The Maelstrom: Network Service Debugging via "Ineffective Procedures"

Dr. Alva L. Couch – Tufts University
Noah Daniels – Analog Devices

ABSTRACT

The process of network debugging is commonly guided by "decision trees," that describe and attempt to address the most common failure modes. We show that troubleshooting can be made more effective by converting decision trees into suites of "convergent" troubleshooting scripts that do not change network attributes unless these are out of compliance with accepted norms. "Maelstrom" is a tool for managing and coordinating execution of these scripts. Maelstrom exploits convergence of individual scripts to dynamically infer an appropriate execution order for the scripts. It accomplishes this in $O(n^2)$ procedure trials, where n is the number of troubleshooting scripts. This greatly eases adding scripts to a troubleshooting scheme, and thus makes it easier for people to cooperate in producing more exhaustive and effective troubleshooting schemes.

Introduction

In maintaining complicated service networks, one pressing problem is to determine and actively eliminate causes of network service disruptions. It is currently easy to automatically detect network problems through a variety of monitoring techniques [14, 16, 28]. But taking the next step of automatically remedying network problems has so far proven impractical. A human troubleshooter must often engage in involved scientific inquiry to infer 'causes' from observed 'effects.' A complex problem can take weeks to solve, and may be solved without ever revealing the true 'cause' of the problem.

It is currently easy to automatically detect network problems through a variety of monitoring techniques [14, 16, 28]. But taking the next step of automatically remedying network problems has so far proven impractical. A human troubleshooter must often engage in involved scientific inquiry to infer 'causes' from observed 'effects.' A complex problem can take weeks to solve, and may be solved without ever revealing the true 'cause' of the problem.

While it may be argued that a well-designed network and well-chosen hardware do not fail, this is certainly not true in an environment where one teaches hands-on Computer Science. We face problems almost daily with runaway processes, latent bugs in web scripts, and other disruptions based upon student (or faculty) error. Vendor-supplied servers crash due to latent bugs 'discovered' by users toying with new programming techniques. Cables are stepped upon and equipment is abused. We cannot limit the capabilities of users to prevent such failures without compromising our educational mission, so that we must react to failures on an ongoing basis.

One problem with automating troubleshooting of mission-critical network services such as http, ftp, imap, ssh, etc., is that these services typically depend upon other base services (such as NFS, LDAP, database servers, etc.) that must function before the externally visible services will function. Network devices that connect internal servers to one another must also be examined during the troubleshooting process. Ideally, to automate service troubleshooting, one must integrate and coordinate procedures that analyze and repair almost every device in the network, from server processes down to switches. While it may be argued that a network is poorly designed unless the precedences between services are clearly defined and unvarying, problems persist largely because we do not fully recognize or understand the dependencies between services.

Convergence

Troubleshooting is by nature a *convergent process* where one only repairs a component if it seems not to function properly. Monolithic tools such as Cfengine and its relatives [2, 3, 4, 7, 8, 17, 18] attempt to force every attribute of a given device into compliance with a predefined operational policy. Most of these tools are limited to functioning in environments where one can compile and run programs, and cannot be utilized to maintain closed-source vendor components such as routers, switches, dialup servers, and so on. Babble [10] provides the beginnings of a tool for convergent administration of turnkey network services, but is largely limited to conversing with and controlling one device at a time.

Expanding these tools to cover the problem of network-wide troubleshooting seems impractical. They are already limited by their own size and complexity. Some are perhaps reaching the limits of software complexity from the standpoint of usability, predictability, maintainability, and adaptability to new needs.

In this paper, we study the potential for dividing these currently monolithic administrative processes into smaller pieces that work together to accomplish the same goal. Tools designed to be maintained by a small group of experts may require extensive effort from the programmer who wishes to couple new and reusable convergent processes into an existing tool framework. It is more desirable to be able to contribute independent processes that interoperate easily with others without utilizing traditional software coupling mechanisms such as subroutine calls and interfaces.

