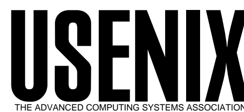
USENIX Association

Proceedings of the 17th Large Installation Systems Administration Conference

San Diego, CA, USA
October 26–31, 2003



© 2003 by The USENIX Association All Rights Reserved For more information about the USENIX Association:
Phone: 1 510 528 8649 FAX: 1 510 548 5738 Email: office@usenix.org WWW: http://www.usenix.org
Rights to individual papers remain with the author or the author's employer.
Permission is granted for noncommercial reproduction of the work for educational or research purposes.
This copyright notice must be included in the reproduced paper. USENIX acknowledges all trademarks herein.

Seeking Closure in an Open World: A Behavioral Agent Approach to Configuration Management

Alva Couch, John Hart, Elizabeth G. Idhaw, and Dominic Kallas – Tufts University

ABSTRACT

We present a new model of configuration management based upon a hierarchy of simple communicating autonomous agents. Each of these agents is responsible for a "closure": a domain of "semantic predictability" in which declarative commands to the agent have a simple, persistent, portable, and documented effect upon subsequent observable behavior. Closures are built bottom-up to form a management hierarchy based upon the pre-existing dependencies between subsystems in a complex system. Closure agents decompose configuration management via a modularity of effect and behavior that promises to eventually lead to self-organizing systems driven entirely by behavioral specifications, where a system's configuration is free of details that have no observable effect upon system behavior.

Introduction

Most system administrators accept that skilled system and network administration involves being a generalist: integrating bits and pieces of intricate and diverse minutiae into the skills to design a system, provide a service, or troubleshoot a problem. The wizards who can perform this integration teach the apprentices who are not yet wizards, and our configuration management tools are built in the image of the wizards, to allow more apprentices to function with less wizards among them. In short, current tools are aimed at teaching humans to manage an inherently complex process, and to embrace and even contribute to that complexity.

We believe that the complexities in a complex system are often illusory. Many are the result of less than thoughtful design, or at least, design not motivated by a goal of decreasing complexity to make systems more manageable. In this paper, we outline a strategy for reducing complexity and intricacy by changing the level of abstraction at which we interact with systems. If we adopt a new mindset and proceed according to a new set of rules, much of the complexity disappears. It is not defaulted or otherwise hidden by clever lingual mechanisms; it is literally gone and need not ever be considered again.

The key to this process is to "close the box" on subsystems that are sufficiently mature and allow them to become self-managing and self-healing in the absence of an administrator. This is a bottom-up process of administrative "practice hardening" in which we build overall system robustness upon a foundation of highly reliable low-level configuration subsystems that are tightly and inextricably coupled with behavior. These subsystems, together with a set of design rules for building interactive networks of subsystems, form a new paradigm for system administration. Almost everyone involved in configuration management is using some or most of these rules; this paper is an attempt to write down all of the known rules in one place.

With a few exceptions, most current configuration management tools function at an inappropriate level of abstraction. Specifications and declarations concern systems and networks, when they should instead document the behavior of *closures* and *conduits* between closures. A "closure" is a "domain of semantic predictability," a structure in which configuration commands or parameter settings have a documented, predictable, and persistent effect upon the externally observable behavior of software and hardware managed by the closure(more precise definitions will be discussed later). Closures are not entities that live just within one machine, but can also span LANs and networks. "Conduits" are methods of communication between closures, by which they can make their needs known to other closures. A conduit can take most any form, from a command-line interface to a custom networking protocol.

The idea of closure is not new. Many closed-source network devices and subsystems already exhibit some form of closure; commands are guaranteed to work and to be free of external effects. Service appliances and network switches are prime examples. We study instead how to create and maintain closure in an otherwise open environment subject to many changes and updates. We seek predictability in an otherwise unpredictable environment: "closure in an open world."

Configuration Management Challenges

Current configuration management systems all suffer from a similar set of problems that arise from the nature of the task. Among these problems, the

most important for our discussion are the problems of *referents*, *unintended consequences*, *hidden preconditions*, *latent variables*, and *incidental complexity*.

The *problem of referents* [11, 18, 19] arises from the complexity of the systems being configured. In a large and complex network, how does one specify how a particular subsystem should behave? This is a matter of referring to the subsystem and its parameters "by name" and assigning values to each parameter. The problem is that any naming scheme complex enough to precisely specify a subsystem is too complex to remember and use effectively. There are many approaches to hiding the problem of referents with clever language structuring [8, 11, 18, 19, 33].

For example, in Distr [11] and Arusha [18, 19], low-level parameter value declarations can be reduced or avoided via parameter defaults specified at a higher level. Environmental acquisition [33] allows parameter values to be inferred from the context in which a host must operate, much as a red automobile typically has red doors. Unfortunately, clever tricks such as value inheritance and environmental acquisition do not eliminate the problem; they simply transform the problem of referents into the equivalent problem of keeping inherited attributes correct in an increasingly complex inheritance scheme. As we will see, the solution is not to refine the solution, but to change the problem.

The *problem of unintended consequences* [15, 17, 40] arises because subsystems are often coupled in unforeseen and even undocumented ways. Very commonly, replacing a dynamic library to repair one application will break another. Similarly, installing one application can edit, e.g., /etc/inetd.conf, so that another application is disabled. We need some way of protecting ourselves from doing things that harm more than help.

Unintended consequences are often the result of *hidden preconditions*. Every management process, whether automated or manual, only works in particular environments. If we forget the environment and conditions under which a process is applicable, the process may have unintended consequences. An ideal process has minimal preconditions, and those preconditions are explicitly defined.

In turn, all hidden preconditions are the result of *latent variables*. A latent variable is a fact about a system that remains unseen until it causes a failure. For example, in a Linux server of about five years ago, the fact that one network card in a Linux server was manufactured by a particular vendor was unimportant, but if one added another identical card, a well-known networking bug would cause packets to be sent on incorrect interfaces. The manufacturer of the initial card is a latent variable that is not perceived to be a problem until it is *expressed* (like a gene) by adding another identical network card.

These problems obscure yet another problem that is not generally acknowledged, but is far more expensive in terms of human effort: the quandary of *incidental complexity*. While our mission as system administrators is to observe and assure particular kinds of system and network behavior, 95% of the information we currently specify (or perhaps inherit and override) during "configuration management" has no impact upon observable behavior.

A system is a graph of multitudes of interdependent minutiae, requiring knowledge of facts such as:

1. Where particular configuration files, programs, and subsystems are stored.
2. Which environment variables affect which applications.
3. Which disks are faster than others.
4. Which machines get which services from which others.

Bothersome facts about this minutiae include that:

1. An effective system administrator must be able to unravel all of it, ergo
2. One must set up patterns that are easy to remember, so
3. It must be consistent from system to system, even though
4. Its initial specification is not only subject to human error, but can also drift over time due to configuration changes.

The result of this culture is that configuration management systems, faced with the *tradition* of human management of irrelevancy, *streamline* this useless management task instead of more appropriately eliminating it entirely.

A pervasive and systemic problem sometimes requires a radical solution. If information is irrelevant to behavior, we leave that information to a better authority than ourselves: an expert system that figures out the various minutiae necessary to assure a behavior. This expert system takes the form of a suite of small and simply constructed configuration engines responsible for particular facets of behavior. These engines communicate with one another through a hierarchy that reflects the primary dependencies between subsystems. They are designed bottom-up and utilized top-down. A human communicates with the top level in order to effect changes at lower levels.

To solve the problem of incidental complexity, these configuration engines split configuration parameters into two distinct sets:

1. *Interior* parameters that are the responsibility of agents, and are read-only at all times (except during troubleshooting).
2. *Exterior* parameters that are the responsibility of the system administrator and are always read-write.

In a human-readable configuration, only exterior parameters matter, which greatly reduces the size of a "configuration." This division also leads to a less-than-obvious solution to the problem of referents. If referents are *behaviors*, instead of *parameters*, the exterior space of referents self-scales to a size a human administrator can easily memorize and internalize.

The agent approach also solves the problems of unintended consequences and latent variables for the *core* of a system, but not overall. Anything that is suitably constrained cannot behave badly, but it is impractical to constrain everything. The highest level of any system might need to remain open and evolving.

If as well, the exterior parameters are controlled by *constraining values* rather than specifying literal values, we have achieved the highest attainable level of abstraction in specification, a strategy first proposed by Burgess [7] and Anderson, et al. [2].

Closures

A ''closure'' is a programming language term [41] for a name-binding environment in which setting a variable and then reading it always gives the value to which it was set, independent of the settings of other things. In a closure, the meanings of names are independent of one another, unique, and persistent. Perhaps the simplest closure is `{ . . . }` in C. For example,

```
{ int x=4; { int x=1,y=5; } }
```

defines two nested closures, each with a distinct idea of the binding between variable `x` and a storage location. In the outer closure, `x` has the value 4, while in the inner closure, a different binding for `x` has the value 1. After the inner closure and inside the outer one, the symbol `x` briefly refers again to the one with value 4.

''Closure'' in a mathematical sense is a property of a mathematical system in which operations within the system do not produce results outside the system, e.g., we say that the integers are ''closed under addition'' because the sum of any two integers is an integer. Likewise, the integers are not closed under division, because 1/2 is not an integer. As another mathematical meaning of closure, in real analysis, a set of numbers is ''closed'' if the limit of any sequence of set members is present in the set.

Generally, in a closed system, no matter what one does under the rules, one obtains a result to which the rules still apply.

We apply the term *closure* to system administration in a form that embodies all of these prior meanings.

Principle 1: A *closure* is a subsystem with highly predictable, reliable, and robust behavior in response to configuration changes.

It is a domain of ''semantic predictability'' in which the behaviors one requests are portrayed exactly as requested and have a precisely predictable effect.

Coming to closure in a relatively static environment is easy; coping with change is the prime adversary of closure. The key to handling change is to base configuration management upon those attributes whose structure changes least frequently. The implementation of services is constantly changing and evolving, but the *nature* of services – what they do and how they behave – changes more slowly, if at all. We thus base our language of configuration upon *observable behavior* rather than parameter settings, where a behavior is observable if one can determine its presence or absence by asking a simple yes/no question. As software is updated and upgraded, overall behavior changes little with each new software revision, while the underlying low-level implementation may require regular and sometimes drastic improvements.

Parts of a Closure

A closure has four parts:

1. A set of conditions that define the environment in which the closure will operate properly. In software engineering terminology (as well as in the theory of program correctness [41]), these are called *preconditions*.
2. A set of *configuration operations* that work predictably when preconditions are met.
3. A set of *conduits* that allow exchange of information with other closures.
4. A map from configuration operations and parameter values to system behaviors, which specifies how each configuration change must affect the behavior of the overall system. In software engineering terminology, these are called *postconditions*.

The effect of a closure is that if one obeys preconditions and uses only closure operations and conduits, then the map from configuration to behavior will remain a definitive description of behavior. If one violates any of these restrictions, the map is no longer guaranteed.

We do not expect that anyone will argue about the value of predictability. The controversial part is that one must limit one's environment and practice to assure that predictability. It is not practical to allow ''anything'' to be done to any machine – the result is almost never maintainable or reproducible. In particular, one must not even attempt to configure a system other than by its approved conduits; to do so invites disaster.

Again, this concept is not new. Every software or hardware product embodies some kind of closure. If one installs the product as recommended (preconditions), and interacts with it using configuration operations listed in the documentation, it will hopefully behave as documented (postconditions).

Closures Without Agents

So how difficult is it to construct a closure? Practically, it is just a matter of limiting one's operations and certifying their effects individually and in concert. It does not require an agent, and can instead consist entirely of manual practices.

For example, at Tufts we certify only a few baseline configurations for desktop computers, including hardware and software specifications. These configurations form a closure in which the results of common upgrades become predictable. If a user deviates from a baseline by employing custom hardware or software, he moves outside the closure and (according to support rules) receives a lower level of support. If it is

impossible to repair his problems, staff will offer to put him back into the closure ('baselining') but will not debug problems encountered by a user who voluntarily leaves the closure. The closure allows us to provide a relatively high level of service to people within the closure, at the cost of aggressively discouraging people from leaving it. This saves much support time by avoiding costly troubleshooting sessions on systems outside the baseline.

