Experience in Implementing an HTTP Service Closure

Steven Schwartzberg – BBN Technologies
Alva Couch – Tufts University

ABSTRACT

One ideal of configuration management is to specify only desired behavior in a high-level language, while an automatic configuration management system assures that behavior on an ongoing basis. We call a self-managing subsystem of this kind a *closure*. To better understand the nature of closures, we implemented an HTTP service closure on top of an Apache web server. While the procedure for building the server is imperative in nature, and the configuration language for the original server is declarative, the language for the closure must be transactional; i.e., based upon predictable and verifiable atomic changes in behavioral state. We study the desirable properties of such transactional configuration management languages, and conclude that these languages may well be the key to solving the change management problem for network configuration management.

Introduction

HTTP servers are complex applications. In crafting a valid configuration file for a server such as Apache, there are many options, often with cryptic names and unclear meanings. Choices for many options seem not to matter to the end-user, e.g., the exact locations of content hierarchies within the filesystem. Choices for other options have critical effects, such as whether to allow CGI programs within a particular directory to execute. Simple typos in the configuration file can be difficult to locate and have unpredictable results. Thus, to assure reliable service, many configuration changes must be made by an experienced system administrator.

We manage an HTTP server cluster where the majority of responsible system administrators and content providers have historically been relatively inexperienced students. As a result, there has been considerable service downtime due to misconfiguration of the server, giving content files inappropriate names or MIME types, inappropriately protecting content, and even allowing servers in the cluster to differ in configuration.

Content providers often make serious errors in naming files and setting permissions for HTTP content. Either content is protected too restrictively to be available, or content protections are permissive enough to pose a security risk. Users also have difficulties ensuring that files have extensions that match their content. Typical examples include inadvertently exposing private contents of scripts by giving them incorrect extensions or filing them in an inappropriate directory, or making content directories world-writable, thus posing a security risk.

Naive editing of HTTP configuration files can also cause unexpected and costly downtime for web servers. When many virtual domains inhabit one server, one configuration error can be catastrophic, as it will bring down all of the domains. In practice, it can take a lot of time to recover from an error if changes have not been carefully tracked. At times, low-traffic virtual domains have been down for weeks due to undocumented changes that had hidden effects.

An HTTP Closure

We seek to solve this problem by surrounding our HTTP service with a closure, as described in [17]. A closure is a self-managing component of an otherwise open system. We intended our closure to:

1. Allow relatively untrained users to reliably create relatively complex configurations involving virtual domains and aliases.
2. Allow reliable creation and deletion of virtual domains.
3. Ensure appropriate protections and MIME types for content.
4. Protect against unauthorized changes to content or configuration.

To accomplish this, we had to make a radical departure from the way HTTP servers are usually administered.

At the beginning, we were inspired by several related projects. DryDock [20] is a content-management system that allows the submission of web pages to a web server, after they have checked by a human being. While it provides some desired features, including content validity checking, DryDock is more of a content checking and approval method than a web server configuration tool. TemplateTree II [30] can help configure the webserver configuration file, by automatically filling in blanks in a pre-determined template, but seems to stop short of being able to handle advanced features such as virtual domains. Each of

the virtual domains requires the addition of a new template to the file. Charlie [38] is a content-mapping webserver in which URL's are explicitly mapped to files by a declaration file, and service is not determined by filesystem structure. Our initial goal was to combine these ideas into a reasonable HTTP closure in which:

1. Content is specified by mappings between URLs and files (Charlie).
2. The appropriate configuration is generated from intermediate data (TemplateTree II).
3. Content is checked for type validity before being published (DryDock).

Our hope was that the resulting synthesis would be easier to use than any of its predecessors.

Recently, others have attempted part of this process independently. The Virtualmin [9] environment within the Webmin [8] web-based administrative environment solves the problem of defining virtual servers neatly, but does not deal with the problems of content management and assuring that content is provided with correct MIME types, etc. Both Virtualmin and Linuxconf [19] support dynamically changing the modules that are loaded into Apache, though to our knowledge they do not address the dependency issues we discovered in trying to accomplish this. A third management environment, Comanche [32], was unavailable to us except in source form at time of writing, and we lack knowledge of its capabilities.

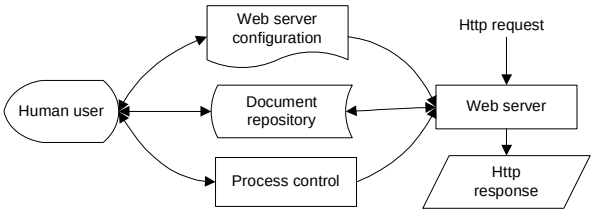

Figure 1: Typical interactions between a user/administrator and web server, where arrows indicate information flow.

HTTP servers are typically administered through direct access to configuration files and documents (Figure 1). One edits a server configuration file directly and then places content directly into directories that the server should expose to the outside world. The configuration file serves not only as a means of control, but also as documentation of defaults and other performance characteristics of the web server. Without fairly complete knowledge of the contents of this configuration file, it can be difficult to publish content. Since the document repository is edited directly, users must also have a good grasp of file protections within the server environment. Current approaches to this problem include simplified graphical user interfaces that expose only part of the capabilities of the configuration file [9, 32].

The configuration of the Apache web server is described by the contents of several files (Figure 2), where arrows indicate couplings between data in differing files or structures. In order for the server to respond correctly to a request, several parts of the configuration must agree in intent. For example, in the figure, to answer the request for URI http://www.foo.edu/, it must be true that:

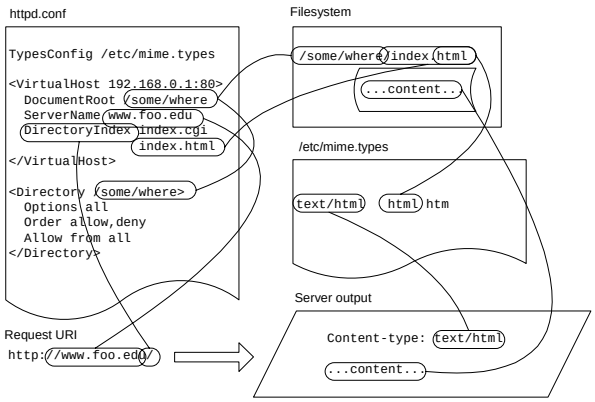

Figure 2: Configuration constraints required in order to answer a request properly on an Apache web server, where lines indicate required agreements between data values.

1. www.foo.edu is a valid virtual domain.
2. www.foo.edu maps to the server's address.
3. www.foo.edu's content is stored in the directory /some/where.
4. It is permissible to publish the content of the directory /some/where.
5. The request is for a directory rather than a file.
6. In returning data for a directory, one first checks for an "index file" that represents directory content.
7. index.html is an appropriate file.
8. index.html exists in the directory /some/where.
9. index.html is readable to the web server.
10. index.html has MIME type text/html because of the .html extension.

If any of these assertions is not true, the request fails. In the figure, many of these constraints are indicated by lines between configuration data that must agree in value.

The net effect of this scheme of constraints is that an Apache server can be quite difficult for a novice to administrate. There are several reasons for this:

1. The effect of a particular declaration depends on other declarations; one needs to understand the global configuration in order to understand the effect of a local declaration.
2. The configuration language – in an attempt to be easy to type – is filled with seemingly convenient defaults that make a configuration file difficult to interpret.
3. Often several distributed declarations determine whether content is provided correctly. For example, in order to serve a directory, one must

specify its protections as a directory, its mapping as a URL, and its MIME type mapping (if different from the default).

To simplify this process, we took direct control of content directories and configuration file away from the user. These are instead controlled by an intervening layer that mediates between user and server (Figure 3).[1] The user interacts directly only with an *image* of the document repository and a command interpreter. This interpreter keeps track of its state and maintains a private document repository of its own to which a user does not have access. User commands cause copying from the user's space into the closure's space in order to publish a document.

It might seem that we have just made the process of publishing more difficult, but in fact we have made it much less error-prone. At several steps during the publishing process, validity checks are made to the document and configuration requests. These checks include:

1. Does the name of each virtual domain correctly map to a valid interface via DNS?
2. Does each file's MIME type roughly agree with its content, as indicated by file magic numbering?
3. Do HTML files contain correct HTML?

Once a document passes these validity checks, it is published reliably, because the closure will take care of placing it in a proper location and protecting it so that the web server can see it. Also, a validate command checks that the web server configuration and all content have not been edited by unauthorized people, by comparing cached MD5 checksums against checksums of current data.

