

ALVA COUCH

From $x=1$ to (setf x 1): what does configuration management mean?



A SYSTEM ADMINISTRATION
RESEARCHER CONSIDERS LESS-
ONS LEARNED FROM LISA '07,
INCLUDING THE RELATIONSHIP
BETWEEN CONFIGURATION
MANAGEMENT AND AUTO-
NOMIC COMPUTING

Alva Couch is an Associate Professor of Computer Science at Tufts University, where he and his students study the theory and practice of network and system administration. He served as Program Chair of LISA '02 and was a recipient of the 2003 SAGE Outstanding Achievement Award for contributions to the theory of system administration. He currently serves as Secretary of the USENIX Board of Directors.

couch@cs.tufts.edu

THE CONFIGURATION MANAGEMENT workshop this year at LISA brushed against autonomic configuration management, but as usual “there were no takers.” The lessons of autonomic control in network management (also called “self-managing systems”) seemed far removed from practice, “something to think about 10 years from now.” Meanwhile, many talks throughout the conference (including the keynote, a guru session, and several technical papers) discussed automatic management mechanisms, although some speakers stopped short of calling these “self-managing” or “autonomic.” Autonomics were almost a theme. But, in my opinion, these speakers made few converts. I stopped to think about why this is true, and I think I have a simple explanation. It’s all about meaning.

The meaning crisis that system administrators face is very similar to the crisis of meaning that plagued the programming languages community in the past: There is a difference in semantics between doing things autonomically and doing things via traditional configuration management. “Semantics” refers to “what things mean.” The difference is so small, and yet so profound, that the community is not fully aware of it. But it places so crippling a wall between autonomics and traditional configuration management that it is worthy of comment in itself.

Operational and Axiomatic Semantics

In programming languages, there is a “semantic wall” between statically typed languages such as C and dynamically typed languages such as LISP. The difference between these languages seems small but is actually profound. The meaning of a C program is easily defined in terms of the operations of the base machine. This is called operational semantics. By contrast, the interactions between a LISP interpreter and the base machine are not useful to understand. Instead, one expresses the meaning of statements via axiomatic semantics: a mathematical description of the observable behavior resulting from executing statements, without reference to the underlying way in which statements are actually implemented.

To understand this subtlety, consider the difference between the semantics of the C statement `x=1` and the LISP statement `(setf x 1)`. For `x=1` there is an empowering operational (also called “bottom-up”) semantic model that “there is a cell named X into which the value 0x00000001 is written.” The operational semantics of `(setf x 1)`, however, are not particularly empowering. There is a symbol named `x` that is created in a symbol table (indexed by name), and there is a numeric atom containing the value 1, and those are associated via the property “symbol-value” of the symbol `x`. At a deeper level, index trees become involved. But those facts about the LISP version of `x` are not important and not empowering except to people developing LISP. The axiomatic (“top-down”) equivalent for the meaning of this statement is that “after `(setf x 1)`, the symbol `x` refers to the atom 1.” The details of implementation are stripped, and only the valuable functional behavior is left.

The Semantic Wall of Configuration Management

We now face a similar semantic wall between systems that exhibit “autonomic” behaviors and systems that “automate” configuration management. The latter utilize operational semantics (like `x=1`), whereas the former utilize axiomatic semantics (like `(setf x 1)`). This difference may seem unimportant, but it is central enough to cripple the discipline.

Current configuration management tools such as BCFG2, Puppet, and Cfengine utilize an operational semantic model similar to that of `x=1` in C. The “meaning” of each tool’s input is “what it does to the configurations of machines.” Regardless of how data is specified, its final destination in a specific configuration file or files is what it “means.” For example, regardless of the way in which one specifies an Internet service, one knows that it must end up as an entry in `/etc/xinetd.conf` or a file in `/etc/xinetd.d`; its “meaning” is defined in terms of that final positioning within the configuration of the machine.

By contrast, autonomic systems are configured via axiomatic semantics; the parameters specified have no direct relationship to the actual contents of files on a machine, nor is the understanding of that correspondence important or empowering, because the relationship between the parameter and the realization of that parameter (in terms of the behavior that it engenders or encourages) is too complex to be useful. For example, a specification that “the Web server must have a response time less than 2 seconds for each request” has little to do with the actual identity of the Web server or how that result might be achieved. In a very deep sense, that information is not useful in understanding the objective.

