

host managed.

This paper is a living document
a project of TerraLuna, LLC.

Foreword

by Steve Traugott

In 1998, Joel Huddleston and I suggested that the entire enterprise infrastructure could be managed by one large “enterprise virtual machine” [the *cfengine* strap]. That paper briefly described part of the management toolset, later named ISconf [isconfig], based on relatively simple makefiles and scripts. It did not seem extraordinary at the time. At the time of the paper, we said that we would likely be using [cfengine] the next time around – I had been following Mark Burgess’ progress since 1994.

At the invitation of Mark Burgess, I joined LISA 2001 [lisa] cfengine workshop to discuss we'd found so far, with possible targets for cfengine 2.0 feature set. The ordering requirements seemed to need more work; I found ordering singly difficult to justify to an audience practiced in use of convergent tools, where ordering is often considered a constraint to be specifically avoided [cfengine-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 comparing ISconf to other tools, I suspect that ordering is a keystone principle in automated system administration. Lance and I, with a lot of help

do this, but making complete backups of hosts is redundant – the same operating system components must be backed up for each host. What we really need are the user data and host boot files (how many copies of /bin/lis do we need to store on tape?). It is usually more efficient to have a master copy and quickly and correctly rebuild each host from it. A tool that maintains an ordered record of the files made after install is one way to do this.

This prediction is particularly important for large organizations using what we call *server farms*. These are hosts that run an automation or administration tool in the context of their own operating environment. Commercial examples in this category include Tivoli, Opsware, and [tivoli, opsware, centerrun]. Open-source

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 on 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 only a few more files, for example – and taking some action

order to be valid, this testing must be done on each production host, due to the factors discussed in §8.47. This testing itself requires either removing the host from production use or exposing users to a degraded machine. Without this validation, we cannot safely put a machine in mission-critical operation.

Congruence

Congruence (Figure 3) is the practice of maintaining production hosts in complete conformance with a fully descriptive baseline (see the section ‘Describing Disk State’). Congruence is measured in terms of disk state rather than behavior. In this state can be fully described, while behavior cannot (§8.59).

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 sets. 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 ordered lifetime journaling required for congruent behavior. Our own `isconf` (described in its own section) is the only specifically congruent tool we have in production use, though `cfengine`, with some extra coding, appears to be usable for administration of congruent environments. We discuss this in more detail in the `cfengine` section.

means by which it is produced.

“Order Matters” because we need the machine-language instructions resulting from 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 not prove correctness of a higher-level program if you do not know the correctness of the lower-level instructions. If the high-level program can modify the underlying layers, then the behavior of the program will change with each modification. Order of modifications appears to be important to predict the behavior of the high-level program. Simply, it is important to ensure that you do not cut off the tree limb *before* you cut through it.

congruent infrastructure. Each change added to the procedure updates the baseline; see the ‘Congruent Infrastructure’ section.

There are tools which can help you maintain the baseline; see the ‘Example Tools and Techniques’ section.

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

The starting state (bare-metal install) description of the disk may take the form of a network :

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

There are several ordering considerations. We won't know that we need the new configuration file until we do step 1. We shouldn't proceed until we know that 2 and 3 were successful. If we do step 4 too early, we may break the ability for cfengine to operate at all. If we do step 4 too late, we may be using the resulting configuration file using the old cfengine, it will fail.

ISconf

As we mentioned in the introduction, *isconf* initially began life as a quick hack. Its basic utility proven itself repeatedly over the last eight *years*, as adoption has grown it is currently managing 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 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 classification of hosts. These top-level facilities also are what govern the actions *ISconf* is to take during boot versus c

Kaines. This version adds more “less” including more fine-grained control applied to target classes and hosts. There are layers of abstraction between the administrators and the target machines; the tool uses various intermediate files that are eventually fed to ‘make.’

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

scripts, each equivalent to an ISconf makefile script. These scripts we then drive within a deterministic ordered framework.

