USENIX Association

Proceedings of the 17th Large Installation Systems Administration Conference

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

USENIX
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

ISconf: Theory, Practice, and Beyond

Luke Kanies – Reductive Consulting, LLC

ABSTRACT

ISconf is a configuration management system (CMS) developed through years of experience automating configuration management. As the number of CMS's available for use increases, attempts have been made to explain why one should use ISconf instead of those other tools; specifically, a stark contrast has been made between tools which depend on *convergence*, like cfengine, and ISconf, which strives for *congruence*. This paper discusses production experience with ISconf, what was good and what was bad, and also analyses recently developed theories used to explain why ISconf might be superior to other tools. It further discusses experience integrating ISconf with cfengine, a process in direct conflict with the accepted ground rules for using ISconf. Based upon this experience, there are serious limitations in practical usage of ISconf, and proposed theories that support its use are insufficient and unconvincing. Although ISconf has been found to be a useful tool, it cannot be considered a sufficiently useful or powerful answer to arbitrary configuration management needs on its own.

Introduction

ISconf [isconf] began life as a makefile [make], as described in *Boostrapping an Infrastructure* by Traugott, et al., [bootstrap]. It was a file specifically ordered and arranged to describe the dependencies and relationships between different hosts and the work that needed to be done on those hosts. Every host was listed as a make target and had a list of dependencies associated with it; one could simply run make -f isconf.mk <hostname>, and make would verify that all of those dependencies had been resolved. This is a very simple concept; Unix system administration can usually be broken down entirely into a sequence of command lines, and this structure merely took advantage of that by maintaining that list of command lines in the order they were executed on a given host.

Although ISconf has gone through two full rewrites since that initial version, it has barely changed at all in practice. It still functions entirely as a means of associating a list of command lines with a given host name, and that list of command lines is still contained in a makefile; all that has changed is how the order of commands is maintained and how the mapping between host names and commands is maintained.

What has changed the most about ISconf in that intervening time is the theory behind it. While it was originally developed out of practice, and its usefulness hinged entirely on experience, attempts have been made to provide a theoretical basis for ISconf's superiority as a tool [order]. While these attempts at theory cannot lessen ISconf's basic usefulness, they have served to muddle the practice of using ISconf, because of that theory's requirement that ISconf not be used in conjunction with any other CMS.

Recent Developments in Theory, and Their Consequences

Initial usage of ISconf demonstrated that it was very simple to develop a list of commands used to build a system, and that if those commands were replayed in the exact order on another host, a host with the same traits as the first could be easily and consistently built. This was found to be very useful, both for disaster recovery and for duplication of system configurations. The theoretical developments around ISconf call this *deterministic ordering*, because they state that the ordered list must be executed in that specific order every time, with no deviation.

This deterministic ordering requires three features that all known versions of make possess: State maintenance, failure on error, and consistent ordering. This has allowed make to be the engine for all of ISconf's functionality with minimal development. Unfortunately, this ordering requirement also introduces a requirement that make cannot meet: any failures must leave the system completely unmodified and in a recoverable, consistent state, meaning that successful steps attempted as part of an unsuccessful sequence must be backed out completely, leaving the system as though the sequence had never been begun. This concept is called *atomicity* [prolog].

State Maintenance

The functionality of make centers around knowing how to build programs which rely on various input files. One develops a list of commands, known as a *stanza*, named for the file one wants to create; if that file does not exist, or is out of date, then those commands should be run. If the commands do not create the named file, then the commands will be run again when make is executed again.

ISconf has a special *stamps* directory devoted to files marking completed commands. All make stanzas executed by ISconf perform their appointed task and then put a stamp file in this directory to mark that they have completed successfully. Given a list of dependencies associated with a host, make will only execute those dependences for which a stamp file does not exist; therefore, make is maintaining a representation of the state that the host is in. Here is an example stanza, to elucidate this process:

```
 enable_ftp:
   cp /etc/inetd.conf /tmp/inetd.conf
   sed 's/#ftp/ftp/' /tmp/inetd.conf \
                    > /etc/inetd.conf
   /etc/init.d/inetd restart
   touch $@
```

This stanza makes a backup of the config file to be modified, then uses sed to uncomment ftp from that backup file and overwrite the main configuration file. Inetd is then restarted. Finally, a file with the same name as the stanza (the macro $@ refers to the name of the stanza being executed in make) is created so the stanza does not get run again.