Closure and Open Systems

Closures are not an attempt to "close systems," and the systems in which closures operate remain otherwise open.

Principle 2: A closure is a highly predictable subsystem of an otherwise open system.

While it is a good thing to be able to extend systems for any use, the act of extending them is often plagued by latent variables. A closure adds sanity to an otherwise open system by protecting a relatively mature subsystem from the effects of open extensibility. We protect the subsystem not by building physical barriers around it, but by agreeing to manage subsystems in a very disciplined and structured way. Closure of whole systems is not always practical. Instead, we locate subsystems that exhibit closure, and agree to leave them alone to perform their appointed tasks.

Again, this is an old idea with a new name. IP/DHCP is an example of such a closure. If we leave it alone and configure it via well-documented procedures, it will assure networking up to the session layer. Applications can leave this function to the operating system, thus ensuring predictability for IP functions. Likewise, network appliances form closures at the service level for file service, web service, proxying, etc.

Closures and Best Practices

There has been much discussion lately about the concept of "best practices" and their role in organizational robustness of service organizations [26, 30].

Principle 3: An ideal (well-designed) closure is an embodiment of best practices that never allows its subsystem to enter an unapproved or invalid state.

The limitations imposed by a closure and its resulting behavior are a standard amenable to validation and verification. Within the standard there is guaranteed predictability and interoperability; outside the standard, anything can happen.

Some closures are simple static preconditions on otherwise open systems. Software for Linux systems is plagued by incompatibilities between the distributions. The Linux Standard Base (LSB) [42] is a closure that limits the structure of the library binding environment for Linux distributions so that vendor software is guaranteed to work. It specifies standards for how distributions locate system files and libraries. Vendor software written to conform to the standards will work properly in any distribution that conforms to the standards.

LSB has the interesting property that its closure properties span distributions; any application that conforms to LSB standards and works properly in any one conforming environment is guaranteed to work in *all* conforming environments. There is no more need for "write once, debug everywhere;" debugging in one conforming environment is sufficient.

Coming to Closure

Though it is possible to buy highly predictable subsystems, a closure is not always something one buys or downloads. It is mostly a change in the way one thinks about the actions one takes in configuring a network. The main content of a closure is "additional rules of practice" that keep one from creating situations in which latent variables can appear. Break these rules, and one no longer has a closure. Follow them, and predictable behavior is assured.

The simplest closure of which we are aware is "Never delete a dynamic library." This discipline assures that you will be free of one latent effect, that of deleting a library that is in use. If one only installs software, and never overrides the contents of a dynamic library, then proper function of most programs is assured. If one ever deletes or replaces a dynamic library, havoc can result [15, 17]. All closures trade something for predictability and robustness. This practice costs one some administrative flexibility and disk space in return for a more stable and predictable system.

Closures take many forms and scales. A network appliance or thin client is a closure at the *system* level: an autonomous component with predictable and immutable behavior. DHCP can be viewed as a closure at the *network* level; it is a domain of predictable behavior in which clients are always assigned reasonable IP addresses. Network closure is also the purpose of Oracle's "StarNet" network middleware layer, among others; in this case the goal is reliable and secure communication with a database management system. A good operating manual often forms a closure at the *human* level: a set of procedures that "always work." All of these are subsystems that are somehow forged to be highly predictable for what they are intended to accomplish, regardless of whatever else is going on in a complex system.

Conduits

A *conduit* is in the context of a closure has the same meaning that it does in middleware. It is an interface between closures that reliably allows information exchange between them, and mediates configuration changes so that mistakes are less likely in either communicating closure.

Conduits can take many forms. Ideal conduits interface between subsystems with no administrative intervention, e.g., the gethostbyname function that interfaces with the domain name service.

An Ideal Closure

Closures vary in the effectiveness with which they eliminate latent effects. Almost everyone has to manage one or more web servers. Let us explore how practice would change if we tried to eliminate all latent effects by building a closure around a web server. This is going to take a little rethinking about how we ''configure'' web servers, but is both possible and practical.

In forming this closure, however, we will have to violate some very common conventions of user interaction, including shell accounts on the web server. We will do this with the confidence that it closes the system against unpredictable effects in ways that a shell-based system cannot be closed. In creating the closure, we will outline general principles one can apply to any situation.

Taxonomy of Behavior

To achieve a closure that ''acts like a web server,'' we will lift details of the configuration of a web server from its phenomenology as a server, i.e., the set of behaviors that make it a web server. The goal will be to describe the server from the outside, and allow the server to configure its internals without human intervention.

Principle 4: An ideal closure's configuration language is derived from a taxonomy of desired behaviors, not from the internal taxonomy of the system.

What comprises a web server's behavior? It has an IP address, responds to one or more names and/or ports, delivers content for each URL one provides, calls CGIs, and perhaps calls extension modules or interacts with database servers. So its configuration attributes should include:

1. *Identity:* IP addresses, virtual names, certificates, ports on which to listen.
2. *Customization:* Names of required modules, libraries, database bindings.
3. *Content:* A map from (virtual name, port) to a hierarchy of files, including service constraints.
4. *Auditing:* Ability to retrieve descriptions of service activity.

For now, let us consider this an exhaustive list of *everything* we provide to the server or get back from it. Everything else is going to be accomplished automatically without intervention from us.

Our pattern-slaved nature as human beings leads us to unavoidably fill in details where they are not needed, so it is difficult at first glance to see just how much we just left out of the web server's configuration. We left out:

1. The operating system to use.
2. The layout of the operating system.
3. The web server to use.
4. The representation of data.
5. The specific locations of files, including source, object, libraries, modules, etc.
6. Performance tuning.
7. Everything the web server requires of its environment that is not seen by the user.

Obviously these details must be ''added back in'' at some later time. Ideally the closure itself provides all these details during its efforts to install itself. We thus minimize what the administrator has to specify and learn.

There is not really any need for an administrator to know where files are kept on a functioning web server. This information is only useful if the server fails and one must troubleshoot it. If closure can arrange for it never to fail or to be self-correcting, then information on its internal structure is no longer needed.

Isolation

Our first configuration action will be to remove normal users and normal configuration tools from the system that will be a web server:

Principle 5: A closure is isolated from subsystems that might create latent effects. ed As part of our contract with the closure, we will ''let it manage itself,'' so we do not need the ability to manage it or interact with it otherwise. This is good, because:

Principle 6: An ideal closure's internal structure is completely opaque to the user; it can vary with circumstances and utilize the most efficient internal representation for a specific environment.

An excellent example of self-optimizing behavior may be found in [2].

Our web closure will presume that it is the sole manager of its configuration. Any violation of that contract will seriously affect our closure's ability to manage itself, because there will be latent effects of changes that the closure did not make during self-management. Our closure's ability to repair itself is greatly tempered by being able to control what changes occur and when. If it has complete control of its configuration, this is a matter of simple feedback algorithms, while if it did not have total control, it would instead be forced to rely on pattern matching and machine learning. We wish to avoid this due to the typical unpredictability of such control mechanisms.

In our particular case, data will be stored in the system in the way that's most efficient for the system, and need not reflect external architecture unless necessary for some internal reason. The reasons for this are obvious; the user and the storage medium are unnecessary constraints that can lower the efficiency with which we can maintain the closure. By removing these constraints, we also increase the size of the space of alternatives for implementing a solution. In particular, we can supply web pages in HTML while the closure stores them in XML.

Conduits

Conduits are the *sole* interface between the administrator and the closure.

Principle 7: A closure is self-centered and requires one to communicate all configuration changes and gather performance data through a gatekeeper conduit.

The gatekeeper conduit is the mediator during contractual disputes between administrator and closure. If you ask a closure to do something it can't do, the gatekeeper will reject it:

Principle 8: A closure is self-policing and validates configuration changes before making them.

This enforces *integrity constraints* that keep the closure configured in a proven way. The key to successful closure is that *every* achievable configuration corresponds to one of a set of *best practices* [26, 30], so that undesirable states are never entered.

The gatekeeper conduit may be a human being or software. The conditions on what a gatekeeper (of any kind) will accept must be carefully documented. Every closure's documentation must describe four things:

1. Prerequisites for setting up a closure, including hardware, software, network resources, etc.
2. Techniques for interacting with the closure as a holistic entity.
3. Postconditions and expectations: what will happen if you do this.
4. Consequences of violating preconditions: what will happen if something breaks.

Documentation is particularly important since it is our *only* clue to how a closure will behave in a specific situation.

For web services, there are at least three kinds of required gatekeeper conduits.

1. Configuration: Determines how the server behaves overall. This can be any kind of user interface that describes behavior other than content.
2. Content: For each virtual service, a conduit that describes data to be served.
3. Auditing: Describes log content.

Configuration is done using nothing more than the usual GUI or CLI (or preferably both) that describes what virtual servers to implement and what special behavior each should exhibit.

The content conduit is dramatically different. It is a closed interface to providing content that can support one or more of:

1. One-way or two-way mirroring of a filesystem available to developers.
2. An explicit CLI or GUI for creating content.
3. A database feed.

This roundabout way of providing content may seem bizarre until one realizes that we are attempting to completely eliminate operational coupling between the closure and the outside world, except through conduits, and arrange for robust service even if conduits are interrupted. Many sites already configure their web servers this way for precisely the same reasons, but call the practice "content staging."

Employing a conduit for content also allows the closure to utilize its own representation for the content that is independent of that provided by users. One thing that one does *not* want to do is to give the closure direct access to the files being updated by users. This creates points of failure that do not exist if these files are filtered through a conduit instead. For example, there is no way that a file provided through a conduit can have an invalid file protection, but a human manually arranging to publish a file can easily mis-protect the file in a variety of ways.

Environment

Few closures can be created in a vacuum. They must take information from the network in order to initially configure themselves, either from the Internet or from a specific host designated as a fileserver. They are also frightfully dependent upon having the resources they expect for disk space, memory, and network bandwidth. All we can do is to limit a closure's dependence upon the outside world in some strategic way:

Principle 9: An ideal closure is self-contained; it only depends upon external resources during configuration and is otherwise uncoupled from its network environment.

This eliminates many latent effects of changes in the environment, such as reboots of other servers, etc. A closure is like a binding contract: get this, do this, and this will happen. There are negative consequences as well as positive ones. Our web server will be very unhappy if it is memory-starved, and non-functional if it is disk-starved in storing content.

Preconditions

For our web server, let us presume that we document the following requirements:

1. 128 MB or more of main memory.
2. 10 GB of available disk.
3. Intel pentium 300 or above.
4. Platform in the RedHat certified list.
5. Extra hardware in the RedHat certified devices list.
6. Initially connected to a network supporting DHCP, with access to an appropriate RedHat repository.

This is more or less all we need to arrange an automatic build of a webserver with no human intervention.

Awareness

Principle 10: An ideal closure is responsible for assuring the integrity of its operating environment.

Documentation of the closure will state specific hardware and resource requirements, and service software will check for those requirements on an ongoing basis and fail otherwise. The reason is obvious; the closure is the best judge of what is needed for it to function. These constraints could be as general as ensuring hardware function or as specific as checking for integrity constraints in the operating environment. If it cannot change its environment to suit, the closure will request service from humans. You must then "adapt to service it."

In the case of our web server, the checks the closure should make are easy to construct. One can arrange

to check for available memory, disk, and devices at boot time, and fail if appropriate devices are not present. The ideal closure assesses its environment and creates its initial configuration based upon what it finds.

Principle 11: An ideal closure is self-organizing and self-installing.

An ideal closure could build itself in entirety on bare metal from a floppy disk and network. This is not at all unreasonable given the current state of the art, though it does limit one's operating system choices. We are aware of many people who have developed boot floppies for various kinds of services, notably cache servers, routers, firewalls, and thin clients.

Security

It is past due for our software to take some of the responsibility for its own security.

Principle 12: An ideal closure is self-hardening and self-repairing, and patches itself for security problems when information on the problems becomes available.