To utilize autonomics effectively, we need to progress from a semantic model in which `x=1` is defined operationally to a semantic model in which `(setf x 1)` is defined axiomatically. This was a big step in programming languages and is an equally daunting step in configuration management. But, as I will explain, not only do current tools not contribute to that progress, they actually work actively against it, by reinforcing practices that trench us needlessly in operational semantics and distance us from the potential for axiomatic meaning.

Abstraction and Meaning

Current approaches to configuration management, such as Cfengine, BCFG2, and Puppet, attempt to close the gap via what some authors call “raising the level of abstraction” at which one specifies configuration. How-

ever, simply raising the level of abstraction cannot scale the semantic wall between operation and behavior. One hard lesson of programming language semantics is that it is not just necessary to “abstract” upward from the machine; one must also create a model of behavior (in an axiomatic sense) with which one can reason at a high level, and with simpler semantic properties than the full operational model. Simply raising the level of abstraction does not automatically create any model other than the existing operational model of “bits on disk.” Without an empowering semantic model, it is no easier to reason about a high-level description based upon operational semantics than it is to reason about a low-level description of the same thing.

Authors of configuration management tools frequently wonder why the level of adoption of their tools is so low. The answer, I think, lies in this issue of semantics. The tools do not “make things easier to understand”; they make things that remain difficult to understand easier to construct. No matter how skillfully one learns to use the tool, one is committed to an operational semantic model, in which one must still understand what “bits on disk” mean in order to understand what a tool does. The tool thus represents “something extra to learn” rather than managing “something that one can afford to forget.”

There is no doubt that current tools save much work and raise the maturity level of a site but, alas, they fail to make the result easier to understand. It is thus not surprising that less experienced administrators with much left to learn about “bits on disk” shy away from having to learn even more than before. If configuration management represents “something else to learn” rather than “something easier to master,” it is no surprise that use of configuration management tools finishes dead last in priority among inexperienced system administrators. If tools are to become attractive, they must represent “less to learn” rather than “more to learn.”

Modeling Behavior

A successful model of configuration semantics would allow one to avoid irrelevant detail and concentrate on important details. The gulf between “automated” and “autonomic” configuration management is so great, however (like the gulf between C and the logic-programming language Prolog), that some intermediate semantics (e.g., those in LISP) might help. If, as well, this intermediate semantics is straightforward enough to be empowering, then we have a semantic layer we can use to bridge between “automatic” and “autonomic” models. The current semantics has the character of C’s $x=1$, whereas this intermediate semantics might have the character of LISP’s (setf x 1).

Consider, for example, the intermediate semantics of file service. The “bottom-up” semantic model of this is that what one writes into configuration files leads to some fixed binding between each client (embodied as a machine) and some file server. The “top-down” model of file service can be expressed in a much simpler way. There is some “service” (that potentially can move from server to server) and some “pool of clients” (that must share the same service), but there is no binding between that service and a particular server, because that would be irrelevant to specifying the goal of providing that service. Instead, there is an expectation of service behavior that is missing from the bottom-up semantic model. That behavioral objective is that whatever file a user writes to the service is persistent across any kind of network event or contingency, and it can be recovered later by reading it back from the service via the same pathname. The way that this behavioral objective is met is not central to reasoning about the requirement. The objective

is not a property of a specific machine, but of the management process itself; if the server changes, the file still (hopefully) persists, as much by the actions of human administrators (in recovering it from backups) as by the actions of software.

By contrast, the bottom-up model of file service is limiting in simple but profound ways. Saying “server X should provide Network File System (NFS) service to client Y” (or even “some server in a pool P should provide NFS to client Y”) is similar to saying $X=1$. There are implicit limiting assumptions about how this might be done. In particular, we have implicitly decided in the former statement that NFS is the objective rather than a means toward an objective. A file service is semantically very much like (setf x 1), in which, by some unspecified method, two operations to which we refer as “write” (such as executing (setf x 1) in LISP) and “read” (analogous to referencing x after the setf) have consistent behaviors. This model is simple, but NFS is relatively complex, and there are many ways of assuring this kind of behavior other than by using NFS.