This closure consists of three components:

1. A *setup script* that initializes the closure on a prebuilt server.
2. An *agent* that interprets the command language and makes changes in configuration over time.
3. A *command language* for describing changes to make in service.

[1]Some authors would call this "middleware." "Middleware" is perhaps one of the most abused terms in the modern computing lexicon. As we provide not an API but instead a message-passing interface, the term "middleware" does not seem to accurately apply.

We chose to place our closure around an Apache web server running inside RedHat Linux 9.0. We assumed that the underlying system would be newly built and functioning on the network prior to invocation of the closure. In an actual closure, all systems with which the closure will interact must also be closures, in order to maintain overall integrity. Since this was our very first closure (which, in hindsight, was probably too ambitious), we had to settle for interacting with an already functioning system.

The initial "build script" determines a few aspects of pre-existing system configuration, such as where the Apache web server is located, whether certain applications are installed on the system, and other vital data necessary in order to construct the closure. The script then creates a managed structure on the disk that has restricted permissions. This structure contains a startup/shutdown script for the web server, a document root, a (private) space for storing closure logs and data files, and a default configuration. Data files include definitions of virtual domains, access rights for users and directories, MD5 checksums of configuration and submitted files, and boilerplate configuration segments describing defaults. Internal data is stored in the Perl Data::Storable format.

After the build script does its work, a command interpreter takes commands and modifies the resulting service automatically. This interpreter is a perl script that acts as a command line interface. This interface is responsible for all ongoing management of the HTTP service. In order to make the closure easy to use for both system administrators and end users (who may not be familiar with computer-related concepts), we decided to create a very limited command set that should be able to provide all the functionality necessary for a typical multiple-domain server.

Most of this process is straightforward; difficulties arose mainly in designing an appropriate command language with which to converse with the closure. Coming up with an appropriate language took considerable thought and required nontrivial changes in the way we think about server configuration as a process.

Command Language 1.X

Our initial command language was patterned after the structure of Apache's configuration file

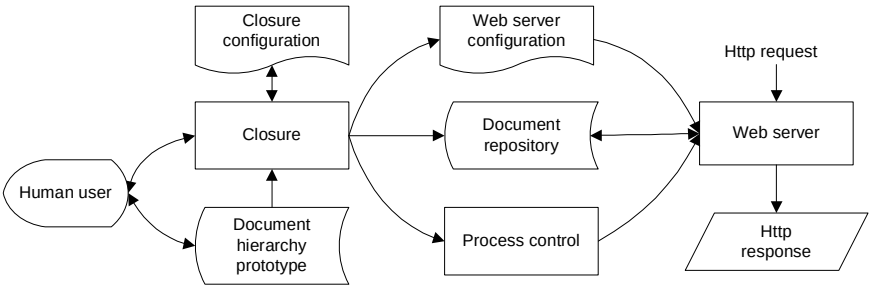

Figure 3: A closure mediates between the user and the Apache configuration, reducing complexity of the configuration process.

httpd.conf. We decided upon a minimal language with typical commands like the following:

```
assert foo.bar.com
```

declares a virtual host and readies it to serve content.

```
retract foo.bar.com
```

removes a virtual host from the server, along with all content that it provides.

```
post /home/prod
        http://foo.bar.com/products
```

makes the directory /home/prod on the current machine appear to be the web directory http://foo.bar.com/products .

```
post /home/couch/foo.html
        http://foo.bar.com/foo.html
```

makes the file /home/couch/foo.html appear as the URL http://foo.bar.com/foo.html. Each URL is associated with a unique directory in a private managed space, and each posted file or directory content is copied there to isolate it from further changes by the user. The type of the first argument – file or directory – determines what post does. In the case of a directory, post copies all subdirectories recursively.

```
retract http://foo.bar.com/products
```

removes any association between that URL and a content directory. It erases content previously associated with the URL via a post command.

```
retract http://foo.bar.com/foo.html
```

removes the mapping that results in content for the above URL. As described in [17], a web server is a mapping between URLs and content; the user need only specify that mapping and the closure takes over to assure it.

Ambiguity

At this point, progress on the project ground to a halt due to seemingly insurmountable difficulties within the command language. *Ambiguity* arose in the command language as a direct effect of *defaults* in the underlying httpd.conf, as well as default behavior for the corresponding HTTP protocol. This ambiguity has several effects:

1. The causal effect of many commands is unclear or determined by context.
2. One needs to understand the history of all commands in effect to know exactly what a single command will do.
3. It is possible for one command to override *part* of another, so that the resulting state cannot be reached via any other sequence.

As the simplest example, consider:

```
post /home/foo.html to
             http://www.foo.com
```

Should this make /home/foo.html available as http://www.foo.com/foo.html or as http://www.foo.com? In the latter case, should the file /home/foo.html be renamed to index.html or left alone in the directory? Then consider:

```
post /home/foo to http://www.foo.com
```

If /home/foo is a file, this has a potentially different effect than if /home/foo is a directory. But we cannot know which it is from the command itself. Finally, consider:

```
post /home/index.html to
      http://www.foo.com/index.html
post /home/index.cgi to
       http://www.foo.com/index.cgi
```

Which of these will be the directory index? The result of answering these questions in any reasonable way was that the effects of the seemingly simple command language were hideously complex to *document* and *understand*.

Creating content for a web server requires complete knowledge of its conventions, including which filenames have special meanings or interpretations. These defaults are set in httpd.conf. Without complete knowledge of the defaults, the user cannot create content properly. By abstracting the defaults into a command language rather than a file, we made the defaults invisible, and thus rendered the problem more difficult than before. We concluded that we had not simplified the problem of managing httpd.conf; we had actually made the management process more difficult than before!

Appropriate Closure Language

The key to these quandaries was to carefully describe desirable attributes of the command language and then redesign to these requirements. But we did not understand the optimal properties of such a command language, and only had the httpd.conf format as an example. Its properties include:

1. *Minimization of ink:* all that is unspecified has (reasonable) defaults.
2. *Hierarchy:* everything is laid out in a carefully designed multi-level hierarchy.
3. *Scoping:* the intent of commands depends upon the context in which they are entered.
4. *Ordering:* ordering of certain commands, including protections, changes intent within the configuration file.

These properties ease the loop of interacting directly with the configuration file, but do *not* ease the process of incrementally describing a configuration through individual and atomic commands. The design of httpd.conf presumes that the administrator has global knowledge of the contents of the whole configuration file. Our closure users, operating from outside the closure, have no such knowledge.

In effect, the syntax of httpd.conf had corrupted our thinking. Used to being able to look at the whole file to answer questions, we presumed that our commands could be patterned after the edits we make to the file and the file copying that we would do without the closure in place. This patterning made configuration more difficult rather than easier. In fact, the exact conveniences and defaults – that make httpd.conf easy to use when one is editing it directly – make a transactional language difficult to use.

To be easy to use, our command language has to have several somewhat different properties:

1. *Clarity*: the intent of a command should be immediately clear from its form.
2. *Independence*: to the extent possible, commands should be independent of one another to avoid conflicts, ambiguities, and difficulties in determining effects. In particular, global defaults should not be subject to change.
3. *Declarative syntax*: Each command represents a state to preserve, rather than an action to perform.
   a. Commands should be *idempotent*, i.e., repeating a command twice in a row should have the same behavioral effect as doing it once.
   b. Commands should be *stateless*, i.e., the behavioral effect of doing a command should not depend upon prior executions of the same command, even if these executions occurred in the remote past. A stateless command is always idempotent; statelessness is a stronger condition.

   Reducibility to assertions: Either an assertion is in effect or not; there is no such thing as being "1/2 in effect." A command that conflicts with a previous command undoes ("retracts") the effect of the conflicting command. Equivalently, any sequence of assertions and retractions is equivalent with a subsequence consisting of assertions alone.
4. *Representability*: at any time, one should be able to get an idea of all commands currently in effect, to understand global contents. Ideally, the representation should be a conflict-free description of the current state of the service, in terms of the unordered list of commands currently in effect. In particular, the representation of service should be free of retractions.

These properties actually arise from mathematical models that we will describe later. For now, it suffices to mention that statelessness and reducibility to assertions imply representability. The remainder of the requirements, clarity and independence, contribute to ease of use.

Statelessness means that a command does not depend upon prior invocations of itself to do its work. Stateless commands cannot be incremental in nature, but must deal with absolute quantities. For example, incrementing a counter is not a stateless command. The reason that statelessness is important is that the user may not have knowledge of prior commands or pre-existing configuration. While a stateful command may have indeterminate results, a stateless command has (roughly) the same effect regardless of when it is executed. The user need not remember anything in order to know what its effect is.

