Why Order Matters: Turing Equivalence in Automated Systems Administration

Steve Traugott – TerraLuna, LLC
Lance Brown – National Institute of Environmental Health Sciences

ABSTRACT

Hosts in a well-architected enterprise infrastructure are self-administered; they perform their own maintenance and upgrades. By definition, self-administered hosts execute self-modifying code. They do not behave according to simple state machine rules, but can incorporate complex feedback loops and evolutionary recursion.

The implications of this behavior are of immediate concern to the reliability, security, and ownership costs of enterprise computing. In retrospect, it appears that the same concerns also apply to manually-administered machines, in which administrators use tools that execute in the context of the target disk to change the contents of the same disk. The self-modifying behavior of both manual and automatic administration techniques helps explain the difficulty and expense of maintaining high availability and security in conventionally-administered infrastructures.

The practice of infrastructure architecture tool design exists to bring order to this self-referential chaos. Conventional systems administration can be greatly improved upon through discipline, culture, and adoption of practices better fitted to enterprise needs. Creating a low-cost maintenance strategy largely remains an art. What can we do to put this art into the hands of relatively junior administrators? We think that part of the answer includes adopting a well-proven strategy for maintenance tools, based in part upon the theoretical properties of computing.

In this paper, we equate self-administered hosts to Turing machines in order to help build a theoretical foundation for understanding this behavior. We discuss some tools that provide mechanisms for reliably managing self-administered hosts, using deterministic ordering techniques.

Based on our findings, it appears that no tool, written in any language, can predictably administer an enterprise infrastructure without maintaining a deterministic, repeatable order of changes on each host. The runtime environment for any tool always executes in the context of the target operating system; changes can affect the behavior of the tool itself, creating circular dependencies. The behavior of these changes may be difficult to predict in advance, so testing is necessary to validate changed hosts. Once changes have been validated in testing they must be replicated in production in the same order in which they were tested, due to these same circular dependencies.

The least-cost method of managing multiple hosts also appears to be deterministic ordering. All other known management methods seem to include either more testing or higher risk for each host managed.

This paper is a living document; revisions and discussion can be found at Infrastructures.Org, a project of TerraLuna, LLC.

Foreword
by Steve Traugott

In 1998, Joel Huddleston and I suggested that an entire enterprise infrastructure could be managed as one large "enterprise virtual machine" (EVM) [bootstrap]. That paper briefly described parts of a management toolset, later named ISconf [isconf]. This toolset, based on relatively simple makefiles and shell scripts, did not seem extraordinary at the time. At one point in the paper, we said that we would likely use cfengine [cfengine] the next time around – I had been following Mark Burgess' progress since 1994.

That 1998 paper spawned a web site and community at Infrastructures.Org. This community in turn helped launch the Infrastructure Architecture (IA) career field. In the intervening years, we've seen the Infrastructures.Org community grow from a few dozen to a few hundred people, and the IA field blossom from obscurity into a major marketing campaign by a leading systems vendor.

Since 1998, Joel and I have both attempted to use other tools, including cfengine version 1. I've also tried to write tools from scratch again several times, with mixed success. We have repeatedly hit indications that our 1998 toolset was more optimized than we had originally thought. It appears that in some ways Joel and I, and the rest of our group at the Bank, were lucky; our toolset protected us from many of the pitfalls that are laying in wait for IAs.

One of these pitfalls appears to be deterministic ordering; I never realized how important it was until I tried to use other tools that don't support it. When left without the ability to concisely describe the order of changes to be made on a machine, I've seen a marked decrease in my ability to predict the behavior of those changes, and a large increase in my own time spent monitoring, troubleshooting, and coding for exceptions. These experiences have shown me that loss of order seems to result in lower production reliability and higher labor cost.

The ordered behavior of ISconf was more by accident than design. I needed a quick way to get a grip on 300 machines. I cobbled a prototype together on my HP100LX palmtop one March *'94* morning, during the 35-minute train ride into Manhattan. I used 'make' as the state engine because it's available on most UNIX machines. The deterministic behavior 'make' uses when iterating over prerequisite lists is something I didn't think of as important at the time – I was more concerned with observing known dependencies than creating repeatable order.

Using that toolset and the EVM mindset, we were able to repeatedly respond to the chaotic international banking mergers and acquisitions of the mid-90's. This response included building and rebuilding some of the largest trading floors in the world, launching on schedule each time, often with as little as a few months' notice, each launch cleaner than the last. We knew at the time that these projects were difficult; after trying other tool combinations for more recent projects I think I have a better appreciation for just how difficult they were. The phrase "throwing a truck through the eye of a needle" has crossed my mind more than once. I don't think we even knew the needle was there.

At the invitation of Mark Burgess, I joined his LISA 2001 [lisa] cfengine workshop to discuss what we'd found so far, with possible targets for the cfengine 2.0 feature set. The ordering requirement seemed to need more work; I found ordering surprisingly difficult to justify to an audience practiced in the use of convergent tools, where ordering is often considered a constraint to be specifically avoided [couch, eika-sandnes]. Later that week, Lance Brown and I were discussing this over dinner, and he hit on the idea of comparing a UNIX machine to a Turing machine. The result is this paper.

Based on the symptoms we have seen when comparing ISconf to other tools, I suspect that ordering is a keystone principle in automated systems administration. Lance and I, with a lot of help from others, will attempt to offer a theoretical basis for this suspicion. We encourage others to attempt to refute or support this work at will; I think systems administration may be about to find its computer science roots. We have also already accumulated a large FAQ for this paper – we'll put that on the website. Discussion on this paper as well as related topics is encouraged on the *infrastructures* mailing list at http://Infrastructures.Org.

Why Order Matters

There seem to be (at least) several major reasons why the order of changes made to machines is important in the administration of an enterprise infrastructure:

A "circular dependency" or control-loop problem exists when an administrative tool executes code that modifies the tool or the tool's own foundations (the underlying host). Automated administration tool designers cannot assume that the users of their tool will always understand the complex behavior of these circular dependencies. In most cases we will never know what dependencies end users might create; see §8.40 and §8.46.

A test infrastructure is needed to test the behavior of changes before rolling them to production. No tool or language can remove this need, because no testing is capable of validating a change in any conditions other than those tested. This test infrastructure is useless unless there is a way to ensure that production machines will be built and modified in the same way as the test machines; see 'The Need for Testing' section.

It appears that a tool that produces deterministic order of changes is cheaper to use than one that permits more flexible ordering. The unpredictable behavior resulting from unordered changes to disk is more costly to validate than the predictable behavior produced by deterministic ordering; see §8.58. Because cost is a significant driver in the decision-making process of most IT organizations, we will discuss this point more subsequently.

Local staff must be able to use administrative tools after a cost-effective (i.e., cheap and quick) turnover phase. While senior infrastructure architects may be well-versed in avoiding the pitfalls of unordered change, we cannot be on the permanent staff of every IT shop on the globe. In order to ensure continued health of machines after rollout of our tools, the tools themselves need to have some reasonable default behavior that is safe if the user lacks this theoretical knowledge; see §8.54.

This business requirement must be addressed by tool developers. In our own practice, we have been able to successfully turnover enterprise infrastructures to permanent staff many times over the last several years. Turnover training in our case is relatively simple, because our toolsets have always implemented ordered change by default. Without this default behavior, we would have also needed to attempt to teach advanced techniques needed for dealing with unordered behavior, such as inspection of code in vendor-supplied binary packages; see the 'Right Packages, Wrong Order' section.

A Prediction

"Order Matters" when we care about both quality and cost while maintaining an enterprise infrastructure. If the ideas described in this paper are correct, then we can make the following prediction:

> The least-cost way to ensure that the behavior of any two hosts will remain completely identical is to always implement the same changes in the same order on both hosts.

This sounds very simple, almost intuitive, and for many people it is. But to our knowledge, isconf [isconf] is the only generally-available tool which specifically supports administering hosts this way. There seems to be no prior art describing this principle, and in our own experience we have yet to see it specified in any operational procedure. It is trivially easy to demonstrate in practice, but has at times been surprisingly hard to support in conversation, due to the complexity of theory required for a proof.

Note that this prediction does not apply only to those situations when you want to maintain two or more identical hosts. It applies to any computer-using organization that needs cost-effective, reliable operation. This includes those that have many unique production hosts; see 'The Need for Testing.' The 'Congruence' section discusses this further, including single-host rebuilds after a security breach.

This prediction also applies to disaster recovery (DR) or business continuity planning. Any part of a credible DR procedure includes some method of rebuilding lost hosts, often with new hardware, in a new location. Restoring from backups is one way to do this, but making complete backups of multiple hosts is redundant – the same operating system components must be backed up for each host, when all we really need are the user data and host build procedures (how many copies of /bin/ls do we really need on tape?). It is usually more efficient to have a means to quickly and correctly rebuild each host from scratch. A tool that maintains an ordered record of changes made after install is one way to do this.

This prediction is particularly important for those organizations using what we call *self-administered hosts*. These are hosts that run an automated configuration or administration tool in the context of their own operating environment. Commercial tools in this category include Tivoli, Opsware, and CenterRun [tivoli, opsware, centerrun]. Open-source tools include cfengine, lcfg, pikt, and our own isconf [cfengine, lcfg, pikt, isconf]. We will discuss the fitness of some of these tools later – not all appear fully suited to the task.

This prediction applies to those organizations which still use an older practice called "cloning" to create and manage hosts. In cloning, an administrator or tool copies a disk image from one machine to another, then makes the changes needed to make the host unique (at minimum, IP address and hostname). After these initial changes, the administrator will often make further changes over the life of the machine. These changes may be required for additional functionality or security, but are too minor to justify recloning. Unless order is observed, identical changes made to multiple hosts are not guaranteed to behave in a predictable way (§8.47). The procedure needed for properly maintaining cloned machines is not substantially different from that described in the section on 'Describing Disk State.'

This prediction, stated more formally in §8.58, seems to apply to UNIX, Windows, and any other general-purpose computer with a rewritable disk and modern operating system. More generally, it seems to apply to any von Neumann machine with rewritable nonvolatile storage.

Management Methods

All computer systems management methods can be classified into one of three categories: divergent, convergent, and congruent.

Divergence

Divergence (Figure 1) generally implies bad management. Experience shows us that virtually all enterprise infrastructures are still divergent today. Divergence is characterized by the configuration of live hosts drifting away from any desired or assumed baseline disk content.

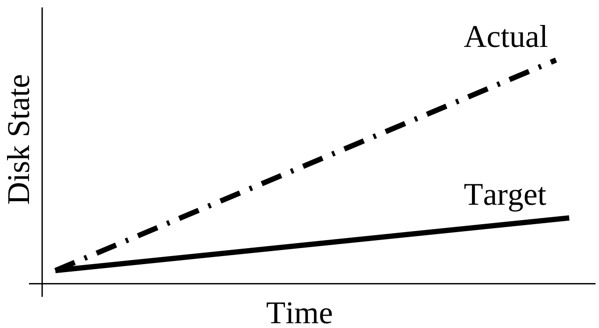

Figure 1: Divergence.

One quick way to tell if a shop is divergent is to ask how changes are made on production hosts, how those same changes are incorporated into the baseline build for new or replacement hosts, and how they are made on hosts that were down at the time the change was first deployed. If you get different answers, then the shop is divergent.

The symptoms of divergence include unpredictable host behavior, unscheduled downtime, unexpected package and patch installation failure, unclosed security vulnerabilities, significant time spent "firefighting," and high troubleshooting and maintenance costs.