In the cfengine version 2 language there are many features, such as the FileExists() evaluated condition, which may reduce the amount of code. So based on our experience over the last few years it

```
# 01 Feb 97 - Block00 is initial distribution
# with ntp etc. added later
Block00: ntp cvs lynx ...
# 15 Jul 98 - got tired of waiting
# cut new baseline image, later added
Block10: Block00 ssh ...
# 17 Jan 99 - new baseline again, including
Block20: Block10 apache kernel-2.2.
```

Listing 2: Baseline

affects this behavior, and how ordered changes can make its behavior more deterministic.

We will expand beyond single machine to the realm of distributed computing and multiple machines, and their associated testing costs. We will discuss how order affects these costs, and how ordered changes provides the lowest cost for managing infrastructure.

Readers who are interested in applying mathematical or theoretical arguments should review the previous section or skip this section.

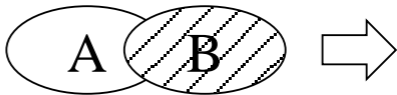
8.1 – A Turing machine (Figure 8.1) reads symbols from an infinite tape, interprets them as instructions, and writes symbols to a hardwired program and rewrites the tape.

ure 6). This arrangement allows the TVM to contain the machine language instructions which describe 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 physical

tions which involve fetching a small amount of code from the remote tape, and storing it on the local program tape as additional and/or replacement instructions (§8.6). We will name the original instruction set **A**. The set of fetched instructions will name **B**, and the resulting merged instruction set will name **AB**. Note that some of the instructions in **B** may have replaced some of those in **A**. Before the fetch, our TVM could be described as an **A** machine, after the fetch we have a new machine – the TVM’s basic functionality is different. It is no longer the same machine.



this, we would need to return the local program 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 the data 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 re-execute 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 on the data tape is only acceptable to an **AB** machine. Machine **A** might reject it or fail to halt (§8.3). The only safe rollback method seems to be some

remote, shared networks, mesystem layer of
portion of disk, as in §8.35.

8.37 – The ASAT described in §8.36 is equivalent to a Turing Virtual Machine (§8.4) 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 networked 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 program can fetch and execute another as in §8.16. Immediate results can control which program is executed next, as in §8.19. The behavior of

behavior such as oscillation (replacing the same over and over) or loop lockup (breaking the tool that it cannot do anything anymore). Deterministic 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 a certain point; there are always new features or bugs 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 enterprise infrastructure.

when we have tested or inspected for
enough.

Due to this lack of assurance, the c
ing orthogonality needs to accrue the p
any errors that result from a faulty p
error cost includes lost revenue, labo
recovery, and loss of goodwill. We m
reduce this error cost, but it cannot be
cost implies that we never make mista
lyzing orthogonality. Because the cost
includes this error cost as well as the c
we know that prediction of orthogon
expensive than either the testing or error

$$\begin{aligned}C_{predict} &> C_{error} \\C_{predict} &> C_{test}\end{aligned}$$

ordered changes.

8.60 – There appears to be a general statement we can make about software systems that run “code” 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.

desire to avoid forward references with
we didn't want to inadvertently base an
cular logic. A much more readable text
produced by reworking these threads in
ear order, though that would likely requ
forward references back in.

For further theoretical study, we re

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

ence (*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*, www.cfengine.org/.

[church] Church, A., “Review of Turing 1936,” *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 System Administration Conference (LISA XV)*, USENIX Association, Berkeley, CA, p. 63, 2001.

- Vol. 11, No. 12, pp. 1477-1490, 1990.
- [rsync] *rsync Incremental File Transfer Utility*, samba.anu.edu.au/rsync.
- [ssh] *SSH Protocol Suite of Network Security Tools*, <http://www.openssh.org>.
- [sup] Shafer, Steven, and Mary Thompson. *Software Upgrade Protocol*, 1989.
- [tivoli] *Tivoli Management Framework*, [tivoli.com](http://www.tivoli.com).
- [turing] Turing, Alan M., "On Computing Machinery with an Application to the Entscheidungsproblem," *Proceedings of the London Mathematical Society, Series 2*, Vol. 42, pp. 230-265, 1936.
- [vonneumann] Von Neumann, John, "First Draft Report on the EDVAC," *IEEE Annals of the History of Computing*, Vol. 15, No. 4, pp. 305-320, 1993.