Representability means that at any time, one can describe the closure via the commands that are currently in effect, which is typically a smaller list than the whole sequence of commands since the closure was created. Reducibility to assertions means in addition that the commands currently in effect do *not* have to contain retract statements, because conflicts cause conflicting assertions to completely disappear from a representation.

Command Language 2.X and Beyond

These considerations caused subtle but profound changes in our command language that both resolve ambiguities and make it easier to use than making manual changes to configuration files and web content. We are currently in the process of implementing these changes.

1. Commands either *augment* or *retract* the effects of other commands. The effect of issuing a conflicting command is to retract the commands with which it conflicts. To assure this, we deprecated using post for files (except for indexes), and required its use on directories, where it recursively applies to subdirectories. In version 1, retracting a directory does not retract its subdirectories; in version 2, subdirectories are retracted as well.
2. When overriding the MIME type for a specific file, the command does not affect other files with the same extension. In version 1, a MIME-type override applied to all of the files of that type in a folder. This caused confusing changes in default MIME-types when new files were uploaded. To assure clarity of intent, the override is now specific to each single file.

Other profound changes are yet to be implemented. The above ideals for language imply that the indexing process for a directory should be *independent of its content.* One should be able to specify an index for an empty directory, or have a directory with no index. In the latter case, one gets a "permission denied" error instead of a directory listing. This effect is accomplished by adding the special keyword index to a post command for the index file.

Thus we plan to change the *concept* of index from being a file with the word index as its name, to being a file with *any* name that just happens to be bound to a directory as a listing operation. This index file can have any name, and be of any file type. This change provides a significant increase in flexibility over the old style, while disambiguating the most frustrating of problems when faced with the problem of insuring consistent operations. If no index file is selected, then either an error page is returned by default, or the security on the system can be made more lax and allow one to display the contents of the requested directory.

Critique

Our current closure does its intended job well, but there are many shortcomings. While several are simply new features to be implemented, some require a relatively deep rethinking of how we interact with systems that provide web content.

First, while the intent of a configuration can be represented by a list of commands, repeating the commands will *not* produce the exact same HTTP service. There is no guarantee that the source directories have not changed in the meantime. Repeating the same set of commands that created the service will instead result in an *updated* service based upon changes in the source directories. Indeed, what we seem to have implemented is the opposite of a content-staging system such as DryDock; we included no protection against inadvertent changes of the *source* repository.

Many issues for dynamic content have yet to be resolved. Various programs, such as Gallery [39], create and/or modify files within the web hierarchy by themselves, which constitutes performing operations outside of the closure. This kind of interaction is not supported in our model, and though it is tolerated, perhaps should not be allowed at all.

The closure deals very poorly with dynamic content stored within the document repository. Repeating commands used to set up a repository will erase any dynamic content created in the meantime within the repository by the action of CGIs. This content is not even *accessible* to the user unless exposed by the web server, and any files posted via the closure will be automatically made read-only.

It could be argued that this fascism is not a bug, but a *feature*; it strictly enforces utilizing external databases to store dynamic content rather than local server files. This is indeed the "best practice" for managing dynamic content, according to many web programmers.

In fact, we can find no reasonable solution to supporting this behavior of CGIs. If we allow dynamic content in document directories, it must be protected from subsequent post commands (that, in normal operation, will delete that content). But if we ignore such files, then the effect of post is not stateless.

Ideally, CGIs should use external data sources for dynamic content. The couplings between CGIs and external data sources should be managed by the closure itself, though the best language for accomplishing this is unknown. Another rather deep question is how to handle enforcing integrity constraints for data sources outside the closure, such as databases utilized by CGI scripts. If we do force all programmers to utilize databases, how do we assure that their scripts bind to functional and allowed data sources? While the Microsoft .NET framework makes this kind of checking easier by separating data source bindings from programs, no such solution exists for Linux and Apache.

Finally, this closure was our very first closure, and thus had no other closures with which to converse (unlike the ideal web service closure described in [17]). Thus our closure must check for external dependencies itself, and cannot correct deficiencies that it finds in its environment. For example, the assert command checks whether the domain being asserted points to this machine in name service, but cannot assure that by modifying the name server. Instead, it must refuse to perform the assert.

Future Work

Plenty of additional work needs to be done on this prototype before it is appropriate for production use. Among the most obvious extensions is to be able to handle ssl (HTTPS) traffic as well as HTTP. This will require improving how the closure deals with constraints. One tricky part of handling ssl, for example, is that one must enforce the constraint that only one virtual server bound to each address can have ssl capabilities. Currently, one must explicitly disable one ssl instance before asserting another.

Separating indexing from directory contents is a bit tricky, as Apache is designed to couple them together. Currently the closure simply uses whatever index file is present in each directory. Implementing the ideal indexing scheme – in which indexing is completely separate from directory contents – requires special care to avoid naming conflicts. This can be handled by storing the index in a name that will not be used otherwise (and cannot be easily entered as a URL), such as "-->index<--.__html__". To avoid confusion in choosing an extension matching the MIME type of the index file, this file is a simple CGI script that will read the real index content from a second cryptic filename "-->content<--.__html__", and display the contents, tagged with the appropriate MIME header.

There are also problems in dynamically managing the modules that Apache loads and requires. A true closure would need to analyze what the user desires from the web server and load the necessary modules by creating a dynamically generated list. We thought we could look into the directory where the modules are located and instruct Apache to load every one it finds as a starting step. We discovered quickly that certain modules are dependent upon others, that the order in which these modules are loaded is important, and that some modules conflict with and preclude the use of others. For example, mod_proxy_ftp.so requires mod_proxy.so to be loaded first or else loading will fail; likewise loading mod_dav_fs.so will fail if mod_dav.so is not loaded first. Using the current scheme for loading modules, there is no way to know in advance what dependencies, if any, a module actually has without trying to load it.

As a temporary solution, the list of modules to be loaded had to be made static, along with the order in which they are loaded. However, this means that the functionality of the closure is currently severely limited, as there are no commands to the closure that can expand the capabilities of the server.

Another ongoing issue is rights management. Currently, a user can only be granted rights to edit a

domain. In addition, we should be able to permit users to update sections of a domain without having rights to other sections. For example, we have started writing code that will allow couch to edit http://www.foo-bar.com/research, but restrict his access to other areas of the domain.

Other low level issues, such as insuring that there is adequate disk space for web content, need to be addressed, though this may need to be performed by talking to a disk closure that has yet to be written. Since varying devices have different access speeds, this could be a future concern, as some content may need to be delivered at a much higher quality of service (QoS) than other content, and one would want those pages to be stored on faster access devices.

Lastly, we would like the application to be able to handle not only a stand-alone web server; it should also be able to scale to a grid or cluster type configuration, which brings a whole new level of difficulties and questions, but ultimately a far more powerful and desirable closure.

Theoretical Background

While we were struggling with the implementation of the prototype closure, another struggle was going on at a different level. Clearly, our language evolved into something relatively useful from something relatively useless. But why do the above language principles work, and what mathematics underlies the design decisions we made? In this section, we explore some of the mathematical underpinnings of closure language design, and tie this work into other work on languages for configuration management. For the non-mathematically inclined, this section can be skipped without loss of continuity.

An overview of the results of this section is shown in Figure 4. Statelessness of individual commands leads to idempotence of sequences of commands. The ability to remove retractions of commands from a sequence and retain equivalent effect is called ''reducibility to assertions.'' This property, in combination with a semantic model that determines preferred order of operations, makes a sequence of commands declarative in character. Statelessness of commands, reducibility to declarations, and a one-one correspondence between configurations and behaviors give rise to Cfengine-like convergence of the declarations, thought of as an operator upon configuration.

There is currently much controversy about whether host configuration languages should be imperative [24, 34, 35] or declarative [1, 4, 5, 6, 7, 10, 22, 23, 26, 27, 33]. A subset of a language is ''imperative'' when it describes procedure or process: ''what should be done'' as an interpretable set of instructions. A subset of a language is ''declarative'' when it describes ''what the result should be'' without specifying the method or procedure with which this result is accomplished. For example, saying ''the car must be blue'' is a declarative statement, while ''paint the car blue'' is an imperative procedure for assuring the truth of the declarative statement. A particular language can exhibit both properties, specifying some things imperatively and others declaratively.