Failure on Error

As make works its way through a list of commands to execute, it exits immediately if any of those commands fail. This fits its original purpose well, because it is assumed that some later file requires the file that just failed to be built. This assumption works well with ISconf, also; if a command fails, then we want ISconf to exit immediately, because it is likely that some later command will depend on the command that just failed.

Thus the fact that make exits immediately upon encountering an error is used in ISconf to maintain ordering; if a command fails but the next command is executed, we will be executing our command list in an order different from that specified in the configuration. Having this failure on error guarantees that all of our hosts have executed up to the point where they completed their command list or they encountered an error, but in no instance did they skip a step or execute a step out of order.

Consistent Ordering

Configuration files for make, also called *makefiles*, are organized in the form of stanzas, each of which have up to three parts. All stanzas have a name, followed by a colon (:). If the stanza has any prerequisites, they will be on the same line as the stanza name, after the colon. If there are any commands associated with the stanza name, they begin on the next line after the stanza name and are indented. Long lines can be continued with an escaped carriage return.

The make program uses a stable topological sort to determine the execution order of commands specified in its configuration files. Make builds a topology of all parent-child relationships, ordered by the declared dependencies and the order in which they appear in the config files, and then linearly walks through these relationships, executing children first and then parents. As long as the parent-child relationships and the order in which they are specified stay the same, make will execute the commands in the same order every time.

ISconf exploits this consistent ordering. It would be straightforward to develop a tool which consistently performs a list of commands in the exact order they are specified, but make conveniently already has that feature, so ISconf takes advantage of it.

Atomicity

Unfortunately, something that ISconf also requires but that make cannot deliver is the atomicity of all procedures. A set of commands must either complete entirely, or the set must fail entirely and not modify the system at all. If a sequence of commands is begun but a failure is encountered, then the entire sequence should be backed out, so that the system is left in an unmodified state. If this cannot be done, then the system is likely to be in an inconsistent or non-functional state. It is possible to make ISconf stanzas idempotent, so that they could be run multiple times with no ill effect, but it is far more difficult and eliminates the simplicity that ISconf provides. If all stanzas were idempotent, then most configuration errors could be easily corrected on the next run of ISconf.

For instance, consider modifying the configuration file of an important service like bind: given a sequence of commands that modify a configuration file and then restart the service, if the restart fails because of a typo in the modification but the modification is not backed out, then the service will be down until a human intervenes.

ISconf requires another level of atomicity; each stanza that ISconf executes must also be atomic. If a stanza has multiple commands that modify a system and that stanza fails, then human intervention will certainly be required to either manually complete the stanza or manually back out the completed portions of the stanza. Given a stanza that creates a directory and then adds it to a system's NFS exports file, consider what happens if the directory creation succeeds but the configuration change fails. Normally one would solve the configuration problem and let ISconf run the stanza again, but the second time the stanza runs, the directory creation fails because the directory already exists. A human must modify the system manually in some way to solve this problem.

Atomicity at both the stanza and the sequence level are required by ISconf, but we see that both are missing, partially because make lacks them and partially because atomicity is a difficult problem [maturity]. ISconf certainly is not any more immune to these problems than other CMS's, and its use of make makes it more susceptible in some ways, especially since atomicity problems can creep into make stanzas in ways which don't show themselves until the stanza has already been run many times.

Practical ISconf Usage

One of ISconf's biggest strengths is that work performed with it is very similar to work performed by hand, so it is very easy to make progress quickly. Unfortunately, whereas a human would naturally take into account the differences between different hosts, those differences have to be coded into ISconf, and it is impossible to code for all of them. This is where ISconf begins to run into real trouble.