The causes of divergence are generally that class of operations that create non-reproducible change. Divergence can be caused by ad-hoc manual changes, changes implemented by two independent automatic agents on the same host, and other unordered changes. Scripts which drive rdist, rsync, ssh, scp, [rdist, rsync, ssh] or other change agents as a push operation [bootstrap] are also a common source of divergence.

Convergence

Convergence (Figure 2) is the process most senior systems administrators first begin when presented with a divergent infrastructure. They tend to start by manually synchronizing some critical files across the diverged machines, then they figure out a way to do that automatically. Convergence is characterized by the configuration of live hosts moving towards an ideal baseline. By definition, all converging infrastructures are still diverged to some degree. (If an infrastructure maintains full compliance with a fully descriptive baseline, then it is congruent according to our definition, not convergent; see the 'Congruence' section.

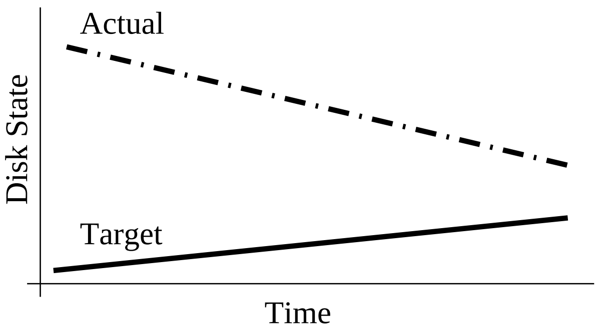

Figure 2: Convergence.

The baseline description in a converging infrastructure is characteristically an incomplete description of machine state. You can quickly detect convergence in a shop by asking how many files are currently under management control. If an approximate answer is readily available and is on the order of a few hundred files or less, then the shop is likely converging legacy machines on a file-by-file basis.

A convergence tool is an excellent means of bringing some semblance of order to a chaotic infrastructure. Convergent tools typically work by sampling a small subset of the disk – via a checksum of one or more files, for example – and taking some action in response to what they find. The samples and actions are often defined in a declarative or descriptive language that is optimized for this use. This emulates and preempts the firefighting behavior of a reactive human systems administrator – "see a problem, fix it." Automating this process provides great economies of scale and speed over doing the same thing manually.

Convergence is a feature of Mark Burgess' Computer Immunology principles [immunology]. His cfengine is in our opinion the best tool for this job [cfengine]. Simple file replication tools [sup, cvsup, rsync] provide a rudimentary convergence function, but without the other action semantics and fine-grained control that cfengine provides.

Because convergence typically includes an intentional process of managing a specific subset of files, there will always be unmanaged files on each host. Whether current differences between unmanaged files will have an impact on future changes is undecidable, because at any point in time we do not know the entire set of future changes, or what files they will depend on.

It appears that a central problem with convergent administration of an initially divergent infrastructure is that there is no documentation or knowledge as to when convergence is complete. One must treat the whole infrastructure as if the convergence is incomplete, whether it is or not. So without more information, an attempt to converge formerly divergent hosts to an ideal configuration is a never-ending process. By contrast, an infrastructure based upon first loading a known baseline configuration on all hosts, and limited to purely orthogonal and non-interacting sets of changes, implements congruence (defined in its own section). Unfortunately, this is not the way most shops use convergent tools such as cfengine.

The symptoms of a convergent infrastructure include a need to test all changes on all production hosts, in order to detect failures caused by remaining unforeseen differences between hosts. These failures can impact production availability. The deployment process includes iterative adjustment of the configuration tools in response to newly discovered differences, which can cause unexpected delays when rolling out new packages or changes. There may be a higher incidence of failures when deploying changes to older hosts. There may be difficulty eliminating some of the last vestiges of the ad-hoc methods mentioned in the section on 'Divergence.' Continued use of ad-hoc and manual methods virtually ensures that convergence cannot complete.

With all of these faults, convergence still provides much lower overall maintenance costs and better reliability than what is available in a divergent infrastructure. Convergence features also provide more adaptive self-healing ability than pure congruence, due to a convergence tool's ability to detect when deviations from baseline have occurred. Congruent infrastructures rely on monitoring to detect deviations, and generally call for a rebuild when they have occurred. We discuss the security reasons for this in the 'Congruence' section.

We have found apparent limits to how far convergence alone can go. We know of no previously divergent infrastructure that, through convergence alone, has reached congruence. This makes sense; convergence is a process of eliminating differences on an as-needed basis; the managed disk content will generally be a smaller set than the unmanaged content. In order to prove congruence, we would need to sample all bits on each disk, ignore those that are user data, determine which of the remaining bits are relevant to the operation of the machine, and compare those with the baseline.

In our experience, it is not enough to prove via testing that two hosts currently exhibit the same behavior while ignoring bit differences on disk; we care not only about current behavior, but future behavior as well. Bit differences that are currently deemed not functional, or even those that truly have not been exercised in the operation of the machine, may still affect the viability of future change directives. If we cannot predict the viability of future change actions, we cannot predict the future viability of the machine.

Deciding what bit differences are "functional" is often open to individual interpretation. For instance, do we care about the order of lines and comments in /etc/inetd.conf? We might strip out comments and reorder lines without affecting the current operation of the machine; this might seem like a non-functional change, until two years from now. After time passes, the lack of comments will affect our future ability to correctly understand the infrastructure when designing a new change. This example would seem to indicate that even non-machine-readable bit differences can be meaningful when attempting to prove congruence.

Unless we can prove congruence, we cannot validate the fitness of a machine without thorough testing, due to the uncertainties described in §8.25. In order to be valid, this testing must be performed on each production host, due to the factors described in §8.47. This testing itself requires either removing the host from production use or exposing untested code to users. Without this validation, we cannot trust the machine in mission-critical operation.

Congruence

Congruence (Figure 3) is the practice of maintaining production hosts in complete compliance with a fully descriptive baseline (see the section on 'Describing Disk State'). Congruence is defined in terms of disk state rather than behavior, because disk state can be fully described, while behavior cannot (§8.59).

By definition, divergence from baseline disk state in a congruent environment is symptomatic of a failure of code, administrative procedures, or security. In any of these three cases, we may not be able to assume that we know exactly which disk content was damaged. It is usually safe to handle all three cases as a security breach: correct the root cause, then rebuild.

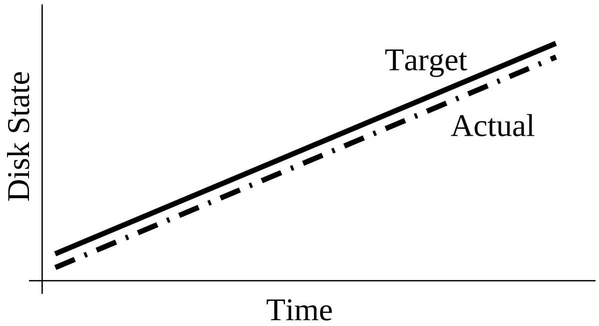

Figure 3: Congruence.

You can detect congruence in a shop by asking how the oldest, most complex machine in the infrastructure would be rebuilt if destroyed. If years of sysadmin work can be replayed in an hour, unattended, without resorting to backups, and only user data need be restored from tape, then host management is likely congruent.

Rebuilds in a congruent infrastructure are completely unattended and generally faster than in any other; anywhere from ten minutes for a simple workstation to two hours for a node in a complex high-availability server cluster (most of that two hours is spent in blocking sleeps while meeting barrier conditions with other nodes).

Symptoms of a congruent infrastructure include rapid, predictable, "fire-and-forget" deployments and changes. Disaster recovery and production sites can be easily maintained or rebuilt on demand in a bit-for-bit identical state. Changes are not tested for the first time in production, and there are no unforeseen differences between hosts. Unscheduled production downtime is reduced to that caused by hardware and application problems; firefighting activities drop considerably. Old and new hosts are equally predictable and maintainable, and there are fewer host classes to maintain. There are no ad-hoc or manual changes. We have found that congruence makes cost of ownership much lower, and reliability much higher, than any other method.

Our own experience and calculations show that the return-on-investment (ROI) of converting from divergence to congruence is less than 8 months for most organizations; see Figure 4. This graph assumes an existing divergent infrastructure of 300 hosts, 2%/month growth rate, followed by adoption of congruent automation techniques. Typical observed values were used for other input parameters. Automation tool rollout began at the 6-month mark in this graph, causing temporarily higher costs; return on this investment is in 5 months, where the manual and automatic lines cross over at the 11 month mark. Following crossover, we see a rapidly increasing cost savings, continuing over the life of the infrastructure. While this graph is calculated, the results agree with actual enterprise environments that we have converted. There is a CGI generator for this graph at Infrastructures.Org, where you can experiment with your own parameters.

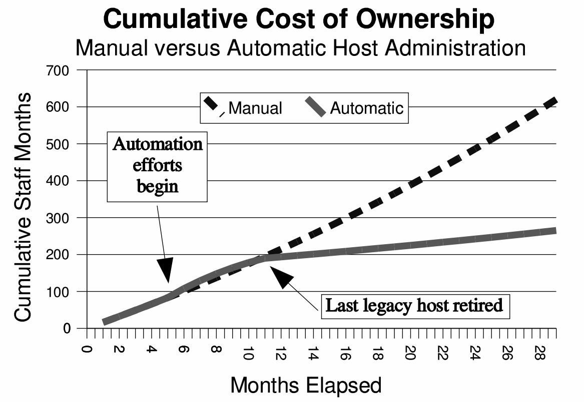

Figure 4: Cumulative costs for fully automated (congruent) versus manual administration.

Congruence allows us to validate a change on one host in a class, in an expendable test environment, then deploy that change to production without risk of failure. Note that this is useful even when (or especially when) there may be only one production host in that class.

A congruence tool typically works by maintaining a journal of all changes to be made to each machine, including the initial image installation. The journal entries for a class of machine drive all changes on all machines in that class. The tool keeps a lifetime record, on the machine's local disk, of all changes that have been made on a given machine. In the case of loss of a machine, all changes made can be recreated on a new machine by ''replaying'' the same journal; likewise for creating multiple, identical hosts. The journal is usually specified in a declarative language that is optimized for expressing ordered sets and subsets. This allows subclassing and easy reuse of code to create new host types; see 'Describing Disk State.'

There are few tools that are capable of the ordered lifetime journaling required for congruent behavior. Our own isconf (described in its own section) is the only specifically congruent tool we know of in production use, though cfengine, with some care and extra coding, appears to be usable for administration of congruent environments. We discuss this is more detail in the cfengine section.

We recognize that congruence may be the only acceptable technique for managing life-critical systems infrastructures, including those that:

- Influence the results of human-subject health and medicine experiments
- Provide command, control, communications, and intelligence (C^3I) for battlefield and weapons systems environments
- Support command and telemetry systems for manned aerospace vehicles, including spacecraft and national airspace air traffic control

Our personal experience shows that awareness of the risks of conventional host management techniques has not yet penetrated many of these organizations. This is cause for concern.

Ordered Thinking

We have found that designers of automated systems administration tools can benefit from a certain mindset:

> Think like a kernel developer, not an application programmer.

A good multitasking operating system is designed to isolate applications (and their bugs) from each other and from the kernel, and produce the illusion of independent execution. Systems administration is all about making sure that users continue to see that illusion.

Modern languages, compilers, and operating systems are designed to isolate applications programmers from ''the bare hardware'' and the low-level machine code, and enable object-oriented, declarative, and other high-level abstractions. But it is important to remember that the central processing unit(s) on a general-purpose computer only accepts machine-code instructions, and these instructions are coded in a procedural language. High-level languages are convenient abstractions, but are dependent on several layers of code to deliver machine language instructions to the CPU.