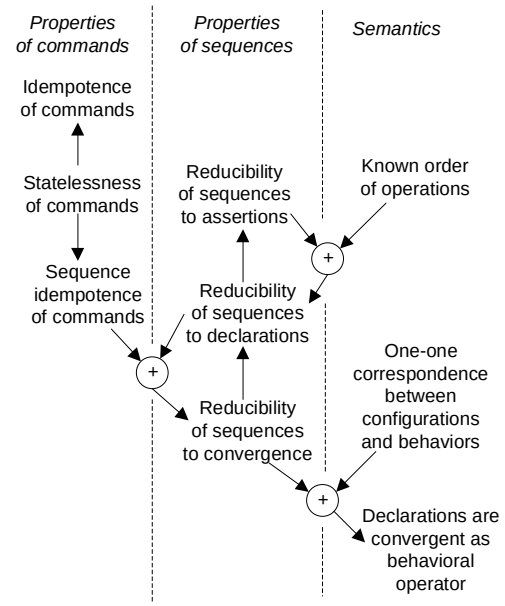

Figure 4: Map of theoretical concepts and their relationships, where arrows indicate logical implication.

Tools support and encourage either imperative or declarative thinking. Proponents of the ''imperative'' tools point out that specifications for these tools are close to the way a human would manually configure a system, while converting human instructions to declarative language requires some reverse-engineering [24]. Proponents of the ''declarative'' tools point out that mechanism is not important; one should specify results, not mechanism, and specifying anything more limits flexible response of the configuration management tool to changing requirements [1, 17]. In practice, a majority of seemingly declarative languages for configuration management allow the direct execution of imperative code as an option when declarative mechanisms fail to be expressive enough [4, 5, 6, 7, 11, 13, 22, 23, 33].

While imperative and declarative mechanisms both work well for creating an initial configuration, neither imperative nor declarative mechanisms proves sufficient to implement a closure as described in [17]. In both paradigms, there are serious problems in dealing with *changes in intent over time*. Imperative management mechanisms suffer from script complexity; it is difficult to make changes in these ''build scripts'' without mistakes [13]. Likewise, undisciplined changes to declarative specifications can lead to unintentional heterogeneity within large networks [17].

Shortcomings of Imperative Scripts

Imperative mechanisms substitute order and reproducibility for understanding of internal dependencies. The script that builds a host is constructed through intensive validation of its behavioral effects, but what the script actually does typically remains poorly understood. This leads to a change control problem as the script is reused over a long lifecycle. Due to lack of understanding of internal dependencies, such scripts can only be safely modified by adding stanzas to the end [24]. Re-imaging a host requires cycling through all of its historical states, including all errors in configuration made by previous scripts. The only alternative is to start over from scratch and validate a new script from the beginning.

The reason that ''order matters'' within the imperative paradigm is that the implicit preconditions for each stanza of a script are that all prior stanzas have been executed in order. ISConf enforces this order for hosts that miss an update by resuming stanza execution exactly where the host last stopped executing them, executing missed stanzas before new ones. In this way, the host passes through a sequence of reproducible states, so that a final desirable state is assured. If ISConf allowed hosts to skip stanzas, scripts would break unpredictably, because the scripts' preconditions would not be assured on hosts on which stanzas were skipped.

Shortcomings of Declarative Languages

Further, careless use of declarative tools can lead to exactly the kind of unpredictable heterogeneity that the imperative tools like ISConf are designed to avoid. Consider configuration elements A, B, and C with initial values $A = a, B = b, C = c$ and configuration files (declarations) d_A, d_B, d_C. Suppose that

1. d_A sets $A = a'$ and leaves all else alone.
2. d_B sets $B = b'$ and leaves all else alone.
3. d_C sets $C = c'$ and leaves all else alone.

Suppose that at any time, a distinct subset of hosts is down (unreachable). At the end of applying d_A, d_B, d_C in sequence, there are now eight kinds of hosts in the network:

1. $A = a', B = b', C = c'$: Up during d_A, d_B, d_C.
2. $A = a, B = b', C = c'$: Down during d_A; up during d_B, d_C.
3. $A = a', B = b, C = c'$: Down during d_B; up during d_A, d_C.
4. $A = a, B = b, C = c'$: Down during d_A, d_B; up during d_C.
5. $A = a', B = b', C = c$: Down during d_C; up during d_A, d_B.
6. $A = a, B = b', C = c$: Down during d_A, d_C; up during d_B.
7. $A = a', B = b, C = c$: Down during d_B, d_C; up during d_A.
8. $A = a, B = b, C = c$: Down during d_A, d_B, d_C.

As time goes on, the unintentional heterogeneity gets worse, a factor of two at a time, every time a station is unavailable for an update.

Two Principles

The above observations can be summarized as two related principles of configuration management that apply to any such process:

> **Principle 1** Once one controls or manages a thing, one cannot forget over time that it is controlled or managed.[2]

For example, ''forgetting'' that A is managed above leads to a heterogeneous population of hosts with two differing values of A [18]. More generally,

> **Principle 2** The discipline with which one changes a declarative configuration file is as important to effective configuration management as the accuracy with which the file expresses intent.

Tools that generate the whole configuration for each host each time [1, 2, 10, 22, 23, 26, 27, 33] neatly avoid this problem, at the cost of being somewhat limited in scope and unable to handle large changes such as software subsystem installation and removal.

Transactional Languages

With these goals in mind, we defined a new kind of configuration language that has both imperative and declarative aspects.

> **Definition 1** A *transactional* configuration language is one in which configuration is expressed as a sequence of atomic (indivisible) changes in behavior from a given and known base state.

For purposes of analysis, a transactional language has at least two basic primitives, assert and retract.[3] The command

```
assert {behavior}
```

causes a behavior to be exhibited, while

```
retract {behavior}
```

causes the behavior to become absent. This choice of primitives is arbitrary but allows us to discuss several effects of transactional language easily. The transactions might as well be SQL queries into databases or even XQUERYs into XML.

A transactional language has somewhat of an imperative quality to it, because order sometimes matters, e.g., the order of assert and retract for the same behavior determines whether that behavior is present. The key to our argument and work is that it is also possible – by design – to give the transactional language a declarative flavor as well.

Reducibility to Assertions

At present, our language has no constraints; most any kind of assert and retract statements are allowed. Our next job is to make it possible to simplify complex command sequences.

[2] ''Be careful what you command, my son. A command, once given, must be repeated forever.'' – Duke Leto Atreides, Frank Herbert's *Dune*

[3] Any resemblance to the primitives with the same names in the programming language Prolog is purely intentional.

Definition 2 A transactional language $\mathcal{L}$ containing only assert and retract statements is *reducible to assertions* if for any sequence of assert and retract statements, there is a subsequence of assert statements alone that has the exact same behavioral effect.

Reducibility means that an assertion cannot apply "halfway." If there is a state in which a retraction cancels part of an assertion, then the assertion is "half right." In a reducible language, retractions cancel assertions either fully or not at all.

As an example of a non-reducible language, suppose that directory indexing is turned on by default and one must manually retract the behavior after asserting the contents of the directory. To make this language reducible to assertions, indexing instead must be off by default.

Reducibility has a subtle but important effect upon language. If a language is reducible, the results of any set of transactions can be expressed with "positive language": what should happen, without mention of what should not happen. Since retractions are order-dependent, but assertions typically are not, making a language reducible to assertions has the primary effect that one can express behavioral outcomes in largely order-independent fashion.

Reducibility to Declarations

The ability to eliminate retractions from a language is just one kind of reducibility:

Definition 3 Let $\mathcal{L}$ be a transactional language and let

$$\mathcal{P} = \{(s_a, s_b) \mid s_a, s_b \in \mathcal{L}\}$$

be a partial order on elements of $\mathcal{L}$, where $(s_a, s_b) \in \mathcal{P}$ exactly when s_a must precede s_b. Then $\mathcal{L}$ is *reducible to declarations* if for every sequence of transactions $(t_1, \ldots, t_n)$, there is a subset $\mathcal{D} = \{d_1, \ldots, d_k\} \subset \{t_1, \ldots, t_n\}$ of the set of transactions (where duplicates are eliminated), where for every total ordering $(e_1, \ldots, e_k)$ of $\mathcal{D}$ consistent with the partial order $\mathcal{P}$, the behavioral result of applying the sequence $(e_1, \ldots, e_k)$ is the same as that of applying the sequence $(t_1, \ldots, t_n)$.

This is a complex and perhaps overly wordy way of expressing a relatively simple idea. A transactional language is reducible to declarations if for every sequence of transactions in the language, there is a subset that does the same thing in any reasonable order in which it is applied. In writing down this subset, "order does not matter" because we already know the partial order $\mathcal{P}$ describing how to appropriately order execution of the particular transactions within the subset.

