Modeling Next Generation Configuration Management Tools

Mark Burgess – Oslo University College
Alva Couch – Tufts University

ABSTRACT

There are several current theoretical models used to discuss configuration management, including aspects, closures, and promises. We examine how these models relate to one another, and develop a overall theoretical framework within which to discuss configuration management solutions. We apply this framework to classify the capabilities of current tools, and develop requirements for the next generation of configuration management tools.

Introduction

Configuration management is the process of constraining the behavior of a network of machines so that each machine's behavior conforms to predefined policies and guidelines and accomplishes predetermined business objectives. Configuration management would be easy if objectives did not change, the number of machines in a network were small, machines were simple in structure, machines were identical, machines did not fail, and no unauthorized parties could alter behavior. Unfortunately, objectives change, networks are large, machines are complex, machines differ, machines fail, and new security holes appear each week, making configuration management a costly part of administering any large network.

The overall goal of configuration management as a practice is to maximize the extent to which systems conform to predetermined expectations, while minimizing the cost of keeping the network's behavior within predetermined guidelines. There are many strategies for accomplishing configuration management, from manually making changes to using powerful and comprehensive software tools to assert, enforce, or monitor configuration changes. Each approach has loyal advocates who consider their approach superior to others, leading to spirited debates in the LISA Configuration Management Workshop.[1]

At the present time, configuration management remains one of the most controversial aspects of system administration. Approaches and tools abound, each with faithful adherents whose dedication to the approach borders on religious fervor [1, 3, 4, 5, 6, 13, 20, 21, 22, 23, 24, 27, 29, 30]. Understanding the key differences between approaches has proven difficult, and many practitioners have asked why it seems so necessary to "re-invent the wheel" [27] in creating completely new configuration management tools from scratch. Many tools seek to enable new collaboration methods(e.g., [23, 27]) to enable sharing of work. So far, collaboration seems to be the exception rather than the rule.[2]

Are authors of new tools really contributing anything new? Why has it proven so difficult to apply configuration management tools to new sites? In trying to answer this question, it has proven difficult to get beyond issues of personal taste and understand *why* there are so many approaches, and what drives each approach. Part of the reason for this is that there has been no coherent language with which to compare and contrast strategies both precisely and fairly. Without this language, advocates of particular approaches seem like zealots; with this language, the reasons behind their thinking can become clear.

The Quandary of Cost

One reason for the diversity of approaches is that the least costly strategy for configuration management is often determined by the nature of the site being managed and its mission [19]. There is, for example, a profound difference between the best configuration management strategy for an academic research lab and for a bank. "Tight" sites such as banks require much more disciplined and expensive strategies than academic research labs, because the cost of downtime is much higher in a bank than in a research lab [26].

To better understand the sources of cost, there have been several tries at creating theoretical models of configuration management. *Closures* attempt to encapsulate parts of network function in black boxes, to reduce configuration management complexity and cost [15, 16], while *promises* model the way autonomous parts of a network exchange information and commit to certain behaviors [10, 11, 12, 14], to allow networks and computers to become more self-

[1]What began as the CFengine Workshop at LISA 2001 inspired many discussions and was more appropriately renamed the Configuration Management Workshop by Paul Anderson shortly afterward.

[2]*AC*: At a configuration management "birds of a feather" session at LISA 2003, an informal poll was taken concerning the number of people in the room using *other* authors' configuration management tools. Of the attendees present, excepting users of CFengine, everyone had written custom tools for the task, and the only user of each tool in the room at the time was its author.

managing and self-sufficient. We present *aspects* as a way of describing the dependencies and constraints that plague configuration management and increase management cost [2]. The diversity and disparity of the contributions has been a hindrance to a feeling of progress in the field. Why are there so many different ways to think about the same basic problem?

In this paper, we discuss the relationships between current models of configuration management with several goals in mind. We define the terms and concepts in each model precisely, and show how they relate to concepts and terms in other models. This leads to an overall theoretical picture of configuration management based upon the union of concepts. This unified theory suggests and clarifies challenges to be addressed by the next generation of configuration management tools.

The plan for the paper is as follows: we introduce the concept of an aspect to capture a configuration management 'unit of planning.' We can think of this as a *requirement*. We then discuss how such aspects can be reliably implemented. This takes us to the concepts of closures and promises. However, there is an obstacle: it is far from clear that we have the authority to require anything of a distributed system, so we must transform a description of requirements into a description based on agreed compliance, or promises. Reviewing briefly the concept of service-oriented computing we show that, if we express aspects and closures in terms of 'promises,' then all of our results apply regardless of whether they are implemented as granted services or as authoritative control scripts. This abstraction makes our theory completely general. We finish by indicating how convergent operational semantics can be expressed as promises, thus completing the picture from high to low level.[3]

How Expensive Could It Be?

It helps to understand from whence configuration management costs arise. The cost of configuration management includes the costs of planning, deploying machines, deploying changes, and troubleshooting changes. Planning includes determining desired behaviors and how to accomplish them. Deployment consists of creating machines with a known initial configuration, to which configuration changes can be applied later. Changes are deployed by modifying machine configurations, network-wide, and changes often cause problems that must be investigated through troubleshooting.

Some of these costs are fixed and difficult to control, while others are somewhat under the control of the system administrator. Planning costs the same amount of staff time regardless of how one decides to manage systems, but deployment costs vary based upon whether the deployment is accomplished automatically or manually. The cost of troubleshooting shows the greatest variability and greatest opportunity for savings. It can be argued that the cost of troubleshooting is the *sum* of staff cost for repairing the problem and staff time and revenue lost *due* to the outage [26]. This observation makes troubleshooting a dominant factor in overall cost of ownership.

Constraints, Dependencies, and Preconditions

One core problem in configuration management is that accomplishing changes is often nontrivial. Often, when a change is made, "something breaks" [28], and troubleshooting is required to determine the cause. For example, installing a new version of a dynamic library has the potential to cause every program that loads that library to stop working properly. In Microsoft Windows, program installers can modify the registry entries of other programs, either intentionally or maliciously, so that installing a new program can lead to seemingly unrelated failures [31]. Programs often invoke other programs. For example, installing an inappropriate version of GhostScript can prevent Xfig from generating Postscript figures.

There are several ways that different authors describe the above situation in words. One can say that "there is a *dependency* between Xfig and the version of GhostScript" or "there is a *constraint* that the versions of GhostScript and Xfig must match." These are equivalent statements. A *precondition* [18, 24] is another name for a dependency; one could say that "a *precondition* for Xfig to function properly is that the appropriate version of GhostScript is installed." The difference between a precondition and a dependency is that a precondition describes relationships between events or occurrences in time, while a dependency or constraint describes relationships between subsystem states.

But the above situation is the easy case. Often, we do not know (or perhaps forget) the dependencies or constraints that must be satisfied. In making changes, it is possible to put systems into states whose behavior is unknown or unverified. Usually, this is because the system is in a different state than we believe it to have, either when applying a configuration change or when trying to use a program. We can say a failure is due to a *hidden dependency* or a *hidden constraint*, or that the success of a command requires (or prohibits) a *latent precondition* [24].

Aspects

We begin our story by proposing a definition of configuration management based upon *aspects*.[4] Our definition of an aspect differs somewhat from that of Anderson [2], who defines it as "a part of configuration specified by one human person or administrator,"

[3]*MB+AC*: for the reader's amusement we have left our (often wry) commentary to one another as bonus material to the director's cut of this paper.

[4]*MB*: on hearing about Promise Theory, Alva turned crimson and sang – "Mark's tongues in aspects."

but we agree with the spirit of his remarks. Instead, we define an aspect of configuration as a bundle of configuration information whose values must be coordinated to satisfy some known set of a priori constraints.

Aspects Are Required Characteristics

Our extended definition has more precise mathematical properties than Anderson's definition, but satisfies the spirit of the original definition; typically Anderson's hypothetical "one person" will be charged with deciding the values for a single aspect.

We pursue this course not so much because aspects are interesting in and of themselves, but because they provide the "glue" and intermediate representation that allows us to discuss the similarities between the seemingly very different concepts of "closures" and "promises," i.e., the concepts we need to actually implement configuration changes. They also better represent the way in which administrators plan and think about distributed management.

We begin with some definitions. It is necessary to understand precisely what we mean by a "configuration parameter" or a "constraint upon configuration parameters" before we can characterize the problem of configuration management more accurately. This may seem overly precise, until one considers that the lack of precise definitions has historically made it difficult to compare configuration management approaches, *because authors have utilized differing terms to describe similar concepts*.