Day to day use of ISconf is affected most by the fact that it relies on make as its execution engine. The best it can hope to do is optimize the mapping between hosts and their execution lists, and possibly simplify some aspects of using make. Most people looking for configuration management systems are seeking some way to better understand the maintenance of their servers, and ISconf is limited in its ability to provide this understanding. ISconf remains useful as long as it is treated as a modest tool with modest goals, but as soon as one begins expecting functionality on par with a full CMS, severe limitations surface.

ISconf's Feature Set

ISconf is an interface to make, and not much else. It is entirely a tool for mapping hosts to commands. If there are functions one needs to perform, those functions must be written. There is no higher-level functionality, no advanced toolset, no powerful configuration language, and no reusable component system. In fact, the ordering that ISconf requires is really the only thing it offers, whereas other configuration management systems usually lack this ordering but provide some semblance of all of the above features. In choosing between ISconf and most other CMS's, this is largely a choice between a tool which easily organizes work and a tool which makes the work itself easier.

It is true that many of these features could be developed over time, especially the advanced toolset. ISconf 3 already has a number of useful utilities. ISconf's use of make as its API to the utilities upon which it depends requires that these utilities must also be capable of functioning independently, because very little information can be passed from ISconf to the utility being executed. This lack of integration also places a lower burden of consistency on the authors of the utilities, which means that utilities are likely to have more localization and individuality than is desirable for a component of a CMS, placing a further burden on the sharing of code. Alva Couch's paper on script maturity [maturity] discusses this topic in more detail.

Because ISconf stanzas depend heavily on the environment for which they were written, the shareable utilities would have to be more abstract than the actual stanzas. So, in order to make it possible to reuse someone else's ISconf code, that code must be written such that it operates irrespective of how it is called by ISconf, which is equivalent to saying it must be able to operate completely independently. Thus, these utilities could be written for any CMS, and cannot be specifically written for ISconf, so ISconf confers no advantage here at all, and is detrimental in that it presents that additional burden of abstraction to those who would like to write shareable utilities.

What ISconf Understands

Similar to how make depends on the writer of the makefile to understand the files it is managing, ISconf depends on its maintainer to understand the commands it is executing. ISconf currently enforces ordering entirely at the command level, not at a symbolic or functional level. This is in stark contrast to tools like psgconf [psgconf], which are specifically developed to manage systems at a symbolic level – much easier for humans to understand – and rely on the CMS to map the symbols into a deployed configuration.

In most ways, this is a limitation of ISconf. At first glance, ISconf makes managing a system as easy as managing a development project, but with use, it quickly becomes more complicated to control. psgconf must understand all files it maintains well enough to rebuild that file from scratch without changing anything else; as a result, an invocation can help one quickly recover from a configuration problem or a lost file.

ISconf provides none of this understanding, which means that more research is required before any change is queued, and any misconfiguration requires time and attention from a system administrator. Consider managing the /etc/services file, which is used for mapping service names to numbers. In psgconf, a specific mapping is maintained in a symbolic form and then converted to a file, so managing this file involves changing this symbolic form, which is very easy for a sysadmin:

```
### Entries for /etc/services in psgconf
port_names {
     "1/tcp" => tcpmux,
     "7/tcp" => echo,
     "7/udp" => echo,
     "9/tcp" => discard,
     "9/udp" => discard,
     "11/tcp" => systat,
        ...
};
```

Any errors in this configuration can be easily repaired, and the file can be instantly regenerated with correct, valid contents.

In ISconf, however, the sysadmin merely queues up a command which does what she thinks is appropriate. If the sysadmin made a typo that was not discovered until the stanza was already deployed, then the file may have to be recovered from backup (during which time the system is in a failure state), or a throwaway stanza must be built to recover from the error; see Listing 1.

You can see that if a few such throwaway stanzas are made, it quickly becomes very difficult to understand exactly how a given ISconf configuration produced the configuration on disk. Most often, the local configuration must be studied, and the list of stanzas to execute (ISconf 3 comes with the utility islist, which provides the ordered execution list for any host) must be followed completely to figure out exactly how a configuration came to be, and thus understand where to correct a problem. Correcting a problem thus includes reverse-engineering a series of scripts as well as understanding the mapping between configuration and behavior. Because all of a stanza's preconditions are only implicitly documented by the stanzas run previously, a stanza's dependencies cannot be assumed and must instead be tested every time.