When an administrator decides that a patch is warranted, the closure itself installs it. This seems scary but is relatively easy to arrange, because the closure already certified the environment in which the closure operates. So we know whether a patch will install correctly if it installs correctly on another host running the same closure.

But the state of current monitoring and response technology goes far beyond simple patching. One can easily construct a web service environment in which available of service is constantly tested and the service environment is restarted whenever it crashes. In many cases, restarts are not emergencies and will simply appear in the administrator's normal log of activities for a server.

Adaptability

A closure is responsible for any flexibility one has in modifying hardware over time.

Principle 13: An ideal closure is self-reliant and deals internally with time-varying hardware changes and resource availability.

This could range from no flexibility, meaning that you can't change anything without a bare-metal rebuild, to total flexibility, that arranges to deal elegantly with complete machine replacement by periodically backing up servers and cloning them on other hardware.

Caveats

There are several invariants of closures that may not be obvious at first glance.

1. Closures are not necessarily portable. They depend upon a contract more specific than to run on any hardware whatever. They can be customized to a specific environment to make a *new* closure.
2. The contract for a closure is a homogeneity constraint. It assures that every closure starts building itself from the same basic set of resources. Closures are scalable provided that homogeneity constraints are satisfied system-wide, and unscalable otherwise. A closure can be replicated without bound once we know how to build systems on which the closure can live.
3. Closures reduce what one has to watch about a network and reduce possible troubleshooting causes. Something else other than the administrator is watching the closure. If anything goes wrong with the closure itself, it will be caught by the closure's own management mechanisms.

The above example is extreme, and much milder examples of closures can be created, e.g., we can make a web server that co-exists with user services by creating a different contract. We can create a closure for IP Telephony by, e.g., standardizing sound cards and drivers and creating an installer and maintenance engine driven by a central knowledge base.

Some seemingly obvious facts about closures are false. The union of several closures need not be a closure. Consider two closures:

1. `rpm --install` commands on an rpm-compliant host with matching rpm repository.
2. `make install` in source directories.

Both of these commands are closures in the sense that there is a sequence of appropriate operations that gives any desired effect in each closed world.

But the union of these two closures is seldom a closure. The problem comes from the way they interact when a `make install` (using autoconf) binds to a dynamic library provided by a distinct `rpm -install`. The rpm closure, which has inverses when considered as a closed world, no longer has inverses when considered as part of the `make install` closure. If this library is removed, the result of the `make install` will break. So the rpm command that installed the library cannot be undone without breaking the `make install` closure.

A Model of Closure

Further design principles require a precise definition of closures. This section is difficult reading and can be skipped without much loss of continuity. Here we precisely define the concept of closure and mathematically illustrate some more esoteric principles of closure design.

Our concept of closure will be a property of a set of software agents. An *agent* is an autonomous process, within a system or network, that accomplishes changes based upon external commands from a system administrator or other agents. Agents can be embodied in physical processes or combined into a single process with multiple functions, as in Stem [25].

In designing a closure, there are a few inescapable and well-justified principles that limit our design choices. To start,

Axiom 1: A usable set of agents must be convergent, consistent, aware, and atomic [14].

1. A *convergent agent* is defined as a software process that will do nothing unless something is amiss, and will correct anything within its domain of change that becomes non-compliant. This property is necessary to limit the intrusiveness of using an agent, and limit downtime due to agent actions. "If it ain't broke, don't fix it."
2. A set of agents is called *consistent* if agents will not undo other agents' actions. This we formerly called *homogeneity* [14]. This is necessary so that a set of agents will, at a particular time, agree upon a state to assert for a system.
3. An agent is called *aware* if it knows whether an error occurred in making a change. This is necessary in order to ensure atomicity, below.
4. In database theory, an action is *atomic* if it either succeeds and accomplishes a change, or fails with no change whatever. Similarly, an *atomic* agent leaves a system completely unchanged when an error occurs during a configuration change.

Convergence and consistency assure that cooperating agents will not interfere with one another or the user. Awareness and atomicity preserve integrity of the underlying system in case of configuration failures, and assures that if the system cannot be forced into compliance with the ideal, that the act of enforcement does not create further problems. In other words, "a closure should do no harm."

Ideal agents treat the process of configuration as if it consists of database transactions [22, 23], where allowable transactions include constraints as well as literal assertions of state. These transactions are subject to integrity constraints and will fail and do nothing if a desired state cannot be achieved.

Alas, the above obvious properties of an agent are not enough to allow us to construct a system management environment from cooperating agents. Informally, we say that "each agent manages a closure." The exact and precise meaning of this takes some work to develop. One deep problem concerns the meaning of *consistency* between agents. The most challenging problem is to design closures so that local consistency between pairs of closures is sufficient to assure global consistency of closures as a set. Pairwise consistency requires up to $O(n^2)$ operations to verify, where n is the number of closures, while global consistency may require up to $O(2^n)$ consistency checks, one for each subset. The rest of this section discusses how to avoid the latter expensive process of consistency checking.

Parameters and Configurations

In understanding the mathematics of closures, we must first define the notion of what a closure does. In the following, subsystems and closures (a special kind of subsystem) are notated in capitals, while their attributes are notated as script capitals subscripted with the subsystem to which they apply, e.g., $\mathcal{V}_A$.

Definition 1: For each *subsystem* A, let $\mathcal{V}_A$ be the set of its configuration parameters.

'V' stands for *variant*. We purposefully leave the concept of parameter relatively unconstrained. To handle complex situations, some parameters may be "latent" or "unexpressed," like "unexpressed genes." For example, in a web server, theoretically every directory on the system has an access list, but only the access lists of directories that serve as web content are expressed by having an impact upon operation. So a subsystem may have an infinite number of possible parameters, though only a finite number are expressed at any time by actually controlling behavior.

We next need to allow parameters to have values.

Axiom 2: Without loss of generality, we can consider each configuration parameter value to be a string from some fixed alphabet Σ.

Since a string can contain any structure, including XML, arbitrary hierarchical relationships can be portrayed.

Definition 2: A configuration c of A is a mapping

$$c : \mathcal{V}_A \rightarrow \Sigma^*$$

where Σ is an alphabet of configuration symbols and Σ^* is the set of all words (i.e., sequences of characters) that can be formed from the alphabet Σ.

Configurations are assertions that indicate what should be true of a system. They are ideals; the actual parameter values on the system may differ due to external influences. For $p \in \mathcal{V}_A$, $c(p)$ represents its value in the configuration c.

Definition 3: For a subsystem A, let $\mathcal{U}_A$ denote the set of all possible configurations.

'U' stands for *universal*. This is a very large set that, while actually finite due to memory bounds, might as well be infinite. Fortunately, we can limit ourselves to studying a reasonably small subset.

Definition 4: For a subsystem A, let $\mathcal{D}_A \subset \mathcal{U}_A$ be the set of all *reasonable* configurations of A. These are configurations that, according to some standard of practice, make sense.

'D' stands for *domain*. These configurations comprise the domain of change that an an agent can understand and manipulate. The contents of $\mathcal{D}_A$ are a set of choice configurations that exhibit appropriate constraints and behaviors. The structure of these constraints will be a design choice in building an agent.

At any particular time t, a subsystem A exhibits exactly one configuration $c_{A,t}$ that may or may not be a member of $\mathcal{D}_A$. In this paper, however, we will study properties of configurations that do not depend upon time, so that our model will ignore the time-varying nature of configuration and concentrate instead upon static structure. In the argument that follows, all choices we make apply to a single time step. One excellent time-varying model of configuration may be studied in [7]; our time-varying model (which differs somewhat) is left as future work.

Policies

The following is borrowed from the work of Burgess [7] and Anderson [2] but used in a new way.

Definition 5: For each subsystem A (at the time step that we consider), a *policy* is a subset $\mathcal{Q}_A \subseteq \mathcal{D}_A$.

The intent of a policy is to describe a set of options for configuring a system. Each element of a policy describes a reasonable configuration that can be applied to a system. Ideally, the options in a policy result in similar behaviors when applied to the system being configured. Like configurations, policies are ideals. They describe what should be set in the configuration of a machine, not what actually appears there due to the effects of time and change.

Definition 6: A subsystem A's configuration is *compliant with a policy* $\mathcal{Q}_A$ (in the time step that we consider) if the actual configuration $c_A \in \mathcal{Q}_A$.

If a system is compliant with a policy, all parameter values specified in one configuration of the policy are echoed in the actual configuration of the machine.

Definition 7: For a particular subsystem A, let $\tilde{\mathcal{P}}_A$ represent the set of all reasonable policies for A.

A reasonable policy is one that has been implemented, validated, and incorporated into a site practice manual. $\tilde{\mathcal{P}}_A$ is a set of sets of configurations. Each $\mathcal{Q}_A \in \tilde{\mathcal{P}}_A$ is a set of configurations, while each configuration $c \in \mathcal{Q}_A$ is a reasonable configuration in $\mathcal{D}_A$. Again, this is not the set of all *possible* policies, but rather the set of all *reasonable* ones that we think will have acceptable results. The former set is staggeringly huge; the latter is no larger than the detail of one's practice manual.

Policies are ''constraint spaces'' in the sense of Burgess [6, 7] or Anderson [2]. A policy is not a definition of what must happen, but rather a list of reasonable options. In the current model, for simplicity, there are no priorities or weights for policy options; any option in the list is fine.

We realize that this is a misuse of the word *policy* (which many authors believe to be ''that which is determined by management'') but at least, it is a consistent misuse among *this* paper's references!

Behavior

One new idea of this paper is to utilize *testing* and *validation* as the definition of external behavior. This continues the work started in [15, 17].

Definition 8: A behavioral test is a yes/no question that determines whether or not a configuration has a particular property. For each subsystem A, let $\mathcal{T}_A$ be a set of behavioral tests that characterize its external behavior.

These are yes/no questions that determine whether a configuration has a particular property or not. Sample tests might include:

- Does A run a web server on port 80?
- Does A not answer tftp requests?
- Does A have a directory named /var/local?

Questions do not have parameters; ''Does A run a web server on port 8080?'' is a different question than the one above.

Closures

Now we are ready to define the exact nature of a closure.

Definition 9: A *closure* is a quintuple (A, $\mathcal{V}_A$, $\mathcal{D}_A$, $\tilde{\mathcal{P}}_A$, $\mathcal{T}_A$) such that for any policy $\mathcal{Q} \in \tilde{\mathcal{P}}_A$ ($\mathcal{Q} \subset \mathcal{D}_A$), any configuration $c \in \mathcal{Q}$, and any test $f \in \mathcal{T}_A$, the value of $f(c)$ does not depend upon the particular choice of $c \in \mathcal{Q}$.

In other words, policy unambiguously determines behavior. To ease notation, we refer to the quintuple (A, $\mathcal{V}_A$, $\mathcal{D}_A$, $\tilde{\mathcal{P}}_A$, $\mathcal{T}_A$) as the closure A.

Note that the idea of closure depends very much upon what we value (or wish to avoid) through testing:

Principle 14: Designing a closure requires choosing first exactly which kinds of behaviors the closure should be able to produce by configuring a system.

This is embodied in the tests $\mathcal{T}_A$ that determine, among other things, which behaviors are important and which are frivolous. Important behaviors correspond to tests; incidental or frivolous behaviors are those for which there is no test included.

Note that this is as much a condition on the structure of reasonable policies as it is upon their effects. For example, if there is one and only one reasonable policy, and tests always have the same results, we have an administratively trivial closure for which there is only one behavioral result. This means that:

Principle 15: Designing a closure requires carefully delimiting what is reasonable and appropriate, as integrity constraints on parameter space.

In most systems, randomly choosing values for parameters results in chaos. The choices for reasonable configurations $\mathcal{D}_A$ and reasonable policies $\tilde{\mathcal{P}}_A$ exclude such chaos.

If we know A is a closure, then there are maps from configuration and policy to behavior.