In reality, on any computer there is only one program; it starts running when the machine finishes power-on self test (POST), and stops when you kill the power. This program is machine language code, dynamically linked at runtime, calling in fragments of code from all over the disk. These ''fragments'' of code are what we conventionally think of as applications, shared libraries, device drivers, scripts, commands, administrative tools, and the kernel itself – all of the components that make up the machine's operating environment.

None of these fragments can run standalone on the bare hardware – they all depend on others. We cannot analyze the behavior of any application-layer tool as if it were a standalone program. Even kernel startup depends on the bootloader, and in some operating systems the kernel runtime characteristics can be influenced by one or more configuration files found elsewhere on disk.

This perspective is opposite from that of an application programmer. An application programmer ‘‘sees’’ the system as an axiomatic underlying support infrastructure, with the application in control, and the kernel and shared libraries providing resources. A kernel developer, though, is on the other side of the syscall interface; from this perspective, an application is something you load, schedule, confine, and kill if necessary.

On a UNIX machine, systems administration tools are generally ordinary applications that run as root. This means that they, too, are at the mercy of the kernel. The kernel controls them, not the other way around. And yet, we depend on automated systems administration tools to control, modify, and occasionally replace not only that kernel, but any and all other disk content. This presents us with the potential for a circular dependency chain.

A common misconception is that ‘‘there is some high-level tool language that will avoid the need to maintain strict ordering of changes on a UNIX machine.’’ This belief requires that the underlying runtime layers obey axiomatic and immutable behavioral laws. When using automated administration tools we cannot consider the underlying layers to be axiomatic; the administration tool itself perturbs those underlying layers; see the ‘Circular Dependencies’ section.

Inspection of high-level code alone is not enough. Without considering the entire system and its resulting machine language code, we cannot prove correctness. For example:

```
print "hello\n";
```

This looks like a trivial-enough Perl program; it ‘‘obviously’’ should work. But what if the Perl interpreter is broken? In other words, a conclusion of ‘‘simple enough to easily prove’’ can only be made by analyzing low-level machine language code, and the means by which it is produced.

‘‘Order Matters’’ because we need to ensure that the machine-language instructions resulting from a set of change actions will execute in the correct order, with the correct operands. Unless we can prove program correctness at this low level, we cannot prove the correctness of any program. It does no good to prove correctness of a higher-level program when we do not know the correctness of the lower runtime layers. If the high-level program can modify those underlying layers, then the behavior of the program can change with each modification. Ordering of those modifications appears to be important to our ability to predict the behavior of the high-level program. (Put simply, it is important to ensure that you can step off of the tree limb *before* you cut through it.)

The Need for Testing

Just as we urge tool designers to think like kernel developers (5), we urge systems administrators to think like operating systems vendors – because they are. Systems administration is actually *systems modification*; the administrator replaces binaries and alters configuration files, creating a combination which the operating system vendor has never tested. Since many of these modifications are specific to a single site or even a single machine, it is unreasonable to assume that the vendor has done the requisite testing. The systems administrator must perform the role of systems vendor, testing each unique combination – before the users do.

Due to modern society’s reliance on computers, it is unethical (and just plain bad business practice) for an operating system vendor to release untested operating systems without at least noting them as such. Better system vendors undertake a rigorous and exhaustive series of unit, system, regression, application, stress, and performance testing on each build before release, knowing full well that no amount of testing is ever enough (§8.9). They do this in their own labs; it would make little sense to plan to do this testing on customers’ production machines.

And yet, IT shops today habitually have no dedicated testing environment for validating changed operating systems. They deploy changes directly to production without prior testing. Our own experience and informal surveys show that greater than 95% of shops still do business this way. It is no wonder that reliability, security, and high availability are still major issues in IT.

We urge systems administrators to create and use dedicated testing environments, not inflict changes on users without prior testing, and consider themselves the operating systems vendors that they really are. We urge IT management organizations to understand and support administrators in these efforts; the return on investment is in the form of lower labor costs and much higher user satisfaction; see §8.42. Availability of a test environment enables the deployment of automated systems administration tools, bringing major cost savings; see Figure 4.

A test environment is useless until we have a means to replicate the changes we made in testing onto production machines. ‘‘Order matters’’ when we do this replication; an earlier change will often affect the outcome of a later change. This means that changes made to a test machine must later be ‘‘replayed’’ in the same order on the machine’s production counterpart; see §8.45.

Testing costs can be greatly reduced by limiting the number of unique builds produced; this holds true for both vendors and administrators. This calls for careful management of changes and host classes in an

IT environment, with an intent of limiting proliferation of classes; see §8.41.

Note that use of open-source operating systems does not remove the need for local testing of local modifications. In any reasonably complex infrastructure, there will always be local configuration and non-packaged binary modifications which the community cannot have previously exercised. We prefer open source; we do not expect it to relieve us from our responsibilities though.

Ordering HOWTO

Automated systems administration is very straightforward. There is only one way for a user-side administrative tool to change the contents of disk in a running UNIX machine – the syscall interface. The task of automated administration is simply to make sure that each machine's kernel gets the right system calls, in the right order, to make it be the machine you want it to be.

Describing Disk State

If there are N bits on a disk, then there are 2^N possible disk states. In order to maintain the baseline host description needed for congruent management, we need to have a way to describe any arbitrary disk state in a highly compressed way, preferably in a human-readable configuration file or script. For the purposes of this description, we neglect user data and log files – we want to be able to describe the root-owned and administered portions of disk. ''Order Matters'' whether creating or modifying a disk:

> A concise and reliable way to describe any arbitrary state of a disk is to describe the procedure for creating that state.

This procedure will include the initial state (bare-metal build) of the disk, followed by the steps used to change it over time, culminating in the desired state. This procedure must be in writing, preferably in machine-readable form. This entire set of information, for all hosts, constitutes the baseline description of a congruent infrastructure. Each change added to the procedure updates the baseline; see the 'Congruence' section.

There are tools which can help you maintain and execute this procedure; see the 'Example Tools and Techniques' section.

While it is conceivable that this procedure could be a documented manual process, executing these steps manually is tedious and costly at best. (Though we know of many large mission-critical shops which try.) It is generally error-prone. Manual execution of complex procedures is one of the best methods we know of for generating divergence (described in its own section).

The starting state (bare-metal install) description of the disk may take the form of a network install tool's configuration file, such as that used for Solaris Jumpstart or RedHat Kickstart. The starting state might instead be a bitstream representing the entire initial content of the disk (usually a snapshot taken right after install from vendor CD). The choice of which of these methods to use is usually dependent on the vendor-supplied install tool – some will support either method, some require one or the other.

How to Break an Enterprise

A systems administrator, whether a human or a piece of software (§8.36), can easily break an enterprise infrastructure by executing the right actions in the wrong order. In this section, we will explore some of the ways this can happen.

Right Commands, Wrong Order

First we will cover a trivial but devastating example that is easily avoided. This once happened to a colleague while doing manual operations on a machine. He wanted to clean out the contents of a directory which ordinarily had the development group's source code NFS mounted over top of it. Here is what he wanted to do:

```
umount /apps/src
cd /apps/src
rm -rf .
mount /apps/src
```

Here's what he actually did:

```
umount /apps/src
   ...umount fails, directory in use;
   while resolving this, his pager goes
   off, he handles the interrupt, then...
cd /apps/src
rm -rf .
```

Needless to say, there had also been no backup of the development source tree for quite some time...

In this example, ''correct order'' includes some concept of sufficient error handling. We show this example because it highlights the importance of a default behavior of ''halt on error'' for automatic systems administration tools. Not all tools halt on error by default; isconf does (see its section).

Right Packages, Wrong Order

We in the UNIX community have long accused Windows developers of poor library management, due to the fact that various Windows applications often come bundled with differing version of the same DLLs. It turns out that at least some UNIX and Linux distributions appear to suffer from the same problem.

Jeffrey D'Amelia and John Hart [hart] demonstrated this in the case of RedHat RPMs, both official and contributed. They showed that the order in which you install RPMs can matter, even when there are no applicable dependencies specified in the package. We don't assume that this situation is restricted to RPMs only – any package management system should be susceptible to this problem. An interesting study would be to investigate similar overlaps in vendor-supplied packages for commercial UNIX distributions.

Detecting this problem for any set of packages involves extensive analysis by talented persons. In the case of [hart], the authors developed a suite of global analysis tools, and repeatedly downloaded and unpacked thousands of RPMs. They still only saw ‘‘the tip of the iceberg’’ (their words). They intentionally ignored the actions of postinstall scripts, and they had not yet *executed* any packaged code to look for behavioral interactions.

Avoiding the problem is easier; install the packages, record the order of installation, test as usual, and when satisfied with testing, install the same packages in the same order on production machines.

While we’ve used packages in this example, we’d like to remind the reader that these considerations apply not only to package installation but any other change that affects the root-owned portions of disk.

Circular Dependencies

There is a ‘‘chicken and egg’’ or bootstrapping problem when updating either an automated systems administration tool (ASAT) or its underlying foundations (§8.40). Order is important when changes the tool makes can change the ability of the tool to make changes.

For example, cfengine version 2 includes new directives available for use in configuration files. Before using a new configuration file, the new version of cfengine needs to be installed. The new client is named ‘cfagent’ rather than ‘cfengine,’ so wrapper scripts and crontab entries will also need to be updated, and so on.

For fully automated operation on hundreds or thousands of machines, we would like to be able to upgrade cfengine under the control of cfengine (§8.46). We want to ensure that the following actions will take place on all machines, including those currently down:

1. fetch new configuration file containing the following instructions
2. install new cfagent binary
3. run cfkey to generate key pair
4. fetch new configuration file containing version 2 directives
5. update calling scripts and crontab entries

There are several ordering considerations here. We won’t know that we need the new cfagent binary until we do step 1. We shouldn’t proceed with step 4 until we know that 2 and 3 were successful. If we do 5 too early, we may break the ability for cfengine to operate at all. If we do step 4 too early and try to run the resulting configuration file using the old version of cfengine, it will fail.

While this example may seem straightforward, implementing it in a language which does not by default support deterministic ordering requires much use of conditionals, state chaining, or equivalent. If this is the case, then code flow will not be readily apparent, making inspection and edits error-prone. Infrastructure automation code runs as root and has the ability to stop work across the entire enterprise; it needs to be simple, short, and easy for humans to read, like security-related code paths in tools such as PGP or ssh.

If the tool’s language does not support ‘‘halt on error’’ by default, then it is easy to inadvertently allow later actions to take place when we would have preferred to abort. Going back to our cfengine example, if we can easily abort and leave the cfengine version 1 infrastructure in place, then we can still use version 1 to repair the damage.

Other Sources of Breakage

There are many other examples we could show, some including multi-host ‘‘barrier’’ problems. These include:

- Updating ssh to openssh on hundreds of hosts and getting the authorized_keys and/or protocol version configuration out of order. This can greatly hinder further contact with the target hosts. Daniel Hagerty [hagerty] ran into this one; many of us have been bitten by this at some point.
- Reconfiguring network routes or interfaces while communicating with the target device via those same routes or interfaces. Ordering errors can prevent further contact with the target, and often require a physical visit to resolve. This is especially true if the target is a workstation with no remote serial console access. Again, most readers have had this happen to them.

Example Tools and Techniques

While there are many automatic systems administration tools (ASAT) available, the two we are most familiar with are cfengine and our own isconf [cfengine, isconf]. In this section, we will look at these two tools from the perspective of Turing equivalence (§8), with a focus on how each can be used deterministically.