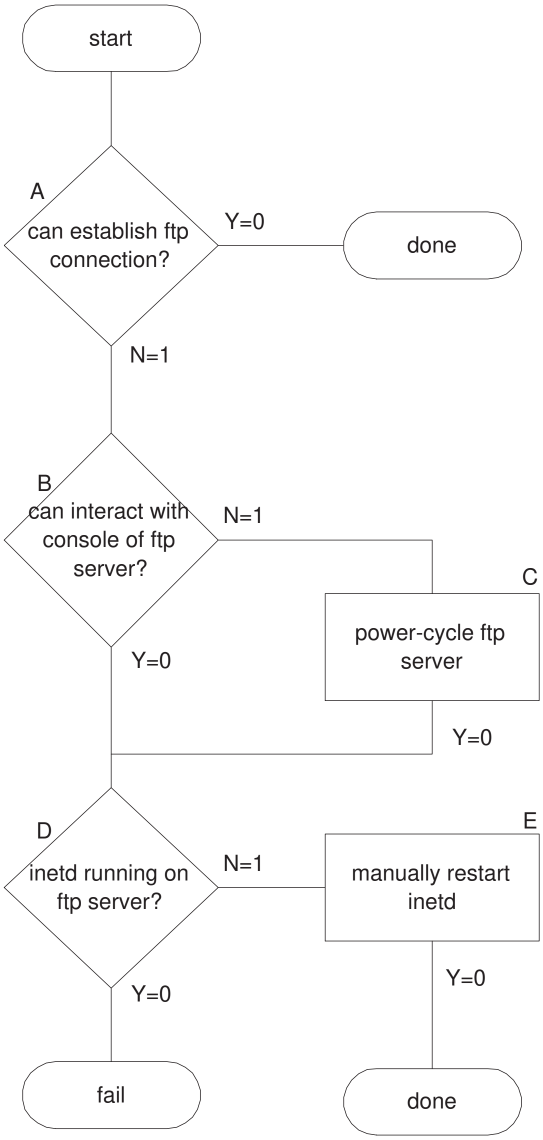

Figure 1: A typical (though oversimplified) troubleshooting decision tree.

Decision trees

As a 'best practice,' many sites pragmatically describe and standardize network troubleshooting procedures as ''decision trees'' that describe tests to make and corresponding actions to take. Usually decision trees are prepared by senior staff (or vendors) to aid junior staff in dealing effectively and autonomously with common problems. A decision tree is nothing more than a flow chart (in actuality a directed acyclic graph (DAG)) of actions, where the result of each action determines the next action to try.

For an example, consider the (very much oversimplified) decision tree in Figure 1. According to the tree, to check whether ftp service is running, one must first check whether one can make an ftp connection from a client machine. If so, everything is working. If not, one must then check whether the console of the ftp server responds to a keypress. If this succeeds, the server is running; otherwise it has probably crashed and needs to be power-cycled. The final measure is to check whether the program inetd is running on the (now perhaps rebooted) ftp server, and to start it if not. This is far too simple to be realistic; it is contrived only to illustrate concepts rather than as a practical application.

Two special actions determine when to give up. The 'done' action indicates success, while the 'fail' action indicates that the decision tree failed to correct the problem. In a realistic situation, failure of a procedure would escalate the problem's priority and refer it to the next level of technical staff.

One could perform most of the steps in this simple tree by convergent administration in Cfengine [2, 3, 4], but for the purposes of this discussion we consider the steps in the tree to be Babble [10] scripts that potentially interact not just with a local host but also with remote serial consoles of devices such as routers, switches, and power interrupters. While Babble suits our needs for some scripting purposes, these scripts could be arbitrary programs in any desired language.

We started this work with the goal of automating the process of interpreting troubleshooting ''decision trees'' like the above to automatically detect and repair network disruptions. We abstracted each 'decision' in the tree into a script with multiple exit values 0, 1, 2, . . ., where the exit value of a script indicates the next script to invoke. Simple actions not involving a decision are considered to be decisions with only one exit code and outcome. The scripts are plumbed together by declaring which ones should be called as a result of the return codes of others. We implemented a tree traversal algorithm in Perl that ''executes the decision tree'' by running scripts and taking decision branches as indicated.

Within our scheme, one would implement the above example as scripts A, B, C, D, E, with exit codes determining branches as indicated in Figure 1. For example, if B's exit code is 1, we invoke C; if the code is 0, we invoke D. Using algorithms from Babble [10], we thought we could represent this tree structure in XML [15]. We endeavored to convert this XML tree into a nested hash and execute the results as a kind of script.

Ineffective Procedures

This approach seemed logical, sensible, and straightforward, but failed miserably in practice. Most decision trees for human use are *ineffective procedures*. They suggest what to do, but are typically based upon a simplified view of network function that is understandable to a human. The typical tree addresses a few failure modes and makes assumptions about the precedences between tasks and couplings between components that may not be true in practice.

Perhaps these limitations of a decision tree arise from its real (and perhaps hidden) purpose.

Proposition 1: **A decision tree is not an embodiment of technical knowledge, but rather a statement of operating policy [9]**.

It is a description of what a human should "try first," not necessarily what must "work first." The order of tests and actions to be taken are tempered not by physical dependencies, but by service expectations and site mission. For example, 'Rebooting' might be a routine expedient at an academic site, but not at a bank!

Thus one must consider any decision tree not as an effective procedure, but merely a *theory* of what to do when. One does not interpret such a tree literally, but instead bases a course of action upon the information in the tree. In practice, this is what we really do with our own troubleshooting procedures; we 'jump around' within the tree, executing the steps not in order of tree appearance, but in order of their likelihood of causing the problem we have detected.