For example, suppose that we have the following transactions:

```
assert A
assert A.X
assert B
assert B.Y
retract B
```

and the partial order:

```
{ (assert A, assert A.X),
  (assert B, assert B.Y) }
```

meaning that one must create A or B before creating their substructures A.X or B.Y. Suppose that retract B retracts the substructure as well; this is allowed. Then we can write an equivalent set of operations as the set { assert A.X, assert A } where the order of this set is unimportant, because we know that assert A.X must follow assert A from the partial order. In our closure, the order constraints are that one must post the contents of parent directories before posting subdirectories; the effect is identical to that in this example.

Whether a language is "declarative" depends upon what we know about the elements of that language and their sequencing. If we are absolutely sure of the appropriate sequences, the order of writing down the elements does not matter; we can resort them into an appropriate order later. A transactional language is reducible to declarations if we can eliminate conflicts from the sequence of declarations so that order does not matter in the resulting reduced set.

Statelessness

While our task requires operations that are reducible to declarations, this is not quite enough:

Definition 4 A transaction or sequence of transactions p is *idempotent* if repeating p twice in succession has the same effect as executing p once.

In other words, once p is successful, doing p again does nothing. More generally,

Let $\mathcal{L}$ be a set of commands. $\mathcal{A}$ is *stateless* if for any command $p \in \mathcal{L}$ and any sequence $q_1, \ldots, q_n \in \mathcal{L}$, applying p followed by $q_1, \ldots, q_n$ followed by p has the same effect as the sequence $q_1, \ldots, q_n, p$. In other words, the initial execution of p before the sequence does not matter.

Statelessness trivially implies idempotence.

The reason statelessness is important is that it is related to idempotence of sequences:

Definition 5 A set of commands $\mathcal{L}$ is *sequence-idempotent* if for any sequence of commands $p_1, \ldots, p_n$ from $\mathcal{L}$, applying the sequence twice has the same effect as applying it once.

This is important because

Proposition 1 If $\mathcal{L}$ is stateless, then $\mathcal{A}$ is sequence-idempotent, i.e., the set of all sequences of commands taken from $\mathcal{L}$ is idempotent.

The proof of this is contained in [16]. This fact allows us to translate between descriptions of a configuration that are command-based and those that are instead declarative:

> **Definition 6** A language $\mathcal{L}$ is *reducible to convergence* if it is reducible to declarations and the declarations, used upon the closure, are idempotent as a sequence.

In other words, given any sequence $(t_1, \ldots, t_n), t_i \in \mathcal{L}$, there is a subset $d_1, \ldots, d_k$ such that if $(e_1, \ldots, e_k)$ is an ordering of $d_1, \ldots, d_k$ conforming to the partial order $\mathcal{P}$, applying the *sequence* $(e_1, \ldots, e_k)$ twice does nothing different than applying it once. In particular, applying this sequence to the configured closure does nothing at all, while applying it to an unconfigured closure reconstructs the closure's current state.

Thus we have the immediate result that:

> **Proposition 2** If $\mathcal{L}$ is both reducible to declarations and stateless, then $\mathcal{L}$ is reducible to convergence.

This is a trivial corollary to Proposition 1. Note that statelessness is a sufficient but not necessarily required condition for sequence idempotence, so that it is a sufficient but not necessarily required condition for being reducible to convergence.

Finally, we relate this to behavior of the overall closure:

> **Proposition 3** Suppose that there is a one-to-one correspondence between behaviors and configurations, and that $\mathcal{L}$ is a set of transactions that change configuration. Suppose that $\mathcal{L}$ is reducible to convergence, $(t_1, \ldots, t_n)$ is a sequence of operations in $\mathcal{A}$ reducible to the declarations $\{d_1, \ldots, d_k\}$, and $(e_1, \ldots, e_k)$ is one order of $d_1, \ldots, d_k$ compliant with the partial order $\mathcal{P}$. Then the operator $e_1 \cdots e_k$ formed by applying $e_1, \ldots, e_k$ in order is convergent in the sense of [4, 5, 6, 7], i.e., it is idempotent as a sequence and interchangeable with the sequence $t_1 \cdots t_n$ in assuring the same behaviors.

Proof: Given $(t_1, \ldots, t_n)$, we know that $\mathcal{L}$ is reducible to convergence, so that $(e_1, \ldots, e_k)$ exists by definition. We also know that from a behavioral standpoint, applying $e_1 \cdots e_k$ has the same *behavioral* effect as applying $t_1 \cdots t_n$. Since there is a one-to-one map between behavior and configuration, these also therefore assert the exact same configuration. Since $e_1 \cdots e_k$ is idempotent, it will not change that configuration. □

This is a bit tricky, as one must require a correspondence between behavior and configuration. In cases where gratuitous differences exist between configuration and behavior, we can still get into a state where $u_1 \cdots u_k$ is not idempotent while $t_1 \cdots t_n$ is. For example, consider the transactions:

```
t1 = { A:=B }
t2 = { B:=4 }
```

and suppose that the value of A affects behavior but the value of B does not. Due to this, $t_1 t_2$ is reducible to t_1, but applying the sequence $t_1 t_2 t_1$ results in differing behavior than applying $(t_1 t_2)$ alone. There are two solutions to this: either disallow use of non-behavioral values in transactions [18], or limit one's self to a definition of configuration in which non-behavioral values do not appear.

Impact of Statelessness and Reducibility

The purport of the above mathematical discussion is that if the commands in a language are stateless, then reducibility to convergence implies reproducibility of effect, i.e., the reduction suffices as a declaration of state. There are several benefits of having a configuration language with these properties:

1. At all times, it is possible to express the effect of a sequence of commands in the same language that the commands use themselves. This eliminates the problem of "semantic distance" [13] in which the language used to declare state differs substantively from the language used to assure it.
2. This effect is expressed in terms of positive and non-conflicting assertions.
3. In a recovery situation, these assertions suffice as commands to reproduce current state.

These are desirable properties for any language, but statelessness would seem a very strong condition upon a language. What kinds of languages have we eliminated from consideration?

A stateless language is simply one in which all assertions are made with absolute (constant) values for parameters (or, at least, parameters that can be converted unambiguously to absolute form, such as relative pathnames). A stateless language cannot allow incrementing or decrementing a configuration parameter, or base one parameter's value upon that of another. This is a stateful (and non-idempotent) change by nature (i.e., $pp \neq p$).

More subtle, the identity of the parameter that gets set by an operation p cannot be a function of some other setting. Suppose we have parameters A, B, C, and that the operation p sets B to 1 if A is 0, and C to 1 otherwise. Suppose that A is initially 0 and the operation q is $A{:=}1$. Then pqp has a different effect $(A = 1, B = 1, C = 1)$ than qp $(A = 1, B = 0, C = 1)$, violating statelessness.

Stateless Transactions and XML

It would seem that reducibility to assertions and statelessness impose rather extreme limits on what a command language can do. An immediate question about stateless transactions is whether one can create a set of representable (reducible and stateless) transactions that can maintain any kind of configuration file. Most configuration files are hierarchical in structure, and any reasonably consistent hierarchical structure can be expressed by an XML document type definition

(DTD), so it suffices to show that one can correctly maintain the contents of XML files via stateless and reducible transactions.

The allowable forms of an XML document are described by its Document Type Definition (DTD). The DTD describes the allowable contents of each kind of XML node by a *DTD rule*. This rule expresses the structure of allowable content for the node as a regular expression in which tokens are node labels. There are three constructions one can use to make a DTD rule:

1. Sequencing: the expression ''A,B,C'' matches a sequence of nodes: a node named A followed by a node named B followed by a node named C.
2. Alternation: ''A|B|C'' matches exactly one of A, B, or C.
3. Repetition: ''A*'' matches zero or more copies of a node named A in sequence. This is the ''Kleene star'' operator.

More complex specifications are regular expressions containing the above operators, e.g., one can say that a node named A contains either a node named B or a sequence of nodes named C followed by a node named D by describing A's content via the regular expression pattern ''B|(C*,D)''. This is described in a DTD by the rule

```
<!ELEMENT A (B|(C*,D))>
```

This rule allows XML such as

```
<A>
<B>...</B>
</A>
```

and

```
<A>
<C>...</C>
<C>...</C>
<D>...</D>
</A>
```

but disallows XML such as

```
<A>
<C>...</C>
</A>
```

(because C cannot appear alone inside A in the above rule). Here ... in the text represents content conforming to (yet to be described) rules for the content of C and D. Other DTD constructions, including + for ''one or more instances,'' are expressible using these constructions: A+ is just ''A,A*''.