In general, some of the techniques that seem to work well for the design and use of most ASATs include:

- Keep the ‘‘Turing tape’’ a finite size by holding the network content constant (§8.23), or versioning it using CVS or another version control tool [cvs, bootstrap]. This helps prevent some of the more insidious behaviors that are a potential in self-modifying machines (§8.40).
- Continuing in that vein, when using distributed package repositories such as the public Debian [debian] package server infrastructure, always

specify version numbers when automating the installation of packages, rather than let the package installation tool (in Debian's case apt-get) select the latest version. If you do not specify the package version, then you may introduce divergence (described in its own section). This risk varies, of course, depending on your choice of 'stable' or 'unstable' distribution, though we suspect it still applies in 'stable,' especially when using the 'security' packages. It certainly applies in all cases when you need to maintain your own kernel or kernel modules rather than using the distributed packages.

We have experienced this repeatedly – machines which built correctly the first time with a given package list will not rebuild with the same package list a few weeks later, due to package version changes on the public servers, and resulting unresolved incompatibilities with local conditions and configuration file contents. Remember, your hosts are unique in the world – there are likely no others like them. Package maintainers cannot be expected to test every configuration, especially yours. You must retain this responsibility; see 'The Need for Testing.'

We use Debian in this example because it is a distribution we like a lot; note that other package distribution and installation infrastructures, such as the RedHat up2date system, also have this problem.

- Expect long dependency or sequence chains when building enterprise infrastructures. If an ASAT can easily support encapsulation and ordering of 10, 50, or even 100 complex atomic actions in a single chain, then it is likely capable of fully automated administration of machines, including package, kernel, build, and even rebuild management. If the ASAT is cumbersome to use when chains become only two or three actions deep, then it is likely most suited for configuration file management, not package, binary, or kernel manipulation.

ISconf

As we mentioned in the introduction, isconf originally began life as a quick hack. Its basic utility has proven itself repeatedly over the last eight *years,* and as adoption has grown it is currently managing more production infrastructures than we are personally aware of.

While we show some ISconf makefile examples here, we do not show any example of the top-level configuration file which drives the environment and targets for 'make.' It is this top-level configuration file, and the scripts which interpret it, which are the core of ISconf and enable the typing or classing of hosts. These top-level facilities also are what governs the actions ISconf is to take during boot versus cron or other execution contexts. More information and code is available at ISconf.org and Infrastructures.Org.

We also do not show here the network fetch and update portions of ISconf, and the way that it updates its own code and configuration files at the beginning of each run. This default behavior is something that we feel is important in the design of any automated systems administration tool. If the tool does not support it, end-users will have to figure out how to do it themselves, reducing the usability of the tool.

ISconf Version 2

Version 2 of ISconf was a late-90's rewrite to clean up and make portable the lessons learned from version 1. As in version 1, the code used was Bourne shell, and the state engine used was 'make.'

In Listing 1, we show a simplified example of Version 2 usage. While examples related to this can be found in [hart] and in our own makefiles, real-world usage is usually much more complex than the example shown here. We've contrived this one for clarity of explanation.

In this contrived example, we install two packages which we have not proven orthogonal. We in fact do not wish to take the time to detect whether or not they are orthogonal, due to the considerations expressed in §8.58. We may be tool users, rather than tool designers, and may not have the skillset to determine orthogonality, as in §8.54.

These packages might both affect the same shared library, for instance. Again according to [hart] and our own experience, it is not unusual for two packages such as these to list neither as prerequisites, so we might gain no ordering guidance from the package headers either.

In other words, all we know is that we installed package 'foo,' tested and deployed it to production, and then later installed package 'bar,' tested it and deployed. These installs may have been weeks or months apart. All went well throughout, users were happy, and we have no interest in unpacking and analyzing the contents of these packages for possible reordering for any reason; we've gone on to other problems.

Because we know this order works, we wish for these two packages, 'foo' and 'bar,' to be installed in the same order on every future machine in this class. This makefile will ensure that; the touch $@ command at the end of each stanza will prevent this stanza from being run again. The ISconf code always changes to the timestamps directory before starting 'make' (and takes other measures to constrain the normal behavior of 'make,' so that we never try to "rebuild" this target either).

The class name in this case (Listing 1) is 'Block12.' You can see that 'Block12' is also made up of many other packages; we don't show the makefile stanzas for these here. These packages are listed as prerequisites to 'Block12,' in chronological order. Note that we only want to add items to the end of this list, not the middle, due to the considerations expressed in §8.49.

In this example, even though we take advantage of the Debian package server infrastructure, we specify the version of package that we want, as in the introduction to the 'Example Tools and Techniques' section. We also use a caching proxy when fetching Debian packages, in order to speed up our own builds and reduce the load on the Debian servers to a minimum.

Note that we get "halt-on-error" behavior from 'make,' as we wished for in 'Right Commands, Wrong Order.' If any of the commands in the 'foo' or 'bar' sections exit with a non-zero return code, then 'make' aborts processing immediately. The 'touch' will not happen, and we normally configure the infrastructure such that the ISconf failure will be noticed by a monitoring tool and escalated for resolution. In *practice,* these failures very rarely occur in production; we see and fix them in test. Production failures, by the definition of congruence (see its special section), usually indicate a systemic, security, or organizational problem; we don't want them fixed without human investigation.

```
Block12: cvs ntp foo lynx wget \
   serial_console bar sudo mirror_rootvg
foo:
apt-get -y install foo=0.17-9
touch $@
bar:
apt-get -y install bar=1.0.2-1
echo apple pear > /etc/bar.conf
touch $@
. . .
```

Listing 1: ISconf makefile package ordering example.

ISconf Version 3

ISconf version 3 was a rewrite in Perl, by Luke Kaines. This version adds more "lessons learned," including more fine-grained control of actions as applied to target classes and hosts. There are more layers of abstraction between the administrator and the target machines; the tool uses various input files to generate intermediate and final file formats which eventually are fed to 'make.'

One feature in particular is of special interest for this paper. In ISconf version 2, the administrator still had the potential to inadvertently create unordered change by an innocent makefile edit. While it is possible to avoid this with foreknowledge of the problem, version 3 uses timestamps in an intermediate file to prevent it from being an issue.

The problem which version 3 fixes can be reproduced in version 2 as follows: Refer to (listing 1). If both 'foo' and 'bar' have been executed (installed) on production machines, then the administrator adds 'baz' as a prerequisite to 'bar,' then this would qualify as "editing prior actions" and create the divergence described in §8.49.

ISconf version 3, rather than using a human-edited makefile, reads other input files which the administrator maintains, and generates intermediate and final files which include timestamps to detect the problem and correct the ordering.

ISconf version 4

ISconf version 4, currently in prototype, represents a significant architectural change from versions 1 through 3. If the current feature plan is fully implemented, version 4 will enable cross-organizational collaboration for development and use of ordered change actions. A core requirement is decentralized development, storage, and distribution of changes. It will enable authentication and signing, encryption, and other security measures. We are likely to replace 'make' with our own state engine, continuing the migration begun in version 3. See ISconf.Org for the latest information.

Baseline Management

In the 'Congruence' section, we discussed the concept of maintaining a fully descriptive baseline for congruent management. In discussing disk state descriptions, we discussed in general terms how this might be done. In this section, we will show how we do it in isconf.

First, we install the base disk image (as previously), usually using vendor-supplied network installation tools. We discuss this process more in [bootstrap]. We might name this initial image 'Block00.' Then we use the process we mentioned in the isconf section to apply changes to the machine over the course of its life. Each change we add updates our concept of what is the 'baseline' for that class of host.

As we add changes, any new machine we build will need to run isconf longer on first boot, to add all of the accumulated changes to the Block00 image. After about forty minutes' worth of changes have built up on top of the initial image, it helps to be able to build one more host that way, set the hostname/IP to 'baseline,' cut a disk image of it, and declare that new image to be the new baseline. This infrequent snapshot or checkpoint not only reduces the build time of future hosts, but reduces the rebuild time and chance of error in rebuilding existing hosts – we always start new builds from the latest baseline image.

In an isconf makefile, this whole process is reflected as in Listing 2. Note that whether we cut a new image and start the next install from that, or if we just pull an old machine off the shelf with a Block00 image and plug it in, we'll still end up with a Block20 image with apache and a 2.2.12 kernel, due to the way the makefile prerequisites are chained.

This example shows a simple, linear build of successive identical hosts with no "branching" for different host classes. Classes add slightly more complexity to the makefile. They require a top-level configuration file to define the classes and target them to the right hosts, and they require wrapper script code to read the config file.

There is a little more complexity to deal with things that should only happen at boot, and that can happen when cron runs the code every hour or so. There are examples of all of this in the isconf-2i package available from ISconf.Org.

Cfengine

Cfengine is likely the most popular purpose-built tool for automated systems administration today. The cfengine language was optimized for dynamic prerequisite analysis rather than long, deterministic ordered sets.

While the cfengine language wasn't specifically optimized for ordered behavior, it is possible to achieve this with extra work. It should be possible to greatly reduce the amount of effort involved, by using some tool to generate cfengine configuration files from makefile-like (or equivalent) input files. One good starting point might be Tobias Oetiker's *TemplateTree II* [oetiker].

Automatic generation of cfengine configuration files appears to be a near-requirement if the tool is to be used to maintain congruent infrastructures; the class and action-type structures tend to get relatively complex rather fast if congruent ordering, rather than convergence, is the goal.

Other gains might be made from other features of cfengine; we have made progress experimenting with various helper modules, for instance. Another technique that we have put to good use is to implement atomic changes using very small cfengine scripts, each equivalent to an ISconf makefile stanza. These scripts we then drive within a deterministically ordered framework.

In the cfengine version 2 language there are new features, such as the FileExists() evaluated class function, which may reduce the amount of code. So far, based on our experience over the last few years in trial attempts, it appears that a cfengine configuration file that does the same job as an ISconf makefile would still need anywhere from 2-3 times the number of lines of code. We consider this an open and evolving effort though – check the cfengine.org and Infrastructures.Org websites for the latest information.

Brown/Traugott Turing Equivalence

> If it should turn out that the basic logics of a machine designed for the numerical solution of differential equations coincide with the logics of a machine intended to make bills for a department store, I would regard this as the most amazing coincidence that I have ever encountered.
> – Howard Aiken, founder of Harvard's Computer Science department and architect of the IBM/Harvard Mark I.

Turing equivalence in host management appears to be a new factor relative to the age of the computing industry. The downsizing of mainframe installations and distribution of their tasks to midrange and desktop machines by the early 1990's exposed administrative challenges which have taken the better part of a decade for the systems administration community to understand, let alone deal with effectively.

Older computing machinery relied more on dedicated hardware rather than software to perform many administrative tasks. Operating systems were limited in their ability to accept changes on the fly, often requiring recompilation for tasks as simple as adding terminals or changing the time zone. Until recently, the most popular consumer desktop operating system still required a reboot when changing IP address.

In the interests of higher uptime, modern versions of UNIX and Linux have eliminated most of these issues; there is very little software or configuration management that cannot be done with the machine "live." We have evolved to a model that is nearly equivalent to that of a Universal Turing Machine, with all of its benefits and pitfalls. To avoid this equivalence, we would need to go back to shutting operating systems down in order to administer them. Rather than go back, we should seek ways to go further forward; understanding Turing equivalence appears to be a good next step.

This situation may soon become more critical, with the emergence of "soft hardware." These systems use Field-Programmable Gate Arrays to emulate

```
# 01 Feb 97 - Block00 is initial disk install from vendor cd,
# with ntp etc. added later
Block00: ntp cvs lynx ...
# 15 Jul 98 - got tired of waiting for additions to Block00 to build,
# cut new baseline image, later add ssh etc.
Block10: Block00 ssh ...
# 17 Jan 99 - new baseline again, later add apache, rebuild kernel, etc.
Block20: Block10 apache kernel-2.2.12 ...
```

Listing 2: Baseline management in an ISconf makefile.

dedicated processor and peripheral hardware. Newer versions of these devices can be reprogrammed, while running, under control of the software hosted on the device itself [xilinx]. This will bring us the ability to modify, for instance, our own CPU, using high-level automated administration tools. Imagine not only accidentally unconfiguring your Ethernet interface, but deleting the circuitry itself...