Exposing Convergent Processes

Most decision trees can be straightforwardly represented as a series of *convergent processes* performed in sequence, each with the purport, "If it is broken, then fix it." A typical convergent process consists of a test and perhaps a configuration change action. If the test succeeds, no action is performed; else an appropriate action is taken in order to ensure that the test succeeds at a later time.

The processes in most decision trees are not convergent in this sense. Sometimes it takes a bit of worthwhile effort to fit an existing decision tree to the convergent process model. For example, one might replace the decision tree above by three convergent processes:

F. Test whether ftp is running, and report success or failure.
G. Test whether the console of the ftp server responds, and power-cycle the server if it seems to have crashed.
H. Test whether inetd is running on the ftp server, and manually restart inetd if it does not seem to be running.

Barriers in the decision tree that prevent unnecessary troubleshooting are no longer necessary. Step F need not fail before one tries steps G and H. None of these steps is damaging to a network that is functioning normally. Step F is a simple test of symptoms that is not even necessary to invoke, because the other two processes are the only 'operative' ones that could potentially change the state of the network and repair a problem.

This convergent representation has advantages over the original decision tree representation. If all three of these processes report success, ftp is running as desired. It no longer matters which process is tried first, as the diagnostic procedures will not take action upon a problem unless the problem has been observed directly and a change is necessary.

In our study we have been unable to generate a network decision tree that cannot be represented as a set of convergent scripts. Network decision trees seldom exhibit more complexity than can be represented straightforwardly in this form. Even when they do, such trees can be expressed in this simpler form by encapsulating more complex processes inside larger "convergent" processes that replace several decisions.

The Curse of Precedence

Transforming a complex decision tree into a sequence of convergent actions is advantageous not only from a human interface standpoint, but also because of the properties of the networks we wish to manage.

The most difficult problem in creating a network debugging decision tree is to determine the precedences between debugging tasks. Many facets of network performance depend upon others. For example, one server may provide NFS service to another that cannot function properly without that service, so one must make NFS work properly before trying to do anything with the dependent server.

Anyone who has tried to manually craft a Makefile for use with the make [23] program will have experienced the complexity of determining and specifying precedences. If the precedences are incorrect or incomplete, make may not have the desired effect, sometimes in subtle ways.

A decision tree is nothing more than a description of the precedences between tasks; in this sense it is a subset of the precedence declarations in a Makefile. But in a network, the precedences between tasks may be constantly changing and are impossible to predefine. As an oversimplified example, consider the problem of a router that utilizes TFTP to boot and then provides routing services for the TFTP host. Suppose we have two convergent processes:

J. Check router function and restart if necessary.
K. Check the TFTP server and restart TFTP if necessary.

In this case, there need not be any fixed precedence between processes J and K; the best precedence depends upon the network's configuration and its current state. If the router is down, then it would be greatly desirable to check the TFTP server first so that the reboot will work, *unless the* TFTP *server is on the other side of the router from the host running the debugging scripts!* The 'best' ordering for these checks must be determined by considering several factors that may or may not be known:

- The current state of the network and current failure mode.
- Dependencies that apply between components while in that state.

Errors in precedence mean that the overall debugging process will fail, so the success of the process is dependent upon the problem being solved. This means in turn that we must work around this potential problem in a decision tree by encoding *both* orders for the above processes into the tree in some fashion.

In constructing Makefiles, determining the correct precedences is a surmountable problem, because those precedences never change. In network debugging, the problem of determining appropriate precedences between tasks is ill-formed, and there may not be a static precedence between tasks that always applies. Thus make's algorithm of 'topologically sorting' tasks into an execution order coherent with a predetermined partial order is not enough to deal with the precedences between troubleshooting tasks. In fact:

Proposition 2: **Precedence between troubleshooting tasks is an abstract ideal that is not well-defined in practice.**

Exponential Complexity

To generate a decision tree that could succeed in repairing problems in all cases, we would necessarily have to foresee all possible failure modes and encode them into the tree. These failure modes determine the order in which steps should be taken. Each possible ordering or precedence would add complexity to the tree, in the worst case resulting in an exponential explosion in the size of the tree.

These negative conclusions engendered a fundamental change in our thinking and approach. Since we did not believe that we could ever construct such a complex tree correctly, we had no confidence that direct automation of any regular decision tree could *ever* be useful. We stopped even thinking about executing a decision tree directly, and began looking for a better way to approach the problem. The key to this approach is that if a set of scripts exhibit a particular set of properties, precedence does not matter. The key concept is *homogeneity of effect*; that scripts do not conflict with one another in the goals they wish to ensure.