This problem becomes even more complicated by ISconf's susceptibility to problems with latent preconditions, where hosts are differentiated in ways that aren't fully understood. For example:

```
add_goodpkg:
  pkginstall GoodPkg
        # adds 'goodpkg 9000/tcp'
        # to /etc/services
  touch $@

add_newservice:
  echo "newservice 9000/tcp" \
            >> /etc/services
  touch $@
```

With the comment attached to the add_goodpkg stanza, it is obvious that these stanzas conflict. But few of the changes done during package installation are obvious, even when the known changes are documented somewhere. If the first stanza is run on a small subset of hosts and the second stanza is run on all hosts, it could take a significant amount of time to track down the source of any problems that result from this conflict. It is possible to overcome this specific problem by using ISconf to generate the /etc/services file similarly to psgconf [psgconf], but this type of problem crops up constantly in more subtle and less manageable forms.

It should be noted that ISconf probably does provide the lowest barrier of entry in terms of managing the /etc/services file, in that it is incredibly easy to modify that file in ISconf, but it provides no extra functionality with that ease of use nor any protections from mistakes, whereas both cfengine [cfengine] and psgconf require more initial investment time upfront but give back significant extra capabilities in return for that investment.

How ISconf Does Its Job

As has already been seen, ISconf relies on make for most of its functionality. Inasmuch as make can be said to have an API (Application Programming Interface), ISconf shares that API. Because make was developed with a very specific purpose, one that is relatively simple when compared to managing operating systems, it has a somewhat limited interface. ISconf inherits all of the limitations of this interface, but tacks some of its own liabilities onto it.

The make program organizes its commands into stanzas; each stanza is essentially self-contained and should perform a single, specific function. ISconf in turn associates these stanzas with hosts and host types. This actually means that ISconf's API is less functional than that of make, because one does not have access to make's full preconfiguration capabilities. Because it is the stanzas that ISconf associates with a host, and not the actual commands, these stanzas function as a mapping between a host and what is done on that host; the commands that a stanza executes could be changed to be something completely different and ISconf would never know, or a stanza name could be changed without changing its functionality and again ISconf would not notice.

What's worse, however, is that this means that each host has two lists associated with it – the list of stanzas and the list of commands actually executed – and neither can ever be modified after the fact. One cannot change the name of a stanza, because make will incorrectly conclude that it is a new stanza and will execute it again. One cannot change the commands that a stanza executes because it is only the old version of the commands that has been tested, and one has no way of really knowing if the new version of the commands will succeed if the host needs to be duplicated or rebuilt. This is a direct result of ISconf managing a list of make stanzas, rather than understanding or controlling what it is actually doing.

This is what really hurts ISconf in the long run. All code and all stanzas that have ever been executed on any host must be maintained until that host no

```
### Entries in /etc/services for ISconf
# if this stanza gets run on all your systems, you are very
# unhappy
add_myservice_bad:
    echo "myservice 888/tcp" > /etc/services  # oops, just overwrote the file
    touch $@

add_myservice_service:
    echo "myservice 888/tcp" >> /etc/services # oops, wrong port number
    touch $@

fix_myservice_service:
    cp /etc/services /tmp/services.bak
    sed 's/myservice 888\/tcp/myservice 8888\/tcp/' \
        /tmp/services.bak > /etc/services
    touch $@
```

Listing 1: Error recovery stanza.

longer exists, even if they are no longer in active use. And even if one has found a new way of managing a given function or file, all of the old methods must be maintained in case they are needed again during disaster or server duplication. This is because all hosts on the network are different in some ways (usually at least hostname and IP address), and although testing can reduce the likelihood of those differences causing problems, it cannot actually eliminate the possibility for problems. Not all dependencies are specifically encoded as such in ISconf; package installation may require a certain amount of free space in /var without there being a check that this space exists. In fact, if this amount of space is proportional to the size of the package being installed, it may not be a problem until an especially large package install is attempted.