Definition 1: Configuration parameter *A* configuration parameter *is a unit of configuration information. It can be manipulated by use of specified* get *and* set *methods, where* get *returns the parameter's value and* set *specifies a new value.*

A parameter's location within the system is not important, we refer to it indirectly, i.e., by a *method* or access service which conceals that specific location. The value of a parameter might be anything from a single scalar value to the contents of a hierarchy of files located somewhere within the filesystem.

We now want to talk about aspect 'types.'

Definition 2: Type of a parameter *The* type *of a configuration parameter p is a label identifiable with the domain of possible values that the parameter can assume, which we notate as D_p.*

Note that a type is a *shorthand* for a *set of options* D_p. Try not to think of an aspect type as a primitive data-type, e.g., like "string" and "integer"; rather think of "parameter set 1" and "parameter set 2" in a system specification, i.e., different configuration concerns. Types have the character of database schemas or XML schemas for configuration parameters.

Definition 3: Parameter constraint *A single constraint on a configuration parameter p may be expressed in two equivalent ways:*

1. *As a restricted set of values, i.e., a subset V_p of the domain D_p of its allowable values.*
2. *As a set of rules R_p that indirectly define the contents of V_p.*

The reader may be confused by this abstract specification of a rather mundane thing. When we make a constraint upon a parameter, e.g., "the hard disk must contain more than 4 GB of space," we are in actuality selecting a subset of hard disks that meet the criterion. By thinking of this set, rather than the rule or formula that defines it, we can avoid messy notation and clarify the concept.

Definition 4: Parameter set constraint *For a parameter set A, we can think of the constraints on the set A as being specified in two ways:*

1. *As the union of the rules R_p for values of parameters $p \in A$.*
2. *As a set of allowable* tuples *of values $T_A \subseteq \Pi_{p \in A} D_p$ where Π denotes cross product.*

In other words, the constraints on a set of things are some subset of the ordered tuples of parameter values, or alternatively, some set of rules that determine those tuples.

Example 1 *It is common for sets of parameters to have constraints between parameter values. The hostname declared for the web server is usually the same name as the name of the physical host running the server. This is a form of tuple constraint.*

A constraint is a specification of a subset of the possible parameter values that we particularly require. Defining rules R_A expresses that, for the parameters P_A, we are disallowing some *tuples* of values and are left with a smaller set of tuples T_A that are suitable. This set may be defined by enumerating possible tuples, or by abstract rules, but the effect is the same: to limit the set of allowable values.

An aspect, then, is a logical grouping (schema) of such parameters whose values are a characteristic of the system that we are trying to manage.

Definition 5: Aspect *An* aspect *A is a pair $\langle P_A, C_A \rangle$, where P_A is a set of configuration parameters and C_A is a set of constraints limiting the values of those parameters. C_A may be expressed as either a ruleset or an enumeration of tuples.*

If a parameter p is part of an aspect, values of the parameter p must conform to the constraints V_A for the overall aspect (specified as a tuple space). If the value of one parameter p is changed, we must choose a new value $v \in V_A$ (the set of allowable tuples), so that $v(p)$ has the value we desire, while all other parameters are adjusted so that the value of the aspect v remains a member of the constrained set V_A of allowable values.

Example 2 *Suppose that two data values are required to have the same value, but are stored in different places and accessed via different means. They are different parameters, but can be* *considered to be members of the same aspect, bound together by the aspect constraint of "value identity." For example, a web server must be configured to answer requests for a numeric IP address that happens to agree with the IP address of the machine running the server.*

If we have two parameters whose constraint is that they must have the same value at all times, then set for one must set the same value for the other, using mechanisms of traditional aspect-oriented programming.

Example 3 *Consider the number of times the hostname of the current host appears inside files in* /etc. *Under our definition, each occurrence is a separate parameter, but the aspect "hostname" embodies all of them, and setting the hostname as an aspect should modify all occurrences of the hostname everywhere it might appear in configuration files. There are many aspects with this quality, of one value stored in many places.*

Again, we have a relationship between setting one parameter and setting several others. Aspects may be more subtle than identity, though.

Example 4 *Consider an aspect dealing with hostname in a local-area network. The hostname of each host must be unique, so we make this property an aspect of the local-area network. Setting the hostname of one specific host to an already-assigned name would require us to set the already-assigned host's name to something different.*

Aspect constraints can be much more complex than this:

Example 5 *Consider an aspect for installing software packages. This aspect has constraints for determining when a package can be considered to work, in terms of dependency packages that must be installed beforehand.*

Aspects can even honor dependencies inside a single software package.

Example 6 *In the aspect called "web service," there are specific requirements and limitations on which modules can be installed in Apache, due to interoperability limits.*

In general, we can model the dependencies, requirements, and documentation of a network as a mesh of overlapping, inter-dependent aspects. Overlaps will be a problem, but we shall solve this matter by reducing aspects to networks of "promises." Our definition of an aspect is very similar to that of a *promise* [14], but at a higher level, and indeed this is no accident. We shall be returning to the reason for this in later sections.

Properties of Aspects

Having introduced aspects, the concept of parameter becomes somewhat redundant: we can meaningfully converse in terms of aspects alone.

Proposition 1: *Any single parameter p is also an aspect $\langle p, D_p \rangle$.*

Proof 1 *The type of a parameter by definition corresponds to a set of domain values, so the result is trivial.*

Proposition 2: *Any set of parameters is also an aspect, with the set of constraints that is the union of their individual constraints/types.*

Proof 2 *Again, this is obvious from the definition.*

Aspects consisting of only type information are rather dull; to make life interesting, we must include constraints about how parameters interoperate or must be related. We can do this most straightforwardly via *composition*:

Lemma 1: Aspect composition *A union of aspects is an aspect.*

Proof 3 *Let $\langle P_A, R_A \rangle$ represent one aspect A (expressed as a set of parameters P_A and a set of constraint rules R_A) and let $\langle P_B, R_B \rangle$ be another aspect expressed in the same fashion. The lemma follows trivially from*

$$\begin{aligned} A \cup B &= \langle P_A, R_A \rangle \cup \langle P_B, R_B \rangle \\ &= \langle P_A \cup P_B, R_A \cup R_B \rangle. \end{aligned}$$

The union of the sets of constraints is a larger (and perhaps more restrictive) set of constraints.

In other words, to make a union of two aspects, take the union of their parameter and constraint sets. Naturally, the behavior of a larger set of constraints is more constrained than a smaller number; as we make successive unions of aspects we arrive at a system with completely determined behavior at top level.

The duality between constraint rules and allowable value sets may seem curious to the reader. A rule $r \in R_A$ limits an aspect, which means that the larger R_A is, the smaller the set of allowable values V_A becomes. The same apparent strangeness occurs in object-oriented programming, where adding constraints to a subclass (via inheritance) *limits* the number of instances that can be considered members of that subclass, compared to the instances that are members of the parent class. Increasing constraints limits the number of acceptable values. Decreasing the number of constraints increases the number of acceptable values. This duality and contravariance between constraints and instances will be exploited in several ways in the rest of the paper.

Definition 6: Value of an aspect *The value v_A of an aspect A is a function from parameters within the aspect to values of those parameters, so that $v_A(p)$ represents the current value of parameter $p \in P_A$.*

Again, this is a simple concept. A value is simply a tuple v_A where fields $v_A(p)$ conform to all constraints of the aspect.

Hard and Soft Constraints

Note that constraints on values take two forms: "hard" and "soft." A "hard" constraint is one whose violation also violates physical law or the preconditions of a software or hardware subsystem. A "soft" constraint is a matter of policy or personal taste or choice.

A "hard aspect" contains only hard constraints:

Definition 7: Hard aspect *A hard aspect is one in which all constraints reflect physical limitations of the configured device and/or its software.*

For example, not using existing partitions in a partition map would lead to a non-functional system. Hard aspects arise both from documentation (of which values "should" work) and direct experience (of what works and does not work).

A "soft aspect" is one that we impose as a matter of policy, even though no physical laws are broken in its absence.

Definition 8: Soft aspect *A soft aspect is one in which all constraints are elective and do not reflect actual physical limitations. We might also call this a policy aspect.*

For example, the place we actually install the web server software is a "soft aspect" of the web service hierarchy; there are no physical reasons we cannot install it anywhere we wish (provided that partitions are large enough, which is a hard aspect!).