Definition 10: For a closure A (which is the quintuple (A, $\mathcal{V}_A$, $\mathcal{D}_A$, $\tilde{\mathcal{P}}_A$, $\mathcal{T}_A$)), let τ_A denote a map from configurations to behaviors

$$\tau_A : \mathcal{D}_A \to (\mathcal{T}_A \times \{\text{true}, \text{false}\})$$

such that for each configuration $c \in \mathcal{D}_A$,

$$\tau_A(c) = \{(f, f(c)) \mid f \in \mathcal{T}_A\}$$

where $f(c)$ is the (boolean) result of applying test f to a system compliant with the configuration c.

The action of the test suite, $\mathcal{T}_A$, on a particular configuration is to tabulate the results of running all tests, filed by test name and result. The result is a set of ordered pairs $\tau_A(c)$. Because A is a closure, each $f \in \mathcal{T}_A$ can appear in exactly one ordered pair, $(f, f(c))$.

Generalizing this to policies that are sets of configurations:

Definition 11: For a closure A, let $\tilde{\tau}_A$ denote a map from policies to behaviors, where
$$\tilde{\tau}_A : \tilde{\mathcal{P}}_A \to (\mathcal{T}_A \times \{\text{true}, \text{false}\})$$
such that for each policy $\mathcal{Q} \in \tilde{\mathcal{P}}_A$,
$$\tilde{\tau}_A(\mathcal{Q}) = \{(f, f(c)) \mid f \in \mathcal{T}_A, c \in \mathcal{Q}\}$$
where $f(c)$ again represents the (boolean) result of applying test f to a system compliant with the configuration c.

As before (since A is a closure), each $f \in \mathcal{T}_A$ appears in exactly one pair, $(f, f(\mathcal{Q}))$, in $\tilde{\tau}_A(\mathcal{Q})$.

Note that both of these definitions are nonsense for non-closures, because in the absence of a closure, the results $\tau_A(c)$ and $\tilde{\tau}_A(\mathcal{Q})$ may differ over time even for constant choices of c and $\mathcal{Q}$! By the definition, A is a closure if and only if $\tilde{\tau}_A$ makes sense as a function.

Provided that A is a closure, the contents of $\tilde{\mathcal{P}}_A$ are constrained. Each element of $\tilde{\mathcal{P}}_A$ must be a subset of some *inverse image* of a unique set of behavioral test results, e.g., for a set of test results $L \subset (\mathcal{T}_A \times \{\text{true}, \text{false}\})$,
$$\tau_A^{-1}(L) = \{c \in \mathcal{D}_A \mid \tau_A(c) = L\}$$
Note that for every valid set of test results $L \subset (\mathcal{T}_A \times \{\text{true}, \text{false}\})$, each test appears only once on the left-hand side.

Definition 12: Let $\mathcal{L}_A$ be the set of all possible test results, i.e., all subsets of $(\mathcal{T}_A \times \text{true}, \text{false})$ where each first coordinate appears exactly once.

Then the inverse images of $L \in \mathcal{L}_A$ under τ_A partition $\mathcal{D}_A$ into disjoint subsets, one per non-empty inverse image. Each element of the partition is a candidate policy for the closure, i.e., a set of configurations with identical behavior. In choosing $\tilde{\mathcal{P}}_A$, for A to be a closure, each element of $\tilde{\mathcal{P}}_A$ must be a subset of one inverse image; otherwise policy would not uniquely determine behavior.

Since multiple policies with the same behavior represent wasted effort, without loss of generality we can set
$$\tilde{\mathcal{P}}_A = \{\ \tau_A^{-1}(L) \mid L \in \mathcal{L}_A\ \}\ .$$
This is the set of all subsets of configurations corresponding to unique observable behaviors $L \in \mathcal{L}_A$. Thus

Principle 16: The structure of parameter space for a closure is dependent upon the limited taxonomy of a finite number of *desirable* behaviors, not upon the *possible* behaviors that result from arbitrary parameter settings.

The space of desirable parameter combinations $\tilde{\mathcal{P}}_A$ for a closure A is almost always much smaller than the set of all possible parameter values. It is instead a compendium of best practices for the target environment [26, 30]. This expertise factor makes the inverse function practical to enumerate by iterating over and testing all policy choices. Otherwise, determining the inverse would be intractable.

It is particularly important to limit predictability to a finite subset of objective tests. Many mistakes in reasoning have been made by presuming that behavior includes "all" that a system can do. The observable behaviors are a finite set, while "all behaviors" include minutiae that are not important, e.g., the host's specific MAC address. By constraining the tests, we remove tests that have no impact upon usability. Failing to constrain tests to a finite set is the root of several posed theoretical problems of system administration [20, 38].

Consistency

One of our tenets is that at any time, the set of all closures operating upon a system must be "consistent." We have not yet precisely determined what consistency means. If the closures have disjoint parameter spaces, then they are trivially consistent because conflicts are impossible. Consistency is only nontrivial when two closures share a resource or parameter. The exact nature of that sharing is yet to be determined, and there is a danger of over-limiting closures so that they become impractical to construct. We must define consistency for configurations, policies, and closures.

Configurations are consistent if they agree upon the values of common parameters.

Definition 13: Two configurations $c \in \mathcal{D}_A$, $c' \in \mathcal{D}_B$ are *consistent* if for every parameter $p \in \mathcal{V}_A \cap \mathcal{V}_B$, $c(p) = c'(p)$.

Definition 14: A collection $\mathcal{M}$ of configurations is consistent if any pair of configurations $c, c' \in \mathcal{M}$ are consistent.

In this case, consistency of pairs is sufficient to assure consistency of arbitrary subsets. This is not true in general.

Consistency of two policies means that the policies can both be applied at the same time without conflict:

Definition 15: For closures A and B with policies $\mathcal{Q}_A$ and $\mathcal{Q}_B$, policy $\mathcal{Q}_A$ *admits* $\mathcal{Q}_B$ ($\mathcal{Q} \rhd \mathcal{Q}_B$) if for every $c \in \mathcal{Q}_A$, there is a $c' \in \mathcal{Q}_B$ such that $c(x) = c'(x)$ for all $x \in \mathcal{V}_A \cap \mathcal{V}_B$.

In words, $\mathcal{Q}_A$ admits $\mathcal{Q}_B$ if for all configurations of A compliant with A's policy $\mathcal{Q}_A$, there is a matching configuration of B compliant with B's policy $\mathcal{Q}_B$. If every possible configuration of either A or B is compatible with some configuration of the other, we have consistency of policies:

Definition 16: Policies $\mathcal{Q}_A$ and $\mathcal{Q}_B$ are *consistent* if $\mathcal{Q}_A$ admits $\mathcal{Q}_B$ and $\mathcal{Q}_B$ admits $\mathcal{Q}_A$ ($\mathcal{Q}_A \rhd \mathcal{Q}_B$ and $\mathcal{Q}_B \rhd \mathcal{Q}_A$).

Consistency of two closures means that the policies for the two agents are coupled, so that on common parameters, a reasonable state for one is also reasonable for the other. If the agent for one policy is invoked before an agent for another consistent policy, the second agent will not change parameters that the first agent already corrected, unless these have in the meantime been changed again by an outside force.

The mystery here is how to choose policies for a large set of closures that will be mutually consistent. Informally, a set of closures $\mathcal{S}$ is *mutually* consistent if for any subset

$$\mathcal{I} = \{I_1, \ldots, I_k\} \subseteq S$$

and any consistent choice for policies $\mathcal{Q}_j \in \tilde{\mathcal{P}}_{I_j}$, there are policies for the remainder of $\mathcal{S}$ that remain consistent with this initial set of policy choices. This definition, however, is too naive; closures are not created equal. Some are more likely to be controlled by administrators than others, so that mutual consistency must be based upon choosing policies for those distinguished closures first.

Dominance

Some closures are more important than others in describing consistency.

Definition 17: Closure A dominates B (written $A \succ B$ or $B \prec A$) if for each policy $\mathcal{Q}_A \in \tilde{\mathcal{P}}_A$, there is at least one policy $\mathcal{Q}_B \in \tilde{\mathcal{P}}_B$, such that $\mathcal{Q}_A$ admits $\mathcal{Q}_B$ ($\mathcal{Q}_A \rhd \mathcal{Q}_B$).

A dominates B if it is possible to bring B into compliance with A by choice of some policy for B. While consistency is a property of a configuration, dominance is a property of the sets of all policies: a constraint on the globally achievable states for both A and B.

Definition 18: Closures A and B are *mutually dominant* if A dominates B and B dominates A.

This is common when the closures are tightly coupled, e.g., DNS and DHCP. Putting a host into DHCP *should* correspond with putting it into DNS and vice-versa.

Some seemingly obvious statements about dominance are false in general.

Proposition 1: Dominance is not necessarily transitive ($C \succ B$ and $B \succ A$ does not necessarily mean that $C \succ A$).

Proof: Construct closures A, B, and C with $\mathcal{V}_A = \{x, y\}$, $\mathcal{V}_B = \{y, z\}$, and $\mathcal{V}_C = \{x, z\}$, and let the policies be assigned as follows (also see Figure 1):

$$\tilde{\mathcal{P}}_A = \{x = 0, y = 0\}$$
$$\tilde{\mathcal{P}}_B = \{y = 0, z = 0\}$$
$$\tilde{\mathcal{P}}_C = \{x = 1, z = 0\}$$

Then $C \succ B$ ($x = 1, y = z = 0$) and $B \succ A$ ($x = y = z = 0$), but $C \not\succ A$ (x values incompatible). □

Dominance is important because it allows behavior to be developed in a stepwise process. If $A \succ B$, then choosing a behavior (via a policy) for A partially determines the allowable behaviors for B, with the remaining options determined by the nature of B or the administrator.

Consistency of Closures

Now we are ready to define consistency of closures. First we need to distinguish between constrained and unconstrained closures in an arbitrary set of closures $\mathcal{S}$:

Definition 19: In a set of closures $\mathcal{S}$, a closure $A \in \mathcal{S}$ is *exterior* if everything that dominates it is mutually dominant with it, and *interior* otherwise.

An exterior closure is one whose parameter values control the parameter values of other closures.

Definition 20: An *exterior cover* for a set of closures $\mathcal{S}$ is a subset $\mathcal{I} \subseteq \mathcal{S}$ where each $u \in \mathcal{I}$ is exterior, every element $s \in \mathcal{S}$ is dominated by some $u \in \mathcal{I}$, and there are no mutually dominant pairs of closures in $\mathcal{I}$.

'I' stands for *interface*. An exterior cover is a set of closures that, once configured, determine the configuration of every closure (though perhaps not uniquely). First, an exterior closure must be "at the highest level" of a dominance hierarchy, with nothing dominating any closure in it that is not mutually dominant with the closure. We require that every element of $\mathcal{S}$ is dominated by something in $\mathcal{I}$, so that the configuration of every element is thereby constrained. A mutually dominant pair is redundant; only one of the pair need be included in order to control both elements of the pair. There are often many choices for an exterior cover due to mutual dominance.

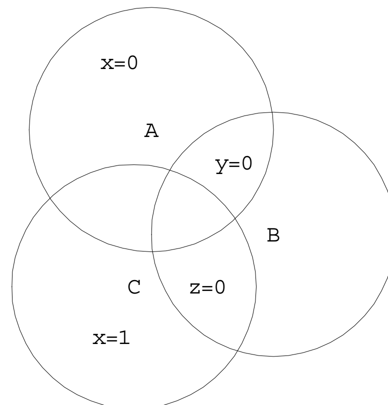


Figure 1: How transitivity of dominance fails.

An exterior cover for a set of closures is a choice for how a human might communicate desires to a set of communicating closures. The exterior closures are the ones with which a human communicates; the interior ones are all implicitly configured by settings for the exterior ones.

We can now characterize what it means for a set of closures to be consistent:

Definition 21: A set of closures S is *consistent* if, given any choice of policies for any exterior cover $\mathcal{I} \subset \mathcal{S}$, there are choices for policies for all non-exterior closures in $\mathcal{S} - \mathcal{I}$ (where $-$ indicates set difference), as well as choices of particular configurations within each policy, so that the set of all resulting configurations is consistent.