Homogeneity

The concept of a 'convergent script' must be refined slightly when one expects it to interoperate and cooperate with other scripts of similar intent. A script must not only be convergent within itself, but must also behave so that the set of all scripts is convergent as a suite. To do this, each script must exhibit three crucial ingredients:

- **awareness:** a script knows whether it succeeded or not in enforcing its requirements. It returns a nonzero exit code to indicate failure and a zero exit code to indicate success.
- **convergence:** applying the script twice changes nothing and has no effect if the network is already in compliance with the script's requirements.
- **homogeneity** (or *consistency*): scripts never undo the changes made by others (though scripts may enforce the *same* changes as others). This is a *global* convergence criterion over all scripts.

These conditions were of course inspired by the convergent processes of Cfengine [2, 3, 4] and our own Slink [7] and Distr [8], and are exactly the behaviors that we previously attempted to ensure by scripting in Prolog [9] instead of Cfengine or PIKT [24], and in Babble [10] instead of Expect [20].

Of these conditions, the first two are relatively easy to ensure for a given script. These are properties of a script in isolation from others. The third property of homogeneity is a *global condition on the suite of all scripts.* These seemingly abstract ideals are easy to provide in scripts. Consider the simple task of forcing /etc/inetd.conf to have a particular form. A script that does this might contain:[1]

```
cp /proto/inetd.conf /etc/inetd.conf
```

This script can be converted into a convergent one by checking that a copy is needed before making the copy:

```
if [ ! cmp /proto/inetd.conf \
           /etc/inetd.conf ] ; then
  cp /proto/inetd.conf /etc/inetd.conf
fi
```

This can be made aware by arranging for it to return the proper exit codes:

```
if [ ! -f /proto/inetd.conf ] ; \
    exit 1
if [ ! cmp /proto/inetd.conf \
           /etc/inetd.conf ] ; then
  cp /proto/inetd.conf /etc/inetd.conf
fi
exit 0
```

To make it homogeneous with other scripts, however, *every* script that modifies /etc/inetd.conf must modify it into this exact form.

Avoiding Precedence

A set of scripts that exhibit awareness, convergence, and homogeneity with one another has unexpected properties that may not be evident at first glance.

Proposition 3: **Scripts that are convergent and homogeneous may be executed in any order without harm to a functioning network.**

[1]It is generally believed that such a file should be maintained by incremental editing rather than by file copying; this example is oversimplified for clarity.

Since they will not repair non-problems, they will do nothing in a functioning network, and since they will agree on results, they will not undo each others' changes.

This means that even if we do not know the dependencies between scripts, we can dynamically discover an order in which they work properly:

Proposition 4: **Given a set of aware, convergent, and homogeneous scripts that repair parts of a network, we can assure network function by cycling through all possible permutations of the scripts.**

Given a little thought, this claim is relatively obvious. If a script is safe to repeat until it works, and innocuous when not needed, one can simply try it in all possible contexts until it works. If there is an order in which the scripts will work, that order will be tried, so that precedences will be satisfied.

In our decision tree example, step H depends upon the success of step G, while step F depends upon the success of step H, so that the appropriate execution order is "GHF". But even if we do not know this order, we can still utilize the tests effectively by repeating them so that all possible orders will be contained in the pattern of repetition. If we execute the steps in the order "FGHFGHF", the appropriate "GHF" subsequence is present in that ordering. The sequence "FGHFGHF" contains *all possible permutations* of F,G, and H as subsequences:

```
(FGHFGHF)
 FGH....
 F.H.G..
 .GHF...
 .G.F.H.
 ..HFG..
 ..H.G.F
```

After this sequence of executions, ftp will be running if it possibly can be made to run by the debugging processes as given.

Trying All Permutations

Given any set of scripts that are aware, convergent, and homogeneous, one can try them in all possible permutations in a very straightforward way. One exploits the following mathematical fact:

Proposition 5: **Given a sequence $S = A_1 \cdots A_n$ of n objects, the sequence containing $n-1$ copies of S followed by the first object A_1 in S contains all permutations of the members of A as subsequences. This sequence contains $n(n-1)+1$ elements.**

Counter to intuition, enumerating all permutations as *subsequences* of a given sequence does not require $O(n!)$ steps, but only $O(n^2)$ steps.

As an aside, this fact is easy to see by inductive proof. The inductive basis case is that for two objects A_1 and A_2, the appropriate sequence is $A_1A_2A_1$. If we presume that the proposition is true for $A_1 \cdots A_{n-1}$, then a sequence S_{n-1} consisting of $n-2$ copies of $A_1 \cdots A_{n-1}$, followed by A_1, contains all permutations of $A_1 \cdots A_{n-1}$ as subsequences. For A_n elements, we construct the sequence S_n of $n-1$ copies of $A_1 \cdots A_n$, followed by A_1. Our claim is that S_n contains all permutations of $A_1 \cdots A_n$ as subsequences.