If we consider that the parameter that we are setting is itself an aspect, and the value we are asserting is a constraint within a new aspect containing that parameter, then our desires for a host's configuration can all be expressed in terms of aspect composition. Indeed,

Proposition 3: Configuration is an aspect *The entire configuration of a host or network can be thought of as the value of a composition of hard and soft aspects, including physical limits, policy choices, and arbitrary choices.*

An aspect generalizes and embraces related alternatives, i.e., one choice is available for each parameter in an actual configuration, whereas an aspect may provide alternatives. We can therefore arrive at a configuration by imposing a sequence of increasingly demanding constraints, from hardware and software limits, tempered by policy decisions, all the way to individual choices that may not matter.

The key idea of aspects is that it is an easy and straightforward way to encapsulate relationships between parameters and subsystems. While a parameter corresponds to a single configuration item, an aspect binds several together with a shared meaning, that might be either localized or distributed. In this respect, aspects will turn out to be related to *roles* in promise theory, which we will also discuss.

Managing Aspects

The concept of an aspect is a compelling mirror of design practice. Implementing configuration management, one must constantly conform to a series of practicality and policy constraints. These constraints commonly *overlap*, making configuration management a constraint satisfaction problem [25]. Worse, the constraints of an aspect may not be known, and we sometimes must make guesses about their nature. We can thus rethink configuration management as a problem of managing aspects.

Clearly, aspects are a mechanism for specifying the *requirements* for a functional system. But there is a presumption here – namely that we can actually require anything at all of a system. As computers and devices become increasingly personalized, a configuration planner becomes increasingly powerless to control autonomous devices; this is an issue which we are forced to confront.[5] An aspect specification is completely free of assumptions about how it will actually be managed, as a physical entity. This conceptual decoupling allows us compile the high level concept into some kind of lower level language – and this brings us to discuss closures and promises below.

Many aspect constraints are simple value choices. We can conform to these constraints most easily by storing the (replicated) specification in a database or file, and replicating the information into several files via a "generative" [18] or template based configuration management strategy, e.g., like LCFG [1, 3, 20].

Since the definition of a parameter arises from the ability to get and set it, two parameters are identical iff they are defined by exactly the same get and set methods. In case of overlaps, it is important to know whether values for two overlapping aspects are reasonable:

Definition 9: Coordinated aspects *Two aspects are mutually coordinated iff they agree to share the possible values of aspect parameters.*

In promise theory one has the notion of a coordination promise as a primitive construction to handle scenarios like this. Compiling aspects into promises will allow us to keep track of the logic of these complexities.

The most difficult aspects to manage are those with "distributed constraints." While a "local" aspect involves one machine, a "distributed" aspect involves some group of machines and their interactions. These have proven difficult to manage in several ways. First, there is a need for *coordination* whenever part of an aspect must change on one host in an aspect group.

An example of a distributed aspect is a client-server relationship. In this relationship, a client has an aspect that identifies the server, while the server has an aspect that defines the service. The union of these aspects and a port-number aspect describes a *binding* between server and client.

[5]*MB*: There is an important crossroads here. As we move towards a service-oriented picture of autonomously managed services, we move into a realm of having no authority to require anything of a server. Thus we must eventually move away from the idea that we are in control, to a view of encouraging voluntary cooperation. This step is taken by reinterpreting aspects in terms of promises [14, 10, 12].

Proposition 4: Bindings are distributed aspects *All service bindings of a client are distributed aspects with both client-side and server-side components.*

Example *7 For a more complex distributed aspect, consider the inherent coupling between DHCP and DNS. If a host is in DHCP, then its MAC address maps to a particular IP address, and if it is in DNS, then its IP address is mapped to a name. Thus the triple (hostname, IP address, MAC address) is a distributed aspect spanning the host itself, the DNS server, and the DHCP server. Once the value of that aspect is defined, it constrains values in all three domains and, by overlap, constrains contents of other configuration files whose aspects overlap. It is often considered good form to place a record for the hostname of a host into /etc/hosts; this would happen because the hostname aspect (on the local host) must be coordinated with the DNS/DHCP aspect (on the distributed network service layer).*

State of the Art

At most sites, distributed aspects are maintained and updated by hand, by distributing policies for *local* aspect control. For example, the policy writer must insure *manually* that the DNS server listed in /etc/resolv.conf is actually a DNS server, and that the zones of that server contain the appropriate SOA records for it to be an authority for the zones for which it is intended to be authoritative.

This difficulty in coordinating distributed aspects is also largely responsible for the incorrect belief that centralized coordination is necessary for effective configuration management. Tools such as LCFG [1, 3], BCFG [20], Puppet, Arusha [23], and others manage distributed aspects through centralized coordination. Mostly this is accomplished by storing values in a single data structure that can be checked on a central server for consistency. The strength of generative configuration management is that identity relationships among aspects (where several parameters must have precisely the same value) are addressed by generating multiple files from the same hierarchy of values, thus solving the aspect consistency problem implicitly.

To our knowledge, with one exception, none of the data models of production configuration management tools are explicitly aware of aspect constraints; they manage aspects by constructing some values as functions of others. Thus the kinds of aspects a typical tool can express are rather simple ones, where there is a functional relationship between the choice of value for one parameter and choices for values of others. The exception is Anderson's prototype implementation using SmartFrog [3], which dynamically computes aspect constraints and chooses among the resulting values. CFengine [4], on the other hand, is explicitly constraint oriented at the low level, but does not cope well with high level or aspect-level constraints.

One problem with contemporary mechanisms is that they take a lot of human labour to set up, and require that the centralized hosts generate configurations by acquiring and maintaining rather intimate knowledge of the hosts that they manage. In order to manage a distributed aspect, a central server must control *both sides* of the aspect. Initial setup takes a lot of time and specialized knowledge, and this has discouraged the use of such mechanisms except at the largest and most complex sites. While this setup is feasible, with some effort, in the networks of today, it cannot scale easily to future networks involving millions of pervasive nodes.

Practical Aspect-Oriented Design

So what does this mean to the practical administrator? Aspects are a way of *thinking* about the configuration management problem. They are a planning tool for distributed characteristics. When aspects overlap or work in concert, coordination is necessary to avoid contradictions. But, looking deeper, there are immediate benefits to thinking about and designing systems in terms of aspects, rather than basing design upon the capabilities of existing tools. In fact, it seems that the most effective way to save money spent on configuration management is not to utilize powerful tools, but to instead *refine the problem description* so that management difficulties are reduced or even eliminated.[6]

For example, it is common practice to run one kind of service per server-host, where possible. Why? In our present model, we understand this: running more than one service can lead to overlapping of aspects, because certain parameters might be needed by more than one service, risking the possibility of contradictions, and making the problem of maintaining and updating the server potentially more complex. Virtualization now provides a low-cost method for implementing the one-host-to-one-service practice, by simulating several independent servers with one physical machine.

Note that there are situations in which very complex systems exhibit no costly overlaps, again as a result of careful analysis. Consider, for instance, a linux workstation image consisting of a pre-tested suite of applications, managed as a unit, such as RedHat Enterprise Edition. We might want to think of the resulting workstation as a product of complex dependencies, but we can pay to have others do that thinking for us and manage each workstation as a unit. Thus we reduce cost by outsourcing the management. This is fine provided that we do not construct our own aspects that interfere with that remote management.[7]

[6]*MB*: Ideally these refinements would be equivalent, but our failure to model CM adequately in the past has led to a gap between common sense and technology.

[7]*AC*: Herein lies an important observation: *system administrators often construct aspect overlaps in crafting the* requirements *for a system.*

Rules for efficient aspect-oriented design of networks are thus simple and straightforward and are easily motivated by promise theory:

1. Factor services onto independent closures, e.g., virtual machines where possible (to eliminate aspect overlaps).
2. One manager for one aspect. Maintain clear separations in the source of aspect control, i.e., avoid interfering with systems that manage their own aspects, e.g., RPM and RedHat Enterprise.
3. Specify replicated parameters at a single source, e.g., one configuration file.
4. Document all remaining overlaps, so that future administrators will not fall into the trap of violating their constraints.

These rules seem intuitive, indeed they arise naturally in Service-Oriented Architectures (SOAs), which we come to shortly.

Simplest is Best?

How do we know when we have made configuration management as simple and straightforward as possible? Simple is relative, but many will agree that simple as possible is when there are minimal aspect overlaps, and the aspects in force are as *unrestrictive* as possible. Conversely, a site is "complex to manage" when overlaps cannot be eliminated and severely restrict choices. If there were an automated and reliable way to enforce an aspect via a software tool, then the complexity of managing that aspect is the complexity of managing the interface to the aspect, not the aspect itself.

