. For example, if a computer is switched off, you cannot force it to install a new version of software. Thus, a model of maintenance based on obligation is, at best, optimistic and, at worst, futile. The second point is that obligations lead to contradictions in networks that cannot be resolved. Two different parties can insist that a third will obey quite different rules, without even being aware of one another.[1]

Realizing these weaknesses has led to a rethink of obligations, turning them around completely into an atomic theory of "voluntary cooperation," or promise theory.[1] After all, if an obligation requires a willing consent to implement it, then voluntary cooperation is the more fundamental point of view. It turns out that a model of promises provides exactly the kind of umbrella under which all of the aspects of system administration can be modeled. The result is an agent-based approach: each system part should keep its own promises as far as possible without external help, expecting as little as possible of its unpredictable environment.

Independence of parts is represented by agents that keep their own promises; the convergence to a standard is represented by the extent to which a promise is kept; and the insensitivity to initial conditions is taken care of by the fact that promises describe outcomes, not initial states.

Promise theory is a wide-ranging description of cooperative model building that thinks bottom-up instead of top-down. It can be applied to humans and machines in equal measure and can also describe human workflows—a simple recipe for federated management.

Promise theory turns out to be a rather wide-ranging description of cooperative model building that thinks bottom-up instead of top-down. It can be applied to humans and machines in equal measure and can also describe human workflows—a simple recipe for federated management. It has not yet gained widespread acceptance, but its principal findings are now being used to restructure some of the largest organizations in banking and manufacturing, allowing them to model complexity in terms of robust intended states. Today, only Cfengine is intentionally based on promise theory principles, but some aspects of Chef's decentralization[10] are compatible with it.

The Limits of Knowledge

There are subtler issues lurking in system measurement that we've only glossed over so far. These will likely challenge both researchers and practitioners in the years ahead. To verify a model, you need to measure a system and check its compliance with the model. Your assessment of the state of the system (does it keep its promises?) requires a trust of the measurement process itself to form a conclusion. That one dependence is inescapable.

What happens when you test a system's compliance with a model? It turns out that every intermediate part in a chain of measurement potentially distorts the information you want to observe, leading to less and less certainty. Uncertainty lies at the very heart of observability. If you want to govern systems by pretending to know them absolutely, you will be disappointed.

Consider this: environmental influences on systems and measurers can lead directly to illogical behavior, such as undecidable propositions. Suppose you have an assertion (for example, promise that a system property is true). In logic this assertion must either be true or false, but consider these cases:

- You do not observe the system (so you don't know);
- Observation of the system requires interacting with it, which changes its state;

▸ You do not trust the measuring device completely; or

▸ There is a dependence on something that prevents the measurement from being made.

If you believe in classic first-order logic, any assertion must be either true or false, but in an indeterminate world following any of these cases, you simply do not know, because there is insufficient information from which to choose either true or false. The system has only two states, but you cannot know which of them is the case. Moreover, suppose you measure at some time *t*; how much time must elapse before you can no longer be certain of the state?

This situation has been seen before in, of all places, quantum mechanics. Like Schrodinger's cat, you cannot know which of the two possibilities (dead or alive) is the case without an active measurement. All you can know is the outcome of each measurement reported by a probe, after the fact. The lesson of physics, on the other hand, is that one can actually make excellent progress without complete knowledge of a system—by using guiding principles that do not depend on the uncertain details.

Back to Stability?

A system might not be fully knowable, but it can still be self-consistent. An obvious example that occurs repeatedly in nature and engineering is that of *equilibrium*. Regardless of whether you know the details underlying a complex system, you can know its stable states because they persist. A persistent state is an appropriate policy for tools such as computers—if tools are changing too fast, they become useless. It is better to have a solid tool that is almost what you would like, rather than the exact thing you want that falls apart after a single use (what you want and what you need are not necessarily the same thing). Similarly, if system administrators cannot have what they want, they can at least choose from the best we can do.

Systems can be stable, either because they are unchanging or because many lesser changes balance out over time (maintenance). There are countless examples of very practical tools that are based on this idea: Lagrange points (optimization), Nash equilibrium (game theory), the Perron-Frobenius theorem (graph theory), and the list goes on. If this sounds like mere academic nonsense, then consider how much of this nonsense is in our daily lives through technologies such as Google PageRank or the Web of Trust that rely on this same idea.