An axiomatic model of file service thus differs drastically from the operational model. The entities are not machines, but users, and the axiomatic formulation is that, for each user, writing content to a path results in that content being available henceforth via that path. For simplicity, we might notate this “behavioral axiom” as:

User -(Path:Content)-> filesystem

to mean that for an entity that is a “User,” interactions with the entity “filesystem” comprise associating a “Path” with “Content” and being able to retrieve that content via that path. “User,” “Path,” and “Content” are types that refer to sets of potential entities, whereas “filesystem” is an entity. The arrow represents a dominance relationship, in which any entity of type “User” is dominant in creating content; “filesystem” is subservient in recording and preserving that relationship.

Modeling Services

One advantage of such a model is that many details that are purely implementation drop out of the model. The most important facet of a DHCP relationship between server and client is that the server specifies the address of the client:

DHCP -(MAC:IPAddress)-> Client

whereas the client is accessible through that address. There are many ways of assuring the latter, but one of the more common is “dynamic DNS,” in which:

DHCP -(Name:IPAddress,IPaddress:Name)->DNS

This means that DHCP specifies the name-IP mapping to DNS in accordance with its data on active clients. DNS returns this to the clients via:

DNS -(Name:IPAddress,IPaddress:Name)->Client

This means that a client asking for an “IPaddress” for a “Name” or a “Name” for an “IPaddress” gets the one that DHCP specified originally. The “Name” to “MAC” mapping is specified by an administrator, e.g.:

Administrator->(MAC:Name)->DHCP

These are all dominance relationships very much like the one that describes file service.

This level of detail is independent of irrelevant detail, such as how this mapping is accomplished. Caching, timeouts, and formats of mappings are (at this level) irrelevant details. The important details include dominance relationships and behavioral predictions, including that the address assigned by DHCP is indeed the address by which the host can be located via DNS.

The beauty of this scheme is that we describe “how things should work” but not “how this behavior is assured.” The former is empowering; the latter is more or less irrelevant if our tools understand the former. But current tools do not understand the former; neither can they assure this behavior without a lot of help from human beings.

Promises, Promises . . .

The astute reader will realize that this notation is very similar to that of promise theory, and the even more astute reader will realize that promise theory does not include a globally valid semantic model. Promises are a concept introduced by Mark Burgess to provide a simple framework for modeling interactions between agents during configuration management. A promise is a declaration of behavioral intent, whose semantic interpretation is up to the individual agent receiving each promise. A promise between agents assumes as little as possible about behavior, while at the same time being as clear as possible about the intention of the promise. The “type” of a promise is a starting point for the agent’s determination of the promise’s “meaning,” which is an emergent property of the promise, tempered by local observation by the receiver of its validity or lack thereof.

My notation, by contrast, globally defines expected interactions and their results. Promises enable local interactions, whereas the notation here attempts to describe overarching intent. Thus my semantics may look as though it describes promises but, because it describes intent as a global invariant, it is not like promises at all. Promise theory is one level up from my model in complexity, in not assuming that agents can be trusted to cooperate.

Coming to Closure . . .

In like manner, anything that implements the semantics of my notation is a closure, in the sense that it exhibits semantic predictability based upon an exterior description of behavior. This is how “closure” is defined.

It is well documented that building a closure is difficult and requires changes in the way we think about and notate a problem, but so is building a LISP interpreter in C, and we managed to do that. Most of the difficulties inherent in both tasks (building a closure or building a LISP interpreter) lies in letting go of lower-level details and scaling the semantic wall without looking back or down.

This is what we currently cannot bring ourselves to do.

And, because this is exactly what autonomic tools do, we are setting ourselves up for a rude awakening in which our tools and practices lag far behind the state of the art.

Science, Engineering, or Sociology?