To show this, we choose one element A_i from the first copy of $A_1 \cdots A_n$, and consider the subsequence of this sequence constructed by starting after A_i and deleting A_i from the rest. The beginning of this subsequence always has the form of the sequence in the inductive hypothesis, and thus contains all permutations of the A's without A_i. Thus the sequence S_n (of which this is a subsequence) contains all permutations *starting* with A_i. As A_i was arbitrary, S_n contains all permutations of the A's.

For example, consider what happens with four scripts A, B, C, and D. If we start with the sequence ABCDABCDABCDA and select, e.g., C from the first group, the subsequence constructed according to the proof is ABDABDA, which has already been shown to contain all permutations of A, B, and D. By repeating this process we can generate all permutations of A, B, C, and D:

```
subsequence     permutations
 ABCDABCDABCDA
 ABCD.BCD.B... (A first)
 .BCD..C...... ABCD, ABDC
 ..C..B.D.B... ACBD, ACDB
 ...D.BC..B... ADBC, ADCB
 .BCDA.CDA.C.. (B first)
 ..CDA..D..... BCDA, BCAD
 ...DA.C.A.... BDAC, BDCA
 ....A.CD..C.. BACD, BADC
 ..CDAB.DAB.D. (C first)
 ...DABA...... CDAB, CDBA
 ....AB.D.B... CABD, CADB
 .....B..A..DA  CBAD, CBDA
 ...DABC.ABC.A  (D first)
 ....ABC..B... DABC, DACB
 .....BC.A.C.. DBCA, DBAC
 ......C.AB..A  DCAB, DCBA
```

This is not the best known solution to the problem of "Permutation Embedding." For n objects there is a sequence of $n^2 - 2n + 4$ elements that contains all permutations of the original objects as subsequences [1, 5, 13, 19, 22, 26, 27]. The methods utilized to construct a sequence with this lower bound are less intuitive than ours, while the size of the sequence remains $O(n^2)$ in all cases. To our knowledge, finding the optimal solution in the general case remains an open problem in combinatorics.

In our case, the objects being permuted are scripts that manage the values of configurable attributes, while the sequence is the execution order for the scripts. If we know that a correct order for the scripts exists, then if we try to execute them in the order suggested above, that order will be achieved at some time during the process. Because of our homogeneity constraint, each script need work only once and scripts will not undo the work of others, so that one iteration of the scripts in the proper order is all that is required. Thus the precedence constraints for all scripts will be satisfied if there is a way to satisfy them at all, and *the algorithm dynamically infers the correct execution order for the scripts as it executes.*

Exploiting Awareness

If we know that the scripts we are running are aware as well as homogeneous, we can make this iterative process more efficient. If they are aware, they know when they succeed. If they are homogeneous, one success for each script suffices. Thus we need only execute each script until it succeeds, and the permutation embedding algorithm can skip invoking scripts that have already succeeded.

Suppose, e.g., that we have six scripts A,B,C,D,E,F, to be executed in that order, and that in actuality D and E must occur before B; E before C; C before A; and A before F. Let us notate a success of A as =A (representing unification with A's postconditions) and a failure of A as !A. Then the algorithm's execution will proceed as follows:

```
!A !B !C =D =E !F
!A =B =C .. .. !F
=A .. .. .. .. =F
```

In the first pass, A, B, C, and F fail, while D and E succeed, because !C implies !A, !D and !E imply !B, !E implies !C, and !A implies !F. In the second pass B and C succeed because their predecessors D and E have succeeded in the first. In the last pass, A and F succeed because all their preconditions are satisfied. So an appropriate total order is D, E, B, C, A, F.

The worst case is that the precedences are the reverse of the order of the list. In this case the algorithm takes $O(n^2)$ executions, where n is the number of scripts:

```
!A !B !C !D !E =F
!A !B !C !D =E ..
!A !B !C =D .. ..
!A !B =C .. .. ..
!A =B .. .. .. ..
=A
```

There are $(n - 1)(n - 2)/2$ script failures due to inappropriate precedence. Once we know the reverse order is more appropriate, however, a subsequent run will take only $O(n)$ executions.

The driving idea here is that

Proposition 6: **The execution order for an aware, convergent, and homogeneous set of scripts need not be predeclared through precedences, but can be discovered by executing the scripts and observing their effects.**

We should note that this does not mean that we can discover the true precedences between scripts, just that we can discover *some execution order that obeys those precedences.* Further, Frode Eika Sandnes shows that knowing this order does not in general aid in inferring the true precedences between actions [25].