At this point, we digress briefly to comment upon the relationship between the distributed aspect management problem and SOAs. This discussion will lay the groundwork for discussing methods of implementing the distributed constraints we have been discussing, and the next concept: closures.

Service-Oriented Architectures

Service Oriented Architectures (SOA) are currently in vogue. They ascend along with a heightened interest in outsourcing and delegation of responsibility in commerce. Clients need to be able to buy and sell services without surrendering their autonomy, or right to decide. SOAs enable the construction of distributed computing applications from a collection of autonomous, cooperating services. One does not expect that all the parts of the system are under the same jurisdiction.

For example, we no longer think of a "web application" as living on a single "web server"; the application is instead composed from the interactions of autonomous components, and linked via middleware that utilizes the Simple Object Access Protocol (SOAP) to expedite requests. The application thus spans several physical machines and perhaps even several enterprises, utilizing components from each. No single administrator controls the configuration of this arrangement.

Learning from SOAs

Let us be clear: we are not advocating the use of web services for configuration management. There are many reasons why this would not be the best solution. However, service architectures embody some compelling ideas that we can utilize in configuration management:

1. Subscribing to a service is not a simple matter of pointing each client at a server. It involves some form of service guarantee as well.
2. The protocols by which one receives a service are defined by the servers of the service, using a transaction that defines required inputs and their formats.

A compelling feature of SOAs is that the process of binding client to server is not just a matter of pointing each client at a server, but involves a two-sided agreement to provide and to utilize services. This means that in an SOA, one *manages service bindings* rather than *managing service references*. With some careful thought, we can apply this practice to non-web services such as DNS, DHCP, and the like. This is a key idea that we will develop further throughout the paper.

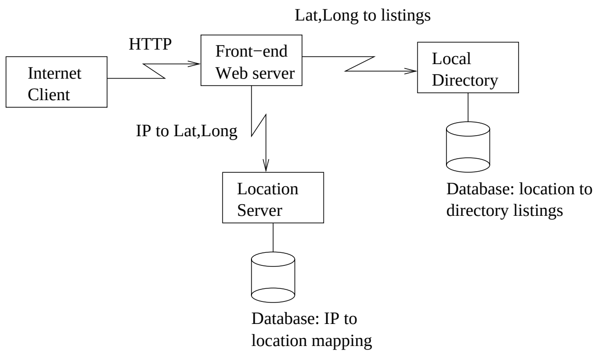

Figure 1: A service-oriented architecture in which a web application is composed from remote location and directory services.

As an example of a service-oriented application, consider Figure 1. There, a web application is created from a front-end server, bound to both a location service that maps IP addresses to latitude and longitude, and a geographically-aware search service that returns results for matching businesses, indexed by latitude and longitude.

This example exhibits many properties of an SOA:

1. Services are *autonomous* and even managed by different corporate entities.
2. Services are coordinated via negotiation between client and server.
3. There is a concept of quality of service that defines how quickly a server should respond to a request.

4. Clients can – at their option – change service providers dynamically, based upon whether a current provider is functional or not.

SOA and Configuration Management

A large part of configuration management involves receiving and utilizing remote services. However, theoretical results for configuration management have primarily concerned the low level practice of controlling bits on local disk or processes on a local system.

By contrast, the service concept deals with the intermediary issues of creating a distributed application. If we can view a computer network as a whole, as if it were an SOA application, then the problem of configuration management becomes primarily a binding problem between services and clients. This binding problem involves both network client transactions (entity to entity or peer to peer) and local transactions on the client machine (to make the machine able and ready to receive services).

There are thus two ways in which the service model applies to configuration management:

- As an information binding between hosts or peers in a network.
- As an information binding between data objects internal to a local host.

The components of an SOA application are known commonly by another name: *closures* [15, 16]. A closure is nothing more than an encapsulation of a service, that is – to some extent – self-managing. While closures do not need to comply with service standards such as SOAP or WSDL or WSAPI, any service that complies with these standards is in fact a closure. It is not surprising, therefore, that closures and SOA applications have some of the same strengths and limitations.

Closures

A closure is a domain of ''semantic predictability'' in which inputs result in outputs with a predictable structure. The central property of a closure is that of freedom from unknown effects; its behavior is completely determined by its transactions with the outside world, defined as input that it receives from various sources.

The configuration of a closure can be thought of as the sum of its transactions with the outside world, so that each output from a closure – in terms of behavior – is a function of all input received so far. Input can take many forms, including transactions, events, streams, etc. The only hard requirement for a closure's input is that it must be equivalent to a serializable source, i.e., one must be able to express ''what happened'' as a series of occurrences, including inputs, events, etc.

Definition 10: *A closure D is a service $\langle C_D, F_D \rangle$ where C_D describes constraints on input and F_D is a function mapping inputs to responses. F_D maps each sequence S of input transactions, each of which obeys constraints C_D, to a unique output $F_D(S)$ (which may be empty).*

This is a strange definition that is difficult to appreciate until one looks at its opposite. A closure is like a service whose output is a *function* of the totality of its input. The alternative is a service whose output is not such a function, i.e., its output varies with respect to other sources than just what you tell it. The crucial property that determines whether we have a closure or not is ''complete knowledge'' of all operations that might change its output. Any system in which we can claim such knowledge is said to be ''closed,'' while a system in which we cannot make the claim remains ''open,'' i.e., closures are ''fully determined.''[8]

Example 8 *The simplest possible closure is one that memorizes a mapping, e.g., a simple version of DNS. Inputs to the closure include queries that inquire about mappings, as well as transactions that change mappings. While no query changes a mapping, transactions do. So the result of a query is always the result of the sum total of the prior transactions that specify mappings. Since these transactions take the form of reloading the configuration file and DNS by nature forgets all but the last such transaction, the result of a query is completely determined by the last transaction of reading DNS configuration. This suffices to make DNS a closure. DDNS is also a closure, provided that we count DDNS assertions as transactional inputs.*

Example 9 *A database server is a closure; the result of a query depends upon all prior commands given to the server, all the way back to ''create database.''*

Example 10 *A ''business data object'' in a service-oriented architecture is a closure; it defines transactions (*SELECT, INSERT, MODIFY, DELETE*) that can change data state and presumes that no other operations will be utilized.*

A thing is not a closure if there is a way that the service response can change without a transaction, or not as a function of transactions.

Example 11 *If human administrators manually make changes to a system that expects to be manipulated only by a strict transaction protocol, closure will be broken.*

[8]*MB*: Closures are a computer science idealization to my mind. They ignore the effect of outside influences that one cannot necessarily control, e.g., mistakes made by inexperienced prying hands, i.e., they conjecture that we have more control over a system than is realistic. But they are still useful ways of talking about operations, that can be made *approximately correct* provided we make sure they are maintained using additional constraints such as iterative, convergent maintenance [7].

The way this usually occurs is for something that can change the output to remain unknown to the person interacting with the system. If you define transactions as ''actions taken by one system administrator'' and – unknown to you – there is *another* system administrator clandestinely configuring the system, you do not have a closure. Thus, rather trivially,

Theorem 1 *Any closure's behavior can be emulated by a set of SQL transactions, in which each closure transaction is translated into an SQL equivalent.*

Proof 4 *First, consider each closure transaction as a command with parameters. Translate those parameters into SQL parameters. The definition of a closure is that its output is a function of its input, where some transactions may be ignored. As SQL is Turing-Universal, the intent of each transaction can be translated into SQL, using the underlying database as the ''Turing tape.'' Thus the behavior of a closure can be emulated by SQL, as it can be emulated in any other programming language.*

In defining a closure, we have intentionally put as little structure into the closure as possible. Structure is imposed by algebraic rules that simplify the bookkeeping we must do to compute a closure output. These rules tell us when an input is *not* operative in producing an output, and define equivalence classes for input streams that produce the exact same output.

Example 12 *Suppose we have a simple closure that does nothing but store and retrieve string parameter values. It has two input input events,* GET *and* SET*, where the last* SET *determines the value of the next* GET *for a parameter. This last sentence says it all:* GET*s do nothing to modify state;* SET*s do modify state. Thus the next value for a* GET *is determined by the sequence of last SETs for each parameter, and the order of these* SET*s is not important once we have deleted previous* SET*s of the same parameter. Thus the simplicity of this closure results from the algebraic property that all transactions are stateless [17].*

The power of closures arises not from the relatively awkward definition, but from the fact that many common closures are easy to describe *algebraically*, in similar fashion to the examples above. Let us consider the algebraic properties of a selected group of closures.