We, our tools, and our practices are faced with a semantic wall. On one side of the wall lie operational semantics. On the other side lie axiomatic semantics. We have two choices: Scale that wall ourselves or let someone else scale

it for us. If we sit still and let others do the climbing, that climbing will be done by systems engineers who understand little of the human part of system administration. If we instead take an active role, higher-level semantics can evolve in accordance with our human needs as system administrators, in addition to the needs of our organizations.

And the way I think we can take an active role may be somewhat surprising. One can take a role in this revolution even if one uses no tools and does everything by hand!

The Power of Commonality

It is easy to forget that the widely accepted Common LISP standard was preceded by a plethora of relatively uncommon LISPs. There are a million different ways to create a LISP language that conforms to the LISP axioms for behavior. But there aren't currently a million LISP implementations to match these interpretations, because high-level semantics become more useful if there is one unique way to describe their meanings in operational terms. Even though the operational semantics of LISP are not particularly easy for the novice to grasp, these same semantics give the expert a strong and universally shared semantic model that aids in performance tuning and in debugging of the interpreter itself. If one person fixes a bug in this common model, everyone using the model benefits from the fix.

This is a hard fact for the typical system administrator to swallow. We pride ourselves in molding systems in our own images. We locate files where we can find them, and we structure documentation according to personal taste. This all comes with "being the gods of the machine," as one system administrator put it. Our tools, molded in our images, support and enforce the view that customization and molding systems to our own understandings is a necessary part of management.

It is not.

There are, in my mind, roughly three levels of maturity for a system administrator:

- Managing a host
- Managing a network
- Managing business process and lifecycle

As one matures, one gradually understands and adopts practices with increasingly long-term benefits of a broader view. But even at the highest level of maturity in this model, one is not done. There remains another level of understanding and achievement:

- Managing the profession

Managing the profession entails doing things as part of one's practice that benefit all system administrators, and not just the administrators at one's own site.

It would have been easy to allow LISP to "fragment" into many languages, at no cost to the individual programmer. There would have been, though, a cost to the profession if there were 100 LISPs. It would have limited sharing and would have stifled development.

But this is exactly the juncture where we sit with configuration management now. There are a million ways to assure behavior, and everyone has a different way. Our tools support and encourage this divergence. It is like having a million different LISPs with the same axiomatic semantics and different implementations, for no particularly good reason!

In other words, a semantic model is not enough to take system administration to the next level. That model must also be shared and common, and it must refer to and be implemented via shared base semantics.

To raise the level of modeling, it is necessary to do the following:

- Avoid incidental complexity and incidental variation.
- Seek shared standards.
- Evolve a common semantic base from those standards.
- Incorporate best practices in that base.

The end product of this process is a set of shared standards that form a common semantic base that tools can implement and support.

What does this mean to you? It is really simple: If something hurts the profession, stop doing it. One example of a hurtful practice is our arbitrarily differing ways for assuring behavior. We place our personal need to remember details over the professional need for standards and consistency.

- There are millions of ideas for where packages and files should be located in a running system. Let's all choose one and stick with it!
- There are millions of ways to configure services, all of which accomplish the same thing. Let's choose one of these.
- Life is much simpler if, for example, we choose as a profession to run each service on an independent virtual server.
- Let us endeavor to leave every system in a state any other professional can understand.

Let us utilize our tools not for divergence, but for convergence to a common standard for providing and maintaining services that is so strong in semantics that we can forget the underlying details and “close the boxes.” Let us support each other in protecting those standards against deviations that foster personal rather than professional objectives. If there is exactly one “best way” to provide a service, then we can all use that way, and the “institutional memory” of the profession as a whole becomes smaller and more manageable.

Will this ever happen? That is not a question of science, but one of sociology. Tool builders build their careers (and livelihoods) by encouraging adoption of “their personal views” on semantic intent. Meanwhile, the tools we have available for configuration management are still at the $x=1$ stage. One can throw abstraction at a problem—without semantics—and the intrinsic difficulty of the problem does not change. Only when we can define function based upon the abstraction, rather than upon its realization, can we move beyond abstraction to a workable semantics for configuration in which the internals of the configuration process become unimportant, as they rightly deserve to be.

This will be hard work, socially and technically, but the end product will be a profession whose common mission is to make all networks sing.