Remember that dominance allows us to determine how to make a *pair* of closures consistent. Consistency of a *set* of closures is a much stronger criterion

than pairwise dominance, and pairwise dominance is not enough to assure consistency. In particular,

Proposition 2: A set $\mathcal{S}$ of closures in which for each A and B, either $A \succ B$ or $B \succ A$, and for which $\succ$ is transitive, is *not* necessarily consistent!

Proof: Construct closures A, B, and C with $\mathcal{V}_A = \{x, z\}$, $\mathcal{V}_B = \{y, z\}$, and $\mathcal{V}_C = \{x, y, z\}$ and let the policies be assigned as follows:

$$\tilde{\mathcal{P}}_A = \{ \{x = 0, z = 0\}, \{x = 1, z = 1\} \}$$
$$\tilde{\mathcal{P}}_B = \{ \{y = 0, z = 1\}, \{y = 1, z = 0\} \}$$
$$\tilde{\mathcal{P}}_C = \{ \{x = y = 0, z = 0\}, \{x = y = 1, z = 1\}, \{x = y = 0, z = 1\}, \{x = y = 1, z = 0\} \}$$

Transitivity of dominance between these closures is obvious from inspection. A and B form an exterior cover, but there is no policy *globally* consistent with $x = 0, z = 0$ in $\tilde{\mathcal{P}}_A$, because y must be 1 in $\tilde{\mathcal{P}}_B$ and 0 in $\tilde{\mathcal{P}}_C$. By the structure of A, B, and C, there are two conflicting assignments for the same parameter y. □

But, with a few more conditions, we can assure global consistency of closures.

Proposition 3: Let $\mathcal{S}$ be a set of closures such that:
1. For each $A, B \in \mathcal{S}$ where $\mathcal{V}_A \cap \mathcal{V}_B$ is nonempty, either $A \succ B$ or $B \succ A$.
2. If $A \succ B$, then $\mathcal{V}_A \supseteq \mathcal{V}_B$.

Then $\mathcal{S}$ is consistent.

Proof: Note first that in this case, for closures A, B, C, if $A \succ B \succ C$, then $\mathcal{V}_A \supseteq \mathcal{V}_B \supseteq \mathcal{V}_C$, so that by dominance, values of $\mathcal{V}_A$ determine the values in $\mathcal{V}_C$. Thus $A \succ C$, and $\succ$ is transitive.

Note also that any conforming set of closures has the structure of a forest of disjoint trees, as illustrated in Figure 2. Dominant closures are ancestors of dominated closures, where a closure is dominant whenever it contains more parameters than another.

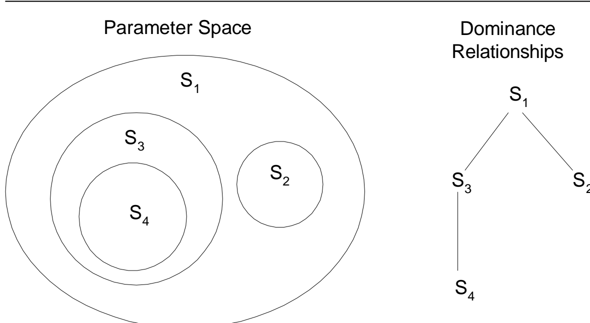


Figure 2: Subset relationships in parameter space exhibit a tree structure.

If $\mathcal{S}$ contains one closure, we are done. If $\mathcal{S}$ contains two closures, then either their parameter sets are disjoint or one closure dominates the other. In the former case, we are done, because the two closures agree on an empty set of common parameters. In the latter case, the dominating closure contains the parameter space of the other, so by definition of dominance, the closures agree on parameter values and are consistent.

If $\mathcal{S}$ contains three closures A, B, C, then if any one has a set of parameters that is disjoint from the other parameter sets, then we have only two closures with common parameters, and we are done by the previous argument. So presume that all three parameter sets have elements in common. By enumerating the possibilities for dominance, one of the three closures must always contain all parameters of the others and dominate the other two, so we are done.

Now assume that the proposition is true for n closures, and consider $n + 1$ closures $\{S_1, S_2, \ldots, S_{n+1}\}$. This set must contain at least one interior closure S_k that is mutually dominant with anything that it dominates; this S_k is a lower bound for the finite set of closures under $\succ$. By induction, the theorem is true for

$$\{S_1, S_2, \ldots, S_{k-1}, S_{k+1}, \ldots, S_{n+1}\}$$

so that the latter set of closures (without S_k) is consistent.

Now consider the possible configurations of S_k. As it is an interior node, either its parameter space is disjoint from that of all other closures (and it stands alone) or it is dominated by some other closure S_m (with which it may be mutually dominant). If its parameter space is disjoint from all others, it is trivially consistent with the rest and we are done. If there is overlap, then choose some other closure S_m that dominates S_k (Figure 3). As S_m is a member of the consistent set of closures except for S_k, setting S_k's configuration from S_m results in a consistent set of configurations overall. We know we can do this, and that it unambiguously instantiates all parameters, because $S_m \succ S_k$ and $\mathcal{V}(S_m) \supseteq \mathcal{V}(S_k)$. □

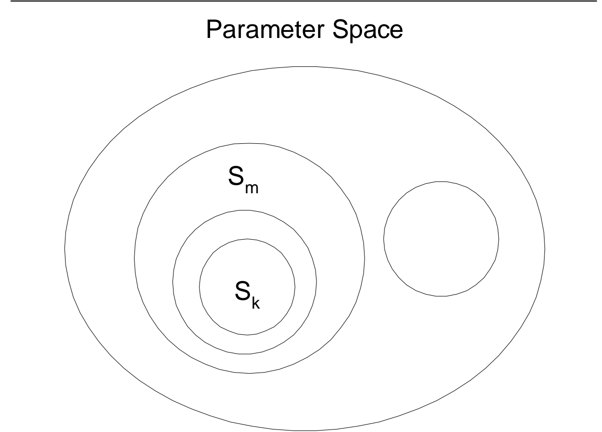


Figure 3: S_k is dominated by S_m.

The proposition describes a convenient way to assure consistency of closures from a condition on all pairs of closures:

Principle 17: To maintain consistency of closures, it is sufficient to maintain copies of parameters and values of dominated closures inside the parameter space of dominating closures.

The proposition and principle are not *optimal*. One can organize one's closures as a tree, with one closure containing the parameters for all the others, and consistency is assured. This does not rule out other schemes for assuring consistency. There are many consistent sets of closures that do not obey this stricture. There is room for much future work on the optimal structure of consistent sets of closures.

For now, without further theory, higher-level closures must maintain copies of lower-level configuration data, or face potential inconsistencies. As a side-effect, this inclusion makes dominance transitive. This copying may seem a bad thing but alas, not all hope is lost, because what dominating closures have to copy is actually rather limited. The structure imposed by the proposition leads to a strict hierarchy (tree) of closures, where one overarching closure contains (or has knowledge of) all of parameter space.

Exterior Parameters

An *exterior parameter* of a closure is a parameter that – according to the previous principle – we must copy upward in a chain of closures.

Definition 22: A set of parameters $E \subseteq \mathcal{V}_A$ is *exterior* if for all $c \in \mathcal{D}_A$, the values in

$$c|_E = \{(p, c(p)) \mid p \in E\}$$

uniquely determine the outcome of tests in $\mathcal{T}_A$.

This is a global constraint on all achievable configurations. $c|_E$ is read "c restricted to domain E." In other words, given the values of parameters in E, all policies compliant with these parameter values have identical behavior. More formally, given $E \subseteq \mathcal{V}_A$ and a configuration c, the configurations that agree with values in $c|_E$ are interchangeable. This means that

$$\{D \in \mathcal{D}_A \mid \forall p \in E, c|_E(p) = D(p)\}$$

all have identical behavior.

A set of parameters is exterior if, given these, the behavior of the closure is completely determined, regardless of the other parameters, modulo the behaviors we can observe. Interior parameter values are dependent upon the exterior parameter values. If a tree falls in the forest and no one hears it, it *does not* make a sound!

Interior parameters most frequently appear in very complex systems, e.g., web servers. For example, the port number of a virtual server is exterior; the name of its root directory may not be. The former is necessary to pass any behavioral test; the latter may change without affecting external behavior at all.

It immediately follows that:

Proposition 4: In a consistent set of closures, behavior is completely determined by the values for all exterior parameters of all closures.

Proof: Let $S_1, \ldots, S_n$ be a consistent set of closures. Let $\mathcal{V} = \bigcup_i \mathcal{V}_{S_i}$ represent the combined set of all parameters. Let $c: \mathcal{V} \to \Sigma^*$ be a consistent configuration for all parameters of all closures.

By definition of exterior parameters, in a particular closure S_i, the values of c restricted to exterior parameters $\mathcal{E}_{S_i}$ completely determine the behavior of that closure. As S_i was arbitrary, this is true overall. □

For consistency's sake, when we make copies of parameters in dominating closures, only the exterior parameters need to be copied in this manner. In other words:

Principle 18: At any level of a dominance hierarchy, all the exterior parameters of dominated closures need to be present as part of the configuration.

This does not mean that these parameters are necessarily exterior at the *current level*; they may be interior and their values under control of the dominant closure or some other dominated closure.

For example, in an intranet closure, the port on which an internal webserver runs is interior to the intranet but exterior to the servers serving the intranet.

Note that the fact that a parameter is exterior means that it determines interior parameters, not that interior parameters do not exist. The exterior command "be a fast webserver" instantiates a large number of interior parameters, including speed of CPU, disk, ethernet, etc. The latter comprise the options that one can take in creating a fast web server, and the possibility of multiple policies that accomplish this allows that "there's more than one way to do it."

Interface

So far, we have considered the internal structure of a closure, including local parameters and propagation of exterior parameters. All configuration management tools so far define their interface in terms of this parameter space. This is a complex and trying task; it is much easier to base the interface on behavior rather than parameterization.

Note that in our model, the reasonable policies $\tilde{\mathcal{P}}_A$ of a closure A map to a set of behaviors

$$\tilde{\tau}_A(\tilde{\mathcal{P}}_A) = \{ \tilde{\tau}(\mathcal{Q}) \mid \mathcal{Q} \in \tilde{\mathcal{P}}_A \}$$

ideally in a one-one fashion, because $\tilde{\mathcal{P}}_A$ is constructed as the inverse image of the test space. This means that *the behaviors determine the reasonable parameter sets that assure each behavior.* Thus it is possible to

Principle 19: Specify behavior, and allow the configuration management system to derive appropriate parameters to assure that behavior.

This idea is borrowed from Anderson [2]. In the current version of LCFG, one maintains grids of computers by specifying constraint spaces for their configuration, from which a reasonable configuration is chosen. For example, one simply asserts that "every network has a DHCP server," and the SmartFrog framework allows stations to vote on the best location for the DHCP server. In the case of LCFG, constraints are still formulated upon configuration parameters, whereas in our case, constraints ideally act upon behavior, independent of the

structure of parameter space. This is admittedly a difficult distinction to make, and one may argue that the structure of LCFG's parameter space accomplishes the same objective.

This has a subtle effect upon parameter propagation upward across dominance boundaries. We already know that it is only necessary to expose exterior parameters, and that behavior induces the appropriate parameter configurations. An upper closure that is informed enough to know the map from behavior to parameter values need not store the values; the desired behaviors are enough to store.

Comparison With Other Approaches

To understand the differences between this approach and typical configuration management systems, consider how we would configure typical network services using the new system. We decompose the problem of providing service into a dominance hierarchy in which the leaves are the files and processes that control service provision. In a typical UNIX system, these files might include /etc/inetd.conf, /etc/services, /etc/protocols, /etc/hosts.allow, /etc/hosts.deny, and other files intended to configure specific services.

In traditional configuration management, the contents of each of these files is generated by custom scripts from an overall specification [1, 2, 18, 19, 22, 23, 36] or edited in place by incremental scripts [20, 37, 38] or convergent rules [4, 5, 6, 12]. In our system, each is instead wrapped by an agent that treats the underlying file more like a database rather than a flat file. This agent takes responsibility for all changes to the file and normalizes the form of the file (by sorting) so that the effects of independent changes to the file do not depend upon the order of changes. These agents are convergent in a stronger sense than Cfengine file editing (as proven mathematically in [16]) and replace convergent and scripted file editing with a more stable and predictable alternative.