Example 13 *We can think of a DNS server as receiving transactions about mappings from around the world, and queries that depend upon those mappings. At any one time, the result of a query is the last received mapping.*

Example 14 *We can think of a file-server as receiving (block-level) transactions to write blocks and returning (block-level) reads. At any time, the result of a read has the content written during the last write of that block.*

Example 15 *A web service closure [15] has inputs consisting of queries and mappings. Queries do not affect mappings, while mappings directly affect which page is returned for a query.*

Several other properties of closures are worth repeating from [15]:

1. Closures are a unit of *independence* in a configuration; the closure only behaves according to the inputs it receives, and no others.
2. Closures can span network nodes and constitute the behavior of peer-peer infrastructures, e.g., DNS.
3. Closures can communicate amongst themselves to create larger closures, e.g., combining web, DNS, DHCP, and routing layers.

We need closures to understand aspect implementation, so let us look at how the two relate.

Closures and Aspects

The main difference between closures and aspects is the use of interior versus exterior constraints. A closure's constraint model is *exterior*; its behavior is defined as a function of its inputs, with no reference to *how* that behavior is assured. An aspect has no explicit concept of behavior; it is instead an interior measure of how something should be configured with implicit consequences; the behavior of an aspect is exterior to its definition. In other words, an aspect is a declarative concept with implicit behavioral consequences.

The mapping between configuration and behavior has been systematically studied in [18] and we adopt the notation of this work here. Behavior is abstractly represented as a subset of a set of tests that can be either true or false. We can think of the current state of a system as a ''subset of known symptoms'' that can be observed. The subset consists of the tests that are true under a given condition. The behavior of an aspect is a relation describing the correspondence between aspect values V and aspect behaviors T: a set of ordered pairs (V, T) where V is an aspect value and T is a subset of tests from a test suite $\mathcal{T}$ that are true when the aspect has the corresponding value. This may be a function of the aspect's value, or instead a relation between an aspect and many possible sets of test outcomes.

Definition 11: *An aspect is closed with respect to a specific set of behaviors $\mathcal{T}$ if there is a map between values for the aspect and behaviors that are exhibited. If not, the aspect is open with respect to $\mathcal{T}$.*

Note that this definition is carefully crafted. There is no such thing as an aspect closed with respect to every behavior; one must select a set of behaviors to observe. Likewise, it is usually possible to find some behaviors that are uniquely determined by aspect choices; these are the *closure behaviors* of an aspect (or even the *closure of the aspect*).

> ***Lemma 2*** *The closure behaviors of an aspect, together with the aspect, form a closure.*

> ***Proof 5*** *The aspect – as a potential closure – has inputs that are the values of configuration parameters and outputs that are sets of behaviors that may or may not be exhibited by those choices. By definition of aspect closure behaviors, these behaviors do not change except as a result of parameter changes, which may or may not be accomplished by other interactions. Considering the stream of inputs, including parameter changes, the definition of a closure is met.*

In other words, *any aspect is a closure with respect to a selected set of behaviors*.

This fact has important implications. The boundaries of closures are determined by what is expected, and what is known, not by what is controlled. Aspects give us a way of easily constructing closures, which we did not have before. The key to constructing a closure is to *think* about an aspect in the proper way, and define "enough" to close it! In other words, *closures are not so much constructed as much as they are discovered.*

Practical Closures

Closures – like aspects – allow one to *think* about the configuration management problem efficiently and effectively. Recall that the complexity of aspect composition depends upon the amount of overlap, and the overlap is determined by the *interface* to the aspect in question. Then add the concept that an aspect is closed if its behavior over a parameter set is predictable.

> ***Principle 1*** *The manageability of an aspect, relative to a fixed set of tools, is increased by limiting variability of parameter values for the aspect.*

In other words, if one can "get away" with a small number of variations, then the aspect becomes easier to think about and manage. And, if one limits sufficiently and circumscribes its behavior accurately enough, it becomes a closure. For example, we might limit the way hosts bind to databases: innocent at first glance, but a good cost-saving mechanism.

In previous closure work, the implication was that closures are *constructed* by building complex interface code. This simple analysis shows that closures are instead *discovered* by factoring otherwise complex systems.

Promises

No general language has been developed to describe how aspects can be managed or described at a low level, nor how closures should communicate in order to implement reliable management changes. So far, the language of closure communications has been in terms of "demands" and "acknowledgments." However, as we have already commented, this idea of making demands is intrinsically at odds with the reality of distributed systems.

The fact that different decision agencies are involved in distributed systems changes many things. One can no longer imagine being in complete control of a network of hosts, making demands, unless everyone agrees to behave in a subordinate fashion and comply with our expectations of them. This brings us to the notion of *promises* and, coincidentally, back to something like a Service Oriented Architecture.

Voluntary Cooperation

If we cannot guarantee behavior by requirement or demand, can we at least make agreements with components of the system to behave in a manner that is acceptable to us? Service Level Agreements (SLA) are one manifestation of this realization for commodity services. These are familiar to most of us. However, Service Level Agreements are too vague and too complex a construction to be useful for analysis. We therefore introduce the atomic idea of a *promise* [10, 12, 14].

We begin with the players in the system that make promises.

> ***Definition 12: Agents*** *An agent is any entity within any system that can make or receive promises, and which computes all decisions autonomously. The information within an agent is not available to any other agent, unless that availability (of information) is promised.*

An autonomous agent cannot be forced or coerced into any behavior against its will.[9] In particular, one cannot demand or require anything of an autonomous agent, one can only suggest or request something of it, by expressing a willingness to receive and use a service that one hopes it will promise to provide.

Because we are developing a language for abstracting cooperative agreements, we are free to apply this model to a variety of scenarios, even where agents represent "dumb" resources like disks or files if we choose. This does not mean that files are intelligent, it simply means that someone is controlling the file's properties and behavior as an independent object, and this abstraction allows us to describe that interaction in low level atomic terms.

> ***Definition 13: Promise*** *A promise is a specification of future state or behavior from one autonomous agent to another. It is thus a unit of policy. A promise is a link in a labeled graph $G = \langle A, L, \Pi \rangle$ in which the set of nodes A are agents, the directed edges or links L are promises and the labels Π are called the promise body. A promise is a private announcement $\pi \in \Pi$ from the sender node s of the promise to the receiver r. We denote it like this:*
>
> $$s \xrightarrow{\pi} r \tag{1}$$
>
> *meaning that s promises π to r.*

[9] *MB*: This property usefully agrees with the security model used by cfengine.

This definition agrees with our commonplace understanding of a promise, but is sufficiently formal that we can use it for analysis. Promises will form a building block for aspect closures, and will allow us to rewrite familiar concepts of configuration management such as *operators* [8] entirely in terms of voluntary cooperation.

Give and Take

A promise is a specification of behavior, but it might be unclear at this point how a promise might describe behavior. There are several principles that apply to this description:

1. A promising agent can only describe its own behavior or behaviors of others that it has directly observed.
2. That behavior includes the properties of a specific set of *interactions* that may occur with neighboring agents.
3. There is no reason to identify an autonomous agent with a "system" or a "machine." One can create many autonomous agents within a single system, as components.

There are two primitive types of promise body from which it is believed all others can be constructed: these are the service and acceptance promises (or promises to given and take). In addition it is convenient to define a third type called the coordination promise [14] as a shorthand.

- A service (giving) promise, whose body is denoted π, is the basic type of promise which denotes a restriction of behavior by the promising agent in the manner of a service:
$$a \xrightarrow{\pi} b \qquad (2)$$
involves an offer of service from a to b and implies a specification of future behavior of a towards b.
- A usage or acceptance promise (taking), denoted $U(\pi)$, is the promise to receive or use a service π promised by another agent.
$$a \xrightarrow{U(\pi)} b \qquad (3)$$
involves a receipt of information and service by the promising agent a from b. It can be related to access control, for instance.
- A coordination (or subordination) promise, denoted $C(pi)$, is the promise to do the same as another agent with respect to a promise body π.
$$a \xrightarrow{C(\pi)} b \qquad (4)$$
involves that b informs a about its actions with respect to promises of type π, and the receipt and usage of that information by a. This promise is a subordination because a is willingly giving up its autonomy in the matter of π by agreeing to follow b's lead. Note that this agreement is made on a peer to peer basis, and implies no *a priori* centralization.