Without loss of generality and to ease notation, we do not consider element ATTRIBUTE declarations in XML. These are easily modeled as subordinate elements of the element to which they apply.

Our initial try at defining XML transactions will be based upon the XPATH language for identifying nodesets within XML files. Every XML file contains nodes that contain other nodes as content. In the file

```
<foo>
  <bar>
    <goo>1</goo>
    <cat>3</cat>
    <goo>4</goo>
  </bar>
</foo>
```

there are five nodes, including one foo, one bar, two goos, and one cat. A *nodeset* within an XML file is a set of nodes within the file having common attributes. XPATH is a language for identifying nodesets, using a notation similar to that used to identify files within a filesystem. For example, the XPATH /foo/bar refers to all nodes named bar within a top-level node named foo, while /foo/bar[2] refers to the second such node named bar in sequence. The special component * refers to a component with any name; /foo/*[5] refers to the fifth component of the content of foo with any name. While this simple subset of XPATH suffices for our discussion, there are many other options too numerous to cover here. All that we need to remember for now is that an XPATH determines a set of nodes within the document for which the assertion controls resulting content. This set of nodes is uniquely determined by the current content of an XML document and an XPATH, and may be empty.

The general form of an XML transaction is:

```
assert <nodeset> <content>
```

where <nodeset> specifies a set of nodes in the file to transform (in XPATH notation) and <content> is XML content that should replace any existing content in the nodeset. <content> can be empty, in which case the assertion has the effect of retracting content from the node. The assertion succeeds if the requested transaction is possible (the nodeset defined by the XPATH <nodeset> is non-empty) and the resulting transformed XML document conforms to the document's DTD; otherwise it fails and does nothing to the document at all. Because DTDs describe the content allowable for each node, and because each assertion provides that content, either all replacements are legal or all replacements fail, together.

For example, in the file

```
<foo>
  <bar>
    <goo>1</goo>
    <cat>3</cat>
    <goo>4</goo>
  </bar>
</foo>
```

performing the command

```
assert /foo/bar/goo[2] <ho>5</ho>
```

would result in the document

```
<foo>
  <bar>
    <goo>1</goo>
    <cat>3</cat>
    <goo><ho>5</ho></goo>
```

```
  </bar>
</foo>
```

(provided that the resulting document is acceptable according to the DTD for the document). The XPATH /foo/bar/goo[2] matches the second instance of goo inside an instance of bar inside an instance of foo. The XPATH /foo/bar/goo would match *both* instances, so that the assertion

```
assert /foo/bar/goo <ho>5</ho>
```

will produce the document

```
<foo>
  <bar>
    <goo><ho>5</ho></goo>
    <cat>3</cat>
    <goo><ho>5</ho></goo>
  </bar>
</foo>
```

Since one can re-assert the contents of the top-level node (via assert /), every state of the XML file is reachable via such assertions. Also, such assertions are idempotent; either an assertion succeeds (if its *total* effect conforms to the document's DTD) or it does not succeed and does nothing to the document. If it does or does not succeed, repeating it immediately has the exact same effect (because the whole assertion is variable-free).

It is a bit more difficult to see that

Proposition 4 The set of all possible assertions $\mathcal{A}$ of the form

```
assert <nodeset> <content>
```

is stateless and reducible to assertions.

Proof: Consider a transaction $T \in \mathcal{A}$ and a sequence S of transactions in $\mathcal{A}$. Note that all content of T is constant; there are no variables that can change state within the transaction itself. Note also that no transaction in $\mathcal{A}$ can change the number of elements in the content of an element unless it also asserts all of the content of that element.

Consider what happens when one applies *TST* in sequence. Either the nodeset determined by the XPATH in T changes or it stays the same. If it stays the same, then T has the same effect by definition, so that the first T need not be executed and T is stateless. If the nodeset changes, however, it must have changed as a result of assertions that change the whole content of nodes. These assertions must override the prior values of T whenever they affect its nodeset. So in either case, T is stateless. As T and S were arbitrary, the whole language $\mathcal{A}$ is stateless.

$\mathcal{A}$ is trivially reducible to assertions, as it has no retract statements at all! □

Statelessness and Semantics

So far, we have a very awkward system for editing XML. It would be convenient to add two new primitives

```
add <nodeset> <content>
```

and

```
subtract <nodeset> <content>
```

that augment and remove sequenced content from a node. add puts new content into a sequence, while subtract removes matching content from a sequence. The success of both operations is again dependent upon conformance between the result and the document's DTD.

Without further constraints, we no longer have statelessness. To have statelessness, we must also have idempotence, but the add primitive is not even idempotent; adding something twice results in two entries for the item instead of one. For example, consider the XML document

```
<foo>
  <bar>
    <goo>10</goo>
    <goo>16</goo>
  </bar>
</foo>
```

and the effect of two statements:

```
add /foo/bar <goo>20</goo>
add /foo/bar <goo>20</goo>
```

After these statements (and with no further constraints) the resulting document would contain:

```
<foo>
  <bar>
    <goo>10</goo>
    <goo>16</goo>
    <goo>20</goo>
    <goo>20</goo>
  </bar>
</foo>
```

instead of

```
<foo>
  <bar>
    <goo>10</goo>
    <goo>16</goo>
    <goo>20</goo>
  </bar>
</foo>
```

(which would be the required result for a stateless add operation).

The DTD for an XML document is syntactic; it describes what can be written but not what the written text means. What is missing from our model is a notion of *semantics* that would tell us when two configurations are equivalent. A model of semantics, in turn, allows one to understand what add and subtract should do in order to remain stateless.

Note that idempotence and statelessness are not properties of what operations do, but of *what the results mean*. If operations act on a configuration file to produce the same contents, it is rather obvious that they result in the same behavior. However, sequences of operations that produce differing configuration files may produce the same behavior, e.g., if the differences in configuration do not produce differences in behavior.

In adding information to a sequence, one must ask several questions. Does order of the sequence matter or not? Do duplicates matter, or does the first or last instance of a duplicate override the others? What constitutes a duplicate entry? These are *semantic* questions that go beyond the simple *syntax* described by a DTD.

Preserving statelessness in using substructure addition and subtraction is a matter of both understanding semantics and limiting operations to fit. If members of a sequence are pure declarations, so that order does not matter and duplicates are ignored, then add and subtract should behave accordingly. Trying to add a duplicate should have no effect.

More subtle, the semantic definition of a duplicate often involves the notion of a unique key. For example, suppose that in Apache we are defining access rights to directories. Obviously, the directory to which we are defining access constitutes a unique key; we should not be able to define two different ideas of protection for one directory.

This means that we need several context-sensitive notions of what add and subtract should do.

1. If order of assertions matters and cannot be inferred from assertion content, the game is over. Language is imperative and statelessness is impossible.
2. If order of assertions does not matter and duplicates can occur, then add cannot be stateless, because it is not even idempotent.
3. If order of assertions does not matter and duplicates should not occur, the add command should have the form

   ```
   add <nodeset> <key> <content>
   ```

   where <nodeset> determines a set of nodes on which to operate, <key> is an XPATH *relative* to each node that determines a key that should be unique, and <content> is content to add. For example, given the XML

   ```
   <foo>
     <bar>
       <goo>10</goo>
       <goo>16</goo>
     </bar>
   </foo>
   ```

   the command

   ```
   add /foo/bar goo <goo>10</goo>
   ```

   does nothing at all, because 10 is already a key, while

   ```
   add /foo/bar goo <goo>20</goo>
   ```

   results in the document

   ```
   <foo>
     <bar>
       <goo>10</goo>
       <goo>16</goo>
       <goo>20</goo>
     </bar>
   </foo>
   ```

   The extra goo in the assertion indicates that the *content* of goo is the key that should be unique.

The moral of this section is that unless the XML being created is truly a declaration (i.e., order does not matter and duplicates should not exist), statelessness is impossible in the transactional language that updates it.

Application to Configuration Management

The impact of this mathematical theory upon the general problem of configuration management is subtle but inescapable. So far, we have largely ignored the problem of change management in host configuration management. Generative tools (that create an entire configuration from templates) largely avoid the issue of change management by erasing everything and starting over each time. However, these tools are relatively limited in scope, as they cannot handle major changes such as package management: installation and removal of software subsystems. Convergent tools allow one to become sloppy and forget that a component is managed, while imperative tools deal poorly with undoing changes.

In the sub-problem of managing the configuration of a web server, change management is a central concern, so that we must adjust our practice and language to ease that task. The result, however, is that we created a framework for change management that applies to the more general problem of network configuration management. In fact, we have created an ‘‘assembly language’’ that is the lowest level of a new strategy for configuration management.