Higher-level configuration management is accomplished by agents that become clients of file-level agents. A "service" agent interacts with the file agents for the files that control a service, and with "process" agents that manages the existence and restart of particular daemons. The service agent receives data on which services should be present and propagates that to the configuration files by conversing with appropriate file agents. Non-behavioral attributes of configuration (e.g., exactly where specific files live) are handled by the closures that manage content of the files, and are no longer part of the configuration of a system.

We are all "washed by the very same rain" [29] and many recent innovations in configuration management have very similar characteristics to these agents. The behavior of agents is similar to distributed file generation procedures in LCFG [1, 2], Arusha [18, 19], and Psgconf [36], and "embody expert knowledge about how to configure a system" [39]. LCFG author Paul Anderson is considering reforming LCFG's language around specifying "constraints" rather than "parameters." In many ways, everyone is converging upon one way of thinking: specify external behavior, and leave everything else to the configuration tool.

There is one extreme contrast between our model and every other model of configuration management. In every other model, *closures are systematically violated* by reverse-engineering appropriate parameter values and setting these values directly by file generation or editing. This seems convenient but has a deadly result. It means that the administrator configuring the management tool has to have *complete knowledge* of the semantics of interior and exterior parameter space of *all* closures in order to automate administration, including interior parameter semantics for all possible versions of that parameter space (e.g., for multiple operating systems and hardware platforms). We suggest a controversial departure from this:

> **Principle 20**: Even advanced configuration management tools should utilize conduits rather than direct file editing to manage closures.

Tools must avoid end runs around protections and integrity constraints built into closures. The management tool need not know the fine-grained semantics of interior parameters, and can concentrate on exterior behavior. The result is to naturally *distribute knowledge* about the system in two places: a high-level description of behavior in the configuration management tool, and a low-level description of how to assure behavior (via interior parameters) in the closure itself. This distribution makes it possible to configure networks with a portable high-level language that becomes customized when applied to specific architectures or platforms.

There are many examples where use of an advanced configuration management tool such as Cfengine destroys a closure. The most obvious problems occur in trying to use Cfengine for package management [20]. The master RPM repository for a particular version of RedHat Linux, together with the rpm command that unravels and installs it, is a strong closure [17], but only if the rpm command is used as intended, and only when no other effects are interposed (such as editing the files that rpm edits during post-install scripts). Interspersing arbitrary configuration commands between a series of package installs can lead to validation drift [17] and eventual failure of the system.

Initial Prototypes

To prove the concept of a closure, we decided to create three prototypes in the context of a group Masters project in System Administration at Tufts University. Concurrently, Prof. Couch continued to develop the mathematical theory of closures. The most important lesson we learned is that in a race between developing theory and prototypes, theory almost always

matures faster. The emerging theory led us to critique the initial designs of the closures in quite unexpected ways, and led to a stronger understanding of the theory as well.

We began our study of closures by trying to replace the somewhat problematic "file editing" features of Cfengine with a more strongly convergent substitute agent, through which all editing transactions could flow under control of Cfengine. This agent understands the structure of the file being edited, as well as the desired output format for the file. Upon receiving editing commands inspired by SQL, the agent parses the file, makes changes, and rewrites the file according to specifications.

The second prototype is a conduit for editing configuration files that understands policy and integrity constraints on file contents. XML [3, 9, 27, 44] declarations describe the format of the incoming file, the desired output, and the range of allowable changes to the file. The file is parsed through a stream-parser based upon Babble [13] and the output generated through use of XSLT stylesheets. This is similar to TemplateTree II [34] except that the whole configuration file is parsed and all configurable variants exposed.

The third prototype uses the first prototype to handle high-level configuration data. It expresses changes in service as changes in configuration, and converses with the relevant file and process agents to affect the changes. Its own high-level file format is amenable to editing via the second prototype, so that the circle is complete and one can specify low-level effects with high-level commands.

The focus of all these early prototypes is basic service provision and security. The long-term goal of this work is to create agents that completely encapsulate the process of configuring an Apache web server, so that virtual services can be created flawlessly without any chance of disrupting other virtual services.

Encapsulating File Editing

It is generally agreed by practitioners that on-the-fly file editing is a weakness of many configuration management strategies. Many avoid explicit editing by generating all file contents from databases. As an exercise, we studied instead how one could encapsulate file editing into a closure that protects against typical file-editing mistakes.

Atomicity

As a first attempt to solve this problem, we built a prototype *flat file configurator* (FFC) that encapsulates the idea of editing a field-based configuration file from /etc, such as services, inetd.conf, etc. This particular closure made all updates atomic. The closure is an agent that maintains the contents of a file via database-like commands with a syntax inspired by SQL. These commands include:

- load /etc/services: Make contents available for editing.
- in /etc/services: Set a context in which edits will occur.
- insert what (service='tftp', port='8888', proto='tcp') : Add a line to the in-memory version of /etc/services.
- delete where (service='dns' and proto='udp') : Delete a line from the in-memory version of /etc/services.
- update who (first 5) where (proto='tcp') what (proto='udp') : Change the first five entries in the file having protocol tcp to protocol udp.
- update who (last 1) where (service='dns') what (proto='udp'): Change the last line where service is dns to have protocol udp.
- display: Show contents of file on terminal.
- display as XML: Show contents as XML.
- write: Update original file with in-memory data.
- write to /etc/services.new: Posts results to a new file.
- end in: Terminate editing context.

These commands preserve the order of the input file when updating and deleting lines, and insert new lines at the end. Updates leave data in the same position in the file. Although the closure does not allow completely duplicate lines, it is possible to create files that have two port entries for the same service and protocol. This usually results in an invalid services file.

This is a fine first attempt, but new theory shows that it is not a particularly strong closure. There is a much better approach.

Invariance

The closure above solves the problem of editing but leaves the problem of file integrity unaddressed. A suitable closure for /etc/services must preserve three invariants of the file:

1. The file is sorted in a deterministic order by port number, protocol, then service name.
2. The file has at most one entry for each service and protocol combination.
3. There is no other content to the file.

One reason for this approach is to assure that a list of edits, applied to two copies of the same initial file, will both have identical Tripwire [43] or Aide [31] signatures.

To make a better closure, we must limit the set of commands to those that cannot violate these invariants, and construct them so that they do not:

- assert what (service='tftp', port='8888', proto='tcp'): Insures that there is a line for service tftp. This will change an existing line if it is present.
- retract where (service='tftp'): Insures that there is no line describing service tftp. This will delete a line only if present.
- commit: Commits changes to the file to disk.

These commands change content in a stateless manner [16] so that the result of editing is invariant of the order of individual assertions and retractions of data.

The lesson learned from this example is that the strength of a closure is proportional to the lack of irrelevancy in the command set and its inability to express less-than-ideal states. Expressive power is an enemy of closure and predictability. This new command set has the following properties:

- Commands always succeed and there is no error state (barring hardware failure)
- Every command preserves the invariants of keeping the file in sorted order and having one line per service.
- The commands are not based upon a parameter model of the file, but upon a *holistic* model of the meaningful states of the file.

While it has been proposed that we should think of individual files as databases [22, 23], this shows that thinking of them in that way is too powerful, and only by limiting the operations on a file to a small set that preserves invariants can one truly construct a closure on the file. The database model is too powerful and flexible to create the appropriate effect.

The above attempt is a closure, but is still not a particularly strong one. It preserves the integrity of the structure of the file, but does not reflect content policy above and beyond that integrity. To go further, knowledge of this policy and of best practices for content of /etc/services must be included in the agent. It does not matter whether the closure *understands* whether a state is valid or not; we simply have to arrange by some means that no invalid state can be assured by the agent. This is not understanding; just syntax.

Policies

The prototype above does not go far enough to eliminate errors. The strength of a true closure for /etc/services lies in its ability to avoid invalid or undesirable states. For most sites, this means that the ports of several well-known services should not be possible to change. As well, it should not be possible to delete or modify certain well-known service records whose absence will create havoc, e.g., time. These assumptions – part of typical site policies – are not representable easily in database form, and the database analogy for closures again breaks down.

To implement this kind of closure, the command set has to be insensitive to certain kinds of changes and may not require full data in order to make an assertion of state. For example, the well-behaved closure agent might presume that

```
assert service=ssh
```

always implies protocol tcp and port 22, so these do not need to be specified. If one tells this closure to

```
assert service=ssh, port=33
```

then it has every right to ignore the request and/or raise an exception. This is an assertion of invalid state according to a site policy that is itself an expression of accepted practice according to the site's practice manual. Likewise, the same closure will reject suggestions like

```
retract service=ldap
```

so that a user has some hope of logging into the host. The feature of limiting configuration actions based upon site policy was first developed in Slink [10].

Implementation

Implementing such a closure is actually relatively straightforward. One first describes the structure of a flat file as an XML declaration, as in Babble [13] (Figure 4). This defines the variant parts of a typical line in /etc/services in the same way that Babble defines the variants in a terminal transaction stream. This declaration is used both to parse and regenerate the file during each editing transaction. Similar declarations could be used to constrain content in creating the proposed changes above, including listings of defaults.

```
<file fileName="services">
  <line>
    <comment marker='#' />
    <var name='service'
         regexp="[A-Za-z0-9_—]+" />
    <whitespace/>
    <var name='port'
         regexp='[0-9]+' />
    <literal>/</literal>
    <var name='proto'/>
    <whitespace />
    <var name='alias'
         regexp='[^#]*'
         optional="true" />
    <var name='comment'
         regexp='#.*'
         optional="true" />
  </line>
</file>
```

Figure 4: Declaring database structure of the services file.

Prototype Status

This is the most evolved of all the prototypes, and is fully functional. It is currently implemented as a single Perl script with a command-line interface. A second configuration file can bind particular structural configurations to the various files in /etc (though at present, only the /etc/services interface has been extensively tested).

Critique

File-editing is a necessary process, but a poor level at which to form a really strong useful closure. Some part of a closure must understand more than the syntax of the file. Basing one's whole strategy upon editing, as in Cfengine, does not form strong closures, even if the editing itself is strongly closed as above. More constraints are required than pure syntax can provide.

Constrained Editing

The second prototype implements an interactive web-based editor for configuration files that understands structural limits of each kind of file. The editor makes it possible to add any legal entry to the file

according to a predeclared policy, but prevents adding any illegal entries to the file.

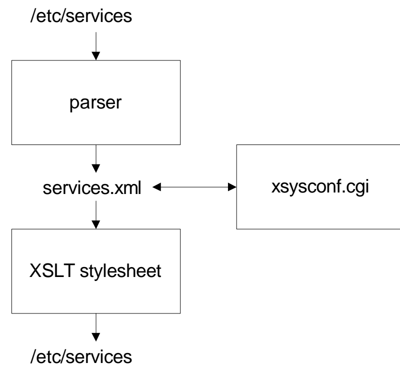


Figure 5: Editing a flat file via XML.

The parser is configured to edit a particular kind of file by specifying the constraints upon the file in XML, using a syntax again similar to that of BABBLE [13] but more complex than that for the flat-file configurator above (FFC). For /etc/services, the constraint file is shown in Figure 6. The var fields describe variant text, while the repeat, optional, choice, and text elements have the obvious meanings. The comments in each kind of variable are a description provided to the user as help text. While this file may seem complex, it is nothing more than a detailed and highly documented regular expression! It says that a services file consists of services entries, one per line, where each line consists of a service name, a port, a forward slash, a protocol name, and optionally a list of aliases and/or a comment.

Using this declaration, the parser converts the flat format of the source file to XML. The /etc/services source file:

```
ssh 22/tcp ssh-2 ssh2 # secure shell
telnet 23/tcp ncsa-telnet
```

is translated into the XML in Figure 7. It is upon this intermediate form that editing takes place. There is presently one design flaw in the prototype: if no alias exists when the initial record is created, none can be added.

The editing process is a web-based CGI script that presents editing options as pulldown menus and text fields (Figure 8). One can change or add fields by pressing the appropriate buttons or changing values and pressing ‘‘write.’’