Two simple but common examples are presented to illustrate this point, both of them very common functions encountered in usage of ISconf. The first is usage of ISconf to manage packages on Sun Solaris systems. Initial package management was done by the author by creating a stanza for every package that needed to be installed, and then doing all the necessary work in each package.

```
add_cvs:
  mount -t nfs \
      server:/export/pkg /mnt
  pkgadd -A /mnt/pkgadd_noask \
              -d /mnt/cvs all
  touch $@
```

After a few of these stanzas were created, it became clear that some method of reducing code duplication needed to be developed. In response, a single script responsible for all package management was created, as well as a way of referring to that script using wildcards in make.

```
pkg/%:
  mkdir -p pkg
  pkginstall --isconf $@
  stamp $@
```

This script expects a complicated stanza name like pkg/add__java__1.4.1 to be passed to it, and it parses that name to figure out what it is supposed to do. However, because the initial package stanzas were already executed on all Solaris systems, they could never be replaced with this new method. Doing so would risk the new method failing because of its own existing preconditions, such as the new method requiring a package installed using the old method.

This is actually a relatively innocuous example, because the two methods are at least compatible. The author has also encountered situations where an incompatible method of performing a function needed to be developed, such as for managing the contents of a file.

The second example is even more common: Deployment of a script with a subtle logic bug which does not get discovered until it is already deployed. The script works fine on the first hosts on which it is deployed, but when the script is deployed again on a differing set of hosts, it fails in an unforeseen way. The author has written a script for automating usage of Sun's DiskSuite application to mirror a system's boot disk; this script was originally developed entirely on systems with SCSI disks, and thus was only tested on them. When Sun came out with servers with IDE disks, this script failed horribly. However, the theoretical foundations of ISconf required that the script be copied and a new version modified to have the new functionality, rather than merely upgrading the existing version, because again, there is no real way of knowing that the new version would succeed on systems that the old version had already been run on.

This is an example of *software rot*, in that the assumptions of a program became out of date. When the script was originally written, it was not possible to purchase a Sun server with IDE disks, so it was assumed that all disks would be SCSI disks. When Sun began shipping systems with IDE disks, this assumption became invalid. This is a very general problem, because it is not really possible for a script or program to truly understand or state all of its preconditions, and many of those preconditions are only discovered when they are not met in some new deployment, long after the script was first put to use.

These two examples illustrate that ISconf encourages a number of very bad habits: keeping buggy old scripts around, not implementing better management methods merely because they would necessitate what amounts to extra bookkeeping, accumulation of cruft in configuration files, and many others. Worse, ISconf encourages its users to break its own rules; it is unlikely that anyone can read the above examples and think "yeah, gotta follow the rules here." Most sysadmins would prefer to spend the time now to get the new script and the new package management methods working, then replace all of the old instances in specific violation of the principles of ISconf, rather than deal with having cruft in the configuration files for literally years. Having a CMS which has strict rules but encourages its users to break them is self-defeating, and is one of ISconf's biggest liabilities. In fact, this aspect of ISconf is what led me to begin searching for other tools, either to replace or complement ISconf.

Are These Problems Really Intrinsic?

Given that many of the problems discussed in this paper are a result of ISconf's interface to make, it could be argued that these problems are a result of current implementations but are not necessarily intrinsic to ISconf. Unfortunately for those so inclined, replacing make with another tool which better met ISconf's needs could only mitigate the problems, not solve them; ISconf is just a means of associating hosts with the specific commands run on those hosts and the problem concerns the commands, not the framework. Any attempt at abstracting those specific commands into symbolic information more useful to humans is in direct violiation to ISconf's ordering principles; after all, how can one really verify that work was done in a consistent order if one does not even know exactly what is being done?

A maturity model has been developed for scripts [maturity] that is useful for assessing whether a script will consistently result in a valid, functioning configuration. While it is possible to write from scratch a script that scores highly in this model, it is much safer and easier to rely on proven high-quality tools which fit this model. ISconf's reliance on its users to develop necessary functions makes it less likely that the utilities used will be mature according to the referenced model, which results in a greater likelihood of deployment problems. Other CMS's might not provide the simplicity that ISconf provides, but they are much more likely to have all of their work score highly on the maturity model, which makes for a more stable infrastructure.