We have synthesized a thought experiment to demonstrate some of the implications of Turing equivalence in host management, based on our observations over the course of several years. The description we provide here is not as rigorous as the underlying theories, and much of it should be considered as still subject to proof. We do not consider ourselves theorists; it was surprising to find ourselves in this territory. The theories cited here provided inspiration for the thought experiment, but the goal is practical management of UNIX and other machines. We welcome any and all future exploration, pro or con.

In the following description of this thought experiment, we will develop a model of system administration starting at the level of the Turing machine. We will show how a modern self-administered machine is equivalent to a Turing machine with several tapes, which is in turn equivalent to a single-tape Turing machine. We will construct a Turing machine which is able to update its own program by retrieving new instructions from a network-accessible tape. We will develop the idea of configuration management for this simpler machine model, and show how problems such as circular dependencies and uncertainty about behavior arise naturally from the nature of computation.

We will discuss how this Turing machine relates to a modern general-purpose computer running an automatic administration tool. We will introduce the implications of the self-modifying code which this arrangement allows, and the limitations of inspection and testing in understanding the behavior of this machine. We will discuss how ordering of changes affects this behavior, and how deterministically ordered changes can make its behavior more deterministic.

We will expand beyond single machines into the realm of distributed computing and management of multiple machines, and their associated inspection and testing costs. We will discuss how ordering of changes affects these costs, and how ordered change apparently provides the lowest cost for managing an enterprise infrastructure.

Readers who are interested in applied rather than mathematical or theoretical arguments may want to review the previous section or skip this one altogether.

8.1 – A Turing machine (Figure 5) reads bits from an infinite tape, interprets them as data according to a hardwired program and rewrites portions of the tape based on what it finds. It continues this cycle until it reaches a completion state, at which time it halts [turing].

8.2 – Because a Turing machine's program is hardwired, it is common practice to say that the program *describes* or *is* the machine. A Turing machine's program is stated in a descriptive language which we will call the *machine language*. Using this language, we describe the actions the machine should take when certain conditions are discovered. We will call each atom of description an *instruction*. An example instruction might say:

> If the current machine state is 's3,' and the tape cell at the machine's current head position contains the letter 'W,' then change to state 's7,' overwrite the 'W' with a 'P,' and move the tape one cell to the right.

Each instruction is commonly represented as a quintuple; it contains the letter and current state to be matched, as well as the letter to be written, the tape movement command, and the new state. The instruction we described above would look like:

```
s3,W → s7,P,r
```

Note that a Turing machine's language is in no way algorithmic; the order of quintuples in a program listing is unimportant; there are no branching, conditional, or loop statements in a Turing machine program.

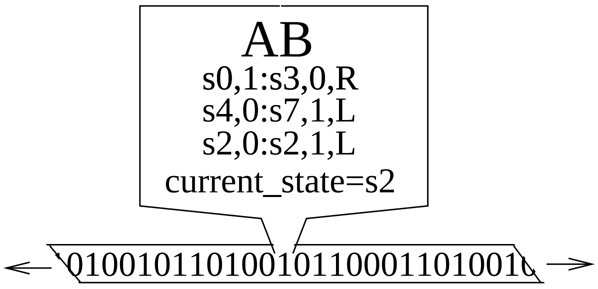

Figure 5: Turing machine block diagram; the machine reads and writes an infinite tape and updates an internal state variable based on a hardwired or stored ruleset.

8.3 – The content of a Turing tape is expressed in a language that we will call the *input language*. A Turing machine's program is said to either *accept* or *reject* a given input language, if it halts at all. If our Turing machine halts in an accept state, (which might actually be a state named 'accept') then we know that our program is able to process the data and produce a valid result – we have validated our input against our machine. If our Turing machine halts because there is no instruction that matches the current combination of state and cell content (§8.2), then we know that our program is unable to process this input, so we reject. If we never halt, then we cannot state a result, so we cannot validate the input or the machine.

8.4 – A Universal Turing Machine (UTM) is able to emulate any arbitrary Turing machine. Think of this as running a Turing ''virtual machine'' (TVM) on top of a host UTM. A UTM's machine language program (§8.2) is made up of instructions which are able to read and execute the TVM's machine language instructions. The TVM's machine language instructions are the UTM's input data, written on the input tape of the UTM alongside the TVM's own input data (Figure 6).

Any multiple-tape Turing machine can be represented by a single-tape Turing machine, so it is equally valid to think of our Universal Turing Machine as having two tapes; one for TVM program, and the other for TVM data.

A Universal Turing Machine appears to be a useful model for analyzing the theoretical behavior of a ''real'' general-purpose computer; basic computability theory seems to indicate that a UTM can solve any problem that a general-purpose computer can solve [church].

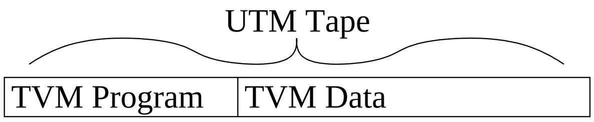

Figure 6: The tape of a Universal Turing Machine (UTM) stores the program and data of a hosted Turing Virtual Machine (TVM).

8.5 – Further work by John von Neumann and others demonstrated one way that machines could be built which were equivalent in ability to Universal Turing Machines, with the exception of the infinite tape size [vonneumann]. The von Neumann architecture is considered to be a foundation of modern general purpose computers [godfrey].

8.6 – As in von Neumann's ''stored program'' architecture, the TVM program and data are both stored as rewritable bits on the UTM tape (§8.4, Figure 6). This arrangement allows the TVM to change the machine language instructions which describe the TVM itself. If it does so, our TVM enjoys the advantages (and the pitfalls) of self-modifying code [nordin].

8.7 – There is no algorithm that a Turing machine can use to determine whether another specific Turing machine will halt for a given tape; this is known as the ''halting problem.'' In other words, Turing machines can contain constructions which are difficult to validate. This is not to say that every machine contains such constructions, but that that an arbitrary machine and tape chosen at random has some chance of containing one.

8.8 – Note that, since a Turing machine is an imaginary construct [turing], our own brain, a pencil, and a piece of paper are (theoretically) sufficient to work through the tape, producing a result if there is one. In other words, we can inspect the code and determine what it would do. There may be tools and algorithms we can use to assist us in this [laitenberger]. We are not guaranteed to reach a result though – in order for us to know that we have a valid machine and valid input, we must halt and reach an accept state. Inspection is generally considered to be a form of testing.

Inspection has a cost (which we will use later): $C_{inspect}$.

This cost includes the manual labor required to inspect the code, any machine time required for execution of inspection tools, and the manual labor to examine the tool results.

8.9 – There is no software testing algorithm that is guaranteed to ensure fully reliable program operation across all inputs – there appears to be no theoretical foundation for one [hamlet]. We suspect that some of the reasons for this may be related to the halting problem (§8.7), Gödel's incompleteness theorem [godel], and some classes of computational intractability problems, such as the Traveling Salesman and NP completeness [greenlaw, garey, brookshear, dewdney].

In practice, we can use multiple test runs to explore the input domain via a parameter study, equivalence partitioning [richardson], cyclomatic complexity analysis [mccabe], pseudo-random input, or other means. Using any or all of these methods, we may be able to build a confidence level for predictability of a given program. Note that we can never know when testing is complete, and that testing only proves *incorrectness* of a program, not correctness.

Testing cost includes the manual labor required to design the test, any machine time required for execution, and the manual labor needed to examine the test results: C_{test}.

8.10 – For software testing to be meaningful, we must also ensure code coverage. Code coverage requirements are generally determined through some form of inspection (§8.8), with or without the aid of tools. Coverage information is only valid for a fixed program – even relatively minor code changes can affect code coverage information in unpredictable ways [elbaum]. We must repeat testing (§8.9) for every variation of program code.

To ensure code coverage, testing includes the manual labor required to inspect the code, any machine time required for execution of the coverage tools and tests, and the manual labor needed to examine the test results. Because testing for coverage includes code inspection, we know that testing is more expensive than inspection alone: $C_{test} > C_{inspect}$.

8.11 – Once we have found a UTM tape that produces the result we desire, we can make many copies of that tape, and run them through many identical Universal Turing Machines simultaneously. This will produce many simultaneous, identical results. This is not very interesting – what we really want to be able to do is hold the TVM program portion of the tape constant while changing the TVM data portion, then feed those differing tapes through identical machines. The latter arrangement can give us a form of distributed or parallel computing.

8.12 – Altering the tapes (§8.11) presents a problem though. We cannot in advance know whether these altered tapes will provide valid results, or even reach completion. We can exhaustively test the same program with a wide variety of sample inputs, validating each of these. This is fundamentally a time-consuming, pseudo-statistical process, due to the iterative validations normally required. And it is not a complete solution (§8.9).

8.13 – If we for some reason needed to solve slightly different problems with the distributed machines in §8.11, we may decide to use slightly different programs in each machine, rather than add functionality to our original program. But using these unique programs would greatly worsen our testing problem. We would not only need to validate across our range of input data (§8.9), but we would also need to repeat the process for each program variant (§8.10). We know that testing many unique programs will be more expensive than testing one: $C_{many} > C_{test}$.

8.14 – It is easy to imagine a Turing Machine that is connected to a network, and which is able to use the net to fetch data from tapes stored remotely, under program control. This is simply a case of an multiple-tape Turing machine, with one or more of the tapes at the other end of a network connection.

8.15 – Building on §8.14, imagine a Turing Virtual Machine (TVM) running on top of a networked Universal Turing Machine (UTM) (§8.4). In this case, we might have 3 tapes; one for the TVM program, one for the TVM data, and a third for the remote network tape. It is easy to imagine a sequence of TVM operations which involve fetching a small amount of data from the remote tape, and storing it on the local program tape as additional and/or replacement TVM instructions (§8.6). We will name the old TVM instruction set **A**. The set of fetched instructions we will name **B**, and the resulting merger of the two we will name **AB**. Note that some of the instructions in **B** may have replaced some of those in **A** (Figure 7). Before the fetch, our TVM could be described (§8.2) as an **A** machine, after the fetch we have an **AB** machine – the TVM's basic functionality has changed. It is no longer the same machine.

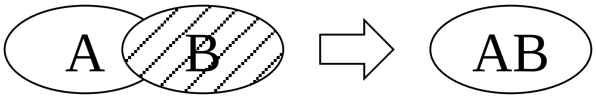

Figure 7: Instruction set B partially overlays instruction set A, creating set AB.

8.16 – Note that, if any of the instructions in set **B** replace any of those in set **A**, (§8.15), then the order of loading these sets is important. A TVM with the instruction set **AB** will be a different machine than one with set **BA** (Figure 8).

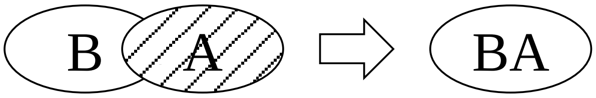

Figure 8: Instruction set BA is created by loading B before A; A partially overlays B this time.