The Maelstrom

Maelstrom is a tool that implements the above algorithm for dispatching debugging scripts. The input to Maelstrom is a configuration file that describes the *probable* precedences between a collection of debugging scripts. These need not be the actual precedences between scripts; they represent just a *suggestion* of the best order in which to execute the scripts. Maelstrom begins by ordering these scripts into a total order consistent with the partial order declared in the file, in the manner of make [23].

The scripts that Maelstrom controls are expected to know whether they succeed or fail. A script tells Maelstrom that it succeeded in its task by returning an exit code of 0. Maelstrom interprets such a response as an indication that the script either found the network in compliance with the script's requirements or made the network comply by making changes. Maelstrom makes no distinction between these cases. If a script returns a nonzero exit code, Maelstrom interprets this as an indication that the script could not succeed for some external reason. Such scripts are tried again at a later time to determine if other scripts make it possible for them to succeed.

Maelstrom attempts to make all the scripts succeed – indicated by an exit code of 0 – by trying a sufficient number of permutations of the scripts. Given n scripts, each script is tried at most n times. When a script succeeds, it is not tried again. If precedences are correct, each script is executed exactly once and succeeds immediately, while if precedences are completely backward, each script is executed n times with $n - 1$ failures preceding one success. At the end of trying all possible permutations of the scripts, Maelstrom gives up and reports the scripts that failed consistently. If this happens, then the scripts have failed to repair the network and some other script (or human intervention) is required in order to repair the problem.

Crafting the Perfect Storm

For convenience and ease of use, Maelstrom's configuration file looks much like a Makefile. The directive:

```
c1 :  c2 :  c3
```

describes the precedences between scripts c1, c2, and c3: c3 before c2 before c1. The difference between this and a Makefile is that there are nothing but scripts here; there are no other files or intermediary results listed. Scripts can have arguments. Two script invocations are considered identical if they have the exact same arguments, so that a script must succeed once with each set of arguments. Thus the declaration:

```
c1 -f : c2 --tftp
c2 --tftp : c2 --comsat
```

describes the relationships between *three* scripts, "c1 -f", "c2 --comsat" and "c2 --tftp".

Commands in Maelstrom are repeatedly executed until they succeed, as indicated by a return code (exit status) of 0. If they do not succeed, they are repeated in sequence according to the total order suggested by a topological sort of the partial order described by the file. During this process, Maelstrom skips any scripts that previously succeeded during the repetition. In the last example above, the total order is:

A. c2 --comsat
B. c2 --tftp
C. c1 -f

Maelstrom would execute these in the order "ABCABCA," which as above contains all permutations of A, B, and C as subsequences.

Maelstrom and make

It should be obvious by now that Maelstrom uses a very different execution algorithm than make. First, Maelstrom presumes that all of the scripts it controls use exit codes to indicate whether they should be repeated. In make, a non-zero exit code instead indicates an irrecoverable error and causes a full stop. The most crucial difference, though, is that make utilizes centralized or *global knowledge* of precedence in order to schedule its tasks, while Maelstrom's knowledge is a local result of the behaviors of the scripts it controls. This means that to emulate make with Maelstrom, some of the tasks done by make must move into the scripts that Maelstrom dispatches.

For example, consider the trivial Makefile:

```
foo: foo.o bar.o
        g++ -o foo foo.o bar.o
foo.o: foo.c
        g++ -c foo.c
bar.o: bar.c
        g++ -c bar.c
```

This describes how to make an executable foo from source files foo.c and bar.c. To simulate this in Maelstrom, we break it into three convergent tasks c1, c2, and c3: one for each compilation command in the Makefile. The convergent process of creating foo.o from foo.c is easy to express in the (shell) script c2:

```
if [ -nt foo.o foo.c ]; \
    exit 0
g++ -c foo.c
exit 0
```

(-nt x y is true if x is newer than y). Likewise, compiling bar.c into bar.o can be accomplished in a script c3:

```
if [ -nt bar.o bar.c ]; \
     exit 0
g++ -c bar.c
exit 0
```

Difficulties, arise, however, in performing the final step of creating foo from foo.o and bar.o. In make, the states and dates of all files are known, and this global knowledge is used to assure that each command operates upon up-to-date data. Maelstrom knows none of this, so we must compensate for its lack of awareness by checking all prerequisites inside the compilation script before doing the final compilation:

```
if [ -nt foo.c foo.o ]; \
  exit 1
if [ -nt bar.c bar.o ]; \
  exit 1
if [ -nt foo foo.o \
  -a -nt foo bar.o ]; \
  exit 0
g++ -o foo foo.o bar.o
exit 0;
```

The script that combines all results must compensate for Maelstrom's lack of global knowledge by obtaining that knowledge for itself. If we call this script c1, then the Maelstrom configuration:

```
c1 : c2
c1 : c3
```

has the same result as the above Makefile.