Note, however, that the robustness advocated in this article, using the principle of atomization and independence of parts, is in flat contradiction with modern programming lore. We are actively encouraged to make hierarchies of dependent, specialized objects for reusability. In doing so we are bound to build fragilities and limitations implicitly into them. There was a time when hierarchical organization was accepted wisdom, but today it is becoming clear that hierarchies are fragile and unmanageable structures, with many points of failure. The alternative of sets of atoms promising to stabilize patches of the configuration space is tantamount to heresy. Nevertheless, sets are a more fundamental construction than graphs.

For many system administrators, these intellectual ruminations are no more pertinent than the moon landings were to the users of Teflon pans. They do not see themselves in these issues, which is why researchers, not merely developers, need to investigate them. Ultimately, I believe there is still great progress to be made in system administration using these approaches. The future of system administration lies more in a better understanding of what we already have to work with than in trying to oversimplify necessary complexity with industrial force.

Conclusion

It is curious that embracing uncertainty should allow you to understand something more fully, but the simple truth is that working around what you don't know is both an effective and low-cost strategy for deciding what you actually can do.

Major challenges of scale and complexity haunt the industry today. We now know that scalability is about not only increasing throughput but also being able to comprehend the system as it grows. Without a model, the risk of not knowing the course you are following can easily grow out of control. Ultimately, managing the sum knowledge about a system is the fundamental challenge: the test-driven approach is about better knowledge management—knowing what you can and cannot know.

Whether system administration is management or engineering is an oft-discussed topic. Certainly without some form of engineering, management becomes a haphazard affair. We still raise computers in captivity and then release them into the wild, but there is now hope for survival. Desired states, the continual application of "dumb" rule-based maintenance, and testing relative to a model are the keys to quantifiable knowledge. C

Related articles on queue.acm.org

A Plea to Software Vendors from Sysadmins—10 Do's and Don'ts
Thomas A. Limoncelli
http://queue.acm.org/detail.cfm?id=1921361

Self-Healing in Modern Operating Systems
Michael W. Shapiro
http://queue.acm.org/detail.cfm?id=1039537

A Conversation with Peter Tippett and Steven Hofmeyr
January 10, 2009
http://queue.acm.org/detail.cfm?id=1071725

References

1. Burgess, M. An approach to understanding policy based on autonomy and voluntary cooperation. Submitted to *IFIP/IEEE 16th International Workshop on Distributed Systems Operations and Management* (2005).
2. Burgess, M. Computer immunology. In *Proceedings of the 12th System Administration Conference*, 1998.
3. Burgess, M. Configurable immunity for evolving human-computer systems. *Science of Computer Programming 51*, 3 (2004), 197–213.
4. Burgess, M. On the theory of system administration. *Science of Computer Programming 49* (2003), 1–46.
5. Cfengine; http://www.cfengine.org.
6. Couch, A., Daniels, N. The maelstrom: network service debugging via 'ineffective procedures.' *Proceedings of the 15th Systems Administration Conference* (2001), 63.
7. Couch, A., Gilfix, M. It's elementary, dear Watson: Applying logic programming to convergent system management processes. In *Proceedings of the 13th Systems Administration Conference* (1999), 123.
8. Dijkstra, E. http://en.wikipedia.org/wiki/Guarded_Command_Language.
9. Hagemark, B., Zadeck, K. Site: A language and system for configuring many computers as one computer site. *Proceedings of the Workshop on Large Installation Systems Administration III* (1989); http://www2.parc.com/csl/members/jthornton/Thesis.pdf.
10. Opscode; http://www.opscode.com/chef.
11. Puppet Labs; http://www.puppetlabs.com/.
12. Sloman, M. S., Moffet, J. Policy hierarchies for distributed systems management. *Journal of Network and System Management 11*, 9 (1993), 404.

Mark Burgess is a professor of network and system administration, the first with this title, at Oslo University College. His current research interests include the behavior of computers as dynamic systems and applying ideas from physics to describe computer behavior. He is the author of Cfengine and is the founder, chairman, and CTO of Cfengine, Oslo, Norway.