As has been mentioned multiple times, ISconf suffers from the fact that it is based entirely on preconditions, but it does not and can not actually code in all of those preconditions. This means that all ISconf stanzas are only valid in the specific environment in which they've been tested, which is created by the sequence of stanzas up to the one in question. Any other use of the stanza is not likely to result in desirable results. The commands executed by the stanza might be independent, but the stanza itself, as run by ISconf, requires the sequence that has already been executed.

Notes on Undecidability and Turing Equivalence

Traugott, et al., recently published a paper [order] claiming that all self-modifying systems, which covers all modern configuration management systems, can be equated to Universal Turing Machines, and thus are susceptible to what is called the Halting Problem [halting], which is a specific instance of the mathematical problem of Undecidability [undecidability]. Traugott, et al., go on to claim that ISconf is actually the only CMS immune to this problem.

While I do not find this comparison particularly worthy of investigation, the claim that ISconf is somehow immune to a problem to which other CMS's are susceptible deserves a closer look. ISconf's theoretical immunity to the Halting Problem arises from its requirement that all actions must be tested by a human before being deployed in production. The implication is that once a given command sequence has been tested in a non-production environment, it can be deployed in a similar production environment and remain immune to any vestiges of the Halting Problem.

Again without addressing the validity of the claim that CMS's are susceptible to the halting problem, if it can be shown that a successfully tested ISconf sequence is still not guaranteed to succeed, then this testing ceases to be any special feature of ISconf, and thus cannot provide any differentiation from other CMS's.

Theoretically speaking, it is trivial to create a command sequence which can succeed during testing but will fail in production: simply build in prerequisites which only exist in the test environment, such as IP addresses, host names, or domain names. Practically speaking, I have seen a number of sequences which succeeded in testing, or even in initial production deployments, which later failed because of differences in pre-existing conditions. I have encountered problems with host installation, package installation, and hostname length, all of which tested just fine but failed in production.

From both theoretical and practical perspectives, it is obvious that testing an ISconf command sequence provides nothing like a guarantee, so this testing can no more protect ISconf from fundamental limitations of the given system than testing with other CMS's can. In fact, because ISconf is so dependent on the bits on the disk and has no higher level facilities for managing objects above the most basic level, it is likely to be more susceptible to fundamental limitations, because it is operating at the same level as those limitations, rather than at a level slightly above them, as psgconf tries to do.

Where to Go From Here

It is apparent that ISconf has fundamental, intrinsic flaws, but the realization that ISconf's rules must be bent allows one to take advantage of what ISconf can do while going elsewhere for more complicated demands. I have found ISconf's ability to map work lists to host names very useful, especially with the typing system as developed in ISconf 3, but found its functionality lacking. Even better, after experimentation with cfengine I found that there were symbolic similarities between ISconf's concept of a host type and cfengine's concept of a host class, and in fact, the format of the two details in the tools was almost exactly the same.

Because I was dissatisfied with the current functionality of ISconf, but wanted to keep the existing work lists, an attempt was made to integrate ISconf with cfengine. Interestingly, significant research effort has been spent in justifying that these two tools are incompatible in their approaches to system management, yet the author found them to be completely orthogonal, and very compatible.

Cfengine Integration

Cfengine is a very useful tool, especially for its its ability to discover the state of the system on which it is running and then perform actions based on that state. However, some actions are obviously intrinsically ordered (one cannot format a volume that has not been created), and ordering arbitrary events in cfengine is inordinately complicated. Thus, it seemed that cfengine's higher-level capabilities could benefit from ISconf's functionality.

Although I was satisfied with ISconf's ability to perform most work, there was a significant amount of functionality within cfengine that ISconf lacked, such as the ability to verify file permissions and restart processes which died. However, I did not want to just implement cfengine independent of ISconf, because cfengine would need to know how hosts were different and then be able to behave differently according to those differences. Thus, I decided that if I could integrate cfengine and ISconf, such that ISconf had access to all of cfengine's states, and cfengine had access to all of ISconf's host types, I could get the best of both tools without having to store the same information (such as a host being an oracle server) in more than one place.