This may seem like a limitation until we realize that in Maelstrom's problem domain, the kind of global knowledge that make utilizes is typically unavailable or approximate. Since troubleshooting scripts, for safety, *must* check that their prerequisites are satisfied before continuing, Maelstrom's lack of global knowledge does not impact its ability to solve troubleshooting problems.

Implementing Policy

Sometimes decision trees are effective procedures. Thus we have made it possible to implement traditional decision trees (such as the one at the beginning of this paper) using Maelstrom's syntax. Decision trees are implemented by specifying execution rules for scripts based upon the exit codes of others.

Three forms of policy control in Maelstrom allow forcing execution of specific scripts immediately after the failures or successes of others. Forced evaluation is a policy decision based upon what is most important in a network. One might wish, e.g., to minimize the impacts of a reboot by immediately checking specific services related to the reboot, before checking other facets of operation. Unlike the core scripts Maelstrom invokes, these are *not* interpreted as suggestions or theory, but directly control what Maelstrom does in specific situations.

All the execution controls mimic shell syntax for ease of use. Short-circuit 'and' and 'or' work as they do in the shell:

```
A || B
```

causes script B to be invoked only if script *A* fails. Likewise,

```
A && B
```

causes script B to be invoked only if script *A* succeeds. Equivalently,

```
!A && B
```

has the obvious meaning. An advanced syntax allows reacting directly to exit codes:

```
A [ 23=>B; 34=>C ]
```

means that if the exit code of A is 23, execute B, and if the exit code of A is 34, execute C.

Primary and Secondary Scripts

Scripts that are called as the result of deterministic rules are treated differently by Maelstrom than the scripts to which such a rule applies.

- *Primary* scripts are those that appear alone, in precedence declarations, or on the left-hand side of the above conditional execution forms (e.g., A in A||B or A&&B or A[B;C]).
- *Secondary* (or incidental) scripts do not appear in any of the primary contexts.

Maelstrom attempts to make all the primary scripts in its configuration file succeed once. Secondary scripts are not included in Maelstrom's algorithm, and only execute under the conditions in which they are declared. This allows one to code traditional decision trees into Maelstrom processes without invoking the Maelstrom algorithm upon their components.

For example, the tree at the beginning of this paper, containing scripts A, B, C, D, E could be coded explicitly as:

```
A[1=>B[1=>C;0=>D[1=>E]]]
```

or by realizing that X[1=>Y] is just X||Y, more simply as:

```
A||B[1=>C;0=>D||E]
```

In this case:

- A is a primary command and becomes part of Maelstrom's main task.
- B, C, D, E are secondary scripts that are only invoked when A fails. They are not part of Maelstrom's list of scripts that *must* succeed.

Using the [] syntax, very complex trees are possible, though for simplicity any one script invocation can be assigned only one set of actions. Writing the two declarations:

```
A || B
A [ 1=>C ; 2=>D ]
```

together is illegal, because there can only be one response to an invocation of A.

A Maelstrom command is parsed by first splitting it into phrases separated by the precedence operator ':'. These phrases consist of a list of commands separated by ';'. Each command can optionally be associated with an action by use of the decision operators '&&', '||', or '[]', where the latter can themselves contain lists of commands and associated actions. A decision can appear after a command in any context, e.g.,

```
A [ B [ C ; D ] ; E || F ] : G && H
```

is precisely equivalent to

```
A : G
A [ B [ C ; D ] ; E || F ]
G && H
```

and roughly equivalent to:

```
A : G
A [ B ; E ]
B [ C ; D ]
E || F
G && H
```

In the first two declarations there are two primary commands A and G, while in the last one the primary commands also include B and E (because they appear outside the context of a conditional command).

Testing the Wind

Maelstrom was much more difficult to test than to write, because few of us would enjoy releasing a tool of such powers to do damage to our network during testing! Thus testing required writing a simulator whose input was a known set of task precedences, to see if Maelstrom could sort them out without foreknowledge of their precedences. It does this as expected.

Quite obviously, the success of Maelstrom depends upon the quality and reliability of the scripts it dispatches. To date, we have not deployed Maelstrom in production, because we are not yet confident of our scripts' convergent properties. The scripts that we envision using in production are largely targeted at restoring specific mission-critical services, and not at addressing systemic failures or network connectivity as yet. A typical script tries a netcat from a remote device to see if a service is working *from outside a server* and then attempts to repair any problems *inside* the server, as is possible by using Babble.

Safety

Even in the simple examples of this paper, there are conditions in which a troubleshooting script can make things worse by its actions. This can happen if a corrective action is too extreme or depends upon external resources that are themselves down at the moment. If our scripts are convergent in the Cfengine sense and there are no hidden constraints, Maelstrom is relatively safe. If, e.g., a reboot depends upon hidden constraints that have not been assured, such as a service required for reboot, Maelstrom may reboot a server even though this makes the network state worse, and may well make future troubleshooting impossible without operator intervention.