8.17 – It is easy to imagine that the TVM in §8.15 could later execute an instruction from set **B**, which could in turn cause the machine to fetch another set of one or more instructions in a set we will call **C**, resulting in an **ABC** machine:

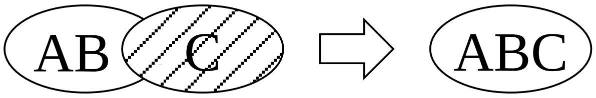

Figure 9: If

8.18 – After each fetch described in §8.17, the local program and data tapes will contain bits from (at least) three sources: the new instruction set just copied over the net, any old instructions still on tape, and the data still on tape from ongoing execution of all previous instructions.

8.19 – The choice of next instruction to be fetched from the remote tape in §8.17 can be calculated by the currently available instructions on the local program tape, based on current tape content (§8.18).

8.20 – The behavior of one or more new instructions fetched in (§8.17) can (and usually will) be influenced by other content on the local tapes (§8.18). With careful inspection and testing we can detect some of the ways content will affect instruction fetches, but due to the indeterminate results of software testing (§8.9), we may never know if we found all of them.

8.21 – Let us go back to our three TVM instruction sets, **A**, **B**, and **C** (§8.17). These were loaded over the net and executed using the procedure described in §8.19. Assume we start with blank local program and data tapes. Assume our UTM is hardwired to fetch set **A** if the local program tape is found to be blank. If we then run the TVM, **A** can collect data over the net and begin processing it. At some point later, **A** can cause set **B** to be loaded. Our local tapes will now contain the TVM data resulting from execution of **A**, and the new TVM machine instructions **AB**. If the TVM later loads **C**, our program tape will contain **ABC**.

8.22 – If the networked UTM machine constructed in §8.21 always starts with the same (blank) local tape content, and the remote tape content does not change, then we can demonstrate that an **A** TVM will always evolve to an **AB**, then an **ABC** machine, before halting and producing a result.

8.23 – Assuming the network-resident data never changes, we can rebuild our networked UTM at any time and restore it to any prior state by clearing the local tapes, resetting the machine state, and restarting execution with the load of **A** (§8.21). The machine will execute and produce the same intermediate and final results as it did before, as in §8.22.

8.24 – If the network-resident data does change, though, we may not be able to rebuild to an identical state. For example, if someone were to alter the network-resident master copy of the **B** instruction set after we last fetched it, then it may no longer produce the same intermediate results and may no longer fetch C (§8.19). We might instead halt at **AB**.

8.25 – Without careful (and possibly intractable) inspection (§8.8), we cannot prove in advance whether an **BCA** or **CAB** machine can produce the same result as an **ABC** machine. It is *possible* that these, or other, variations might yield the same result. We can validate the result for a given input (§8.3). We would also need to do iterative testing (§8.12) to demonstrate that multiple inputs would produce the same result. Our cost of testing multiple or partially ordered sequences is greater than that required to test a single sequence: $C_{partial} > C_{test}$.

8.26 – If the behavior of any instruction from **B** in (§8.22) is in any way dependent on other content found on tape (§8.18, §8.19, §8.20), then we can expect our TVM to behave differently if we load **B** before loading **A** (§8.16). We cannot be certain that a UTM loaded with only a **B** instruction set will accept the input language, or even halt, until after we validate it (§8.3).

8.27 – We might want to rollback from the load or execution of a new instruction set. In order to do this, we would need to return the local program and data tape to a previous content. For example, if machine **A** executes and loads **B**, our instruction set will now be **AB**. We might rollback by replacing our tape with the **A** copy.

8.28 – Due to (§8.26), it is not safe to try to rollback the instruction set of machine **AB** to recreate machine **A** by simply removing the **B** instructions. Some of **B** may have replaced **A**. The **AB** machine, while executing, may have even loaded **C** already (§8.21), in which case you won't end up with **A**, but with **AC**. If the **AB** machine executed for any period of time, it is likely that the input data language now on the data tape is only acceptable to an **AB** machine – an **A** machine might reject it or fail to halt (§8.3). The only safe rollback method seems to be something similar to (§8.27).

8.29 – It is easy to imagine an automatic process which conducts a rollback. For example, in §8.27, machine **AB** itself might have the ability to clear its own tapes, reset the machine state, and restart execution at the beginning of **A**, as in §8.23.

8.30 – But the system described in §8.29 will loop infinitely. Each time **A** executes, it will load **B**, then **AB** will execute and reset the local tapes again. In practice, a human might detect and break this loop; to represent this interaction, we would need to add a fourth tape, representing the user detection and input data.

8.31 – It is easy to imagine an automatic process which emulates a rollback while avoiding loops, without requiring the user input tape in §8.30. For example, instruction set **C** might contain the instructions from **A** that **B** overlaid. In other words, installing **C** will "rollback" **B**. Note that this is not a true rollback; we never return to a tape state that is completely identical to any previous state. Although this is an imperfect solution, it is the best we seem to be able to do without human intervention.

8.32 – The loop in §8.30 will cause our UTM to never reach completion – we will not halt, and cannot validate a result (§8.3). A method such as (§8.31) can prevent a rollback-induced loop, but is not a true rollback – we never return to an earlier tape content. If these, or similar, methods are the only ones available to us, it appears that program-controlled tape changes must be monotonic – we cannot go back to a previous tape content under program control, otherwise we loop.

> You are in a maze of twisty little passages, all alike. – Will Crowther's "Adventure"

8.33 – Let us now look at a conventional application program, running as an ordinary user on a correctly configured UNIX host. This program can be loaded from disk into memory and executed. At no time is the program able to modify the "master" copy of itself on disk. An application program typically executes until it has output its results, at which time it either sleeps or halts. This application is equivalent to a fixed-program Turing machine (§8.1) in the following ways: Both can be validated for a given input (§8.3) to prove that they will produce results in a finite time and that those results are correct. Both can be tested over a range of inputs (§8.9) to build confidence in their reliability. Neither can modify their own executable instructions; in the UNIX machine they are protected by filesystem permissions; in the Turing machine they are hardwired. (We stipulate that there are some ways in which (§8.33 and §8.1) are not equivalent – a Turing machine has a theoretically infinite tape, for instance.)

8.34 – We can say that the application program in §8.33 is running on top of an *application virtual machine* (AVM). If the application is written in Java, for example, the AVM consists of the Java Virtual Machine. In Perl, the AVM is the Perl bytecode VM. For C programs, the AVM is the kernel system call interface. Low-level code in shared libraries used by a C program uses the same syscall interface to interact with the hardware – shared libraries are part of the C AVM. A Perl program can load modules – these become part of the program's AVM. A C or Perl program that uses the system() or exec() system calls relies on any executables called – these other executables, then, are part of the C or Perl program's AVM. Any executables called via exec() or system() in turn may require other executables, shared libraries, or other facilities. Many, if not most, of these components are dependent on one or more configuration files. These components all form an *AVM chain* of dependency for any given application. Regardless of the size or shape of this chain, all application programs on a UNIX machine ultimately interact with the hardware and the outside world via the kernel syscall interface.

8.35 – When we perform system administration actions as root on a running UNIX machine, we can use tools found on the local disk to cause the machine to change portions of that same disk. Those changes can include executables, configuration files, and the kernel itself. Changes can include the system administration tools themselves, and changed components and configuration files can influence the fundamental behavior and viability of those same executables in unforeseen ways, as in §8.10, as applied to changes in the AVM chain (§8.34).

8.36 – A self-administered UNIX host runs an automatic systems administration tool (ASAT) periodically and/or at boot. The ASAT is an application program (§8.33), but it runs as root rather than an ordinary user. While executing, the ASAT is able to modify the "master" copy of itself on disk, as well as the kernel, shared libraries, filesystem layout, or any other portion of disk, as in §8.35.

8.37 – The ASAT described in §8.36 is equivalent to a Turing Virtual Machine (§8.4) in the ways described in §8.33. In addition, a self-administered host running an ASAT is similar to a Universal Turing Machine in that the ASAT can modify its own program code (§8.6).

8.38 – A self-administered UNIX host connected to a network is equivalent to a network-connected Universal Turing Machine (§8.14) in the following ways: The host's ASAT (§8.36) can fetch and execute an arbitrary new program as in §8.15. The fetched program can fetch and execute another as in §8.17. Intermediate results can control which program is fetched next, as in §8.19. The behavior of each fetched program can be influenced by the results of previous programs.

8.39 – When we do administration via automated means (§8.36), we rely on the executable portions of disk, controlled by their configuration files, to rewrite those same executables and configuration files (§8.35). Like the Universal Turing Machine in §8.32, changes made under program control must be assumed to be monotonic; non-reversible short of "resetting the tape state" by reformatting the disk.

8.40 – An ASAT (§8.36) runs in the context of the host kernel and configuration files, and depends either directly or indirectly on other executables and shared libraries on the host's disk (§8.26).

The circular dependency of the ASAT AVM dependency tree (§8.34) forces us to assume that, even though we may not ever change the ASAT code itself, we can unintentionally change its behavior if we change other components of the operating system. This is similar to the indeterminacy described in §8.20.

It is not enough for an ASAT designer to statically link the ASAT binary and carefully design it for minimum dependencies. Other executables, their shared libraries, scripts, and configuration files might be required by ASAT configuration files written by a system administrator – the tool's end user.

When designing tools we cannot know whether the system administrator is aware of the AVM dependency tree (we certainly can't expect them to have read this paper). We must assume that there will be circular dependencies, and we must assume that the tool designer will never know what these dependencies are. The tool must support some means of dealing with them by default. We've found over the last several years that a default paradigm of deterministic ordering will do this.

8.41 – We cannot always keep all hosts identical; a more practical method, for instance, is to set up classes of machines, such as "workstation" and "mail server," and keep the code within a class identical. This reduces the amount of coverage testing required (§8.10). This testing is similar to that described in §8.13.

8.42 – The question of whether a particular piece of software is of sufficient quality for the job remains intractable (§8.9).

We cannot do this in production. Application developers and QA engineers tend to use test environments to do this work. It appears to us that systems administrators should have the same sort of test facilities available for testing infrastructure changes, and should make good use of them.

8.43 – Because the ASAT (§8.36) is itself a complex, critical application program, it needs to be tested using the procedure in §8.42. Because the ASAT can affect the operation of the UNIX kernel and all subsidiary processes, this testing usually will conflict with ordinary application testing. Because the ASAT needs to be tested against every class of host (§8.41) to be used in production, this usually requires a different mix of hosts than that required for testing an ordinary application.

8.44 – The considerations in §8.43 dictate a need for an *infrastructure test environment* for testing automated systems administration tools and techniques. This environment needs to be separate from production, and needs to be as identical as possible in terms of user data and host class mix.

8.45 – Changes made to hosts in the test environment (§8.44), once tested (§8.12), need to be transferred to their production counterpart hosts. When doing so, the ordering precautions in §8.26 need to be observed. Over the last several years, we have found that if you observe these precautions, then you will see the benefits of repeatable results as shown in §8.22. In other words, if you always make the same changes first in test, then production, and you always make those changes in the same order on each host, then changes that worked in test will work in production.

8.46 – Because an ASAT (§8.36) installed on many machines must be able to be updated without manual intervention, it is our standard practice to always have the tool update itself as well as its own configuration files and scripts. This allows the entire system state to progress through deterministic and repeatable phases, with the tool, its configuration files, and other possibly dependent components kept in sync with each other.

By having the ASAT update itself, we know that we are purposely adding another circular dependency beyond that mentioned in §8.40. This adds to the urgency of the need for ordering constraints such as (§8.45).

We suspect control loop theory applies here; this circular dependency creates a potential feedback loop. We need to "break the loop" and prevent runaway behavior such as oscillation (replacing the same file over and over) or loop lockup (breaking the tool so that it cannot do anything anymore). Deterministically ordered changes seem to do the trick, acting as an effective damper.