When this integration project was begun, ISconf used Damian Conway's excellent Parse::RecDescent [recdescent] perl module as a parsing engine, and the format it used for host type names was slightly incompatible with cfengine classes. Because parsing was becoming very slow (taking up to 30 seconds for some key files), the parsing engine was rewritten using Parse::Yapp [yapp] and Parse::Lex [lex], and the format of the host type names was changed to exactly match cfengine classes. This allowed for a simple point of integration: make all ISconf host types available to cfengine as classes, and make all cfengine classes available to ISconf as host types. Thus, attaching a work list to an oracle_server type in ISconf would allow one to set file permissions based on that oracle_server class in cfengine, and discovering in cfengine that a host is a Sun sparc system would allow ISconf to perform specific actions based on that fact.

Unfortunately, the modifications to ISconf's parser were the easiest step in the integration process. Because of a parsing optimization within cfengine, all classes that might be set at some point must be known at parse time, either through hard setting or through use of the AddInstallable command. After much experimentation and many false steps, a script was written that collects all of the ISconf types for a host and generates a cfengine file which sets those types; if cfengine finds that file, then it includes it, and if it does not find the file, then it exits without doing any work. This way one can guarantee that no work is done without all knowledge being available.

An example configuration (trimmed for clarity) is included in Listing 2.

Existing implementations of ISconf have a preparatory script which refreshes its configuration with the most recent version; integration with cfengine enabled an easy replacement of this external script with cfengine's file transport capabilities, on both the client and server side.

Upon successful integration, there was immediate benefit. I had previously written some simple utilities for managing file permissions and ownerships, but those utilities were lacking in key functionality which cfengine possessed, and certainly did not meet the

```
# cfagent.conf
groups:
    # check to see if the istypes.cf file exists
    istypes = ( IsPlain(${workdir}/inputs/istypes.cf) )

import:
    any::
        ispref.cf # creates the istypes.cf file

    istypes::
        istypes.cf
        main.cf

        isconf.cf
        # import all other files if istypes.cf exists;
        # otherwise, import nothing other than isprep.cf
# ispref.cf
control:

    # create the istypes.cf file, and mark all ISconf
    # types as installable
    AddInstallable = ( ExecResult(${workdir}/bin/istypes) )

# isconf.cf
control:
    actionsequence = ( copy "module:runisconf -cf=${ALLCLASSES}" )

copy:
    ${isconfsource}         dest=/var/isconf
                            r=inf
                            server=${isconfserver}
```

Listing 2: Example configuration.

maturity requirements of most organizations. Integration allowed immediate access to all of cfengine's file permission functions, based entirely on information collected from IScon; in other words, cfengine was able to immediately replace a separate, less capable script without any extra development effort (although it did take some practice with cfengine, obviously).

As this integration has progressed, an interesting conceptual transformation happened; the IScon host types conceptually became just *facts*, just as Couch, et al., observed that cfengine states are Prolog Facts [prolog]. Rather than being part of a hierarchical typing system, they were just logically true or false, even if they were set that way based on that hierarchical typing system. This allows one to build work lists as a group, without having to consider that work list as a host type. For instance, consider a lengthy application configuration process, such as is necessary for cfengine: installing three different applications, modification of the /etc/services file, creation of the cfengine key pair, downloading the update.conf file, uploading the public key to the cfengine server, and finally running cfengine the first time to get everything started.

It would make no sense to make this into a host type, because all systems will run it. It is clearly just a related, ordered list of work, not a separate type of server. The precepts of IScon require that one add each item in this list to all configured host types, or at least add them to a base type, with no indication that the work is related; however, if one considers IScon types to just be classes, rather than mapping to a server type, then one can very easily configure this as an ordered work list, maybe named cfengine, and then attach it to the necessary host types. This retains the fact that these steps are all related, while still satisfying IScon's ordering requirements.