Types of Promise

Promises are only useful if they can be made about many kinds of issues. To distinguish between kinds of promises, each promise body consists of two parts: a *type* τ which labels the issue being addressed along with its possible domain of variability, and a *constraint* C which tells us which subset of the domain of possibility for that type is being promised.[10]

Example 16 *Consider a configuration promise. Suppose that τ represents a configuration parameter belonging to the promiser and $C \sqsubseteq \tau$ represents a set of allowable values for that parameter that are allowed by policy. Then $pi = \langle \tau, C \rangle$ is a promise that the values C will be adhered to as a value for the parameter described by τ.*

Note that a configuration *parameter* is a syntactic thing, while a *promise* about that parameter constitutes a form of *knowledge*. It is best to think about active promises as a form of "distributed knowledge" about a system. When an entity promises something, it limits its behavior in observable ways. The union of "promises made" is a form of distributed system state.

Example 17 *Suppose that τ represents a subset of parameters, belonging to a single host, within a distributed aspect and C represents some constraints on those parameters. Then the promise represents a policy atom on the particular promising host that expresses its personal part of that aspect. In other words, a promise expresses limits imposed upon an aspect by one individual agent.*

Promises can be combined into knowledge about the network. The method of combination of promise information is specific to the kind of promise.

Example 18 *Suppose that we take the point of view of a single autonomous agent A_0. Agent A_1 promises agent A_0 that it is a directory server (constraint C_1), and agent A_2 promises A_0 the same thing for itself (constraint C_2). Then the two promises offer alternatives to the receiver but do not oblige it in any way. The result is that the receiver is free to assume the logical-or of the input promises ($C_1 \vee C_2$).*

Example 19 *Suppose we again take the point of view of a single autonomous agent A_0. Suppose agent A_1 promises to A_0 some information (i.e., it is constrained to provide that information) and that the information is part of a distributed aspect, e.g., it informs A_0 where to find DNS service. This information in no way obliges A_0 to use that information. However, if A_0 promises A_1 to use that information, it is constrained to follow A_0's suggestion.*

The important point from these examples is that what an agent does with promises, and the meaning of combining them, is entirely up to the individual agents. Autonomous agents, like closures, have the property of being capable of engaging in arbitrary reasoning based upon the inputs they are given.

[10] *AC*: It seems that no matter how flexible I am about interpreting this definition, a promise is more general than that! "But wait, there's more!"

Promise Notation

Promise graphs become complex quickly and are difficult to notate other than in pictures. To ease notation, we adopt some simple notational conventions. First, if a *set* of nodes is involved in making the same promise, we unambiguously represent the set of promises as a single promise between sets. If S and R are sets, then $S \xrightarrow{\pi} R$ means means the set of promises $s \xrightarrow{\pi} r$, for $s \in S$ and $r \in R$. Similarly, $S \xrightarrow{\pi} r$ is the set of promises $s \xrightarrow{\pi} r$, for $s \in S$, and $s \xrightarrow{\pi} R$ represents the set of promises $s \xrightarrow{\pi} r$ for $r \in R$.

It is also often interesting to know which kinds of promises have been made, without knowing necessarily who made them or to whom they were made. We write $S \xrightarrow{\pi} (.)$ to mean that each $s \in S$ has made the promise π to some unknown set of hosts, and $(.) \xrightarrow{\pi} R$ to mean that some node has promised π to each $r \in R$.

Roles

An important concept in promises is that of a *role*. A role is a kind of emergent pattern that we can identify in the promises made or received by agents.

Definition 14: Role *Suppose S and R are sets of autonomous agents and there is a promise of type τ between each node in $s \in S$ and each node in $r \in R$. Then S and R are said to form role-sets of type τ. S is said to have a sender role of type τ while R is said to have a receiver role of type τ.*

Example 20 *Let S be the set of web servers and R be the set of web clients that can access the servers S. Then $S \xrightarrow{web} R$ describes two roles: $S \xrightarrow{web} (.)$ (S are "web servers") and that $(.) \xrightarrow{web} R$ (R are "web clients").*

If a client receives an offer of service but does not promise to use that service, the role of the client is limited to that of being promised the service. The client would have to formally agree to *use* the service in order to be classified as a service client according to the role model (since this implies a binding commitment).

Example 21 *The simplest example of a role is that of a file server that serves home directories. The file server promises to serve up home directories to clients, and the clients in turn utilize that service in order to allow users interactive access to their files. The fact that the file server's promise is implicit, i.e., determined by use rather than by an explicit communication, is not important. The role of "file server" is an emergent property of how the server is used, not a matter of intent.*

In promise theory, one can do many things with roles. They can be composed to form composite roles (i.e., through the holding or use of more than one promise):

Definition 15: Composition of roles *Suppose R_1 and R_2 are role-sets with respect to types τ_1 and τ_2, where the direction of each τ_i may vary. Then $\tau_1 \cap \tau_2$ is also a role.*

Example 22 *Suppose that a web server s sends a promise $s \xrightarrow{web} R$ to a set of clients R. Those clients who received the promise form one role R. Those clients who also responded with a promise to use form another role $R' \subset R$. The clients who for some strange reason respond with a promise to use without a matching promise to serve form a third role R'' disjoint from R. There is no particular reason that an agent cannot promise to use a service that does not exist. The distinction between these roles is whether one or two promises were made.*

In the strictest interpretation of promise theory, roles are distributed aspects and therefore cannot be forced or decided by anyone. However, in practices roles can be identified empirically (a postiori), or be decided as a design decision in advance (a priori) if we are in the fortunate circumstance of controlling several (formally) independent agents.

Lemma 3: Agents are trivially roles *Let A be the set of agents. A is a role.*

Proof 6 *Consider the empty set of promises $\varnothing$. Every agent in the graph sends and receives this set of 'no promises' in addition to any other promises it might send or receive, thus the pattern of no promises is identified as a subset within the promise graph at every node. Hence every agent node plays a role of 'no promise,' which we can re-name 'autonomous agent.'*

Promise theory is essentially a model for the planning and analysis of generalized *services*. The challenge is to use promises to see how configuration management, perceived as a service, can be carried out by autonomous agents. We refer readers to [10, 12, 11] for more information about promises.

Promises and Closures

The relationship between promises and closures is subtle but straightforward. In all that has been published about closures, little has been said about the language utilized by closures to communicate with one another. The concept of autonomous agent, utilized in promise theory, is roughly the same as a the concept of closure, though closure is more restrictive as a concept. Promises, as an inter-agent language, are an ideal mechanism with which closures can communicate.

Theorem 2 *Closures are a subclass of autonomous agents.*

Proof 7 *Closures require that all transactions are functions of prior transactions and nothing else. This is more restrictive than the definition of an autonomous agent, which requires actions based upon autonomy and previous history, but does not limit the sources of information utilized for such actions.*

Transactions include such things as making promises, but this is more restrictive than the restriction of autonomy. An agent's output could change via other mechanisms than promises or transactions, e.g., resource requirements. Also, while agents act asynchronously and without any notion of transaction, closures rely upon transactions and transaction serialization to arrive at a notion of internal state (in terms of the sequence of past transactions). So in general, a closure is an agent, but not all agents are closures. In like manner, promises are appropriate closure interactions, but not all closure interactions are promises; some are transactions in the traditional sense of being tightly coupled and not subject to debate or choice.

Client cooperation is an important way of building distributed services. On the one hand, we would like to 'demand' the compliance of services around the network, since we are used to "control" rather than "cooperation," but we cannot.

Example 23 *Consider the case of a client binding to a DNS server (see Figure 2). The client can ask a candidate server for a "promise" of service. If the DNS candidate responds with an acknowledgment, this means that its side of the distributed aspect called DNS is ready to converge to a coherent and functional state. Then, when the client adds the DNS server to its resolver table, the distributed aspect becomes complete and functional.*

In the figure, notice that the client makes no promises to the server. They have no agreement. Rather, the master server promises to use and requests the client sends, and to reply to them if they arrive. The relationship between master and slave is more complex. Slave status is acquired by the slave agent subordinating itself with a C(DNS) promise. This means it will make the same promises about DNS that the master will. It agrees to use the zone data sent by the master. The client promises a policy adjudicator that it will contact the master server, and if there is a timeout, it will contact the slave.

Promises and Aspects

We have seen in a previous section how to view the values of aspects in a network as synonymous with its configuration. We now study service binding aspects in more detail. We show, particularly, that there is no way to separate the function of a service binding from the guarantees of function that a server can provide and, in turn, the promises the server can keep. In this way, a "promise kept" is stronger than any current mechanism for centralized control of service bindings.