We stipulate that this is not standard practice for all ASAT users. But all tools must be updated at some point; there are always new features or bug fixes which need to be addressed. If the tool cannot support a clean and predictable update of its own code, then these very critical updates must be done "out of band." This defeats the purpose of using an ASAT, and ruins any chance of reproducible change in an enterprise infrastructure.

8.47 – Due to (§8.45), if we allow the order of changes to be A, B, C on some hosts, and A, C, B on others, then we must test both versions of the resulting hosts (§8.13). We have inadvertently created two host classes (§8.41); due to the risk of unforeseen interactions we must also test both versions of hosts for all *future* changes as well, regardless of ordering of those future changes. The hosts have diverged (see the 'Divergence').

8.48 – It is tempting to ask "Why don't we just test changes in production, and rollback if they don't work?" This does not work unless you are able to take the time to restore from tape, as in §8.27. There's also the user data to consider – if a change has been applied to a production machine, and the machine has run for any length of time, then the data may no longer be compatible with the earlier version of code (§8.28). When using an ASAT in particular, it appears that changes should be assumed to be monotonic (§8.39).

8.49 – It appears that editing, removing, or otherwise altering the master description of prior changes (§8.24) is harmful if those changes have already been deployed to production machines. Editing previously-deployed changes is one cause of divergence. A better method is to always "roll forward" by adding new corrective changes, as in §8.31.

8.50 – It is extremely tempting to try to create a declarative or descriptive language **L** that is able to overcome the ordering restrictions in §8.45 and §8.49. The appeal of this is obvious: "Here are the results I want, go make it so."

A tool that supports this language would work by sampling subsets of disk content, similar to the way our Turing machine samples individual tape cells (§8.1). The tool would read some instruction set **P**, which was written in **L** by the sysadmin. While sampling disk content, the tool would keep track of some internal state **S**, similar to our Turing machine's state (§8.2). Upon discovering a state and disk sample that matched one of the instructions in **P**, the tool could then change state, rewrite some part of the disk, and look at some other part of the disk for something else to do. Assuming a constant instruction set **P**, and a fixed virtual machine in which to interpret **P**, this would provide repeatable, validatable results (§8.3).

8.51 – Since the tool in §8.50 is an ASAT (§8.36), influenced by the AVM dependency tree (§8.34), it is equivalent to a Turing Virtual Machine as in §8.37. This means that it is subject to the ordering constraints of §8.45. If the host is networked, then the behavior shown in §8.15 through §8.20 will be evident.

8.52 – Due to §8.51, there appears to be no language, declarative or imperative, that is able to fully describe the desired content of the root-owned, managed portions of a disk while neglecting ordering and history. This is not a language problem: The behavior of the language interpreter or AVM (§8.34) itself is subject to current disk content in unforeseen ways (§8.35).

We stipulate that disk content can be completely described in any language by simply stating the complete contents of the disk. This is still a case of ordering, a case in which there is only one change to be made. Cloning, discussed in the third section, is an applied example of this case. This class of change seems to be free of the circular dependencies of an AVM; the new disk image is usually applied when running from an NFS or ramdisk root partition, not while modifying a live machine.

8.53 – A tool constructed as in §8.50 *is* useful for a very well-defined purpose; when hosts have diverged (§8.47) beyond any ability to keep track of what changes have already been made. At this point, you have two choices; rebuild the hosts from scratch, using a tool that tracks lifetime ordering; or use a convergence tool to gain some control over them.

8.54 – It is tempting to ask ''Does every change really need to be strictly sequenced? Aren't some changes orthogonal?'' By *orthogonal* we mean that the subsystems affected by the changes are fully independent, non-overlapping, cause no conflict, and have no interaction each other, and therefore are not subject to ordering concerns.

While it is true that some changes will always be orthogonal, we cannot easily prove orthogonality in advance. It might appear that some changes are ''obviously unrelated'' and therefore not subject to sequencing issues. The problem is, who decides? We stipulate that talent and experience are useful here, for good reason: it turns out that orthogonality decisions are subject to the same pitfalls as software testing.

For example, inspection (§8.8) and testing (§8.9) can help detect changes which are *not* orthogonal. Code coverage information (§8.10) can be used to ensure the validity of the testing itself. But in the end, none of these provide assurance that any two changes *are* orthogonal, and like other testing, we cannot know when we have tested or inspected for orthogonality enough.

Due to this lack of assurance, the cost of predicting orthogonality needs to accrue the potential cost of any errors that result from a faulty prediction. This error cost includes lost revenue, labor required for recovery, and loss of goodwill. We may be able to reduce this error cost, but it cannot be zero – a zero cost implies that we never make mistakes when analyzing orthogonality. Because the cost of prediction includes this error cost as well as the cost of testing, we know that prediction of orthogonality is more expensive than either the testing or error cost alone:

$$C_{predict} > C_{error}$$
$$C_{predict} > C_{test}$$

8.55 – As a crude negative proof, let us take a look at what would happen if we were to allow the order of changes to be totally unsequenced on a production host. First, if we were to do this, it is apparent that some sequences would not work at all, and probably damage the host (§8.26). We would need to have a way of preventing them from executing, probably by using some sort of exclusion list. In order to discover the full list of bad sequences, we would need to test and/or inspect each possible sequence.

This is an *intractable* problem: the number of possible orderings of M changes is M!. If each build/test cycle takes an hour, then any number of changes beyond seven or eight becomes impractical – testing all combinations of eight changes would require 4.6 years. In practice, we see change sets much larger than this; the ISconf version 2i makefile for building HACMP clusters, for instance, has sequences as long as 121 operations – that's 121!/24/365, or 9.24×10^{196} years. It is easier to avoid unsequenced changes.

The cost of testing and inspection required to enable randomized sequencing appears to be greater than the cost of testing a subset of all sequences (§8.25), and greater than the testing, inspection, and accrued error of predicting orthogonality (§8.54):

$$C_{random} > C_{predict} > C_{partial}$$

8.56 – As a self-administering machine changes its disk contents, it may change its ability to change its disk contents. A change directive that works now may not work in the same way on the same machine in the future and vice versa (§8.26). There appears to be a need to constrain the order of change directives in order to obtain predictable behavior.

8.57 – In contrast to §8.52, a language that supports execution of an ordered set of changes appears to satisfy (§8.56), and appears to have the ability to fully describe any arbitrary disk content, as in the disk description section.

8.58 – In practice, sysadmins tend to make changes to UNIX hosts as they discover the need for them; in response to user request, security concern, or bug fix. If the goal is minimum work for maximum reliability, then it would appear that the ''ideal'' sequence is the one which is first known to work – the sequence in which the changes were created and tested. This sequence carries the least testing cost. It carries a lower risk than a sequence which has been partially tested or not tested at all.

The costs in §8.8, §8.9, §8.25, §8.54, and §8.55 are related to each other as shown in Figure 10. This leads us to these conclusions:

- Validating, inspecting, testing, and deploying a single sequence (C_{test}) appears to be the least-cost host change management technique.
- Adequate testing of partially-ordered sequences ($C_{partial}$) is more expensive.

- Predicting orthogonality between partial sequences ($C_{predict}$) is yet more expensive.
- The testing required to enable random change sequences (C_{random}) is more expensive than any other testing, due to the N! combinatorial explosions involved.

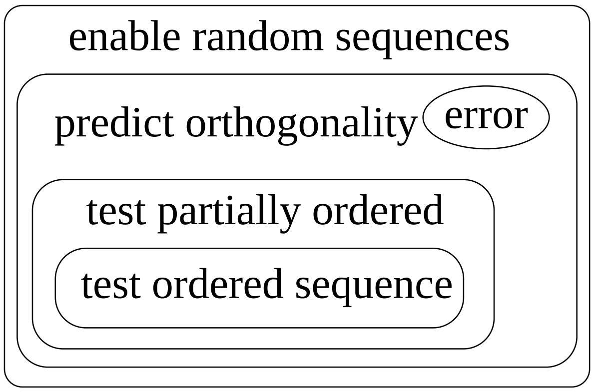

Figure 10: Relationship between costs of various ordering techniques; larger set size means higher cost.

8.59 – The behavioral attributes of a complex host seem to be effectively infinite over all possible inputs, and therefore difficult to fully quantify (§8.9). The disk size is finite, so we can completely describe hosts in terms of disk content (as discussed when describing disks), but we *cannot* completely describe hosts in terms of behavior. We can easily test all disk content, but we do not seem to be able to test all possible behavior.

This point has important implications for the design of management tools – behavior seems to be a peripheral issue, while disk content seems to play a more central role. It would seem that tools which test only for behavior will always be convergent at best. Tools which test for disk content have the potential to be congruent, but only if they are able to describe the entire disk state. One way to describe the entire disk is to support an initial disk state description followed by ordered changes.

8.60 – There appears to be a general statement we can make about software systems that run ‘‘on top of’’ others in a ‘‘virtual machine’’ or other software-constructed execution environment (§8.34):

> If any virtual machine instruction has the ability to alter the virtual machine instruction set, then different instruction execution orders can produce different instruction sets. Order of execution of these instructions is critical in determining the future instruction set of the machine. Faulty order has the potential to remove the ability for the machine to update the instruction set or to function at all.

This applies to any application, automatic administration tool (§8.37), or shared library code executed as root on a UNIX machine (it also applies to other cases on other operating systems). These all interact with hardware and the outside world via the operation system kernel, and have the ability to change that same kernel as well as higher-level elements of their ‘‘virtual machine.’’ This statement appears to be independent of the language of the virtual machine instruction set (§8.52).

Conclusion and Critique

One interesting result of automated systems administration efforts might be that, like the term ‘computer,’ the term ‘system administrator’ may someday evolve to mean a piece of technology rather than a chained human.

Sometime in the last few years, we began to suspect that deterministic ordering of host changes may be the airfoil of automated systems administration. Many other tool designers make use of algorithms that specifically avoid any ordering constraint; we accepted ordering as an axiom.

With this constraint in place, we built and maintained thousands of hosts, in many mission-critical production infrastructures worldwide, with excellent results. These results included high reliability and security, low cost of ownership, rapid deployments and changes, easy turnover, and excellent longevity – after several years, some of our first infrastructures are still running and are actively maintained by people we’ve never met, still using the same toolset. Our attempts to duplicate these results while neglecting ordering have not met these same standards as well as we would like.

In this paper, our first attempt at explaining a theoretical reason why these results might be expected, we have not ‘‘proven’’ the connection between ordering and theory in any mathematical sense. We have, however, been able to provide a thought experiment which we hope will help guide future research. Based on this thought experiment, it seems that more in-depth theoretical models may be able to support our practical results.

This work seems to imply that, if hosts are Turing equivalent (with the possible exception of tape size) and if an automated administration tool is Turing equivalent in its use of language, then there may be certain self-referential behaviors which we might want to either avoid or plan for. This in turn would imply that either order of changes is important, or the host or method of administration needs to be constrained to less than Turing equivalence in order to make order unimportant. The validity of this claim is still an open question. In our deployments we have decided to err on the side of ordering.

On tape size: one addition to our ''thought experiment'' might be a stipulation that a network-connected host may in fact be fully equivalent to a Universal Turing Machine, including infinite tape size, if the network is the Internet. This is possibly true, due to the fact that the host's own network interface card will always have a lower bandwidth than the growth rate of the Internet itself – the host cannot ever reach ''the end of the tape.'' We have not explored the implications or validity of this claim. If true, this claim may be especially interesting in light of the recent trend of package management tools which are able to self-select, download, and install packages from arbitrary servers elsewhere on the Internet.