Another process that became far easier with an integrated cfengine and IScon was the testing that IScon requires so much. To test a new stanza in IScon, one must do a partial roll-out of that stanza, but one can only do that to entire host types, both development and testing. The method IScon uses to get around this problem is requiring multiple domains (e.g., test and prod), different fileservers for each domain, and then deployment of the new stanza first on the test fileservers then on the production fileservers. This partial deployment requires that all fileservers be treated differently because they must all have different branches of the IScon configuration. If two people want to do partial roll-outs of code in two unrelated host types, they basically cannot.

Once integrated with cfengine, however, testing new IScon stanzas becomes far easier. All test and production hosts should be marked so in either cfengine or IScon (depending on configuration needs); when it comes time to test a new sequence of work, it can be created as an independent work list, and cfengine can initially enable that work list only for test systems. Once the test is complete, cfengine or isconf can then easily enable it for all hosts. For instance:

```
# in a cfengine config file
test_domain_com::
     AddClasses = ( mytestworklist )
```

If the test list succeeds, then it can be deployed on all hosts either through IScon, by adding it to another IScon type, or through cfengine by eliminating the test_domain_com:: portion of the file.

For the future, the author would like to begin replacing some of IScon's configuration files with cfengine. For instance, instead of having a hard mapping of host names to host types in IScon's hosts file, the mapping could be maintained within cfengine. This would enable cfengine's logical testing capabilities earlier in the IScon process, but would also sacrifice IScon's ability to associate arbitrary variables with hosts and host types (cfengine does not currently have a facility for passing arbitrary variables to its modules or shellcommands).

Conclusion

IScon is billed as an enterprise configuration management system, capable of managing all aspects of system administration, but it is found instead only to be capable of ordering work done by some other system. This finding does not invalidate IScon's usefulness, but significantly reduces its scope while at the same time allowing it to be used with other tools which have greater functionality. Most of the existing theory explaining IScon's usefulness is found wanting, and the purported conflict between tools like IScon which preach congruence and tools like cfengine which preach convergence is found not to exist. Turning IScon into a cfengine module is found to make both tools significantly better.

Biography

Luke Kanies graduated from Reed College in 1996 with a Bachelor's degree in Chemistry. He has since been honing his skills at using Unix to do less work through automation and abstraction. He currently runs a consulting company, Reductive, LLC. Reach him via email at luke@madstop.com .

Availability

IScon 3, by Luke Kanies, is available via the web at http://www.sourceforge.net/projects/isconf . Cfengine, by Mark Burgess, is available via the web at http://www.cfengine.org . PSGConf, by Mark Roth, is available at http://www-dev.cites.uiuc.edu/psgconf/.

References

[bootstrap] Traugott, S. and J. Huddleston, "Bootstrapping an infrastructure," *Proc. LISA XII*, 1998.

[cfengine] Burgess, M. "Cfengine: a site configuration engine," *USENIX Computing Systems*, Vol. 8, Num. 3, 1995.

[decidability] *Decidability*, http://wombat.doc.ic.ac.uk/foldoc/foldoc.cgi?decidability .

[isconf] *ISconf: The Infrastructure Configuration Engine*, http://isconf.org .

[lex] Verdret, P., *Parse::Lex perl module*, http://search.cpan.org/author/PVERD/ParseLex-2.15/ .

[make] *Make*, http://www.gnu.org/software/make/|.

[maturity] Couch, A., "An Expectant Chat on Script Maturity," *Proc. LISA XIV*, 2000.

[order] Traugott, S. and L. Brown, "Why Order Matters: Turing Equivalence in Automated Systems Administration," *Proc. LISA XVI*, 2002.

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

[psgconf] Roth, M., *PSGConf*, http://www-dev.cites.uiuc.edu/psgconf/ .

[recdescent] Conway, D., *Parse::RecDescent perl module*, http://search.cpan.org/dist/Parse-RecDescent/ .

[turing] Turing, A., "On Computable Numbers, with an Application to the Entscheidungsproblem," *Proceedings of the London Mathematical Society*, Series 2, Vol. 32, pp. 230-265, 1936-37.

[yapp] Desarmenien, F., *Parse::Yapp perl module*, http://search.cpan.org/author/FDESAR/Parse-Yapp-1.05/ .