First, we need a mechanism for "semantic grounding" of the promises on each host. Let Ω be an oracle that describes host documentation and the reasonable constraints of single-host configuration. Ω is the union of all local aspects, and could be described as the source of a union of individual overlapping promises $\langle \tau, C_\tau \rangle$. In other words, constraints arising from the documentation of a system are promises of the form $h \xrightarrow{\text{U(data)}} \Omega$, where h is the local host (i.e., the host promises to comply with documentation).

Proposition 5: *The class of promises that the grounding agent makes are a role that determine the kind of machine being configured.*

Proof 8 *All machines with the same kind of architecture have the same grounded promises, hence they are members of a role by definition.*

The documentation Ω is nothing more than an embodiment or symbol of the constraints arising from the system itself.

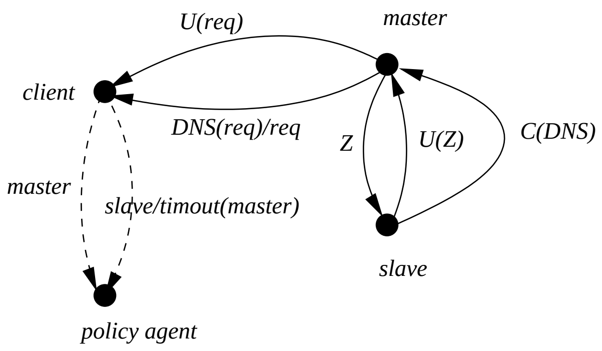

Figure 2: A promise graph for a DNS lookup.

Second, we consider the local policy on a machine as having a different form of grounding.

Definition 16: *Let Γ represent a declaration of local policy, which may change. Promises of the form $h \xrightarrow{U(data)} \Gamma$ determine desirable behavior on the local host.*

The relationship of Ω to Γ is that of a hard aspect to a soft one; limits versus desires.

In formal promise theory, Ω and Γ are possibly hidden parts of the agent; here we make them explicit only to describe the relationship between aspects and promises. Ω represents all of the hard aspects, i.e., the things about the system that are not negotiable; Γ represents soft aspects, determined by policy. These aspects are inputs to the agent's view of the world, not as binding obligations, but instead as information that the agent can use, along with all other promise information with which it is provided. Documentation and experience are as much promises as are messages from an external agent; they are guarantees of specific behavior for the underlying systems.

The point of this discussion is that for all practical purposes, everything an agent needs to do or know about the world can be expressed by some kind of promise. Some of these promises come from other agents.

We emphasize once again that agents in promise theory are not to be confused with configuration agents (e.g., cfagent), their closest interpretation would be individual configuration objects such as files or processes.

Distributed Aspects

A distributed aspect has often been viewed as "pointing" clients to a specific server. We take a different view, in which both client and server take responsibility in a more fundamental way. The key to our argument is the following simple idea:

> ***Proposition 6:*** *A binding is a transaction between service provider and service consumer, in which the server guarantees to reliably provide a service while a client guarantees that it will also reliably consume the service.*

This simple idea has such subtle ramifications that it must be studied in some detail to understand the text that follows.

> ***Example 24*** *In a typical configuration management scenario, a binding is a simple concept of "naming a server" in some context. We "point" our resolver at a "DNS server," or "point" our outgoing mail at a "mail relay." This "pointing" is a matter of blind faith; we assume at some level that the servers we are pointing to are actually providing the service we require, and that some mechanism, either ours or someone else's, has configured them properly to provide that service.*

We wish to challenge this idea of binding in an extremely straightforward (and even seemingly trivial) way.

> ***Principle 2*** *A distributed aspect (e.g., a client-server binding) is configured correctly only if both sides of the client-server relationship are both conversing with the appropriate server and functioning properly as server and client.*

This may seem silly as a principle. Everyone knows this, except that we usually configure the server and client *separately* and manage the two entities as *separate aspects*. It is unfortunate that we also tend to think of these aspects as separate entities as well. In a sense, we do not acknowledge the *distributed aspect* that consists of both of these functioning together, correctly. It is this aspect, not the individual servers, that we are responsible for managing. In other words, a promise is more than a pointer. It is a "guarantee," somewhat like that contained in an SLA, that a service is up and running and answering queries.

This simple way of thinking leads to a drastically different understanding of configuration management as a practice. The "master-slave" view of configuration management is that we have to make all the servers work correctly, and point clients at servers (taking it on faith that the servers will function properly when pointed to), and everything will just work. The reality is that each binding between a client and a server is something that must work properly *as a distributed aspect.* This may require some coordination between server and client that we tend to ignore, but that is crucial to network function.

> ***Example 25*** *A very simple example of a distributed aspect with non-trivial behavior arises from incompatibilities between server and client parts of NFSV3 when utilized over a specific router between a Sun file-server and a Linux client. The aspect, to function properly, must utilize NFSV2 instead. The reason that this is true is not a function of either the server or the client, but of the router between them! Correct function of the client and server is not relevant to function of the aspect; a third piece of the puzzle constrains behavior further and – without that piece – two perfectly configured hosts fail to interoperate. Most important,* this behavior of the binding remains invisible unless one looks at the behavior of the whole binding, rather than the behavior of its endpoints.

Using Promises

A next-generation configuration management tool might utilize promises in an extremely straightforward way. A configuration tool (let us avoid the confusion of calling it an agent) runs on each host to manage service bindings. The tool running on a host providing a service declares this fact via a number of promises. All service bindings are based upon promises received. Among promises received, arbitrary choices are made as to which servers to use, or perhaps some primitive form of distance calculation is utilized to determine the "nearest" server from among several candidates.

It is important to note that every promise received corresponds to a *functional* machine providing a service, not just a pointer to a machine that may or may not be working at the time. So the problem of pointing machines to non-existent services disappears. Every use-promise informs a server about which agents to contact if an outage is expected. This gives the clients time to re-organize their bindings to point to usable servers during the outage.

> ***Example 26*** *Consider the often costly problem of maintaining default printers for desktop workstations and remote users. We want the default printer to be "near" to the user or desktop, presenting an ongoing and expensive management problem as printers and desktops are installed or retired. Now consider the same binding problem and apply promise theory, running an agent to report upon the status of each printer, and bind those agents into a role. The centralized database of nearest printers is replaced by a series of local databases, one for each agent, defining the nearest desktops to their printers. Maintaining this information requires only notifying a single local agent of a change, rather than the whole database of nearest printers. Determinism is*

preserved without centralization, and the management problem is naturally distributed to agents within the control of each separate administrative domain. Yes, as the reader might be guessing already, this is a closure as well!

Example 27 *Consider the problem of maintaining resolver bindings in the presence of network changes, and apply the same architecture of distributed agents as above. Each DNS server reports its availability to all consuming agents, and they can bind at will. This enables online load-balancing using caching and stealth servers, without reconfiguring the network for each addition or deletion of server.*

Example 28 *Consider the problem of determining primary gateways for each host. This is already solved through routing protocols which, if one considers them carefully, consist entirely of promises.*

Shedding Light on Configuration Management

A *taxonomy* is a description of the space of options for a thing. In this case, the "thing" in question is the practice of configuration management. Using aspects, closures, and promises, one can describe many current configuration management strategies, and compare them within that theoretical framework. This gives us a fundamental idea of each strategy's strengths and limits.

A typical user of CFengine is using promise theory without knowing it. The cfagent process receives a configuration file from a central server that – in its essence – contains lots of promises. Instructions that bind the host to specific servers can be interpreted as promises (from the master server) that the services will be present and available. The exact same file enforces distributed aspects, and may in fact determine closures, via its contents and ability to correct errors. The complexity of this file is its main weakness; promises, aspects, and closures offer a way to conceptually simplify its contents in the future.

A typical user of configuration scripts uses promises in a much simpler way. The user of a script is – in essence – personally promising that the script will work, which in turn is the same thing as promising that the configuration settings changed by the script are appropriate and will have appropriate effects. Again, the concepts of aspects and closures are implicit and well-hidden within the script; we cannot currently analyze scripts in enough detail to infer the reasons for a change from the script that makes the change.

A typical user of LCFG, BCFG2, or other generative tools depends upon the tool to hide information about promises, aspects, and closures that the tool creates and manages. The strength of these tools is information hiding; the user need not cope with the true complexity of aspects. But at the same time, the centralized planning functions of these tools cannot react automatically to distributed changes (e.g., between autonomously managed domains) so that promises may provide a way to make these tools more adaptive to changes in network state.