Synthesizing a theoretical basis for why ''order matters'' has turned out to be surprisingly difficult. The concepts involve the circular dependency chain mentioned in the section on 'Ordered Thinking,' the dependency trees which conventional package management schemes support, as well as the interactions between these and more granular changes, such as patches and configuration file edits. Space and accessibility concerns precluded us from accurately providing rigorous proofs for the points made in the section on Turning equivalence. Rather than do so, we have tried to express these points as hypotheses, and have provided some pointers to some of the foundation theories that we believe to be relevant. We encourage others to attempt to refute or support these points.

One issue we have not adequately covered is the fact that changing the order of actions can not only break machines, but the actions themselves may not complete. Altering order often calls for altering the content of the actions themselves if success is to be assured.

There may be useful vulnerabilities or benefits hidden in the structure of §8. Even after the many months we have spent poring over it, it is still certainly more complex than it needs to be, with many intertwined threads and long chains of assumptions (Figure 11). One reason for this complexity was our desire to avoid forward references within that section; we didn't want to inadvertently base any point on circular logic. A much more readable text could likely be produced by reworking these threads into a single linear order, though that would likely require adding the forward references back in.

For further theoretical study, we recommend:

- Gödel Numbers
- Gödel's Incompleteness Theorem
- Chomsky's Hierarchy
- Diagonalization
- The halting problem
- NP completeness and the Traveling Salesman Problem
- Theory of ordered sets
- Closed-loop control theory

Starting points for most of these can be found in [greenlaw, garey, brookshear, dewdney].

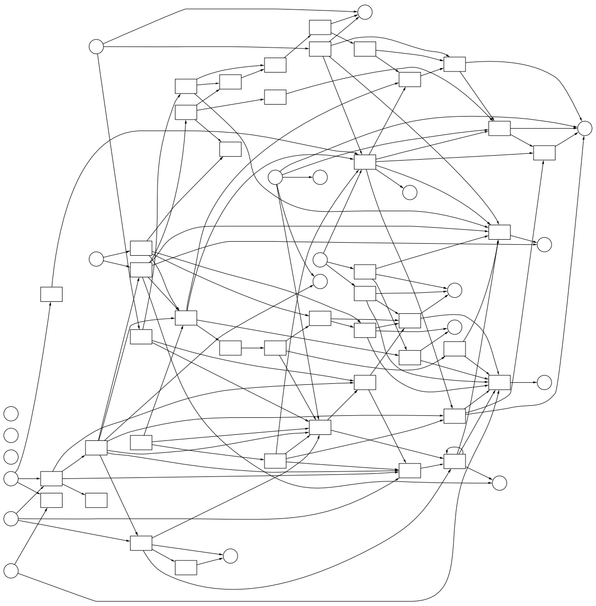

Figure 11: Thread structure of §8.

Acknowledgments

We'd like to thank all souls who strive to better your organizations' computing infrastructures, often against active opposition by your own management. You know that your efforts are not likely to be understood by your own CIO. You do this for the good of the organization and the global economy; you do this in order to improve the quality of life of your constituents, often at the cost of your own health; you do this because you know it is the right thing to do. In this year of security-related tragedies and corporate accounting scandals, you know that if the popular media recognized what's going on in our IT departments there'd be hell to pay. But you know they won't, not for many years, if ever. Still you try to clean up the mess, alone. You are all heroes.

The debate that was the genesis of this paper began in Mark Burgess' cfengine workshop, LISA 2001.

Alva Couch provided an invaluable sounding board for the theoretical foundations of this paper. Paul Anderson endured the intermediate drafts, providing valuable constructive criticism. Paul's wife, Jessie, confirmed portability of these principles to other operating systems and provided early encouragement. Jon Stearley provided excellent last-minute review guidance.

Joel Huddleston responded to our recall with his usual deep interest in any brain-exploding problem, the messier the better.

The members of the *infrastructures* list have earned our respect as a group of very smart, very capable individuals. Their reactions to drafts were as good as rocket fuel. In addition to those mentioned elsewhere, notable mention goes to Ryan Nowakowski and Kevin Counts, for their last-minute readthrough of final drafts.

Steve's wife, Joyce Cao Traugott, made this paper possible. Her sense of wonder, analytical interest in solving the problem, and unconditional love let Steve stay immersed far longer than any of us suspected would be necessary. Thank You, Joyce.

About the Authors

Steve Traugott is a consulting Infrastructure Architect, and publishes tools and techniques for automated systems administration. His firm, TerraLuna LLC, is a specialty consulting organization that focuses on enterprise infrastructure architecture. His deployments have ranged from New York trading floors, IBM mainframe UNIX labs, and NASA supercomputers to web farms and growing startups. He can be reached via the Infrastructures.Org, TerraLuna.Com, or stevegt.com web sites.

Lance Brown taught himself Applesoft BASIC in 9th grade by pestering the 11th graders taking Computer Science so much their teacher gave him a complete copy of all the handouts she used for the entire semester. Three weeks later he asked for more. He graduated college with a BA in Computer Science, attended graduate school, and began a career as a software developer and then systems administrator. He has been the lead Unix sysadmin for central servers at the National Institute of Environmental Health Sciences in Research Triangle Park, North Carolina for the last six years.

References

[bootstrap] Traugott, Steve, and Joel Huddleston, ''Bootstrapping an infrastructure,'' *Proceedings of the Twelfth Systems Administration Conference (LISA XII)*, USENIX Association, Berkeley, CA, p. 181, 1998.

[brookshear] Brookshear, J. Glenn, *Computer Science, An Overview*, (very accessible text), Addison Wesley, ISBN 0-201-35747-X, 2000.

[centerrun] *CenterRun Application Management System*, http://www.centerrun.com .

[cfengine] *Cfengine, A Configuration Engine*, http://www.cfengine.org/ .

[church] Church, A., "Review of Turing 1936," *1937a Journal of Symbolic Logic*, 2, pp. 42-43.

[couch] Couch, Alva, and N. Daniels, "The Maelstrom: Network Service Debugging via 'Ineffective Procedures'," *Proceedings of the Fifteenth Systems Administration Conference (LISA XV)*, USENIX Association, Berkeley, CA, p. 63, 2001.

[cvs] *Concurrent Version System*, http://www.cvshome.org .

[cvsup] *CVSup Versioned Software Distribution package*, http://www.openbsd.org/cvsup.html .

[debian] *Debian Linux*, http://www.debian.org .

[dewdney] Dewdney, A. K., *The (New) Turing Omnibus – 66 Excursions in Computer Science*, W. H. Freeman and Company, 1993.

[eika-sandnes] Eika Sandnes, Frode, ''Scheduling Partially Ordered Events In A Randomized Framework – Empirical Results And Implications For Automatic Configuration Management,'' *Proceedings of the Fifteenth Systems Administration Conference (LISA XV)*, USENIX Association, Berkeley, CA, 2001.

[elbaum] Elbaum, Sebastian G., David Gable, and Gregg Rothermel, ''The Impact of Software Evolution on Code Coverage Information,'' *International Conference on Software Engineering*, pp. 170-179, 2001

[garey] Garey, Michael R., and David S. Johnson, *Computers and Intractability, A guide to the theory of NP-Completeness*, W. H. Freeman and Company, ISBN 0-7167-1045-5, 2002,

[godel] Godel, Kurt, ''Uber formal unentscheidbare Satze der Principia Mathematica und verwandter Systeme,'' *Monatshefte fur Mathematik und Physik*, 38:173-198, 1931.

[godfrey] Godfrey, M. D., and D. F. Hendry, ''The Computer as Von Newmann Planned It,'' *IEEE Annals of the History of Computing*, Vol. 15, No. 1, 1993.

[greenlaw] Greenlaw, Raymond, H. James Hoover, and Morgan Kaufmann, *Fundamentals of the Theory of Computation*, (includes examples in C and UNIX shell, detailed references to seminal works), ISBN 1-55860-474-X, 1998,

[hagerty] Hagerty, Daniel, <hag@ai.mit.edu>, personal correspondence, 2002.

[hamlet] Hamlet, Dick, ''Foundations of Software Testing: Dependability Theory,'' *Software Engineering Notes*, *Proceedings of the Second ACM SIGSOFT Symposium on Foundations of Software Engineering*, Vol. 19, No. 5, pp. 128-139, 1994.

[hart] Hart, John, and Jeffrey D'Amelia, ''An Analysis of RPM Validation Drift,'' *Proceedings of the Sixteenth Systems Administration Conference*, USENIX Association, Berkeley, CA, 2002.

[immunology] Burgess, M., ''Computer Immunology,'' *Proceedings of the Twelth Systems Administration Conference (LISA XII)*, USENIX Association, Berkeley, CA, pp. 283, 1998.

[isconf] *ISconf, Infrastructure Configuration Manager*, http://www.isconf.org and http://www.infrastructures.org .

[jiang] Jiang, Tao, Ming Li, and Bala Ravikumar,

"Basic Notions in Computational Complexity," *Algorithms and Theory of Computation Handbook*, p. 24-1, CRC Press, ISBN 0-8493-2649-4, 1999.

[laitenberger] Laitenberger, Oliver, and Jean-Marc DeBaud, "An Encompassing Life Cycle Centric Survey of Software Inspection," *The Journal of Systems and Software*, Vol. 50, No. 1, pp. 5-31, 2000.

[lcfg] *LCFG: A Large Scale UNIX Configuration System*, http://www.lcfg.org .

[lisa] *Large Installation Systems Administration Conference*, USENIX Association, Berkeley, CA, http://www.usenix.org, 2001.

[mccabe] McCabe, Thomas J. and Arthur H. Watson, "Software Complexity," *Crosstalk, Journal of Defense Software Engineering*, Vol. 7, No. 12, pp. 5-9, December 1994.

[nordin] Nordin, Peter, and Wolfgang Banzhaf, "Evolving Turing-Complete Programs for a Register Machine with Self-modifying Code," *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, Ed. Morgan Kaufmann and L. Eshelman, pp. 318-325, 15-19, ISBN 1-55860-370-0, 1995.

[oetiker] Oetiker, T., "Template Tree II: The Post-Installation Setup Tool," *Proceedings of the Fifteenth Systems Administration Conference (LISA XV)*, USENIX Association, Berkeley, CA, p. 179, 2001.

[opsware] *Opsware Management System*, http://www.opsware.com .

[pikt] "PIKT: 'Problem Informant/Killer Tool'," http://www.pikt.org .

[rdist] Cooper, M. A., "Overhauling Rdist for the '90s," *Proceedings of the Sixth Systems Administration Conference (LISA VI)*, USENIX Association, Berkeley, CA, p. 175, 1992.

[richardson] Richardson, D. J., and L. A. Clarke, "Partition Analysis: A Method Combining Testing and Verification," *IEEE Trans. Soft. Eng.*, Vol. 11, No. 12, pp. 1477-1490, 1985.

[rsync] *rsync Incremental File Transfer Utility*, http://samba.anu.edu.au/rsync .

[ssh] *SSH Protocol Suite of Network Connectivity Tools*, http://www.openssh.org .

[sup] Shafer, Steven, and Mary Thompson, *The SUP Software Upgrade Protocol*, 1989.

[tivoli] *Tivoli Management Framework*, http://www.tivoli.com .

[turing] Turing, Alan M., "On Computable Numbers, with an Application to the Entscheidungsproblem," *Proceedings of the London Mathematical Society, Series 2*, Vol. 42, pp. 230-265, 1936-37,

[vonneumann] Von Neumann, John, "First Draft of a Report on the EDVAC," *IEEE Annals of the History of Computing*, Vol. 15, No. 4, 1993.

[xilinx] *Xilink Virtex-II Platform FPGA*, http://www.xilinx.com .