Prototype Status

Currently, the only fully implemented part is the CGI-based editor for XML. The translators to and from XML are not yet implemented, though the translator into XML is almost identical with that for FFC

```
<xmft:file path="/etc/services">
  <xmft:repeat sorted-by="port" keys="service:port+prot" name="lines">
    <xmft:line>
      <xmft:var type="string" desc="service name" name="service">
        This is the service name. This should correspond to the same
        service in inetd.conf.
      </xmft:var>
      <xmft:whitespace/>
      <xmft:var type="integer" desc="ip port number" name="port">
        This is the Internet Port number of a service. The pair of
        port number and protocol must be unique in the file.
      </xmft:var>
      <xmft:text>/</xmft:text>
      <xmft:choice type="protocol name" name="prot">
        <xmft:option><xmft:text>tcp</xmft:text></xmft:option>
        <xmft:option><xmft:text>udp</xmft:text></xmft:option>
      </xmft:choice>
      <xmft:repeat>
        <xmft:whitespace/>
        <xmft:var type="string" desc="protocol alias" name="alias">
        </xmft:var>
      </xmft:repeat>
      <xmft:optional><xmft:whitespace/></xmft:optional>
      <xmft:optional>
        <xmft:text>#</xmft:text>
        <xmft:optional><xmft:whitespace/></xmft:optional>
        <xmft:var type="string" name="comment">
          This is a comment describing the service defined by this line
        </xmft:var>
      </xmft:optional>
    </xmft:line>
  </xmft:repeat>
</xmft:file>
```

Figure 6: Declaring variant structure of the services file.

above, and the XSLT for generating output is almost identical to that of Finke [23]. Thus we had high confidence in our future ability to implement these, and left them undone in this proof-of-concept prototype.

alias ssh-2 ssh2 add
comment secure shell
port 22 *
prot tcp *
service ssh *
alias ncsa-telnet add
comment
port 23 *
prot tcp *
service telnet *
add record
write file

Figure 8: HTML editing form for services file.

Critique

Again, theory developed in concert with this prototype changed our thinking about it. The missing and crucial element of this editor is the ability to define constraints beyond simple syntactic correctness; the exact same constraints needed for a fully useful version of the FFC above.

Service Configuration

As an example of a high-level agent, we also implemented a service configuration agent. This was our first attempt to build a configuration environment based upon multiple agents acting together. The top-level agent is told what services to configure, and tells lower-level agents what to add to particular configuration files. While this prototype is the most challenging and least complete of the ones we attempted, it does serve to demonstrate that high-level integration is possible.

Implementation

The editor is the most complex of the prototypes, and contains three parts (Figure 5):

1. A *parser* that parses a flat file and converts it to XML.
2. A *CGI-based editor* that uses XSLT to define an interactive web-based editing session for the XML file's contents, in HTML.
3. An *XSLT renderer* that translates the XML file back to a flat file.

Architecture

The service configuration closure involves many component closures, as illustrated in the dominance diagram in Figure 9. Closures above dominate closures below.

1. The *configuration* closure dominates all others.
2. The *service* closure is the highest-level closure for services. It manages all aspects of services, including whether they are running or not, as well as current runtime state.

```
<file path="/etc/services">
  <repeat name="lines">
    <record>
      <repeat>
        <alias mode="optional" thing="var" type="string">ssh-2</alias>
        <alias mode="optional" thing="var" type="string">ssh2</alias>
      </repeat>
      <comment mode="optional" thing="var" type="string">secure shell</comment>
      <port mode="required" thing="var" type="integer">22</port>
      <prot options="tcp,udp" mode="required" thing="choice">tcp</prot>
      <service mode="required" thing="var" type="string">ssh</service>
    </record>
    <record>
      <repeat>
        <alias mode="optional" thing="var" type="string">ncsa-telnet</alias>
      </repeat>
      <comment mode="optional" thing="var" type="string"></comment>
      <port mode="required" thing="var" type="integer">23</port>
      <prot options="tcp,udp" mode="required" thing="choice">tcp</prot>
      <service mode="required" thing="var" type="string">telnet</service>
    </record>
  </repeat>
</file>
```

Figure 7: Translation of a small services file into XML.

3. The *process* closure starts and stops individual processes.

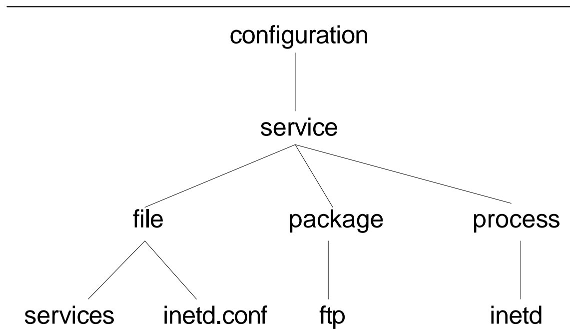


Figure 9: Dominance hierarchy for the service configuration agent.

4. The *file* closure is the lowest-level closure of the system, that only receives commands. It is responsible for editing files to contain appropriate configuration lines.
5. The *package* closure manages installation and removal of packages using RPM.

Many others are also present and planned but omitted due to space constraints. In typical use, the service closure contacts the file closure to affect changes in configuration, the process closure to start and stop service processes, and the package closure to install new service files.

All closures have the same commands assert and retract. Assert makes something available to run within the configuration, while retract makes it unavailable. All closures also have a show command that reports on current status. Most closures have start and stop, that control whether something is executing or dormant.

The Configuration Closure

The configuration closure is the main user interface to the system. For some idea of the form of the

```
assert service tftp
```

creates the service tftp on its normal port, while

```
assert service tftp port=6699
```

creates the service on a different port. The command

```
retract service tftp
```

removes a service, while

```
restart service tftp
```

administrator's interface, the command restarts all tftp daemons for an installed service tftp So far, this looks very similar to print managers such as *lpc*. The difference is in how this is implemented. Rather than doing the work in one large program, the service closure farms out work to subclosures that it dominates. For example, assert ftp results in the chain of assertions shown in Figure 10. Each line ending in a colon determines a closure context in which the commands below it are executed. Ellipses signify detail that is omitted for clarity.

The point of this process – and what distinguishes it from command-line managers such as lpc – is its structure, not its interface. High-level commands are broken down into low-level commands that include

```
configuration:
  assert service ftp
    service:
      assert ftp
        file:
          assert file "
            load /etc/services
            in /etc/services:
              insert what (service='ftp-data', ...)
              insert what (service='ftp', ...)
              write
            end in
          "
        package:
          assert ftp
        file:
          assert file "
            load /etc/inetd.conf
            in /etc/inetd.conf:
              insert what (service='ftp', ...)
              write
            end in
          "
configuration:
  start service ftp
    service:
      process start ftp
        process:
          start ftp
```

Figure 10: Communications between agents in assuring services.

interior parameters implied by the high-level commands. The echoing of the same command-line structure at each level – assert and retract – makes it easy to implement the various closures.

Critique

The only problem with this prototype is that it was the most ambitious of the group, and its design changed much more than the others during development, so that it is not yet fully functional. File editing is problematic, because the closure was built before the file editing closure was completed, and does not communicate well with the existing file editing closure. The theoretical result that exterior parameters of a closure must be present in each closure that dominates it is echoed in the structure of this prototype, even though the prototype was developed completely independent from the theory.

Conclusions

This is the beginning of a new and long journey, but the conclusion of another. For years, we struggled to define causality in a complex system [15, 17, 40]. We conclude that causality in a realistic system is a myth unless we synthesize it ourselves through the discipline of closure. Many principles of practice guide us, but there is still much work to be done, both on examples of closures and more clever principles of practice.

Future Work

It is summer, our masters project activity is a pleasant memory, and all the students involved have graduated, but obviously there is still much to be done. In the immediate future a new team of students will be completing a second generation of prototypes that are deployable in practice, by adding policy and practice awareness. On the theory side, we plan to more extensively study the algebraic properties of closure by applying the theory of semigroups [24, 28, 32], in a manner similar to that of [16].

Another knotty problem is to be able to efficiently define (and ultimately, automatically derive) the sometimes subtle maps between configuration and behavior. This is very much dependent upon the various ''Book of Knowledge'' projects [26, 30] to provide us with the words and practices that the closures should employ. This is really a problem in knowledge representation, and we envision using inductive logic programming [12] to complete exterior specifications with adequate values for interior parameters.

The reason that we carefully say *adequate* rather than *optimal* is that – due to its relation with graph theory – the problem of finding optimal parameter settings is often intractable. As an administrator, one would not even try; one would settle for settings that are *good enough*. Our automated agents should do likewise.

The map between exterior and interior parameters must take a concise, readable form that is easy to validate and certify. While working on our prototypes, we realized that the Stem [25] framework for network programming provides an ideal environment for exploring closures. Future development will focus on placing the closures in a Stem framework, using built-in Stem message passing to implement conduits.

It will be difficult to integrate this method of management with existing configuration management practices. The most difficult problem is to arrange isolation of closures and avoid *any* chance of corrupting a closure outside a conduit. This is beyond the capabilities of current file protection schemes, and might even require a new model of protection for filesystems similar to the model that protects memory at runtime.

We have shown so far that traditional hierarchical control flow can exhibit closure in an expedient fashion, but have not ruled out many other forms of control. The mathematical framework we developed allows many other kinds of communication, including peer-peer and even *clique*, in which one agent of a group arbitrarily appoints itself as the master of the group similarly to the way this is done in LCFG.

These are all difficult problems but – from our experience – all are tractable. Further, once each problem is solved once, the effects last forever. We think it will be worth the effort.

Some Lessons Learned

The theory of closures has some surprising ramifications for practice. Closures are only as strong as the discipline with which they are managed. This discipline includes only making changes via approved mechanisms. For example, the master RedHat repository is a closure if managed properly with rpm, but stops being a closure if one works outside rpm's framework, e.g., with make.

The most astonishing lesson learned is that common automation practices can destroy or weaken closures that already exist. For example, suppose that one wishes to install an RPM package via Cfengine. A common practice is to perform the RPM installation on one host, reverse-engineer its effects, recode the installation into a series of convergent Cfengine declarations, and use Cfengine to assure compliance with the declarations instead of running the original installation on each new host. The problem with this practice is that the original closure was only valid if RPM installation was used; this new method cannot, e.g., predict all of the potential effects of a post-install script when the script is executed on a previously unforeseen kind of host. Worse, the re-engineering has to be redone each time a new version of the package appears. This observation gives some credence to the order-based infrastructure-building strategies discussed in [37, 38].

The fact that ''order matters'' [38] is, however, a symptom of a deeper problem. Every configuration action is (as claimed in [38]) a computer program. As with any computer program, there is a set of preconditions that specify what must be true within a system

before the action will work. In [37, 38], one avoids facing the preconditions by replacing them with order constraint, where the appropriate order of operations is something observed in practice. This is only necessary if preconditions cannot be understood. Order only matters if we do not understand enough about preconditions to be able to vary the order. Our initial theory work suggests that preconditions can be modeled in very much the same way as behaviors; they are a tractable and fairly simple set of conditions upon a much larger and more complex system. With a full understanding of preconditions, order no longer matters. This fact merits further investigation.

Coming to Closure

The current practice of configuration management has high inertia. Many people, trained in the old ways, will find it difficult to adapt to the constraints that closures impose. Many configuration management tools will impede rather than help the process, because of their inherent inability to talk with conduits, as well as the tools' implicit support and encouragement of the closure-weakening reverse-engineering process described above. Vendors will impede the process by failing to provide closures themselves.

How can we evolve current practice toward effective closures? There are several steps:

1. Understand the concept of a closure
2. Identify and treasure the ones already present.
3. Avoid compromising existing closures.
4. Proactively look for domains of application.
5. Deploy win-win closures (that do not enforce limits on users) first.
6. Justify one-sided closures (that limit users) in terms of reduced lifecycle cost.