Conclusions

We have seen in this paper how the concepts of closures and promises – seemingly very different – are actually sides of the same coin. The "glue" by which this comparison is made is the concept of an "aspect," as well as the idea that a configuration is a composition of overlapping aspects. Aspects are important because they are closer to the way in which administrators currently think. As Paul Anderson has noted on several occasions, the challenge for the future is to look for ways to compile high level aspects into low level operations. We believe that this goal is now much clearer from our formalizations. We now have a complete story that captures and unifies all of our state of the art understanding of configuration management:

1. Aspects are constellations of promises.
2. Promises with their agents can form closures.

We identify a progression from high level to low level:

$$\text{High level} \rightarrow \text{Low level}$$
$$\text{Planning} \rightarrow \text{Implementation}$$
$$\text{Aspects} \rightarrow \text{Promises}$$

This progression makes no assumptions about centralization or authority, not does it have to be a linear progression. One can approach it "top-down" or "bottom-up" [8], as one sees fit. Not every aspect is necessarily implementable, if the associated promises are not made (or kept), and we can discover this by attempting the decomposition from high level goals to low level implementation.

It follows from the requirement of convergence that observation is a key element in configuration management [9, 18]. The separation of change management from monitoring is a fundamental mistake in current systems. These issues need to be tightly woven to make reliable bindings with predictable service agreements. It is our belief that a next generation of configuration management tools can do this, utilizing promises, aspects, and closures as conceptual parts of designing and architecting an efficient and robust configuration management strategy.

Acknowledgment

We are grateful to Paul Anderson, Jan Bergstra, and Æleen Frisch for formative discussions. Narayan Desai, Marc Chiarini, Ning Wu, Hengky Susanto, Josh Danziger, and Bill Bogstad gave helpful comments on initial drafts.

Author Biographies

Mark Burgess is a Professor of Network and System Administration at Oslo University College, Norway. He is the author of cfengine and several books and papers on system administration. He can be reached by electronic mail as Mark.Burgess@iu.hio.no .

Alva Couch is an Associate Professor of Computer Science at Tufts University. He is an author of numerous papers on the theory and practice of system administration, and currently serves as Secretary to the USENIX Board of Directors. He can be reached by electronic mail as couch@cs.tufts.edu .

Bibliography

[1] Anderson, P., "Towards a high level machine configuration system," *Proceedings of the Eighth Systems Administration Conference (LISA VIII)*, USENIX Association, Berkeley, CA, p. 19, 1994.

[2] Anderson, P., *System Configuration*, *SAGE Short Topics in System Administration*, 2006.

[3] Anderson, P., P. Goldsack, and J. Patterson, "Smartfrog meets lcfg: Autonomous reconfiguration with central policy control," *Proceedings of the Seventeenth Systems Administration Conference (LISA XVII)*, USENIX Association, Berkeley, CA, p. 213, 2003.

[4] Burgess, M., "A site configuration engine," *Computing systems*, MIT Press, Cambridge MA, Vol. 8, Num. 309, 1995.

[5] Burgess, M., "Automated system administration with feedback regulation," *Software practice and experience*, Vol. 28, p. 1519, 1998.

[6] Burgess, M., "CFengine as a component of computer immune-systems," *Proceedings of the Norwegian conference on Informatics*, 1998.

[7] Burgess, M., "On the theory of system administration," *Science of Computer Programming*, Vol. 49, p. 1, 2003.

[8] Burgess, M., *Analytical Network and System Administration – Managing Human-Computer Systems*, J. Wiley & Sons, Chichester, 2004.

[9] Burgess, M., "Configurable immunity for evolving human-computer systems," *Science of Computer Programming*, Vol. 51, p. 197, 2004.

[10] Burgess, M. and S. Fagernes, "Pervasive computing management: A model of network policy with local autonomy," *IEEE Transactions on Software Engineering*, (submitted).

[11] Burgess, M. and S. Fagernes, "Pervasive computing management: Applied promise theory," (preprint), (submitted).

[12] Burgess, M. and S. Fagernes, "Pervasive computing management: Policy through voluntary cooperation," (preprint), (submitted).

[13] Burgess, M. and R. Ralston, "Distributed resource administration using cfengine," *Software practice and experience*, Vol. 27, p. 1083, 1997.

[14] Burgess, Mark, "An approach to understanding policy based on autonomy and voluntary cooperation," *IFIP/IEEE 16th international workshop on distributed systems operations and management (DSOM), in LNCS 3775*, pp. 97-108, 2005.

[15] Couch, A., J. Hart, E.G. Idhaw, and D. Kallas, "Seeking closure in an open world: A behavioural agent approach to configuration management," *Proceedings of the Seventeenth Systems Administration Conference (LISA XVII)*, USENIX Association, Berkeley, CA, p. 129, 2003.

[16] Couch, A. and S. Schwartzberg, "Experience in implementing an http service closure," *Proceedings of the Eighteenth Systems Administration Conference (LISA XVIII)*, USENIX Association, Berkeley, CA, p. 213, 2004.

[17] Couch, A. and Y. Sun, "On the algebraic structure of convergence," *LNCS, Proceedings 14th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management*, Heidelberg, Germany, pp. 28-40, 2003.

[18] Couch, A. and Y. Sun, "On observed reproducibility in network configuration management," *Science of Computer Programming*, Vol. 53, pp. 215-253, 2004.

[19] Couch, A. L., N. Wu, and H. Susanto, "Towards a cost model for system administration," *Proceedings of the Nineteenth Systems Administration Conference (LISA XIX)*, USENIX Association, Berkeley, CA, pp. 125-141, 2005.

[20] Desai, N., R. Bradshaw, S. Matott, S. Bittner, S. Coghlan, R. Evard, C. Lueninghoener, T. Leggett, J.-P. Navarro, G. Rackow, C. Stacey, and T. Stacey, "A case study in configuration management tool deployment," *Proceedings of the Nineteenth Systems Administration Conference (LISA XIX)*, USENIX Association, Berkeley, CA, p. 39, 2005.

[21] Finke, J., "Automation of site configuration management," *Proceedings of the Eleventh Systems Administration Conference (LISA XI)*, USENIX Association, Berkeley, CA, p. 155, 1997.

[22] Finke, J., "An improved approach for generating configuration files from a database," *Proceedings of the Fourteenth Systems Administration Conference (LISA XIV)*, USENIX Association, Berkeley, CA, p. 29, 2000.

[23] Holgate, M. and W. Partain, "The arushra project: A framework for collaborative UNIX system administration," *Proceedings of the Fifteenth Systems Administration Conference (LISA XV)*, USENIX Association, Berkeley, CA, p. 187, 2001.

[24] Kanies, L., "Isconf: Theory, practice, and beyond," *Proceedings of the Seventeenth Systems Administration Conference (LISA XVII)* USENIX Association, Berkeley, CA, p. 115, 2003.

[25] Narain, Sanjai, "Network configuration management via model finding," *Proceedings of the Nineteenth Systems Administration Conference (LISA XIX)*, USENIX Association, Berkeley, CA, p. 155, 2005.

[26] Patterson, D., "A simple way to estimate the cost of downtime," *Proceedings of the Sixteenth Systems Administration Conference (LISA XVI)*, USENIX Association, Berkeley, CA, p. 185, 2002.

[27] Roth, M. D., "Preventing wheel reinvention: the psgconf system configuration framework," *Proceedings of the Seventeenth Systems Administration Conference (LISA XVII)* USENIX Association, Berkeley, CA, p. 205, 2003.

[28] Sun, Y. and A. Couch, "Global impact analysis of dynamic library dependencies," *Proceedings of the Fifteenth Systems Administration Conference (LISA XV)* USENIX Association, Berkeley, CA, p. 145, 2001.

[29] Traugott, S., "Why order matters: Turing equivalence in automated systems administration," *Proceedings of the Sixteenth Systems Administration Conference (LISA XVI)*, USENIX Association, Berkeley, CA, p. 99, 2002.

[30] Traugott, S. and J. Huddleston, "Bootstrapping an infrastructure," *Proceedings of the Twelth Systems Administration Conference (LISA XII)*, USENIX Association, Berkeley, CA, p. 181, 1998.

[31] Wang, Yi-Min, Chad Verbowski, John Dunagan, Yu Chen, Helen J. Wang, Chun Yuan, and Zheng Zhang, "Strider: A black-box, state-based approach to change and configuration management and support," *Proceedings of the Seventeenth Systems Administration Conference (LISA XVII)*, USENIX Association, Berkeley, CA, p. 159, 2003.