At its core, every configuration can be described in terms of a set of assertions that are true at a given time. In the simplest case, each assertion assigns values to one or more ‘‘configuration parameters.’’ Since configuration values are always specified as absolute quantities in assertions, such assertions are naturally stateless. Most configuration management tools are driven by configuration files containing only stateless assertions of this kind.

By viewing the assertions in such a configuration file as commands to be executed, the only thing we have added in our model is a concept of retraction of assertions. There is a constraint model that describes which assertions conflict with which others, and a rule that keeps current values consistent with one another, e.g., for our web server, we know that asserting a new index file for a directory is going to override the old index declaration. If we retract a virtual server, then all parameters for that server no longer exist.

If the command language is reducible to assertions, no matter what incremental changes we make to the overall configuration through further assertions, the results remain precisely expressible as a set of assertions. Further, the assertions are sequence-idempotent, so that repeating them has the same effect as doing them once. Thus the list of valid assertions is a

good substitute for the policy file found in many configuration management systems.

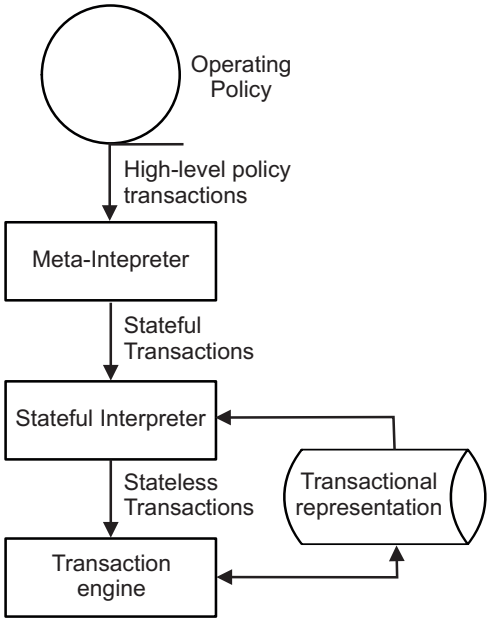

Figure 5: A new model of configuration management, in which low-level statements exhibit statelessness while upper levels encapsulate stateful behavior.

This gives rise to a new model of configuration management (see Figure 5). At the core, a transaction engine interprets a stateless language. This engine interprets stateless commands to effect requested changes in overall configuration. This engine is responsible for maintaining the list of valid assertions and retracting conflicting assertions. This engine deals with stateless commands only.

At the next level, stateful commands are translated into stateless commands for ease of use. While the user thinks in relative terms (“more space”), the transactional system must think in terms of absolute requirements (“2 MB”). A simple memory mechanism here makes the translation between relative and absolute units.

At the third level, meta-commands describing overall intent are translated into the assertions that create that intent. “Become a web server” is translated into the various assertions that cause that to happen.

This rather strange way of accomplishing configuration management has a few rather obvious advantages:

1. The whole history of the configuration of the machine is contained in one transaction stream.
2. At any time, there is a deterministic procedure that can determine what configuration is in effect at that time.
3. Storing the stream and its changes allows one to roll back time, by replaying the stream or selectively retracting the newest assertions, backwards.
4. One can specify changes as incremental operations upon a pre-existing structure.

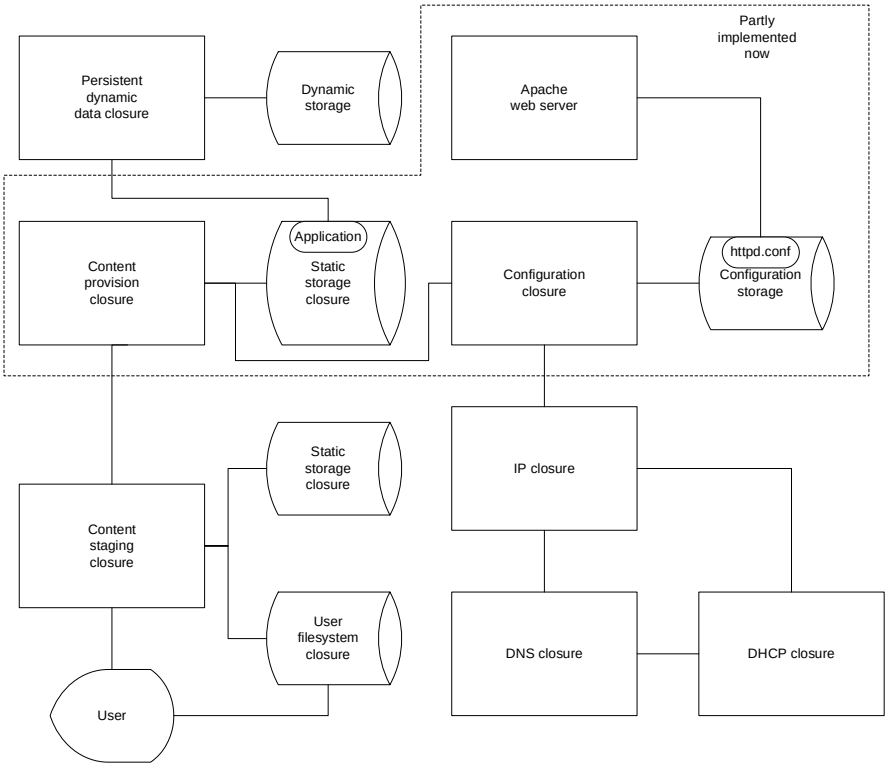

Figure 6: Parts of a complete HTTP closure, where the dotted box indicates completed prototypes.

5. Changes can come from multiple sources (e.g., different administrators or groups) and will be disambiguated at implementation time.
6. At any time, there is a coherent picture of the assertions currently in effect.

These observations are *not* true in general of current convergent administrative tools, including Cfengine. In Cfengine, changes are specified by editing a monolithic file. There is no easy way to undo a configuration step. It is difficult for multiple people to collaborate on a single configuration without conflicts. In order to implement such a mechanism, Cfengine would have to have the ability to return a file to the state before any edits have been applied.

This proposal needs much study before we implement such a language, but is clearly implied by our study of HTTP.

Conclusions

Work on this project has been a long road of discovery. Ad-hoc creation of a closure language – based upon a configuration language – led to much initial confusion. The exact things that make a configuration file easy to read make a command language confusing and difficult to use. Disambiguating that language required a mathematical approach: stateless commands removed the quandaries posed by stateful syntax. The result proves that for a limited problem domain, closures exist.

But we are a very long way from coming to a complete closure. A complete solution seems to have more parts than we could have imagined initially(Figure 6).

1. To assure idempotence of operations, we need content staging, and an independent repository for staged data. There should be two content hierarchies, one for actual provision and the other a cached copy that allows restoring the provided data, e.g., after a crash.
2. It is impossible to deal with dynamic content written into the web hierarchy by traditional means. This must be handled by some kind of dynamic storage closure (that may be a database, or perhaps something else).
3. We had much difficulty verifying that a declared virtual server would work properly according to information in DNS and DHCP. We need the ability to converse with and negotiate with DNS and DHCP closures in order to determine whether declared virtual servers will work properly.

As well, the actual configuration closure should have several parts that are not currently present(Figure 7). A module management subsystem should allow dynamic selection of modules, while a constraint engine disallows impractical choices. Likewise, a virtual server management subsystem should disallow virtual server configurations that cannot work, e.g., declaring more than one ssl server for a single IP address.