Our profession has not sufficiently evolved from arguing from "cost of equipment" to "cost of ownership." [35] If one's thinking is based upon cost of equipment and software, closure makes no sense. When one factors in the cost of administration, closure becomes a cost-saving mechanism.

A Vision of the Future

The evolving theory of closures promises a completely different of role for future system administrators than what they have today. Networks will not consist of "machines," but rather will be built of "closures" that span machines. When one places a new machine into a network, the closures operating within the network will discover it and configure it to talk with them. If this succeeds, no administrator intervention will be required; the machine will simply start working properly. If it fails, one must debug the closures (or the machine) to see why the closures did not properly integrate the machine into the network and install themselves for future use.

In the new scheme, the administrator's job is no longer to remember complex and detailed minutiae, but instead to keep an internal model of the big picture and how things fit together in a large and universal block diagram. Low-level knowledge (how a closure works) is much less important than high-level knowledge (which closures are compatible with which others). Dependency analysis and troubleshooting become jobs of specialists rather than generalists, and are not part of typical practice. Typical practice is to choose closures that work, install them, and watch them work. If they do not work properly, they must be incompatible, and one then chooses another more compatible set. Troubleshooting is to some extent replaced by shopping. Patching and upgrades are largely software functions rather than job functions.

Another fundamental difference in this brave new world is that one must systematically avoid working around the protections created by a closure. To publish content on the world-wide web, one tells a closure to publish it, and never tries this process by hand. There is only one defense against users who would work around closures, which is to make this impossible by, e.g., placing web servers on machines to which users do not have shell access.

The most profound change is in how the administrator can view the profession. We have always thought that system administration was in actuality community building: to paraphrase Burgess, "forming cooperative communities of people and machines." [7] With closures, this community building becomes the substance of the job as well as the dream. Closures are community members that take some of the worry out of keeping the community working properly.

With closures, we can begin to evolve to a methodology for system administration that is more predictable. This will not make the job of administrator obsolete; just a lot more pleasant and rewarding.

Acknowledgements

This work is not the product of individuals, but of a coordinated inquiry among a large and diverse intellectual community. We are grateful to the Large-Scale System Configuration mailing list (lssconf) for providing sustained discussion of the strategies for configuration management and their impact. Yizhan Sun provided detailed, in-depth comments and corrections to the mathematics. Special thanks to Rob Kolstad and Adam Moskowitz for insightful comments (on the first sketch of the paper) that pushed our thinking to a new level. Special thanks to George Leger and shepherd Æleen Frisch for extremely careful proofreading and copy editing. Special thanks to Mark Burgess, Paul Anderson, Steve Traugott, Mark Roth, and Luke Kanies, for providing the main fuel for this debate with their ground-breaking work. This work was supported in part by a generous equipment grant from Network Appliance Corporation.

Author Biographies

Alva L. Couch attended the North Carolina School of the Arts in his home state as a high school

major in bassoon and contrabassoon performance. He received an S.B. in Architecture from M. I. T. in 1978, after which he worked for four years as a systems analyst and administrator at Harvard Medical School.

Returning to school, he received an M.S. in Mathematics from Tufts in 1987 and a Ph.D. in Mathematics from Tufts in 1988. He joined the the faculty of Tufts Department of Computer Science in the fall of 1988, and is currently an Associate Professor of Computer Science.

Prof. Couch is the author of several software systems for visualization and system administration, including Seecube (1987), Seeplex (1990), Slink (1996), Distr (1997), and Babble (2000). He can be reached by surface mail at the Department of Computer Science, 161 College Avenue, Tufts University, Medford, MA 02155. He can be reached via electronic mail as couch@cs.tufts.edu . His work phone is (617)627-3674.

John Hart has worked with Prof. Couch on configuration management for more than five years, first authoring the 'YoKeL' configuration language (Yet anOther Configuration Engine and Language) front-end to Prof. Couch's Prolog system administration interface. He received a Bachelor's degree in Computer Engineering from Tufts University in 2001 and a Masters degree in Computer Science from Tufts in 2003. While at Tufts, he worked as an undergraduate and graduate teaching assistant, LAN administrator, and as a consultant on web design, system administration, and other tasks. John can be reached via surface mail at 215 N. 2nd St., Olean, NY 14760, or electronically at jhart@cs.tufts.edu .

A native of Montana, Elizabeth G. Idhaw lived in Sanford, Maine until graduating high school. Her interest in computer engineering brought her to Lehigh University in Pennsylvania, where she earned a Bachelor of Science degree in that field. After graduating in 1998, she moved to Massachusetts to work as an AlphaServer Hardware Engineer for Compaq Computer Corporation in Marlborough. She returned to school in 2001 for graduate study in Computer Science at Tufts University. With a Master of Science degree under her belt, she began work as a Senior Network and Distributed Systems Engineer for The Mitre Corporation of Bedford, Massachusetts. She can be reached by electronic mail at greenlee@cs.tufts.edu .

Dominic H. Kallas hails from Arlington, Massachusetts. He received his Bachelor of Science degree in Electrical Engineering from Tufts University in 1999 and continued to a Master of Science degree in Electrical Engineering, completed in 2003. His concentration is in Wireless Communications, and he is interested in network design and signal processing. He welcomes e-mail at dkallas@cs.tufts.edu.

References

[1] Anderson, P., "Towards a High-Level Machine Configuration System," *Proc. LISA-VIII*, Usenix Assoc., 1994.

[2] Anderson, P., P. Goldsack, and J. Patterson, "SmartFrog Meets LCFG: Autonomous Reconfiguration with Central Policy Control," *Proc. LISA XVII*, USENIX Assoc., pp. 219-228, San Diego, CA, 2003.

[3] Bohlman, E., "Parsing XML, Part 1," http://www.perlmonth.com/perlmonth/issue4/perl_xml.html .

[4] Burgess, Mark, "A Site Configuration Engine," *Computing Systems*, Vol. 8, 1995.

[5] Burgess, Mark and R. Ralston, "Distributed Resource Administration Using Cfengine," *Software: Practice and Experience*, Vol. 27, 1997.

[6] Burgess, Mark, "Computer Immunology," *Proc. LISA-XII*, Boston, MA, Usenix, 1998.

[7] Burgess, Mark, "Theoretical System Administration," *Proc. LISA-XIV*, New Orleans, LA, Usenix, 2000.

[8] Cons, Lionel and Piotr Poznanski, "Pan: A High-Level Configuration Language," *Proc. LISA XVI*, USENIX, Philadelphia, PA, 2002.

[9] Cooper, C., "XML::Parser – A Perl Module for Parsing XML Documents," http://search.cpan.org/author/COOPERCL/XML-Parser-2.31/Parser.pm .

[10] Couch, Alva, "SLINK: Simple, Effective Filesystem Maintenance Abstractions for Community-based Administration," *Proc. LISA X*, Usenix, 1996.

[11] Couch, Alva, "Chaos out of Order: A Simple, Scalable File Distribution Facility for 'Intentionally Heterogeneous' Networks," *Proc. LISA XI*, Usenix, 1997.

[12] Couch, Alva and M. Gilfix, "It's Elementary, Dear Watson: Applying Logic Programming to Convergent System Management Processes," *Proc. LISA XIII*, Usenix, 1999.

[13] Couch, Alva, "An Expectant Chat about Script Maturity," *Proc. LISA XIV*, Usenix, 2000.

[14] Couch, Alva and Noah Daniels, "The Maelstrom: Network Service Debugging via 'Ineffective Procedures' ," *Proc. LISA XV*, Usenix, 2001.

[15] Couch, Alva and Y. Sun, "Global Impact Analysis of Dynamic Library Dependencies," *Proc. LISA XV*, Usenix, San Diego, CA, 2001.

[16] Couch, A. and Y. Sun, "On the Algebraic Structure of Convergence," to appear in *Proc. DSOM'03*, Elsevier, Heidelberg, DE, Oct., 2003.

[17] Hart, J. and J. D'Amelia, "An Analysis of RPM Validation Drift," *Proc. LISA XVI*, Usenix Assoc., San Diego, CA, 2002.

[18] Holgate, M. and W. Partain, "The Arusha Project: A Framework for Collaborative Unix System Administration," *Proc. LISA XV*, Usenix, San Diego CA, 2001.

[19] Holgate, M., W. Partain, et al., *The Arusha Project Web Site* http://ark.sourceforge.net .

[20] Kanies, L., "Practical and Theoretical Experience with ISconf and Cfengine," *Proc. LISA XV*, USENIX Assoc., San Diego CA, 2001.

[21] Sandnes, Frode Eika, "Scheduling Partially Ordered Events in a Randomised Framework – Empirical Results and Implications for Automatic Configuration Management," *Proc. LISA XV*, Usenix, San Diego CA, 2001.

[22] Finke, Jon, "An Improved Approach for Generating Configuration Files from a Database," *Proc. LISA XIV*, Usenix, 2000.

[23] Finke, Jon, "Generating Configuration Files: The Director's Cut," *Proc. LISA XVII*, USENIX Assoc., pp. 201-210, San Diego, CA, 2003.

[24] Grillet, P. A., *Semigroups: An Introduction to the Structure Theory*, Marcel Dekker, Inc, New York, NY, 1995.

[25] Guttman, U., "Stem: A Sysadmin Enabler," *Proc. LISA XVI*, Usenix, Philadelphia, PA, 2002.

[26] Halprin, G., et al., "SA-BOK (The Systems Administration Body of Knowledge)," http://www.sysadmin.com.au/sa-bok.html .

[27] Harold, E. and S. Means, "XML in a Nutshell, 2nd Edition," O'Reilly, Inc, 2002.

[28] Howie, J. M., *An Introduction to Semigroup Theory*, Academic Press, 1976.

[29] Humphries, Pat, *Same Rain (Audio CD)*, Moving Forward Music, 1992.

[30] Kolstad, R., et al., *The Sysadmin Book of Knowledge Gateway*, (private site).

[31] Lehti, Rami, "AIDE – Advanced Intrusion Detection Environment," http://www.cs.tut.fi/rammer/aide.html .

[32] Ljapin, E. S., *Semigroups*, American Mathematical Society, Providence, RI, 1963.

[33] Logan, Mark, Matthias Felleisen, and David Blank-Edelman, "Environmental Acquisition in Network Management" *Proc. LISA XVI*, Usenix, Philadelphia, PA, 2002.

[34] Oetiker, T.,"TemplateTree II: The Post-Installation Setup Tool," *Proc. LISA XV*, Usenix, San Diego, CA, 2001.

[35] Patterson, J., "A Simple Model of the Cost of Downtime," *Proc. LISA XVI*, Usenix, Philadelphia, PA, 2002.

[36] Roth, M. D., "Preventing Wheel Reinvention: The Psgconf System Configuration Framework," *Proc. LISA XVII*, pp. 211-218, USENIX, San Diego, CA, 2003.

[37] Traugott, Steve and Joel Huddleston, "Bootstrapping an Infrastructure," *Proc LISA XII*, Usenix, Boston, MA, 1998.

[38] Traugott, Steve and Lance Brown, "Why Order Matters: Turing Equivalence in Automated Systems Administration," *Proc. LISA XVI*, Usenix, Philadelphia, PA, 2002.

[39] Sapuntzakis, C., D. Brumley, R. Chandra, N. Zeldovich, J. Chow, J. Norris, M. S. Lam, and M. Rosenblum, "Virtual Appliances for Deploying and Maintaining Software," *Proc. LISA XVII*, pp. 186-200, USENIX, San Diego, CA, 2003.

[40] Wang, Yi-Min, Chad Verbowski, John Dunagan, Yu Chen, Chun Yuan, Helen J. Wang, and Zheng Zhang, "STRIDER: A Black-box, State-based Approach to Change and Configuration Management and Support," *Proc. LISA XVII*, pp. 165-178, Usenix, San Diego, CA, 2003.

[41] Watt, D., *Programming Language Syntax and Semantics*, Prentice Hall, 1991.

[42] *The Linux Standard Base Project*, http://www.linuxbase.org .

[43] Tripwire, Inc, *The Tripwire Security Scanner*, http://www.tripwire.com .

[44] XML Working Group, *XML Schema Specification*, http://www.w3c.org .