We also acknowledge that the end product may not be a single http service closure, but several differing ones for different applications. Making the language simple enough to use in one application may preclude its use in another. For example, a closure whose language is simple enough for use by untrained

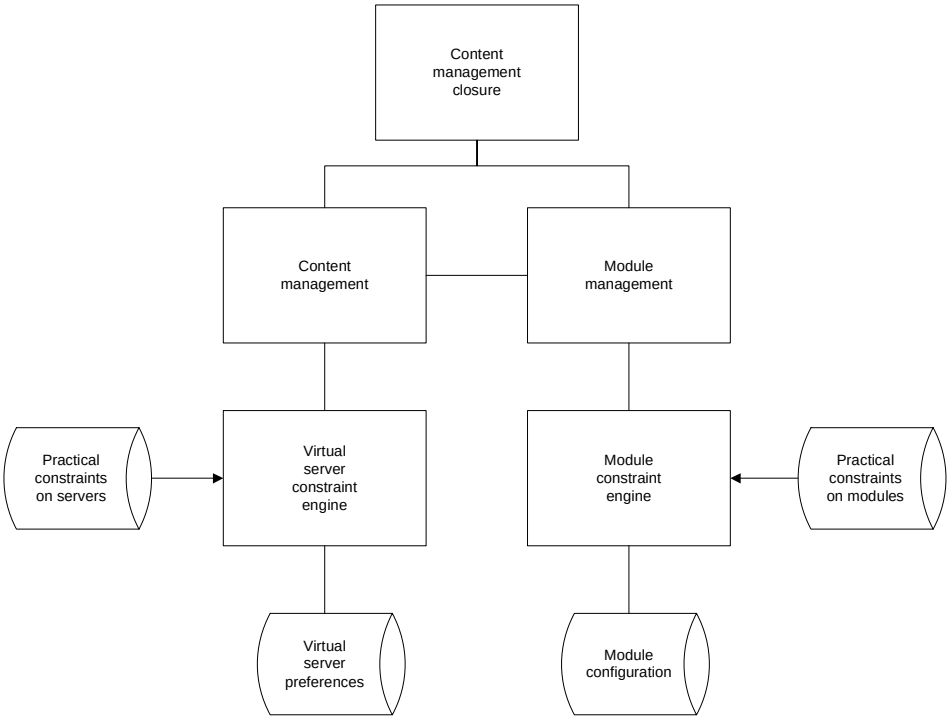

Figure 7: Parts of the configuration management closure in Figure 6.

people may not be expressive enough to be appropriate for experts.

This is only the beginning of our journey, and there are several caveats for those who would follow this path.

1. First, beware of seemingly stateless languages whose *environment* is stateful. In our language, the act of copying files is stateless, but the files themselves can change between copies. So the copying commands are not truly idempotent.
2. Second, beware of implementing the top level of a closure before the lower levels. We cannot really *assure* behavior, because our closure is not built on top of a foundation of lower-level closures. Assurance and trust must rise from the bottom levels rather than being imposed from above. This is the obvious way to settle quandaries such as how to validate that virtual servers will receive requests, etc.
3. Third, beware of making a closure that works around limitations that should rightly be there with good reason. We chose not to allow CGI scripts to create dynamic content within the closure. This seems a limitation, but actually reflects best practices for web content creation. If CGIs ''should'' be using databases, why should we allow them not to? Ideally these CGIs should be communicating with ''data persistence closures,'' otherwise known as database management systems!
4. Creating a closure that reacts predictably to configuration commands requires discipline in creating the command set. But many more disciplines are required, and some features to which we are accustomed in existing paradigms – like CGI scripts editing server files – must cease in order for the closure to become reliable.

In the final estimation, it remains unclear whether closures are the solution to the complexity of configuration management, and unclear whether reasonable stateless languages exist for other applications. The most important lesson of this study is that practice must adapt to allow closure to exist. Without a fundamental change in language, the HTTP closure would have been impossible to create.

Ours was not just a journey of software design, but also of evolving thinking. The future of that thinking remains unknown. It is likely that as we try to create practical closures, more radical shifts in thinking and practice will be required. The answer seems to lie in the simple statement that language must carefully conform to needs. The need for simple subsystems that are easily managed will no doubt lead to ''paths where no one thought.''

Acknowledgements

Alva would like to thank the configuration management community for their continuing support; notably Mark Burgess, John Sechrest, and Paul Anderson. Without their encouragement, the difficult parts of this project would have seemed impossible. Thanks also to long-time colleague David Krumme who was the initial sounding board for many of these ideas.

Steven would like to thank his family and friends for their support, including Ken D'Ambrosio for being a sounding board and an invaluable resource in refining and codifying many concepts into the final form. Thanks to Lara Ullman, Micha Rieser, and John Knoerzer for being there as a sympathetic ear, and forcing a strung out graduate student to take a break when one was truly needed. Thanks must also go out to Lynne Baer for her patience and unwavering faith in me. Lastly, he would like to thank everyone reading this paper, this being my first foray into publication; hopefully you have enjoyed reading it.

Author Information

Steven Schwartzberg has been working in the system administration field for seven years in various capacities ranging from a techie at a small ISP to managing a team at a small technology start-up. He recently received his Masters of Science from Tufts University in the field of Computer Science. He is currently working as a system engineer at BBN Technologies in the Decision and Security Technologies Division, and can be contacted via electronic mail as sschwart@bbn.com .

Alva L. Couch was born in Winston-Salem, North Carolina where he attended the North Carolina School of the Arts as a high school major in bassoon and contrabassoon performance. He received an S.B. in Architecture from MIT 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 became a member of the faculty of Tufts Department of Computer Science in the fall of 1988, and is currently an Associate Professor of Computer Science at Tufts. 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 .

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., San Diego, CA, 2003.

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

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

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

[6] Burgess, M., "Computer Immunology," *Proc. LISA XII*, USENIX Assoc., 1998.

[7] Burgess, M., "Theoretical System Administration," *Proc. LISA XIV*, USENIX Assoc., 2000.

[8] Cameron, Jamie, "Webmin," http://www.webmin.com/index.html .

[9] Cameron, Jamie, "Swell Technology Virtualmin Virtual Hosting Management," http://www.swelltech.com/virtualmin/ .

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

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

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

[13] Couch, Alva, "An Expectant Chat About Script Maturity," *Proc. LISA XIV*, USENIX Assoc., 2000.

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

[15] Couch, A., and Y. Sun, "Global Impact Analysis of Dynamic Library Dependencies," *Proc. LISA XV*, USENIX Assoc., 2001.

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

[17] Couch, A., J. Hart, E. Idhaw, and D. Kallas, "Seeking Closure in an Open World: A Behavioral Agent Approach to Configuration Management," *Proc. LISA XVII*, USENIX Assoc., 2003.

[18] Couch, A. and Y. Sun, "On Observed Reproducibility in Network Configuration Management," to appear in the *Science of Computer Programming* special issue on Network and System Administration, Elsevier, Inc, 2004.

[19] Gélinas, Jacques, "Linuxconf Home," http://www.solucorp.qc.ca/linuxconf/ .

[20] Giridharagopal, Deepak, "DryDock: A Document Firewall," *Proc. LISA XVII*, USENIX Assoc., 2003.

[21] Hart, J. and J. D'Amelia, "An Analysis of RPM Validation Drift," *Proc. LISA XVI*, USENIX Assoc., 2002.

[22] Holgate, M. and W. Partain, "The Arusha Project: A framework for collaborative Unix System Administration," *Proc. LISA XV*, USENIX Assoc., 2001.

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

[24] Kanies, L., "Isconf: Theory, Practice, and Beyond," *Proc. LISA XVII*, USENIX Assoc., 2003.

[25] Sandnes, Frode Eika, "Scheduling Partially Ordered Events in a Randomised Framework – Empirical Results and Implications for Automatic Configuration Management," *Proc. LISA XV*, USENIX Assoc., 2001.

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

[27] Finke, Jon, "Generating Configuration Files: The Director's Cut," *Proc. LISA XVII*, USENIX Assoc., 2003.

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

[29] Logan, Mark, Matthias Felleisen, and David Blank-Edelman, "Environmental Acquisition in Network Management," *Proc. LISA XVI*, USENIX Assoc., 2002.

[30] Oetiker, T., "TemplateTree II: the Post-Installation Setup Tool," *Proc. LISA XV*, USENIX Assoc., 2001.

[31] Patterson, J., "A Simple Model of the Cost of Downtime," *Proc. LISA XVI*, USENIX Assoc., 2002.

[32] Daniel López Ridruejo, "Comanche Downloads," http://www.comanche.org/downloads/ .

[33] Roth, M. D., "Preventing Wheel Reinvention: The Psgconf System Configuration Framework," *Proc. LISA XVII*, USENIX Assoc., 2003.

[34] Traugott, Steve and Joel Huddleston, "Bootstrapping an Infrastructure," *Proc LISA XII*, USENIX Assoc., 1998.

[35] Traugott, Steve and Lance Brown, "Why Order Matters: Turing Equivalence in Automated Systems Administration" *Proc. LISA XVI*, USENIX Assoc., 2002.

[36] 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*, USENIX Assoc., 2003.

[37] 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*, USENIX Assoc., 2003.

[38] Ginger Alliance, ‘‘Charlie – XML Application Framework System,’’ http://www.gingerall.com/charlie/ga/xml/p_ch.xml .

[39] The Gallery Team, ‘‘Gallery – Your Photos on Your Website,’’ http://gallery.sourceforge.net .

[40] XML Working Group, ‘‘XML Schema Specification,’’ http://www.w3c.org .