Maelstrom is currently relatively naive about the limitations of its environment. It can be made safer by giving it more understanding of the imperfections within its scripts, and the hidden couplings between scripts and Maelstrom's environment.

Maelstrom cannot currently compensate for inhomogeneity or lack of convergence in scripts. In the future, there will be stronger precedence operators to control *Maelstrom's* actions in the presence of imperfect scripts. Recall that "c2 : c1" means "c1 might theoretically precede c2." We plan other precedence operations whose main purpose is to compensate for script deficiencies:

- "c2 :: c1" will mean "the *last success* of c1 must precede the *last* invocation of c2."
- "c2 ::: c1" will mean "the *first success* of c1 must precede the *first* invocation of c2."

Both of these are still weaker conditions than the ":" in make. In Maelstrom, we could notate make's concept of strong precedence as follows:

- "c2 :::: c1' could mean that *every success* of c1 must be followed by an invocation (and success) of c2.

We use the colons to limit the number of characters one must escape inside shell commands in the configuration file (currently ':', ';', '[', and ']').

All of these syntactic mechanisms are attempts to compensate for non-homogeneous or non-convergent behavior in scripts.

- The declaration c2 :: c1 means that c1 and c2 are convergent but inhomogeneous, so that c2 must be tried after c1 in order to make the execution result deterministic. This rule means "always clean up after c1 with c2."
- The declaration c2 ::: c1 declares a hard-coded precedence, and means "it is impractical to execute c2 without at least one success of c1. We will use this when there are unavoidable physical dependencies, such as when an intervening router must be tested before the equipment behind it.
- The declaration c2 :::: c1 (which we may *not* implement in the immediate future) compensates for *non-convergent* behavior of c1, by *always* following it as soon as possible with a cleanup routine c2.

To understand the importance of adding these precedence operators to Maelstrom, note that with even the first one (::) we can simulate make with Maelstrom without resorting to more script intelligence. If script c1 is:

```
if [ -nt foo foo.o \
  -a -nt foo bar.o] ; exit 1
g++ -o foo foo.o bar.o exit 0
```

and script c2 is:

```
if [ -nt foo.o foo.c ] ; \
  exit 0
g++ -c foo.c
exit 0
```

and script c3 is:

```
if [ -nt bar.o bar.c ] ; \
  exit 0
g++ -c bar.c
exit 0
```

then the Maelstrom declarations:

```
c1 :: c2
c1 :: c3
```

would accomplish the same *effect* as the Makefile above. Even the relatively weak double-colon operator precedence avoids the need to have script c1 know all the dependencies between its files, as in the former example. This script *might* do redundant compilations, but in the end it will accomplish the exact same result as the Makefile.

Although we discuss the possibility of 'rebooting' as a result of a script, we are not happy with the prospect of automated power-cycling of servers. We are currently developing a tool that allows that kind of dangerous action to be controlled by an electronic mail or two-way pager transaction. The script that wishes to reboot a server asks us whether it should or not, and an operator can mail back a 'yes' or 'no' response.

One weakness of Maelstrom's scheduling is its simplicity. Many colleagues have suggested that Maelstrom should allow one to declare not just precedences, but also "costs" as a measure of how disruptive a particular action will be. One could then try solutions in order of increasing cost. But this would require an even more complex syntax in the configuration file, and theoretical precedences have the same overall effect (through different kinds of declarations).

Observability

The Maelstrom framework allows one to 'glue together' several scripts toward a common goal. Precedences do not have to be specified during the gluing because Maelstrom will sort the scripts into an appropriate order during execution. During the sorting process, Maelstrom *does not determine the actual precedences between scripts, but rather a total order that satisfies those precedences.* The order is observable but the precedences are not [25]. Similarly,

Proposition 7: **While the *effects* of homogeneity are observable, homogeneity itself is an abstract ideal that is unobservable in practice.**

We will illustrate this fact with several examples.

For simplicity, let us consider the case in which all scripts control configuration attributes that have static values in the absence of script effects. This ignores dynamic attributes that can change on their own without script intervention. If the former are problematic, the latter surely are as well.

In this simplified model, there are only *attributes* to be controlled and *parameters* describing ideal settings. A *configuration attribute* is any such data, including text or numeric fields in configuration files or device memory, or even hierarchies of configuration data as handled by Arusha [17] or Babble [10]. A particular script can perhaps do two things with an attribute:

- *read* its value to validate this value against established norms.
- *write* a value into the attribute in order to put it into compliance with such norms.

The value of a particular attribute can be compared with a *configuration parameter* that describes its ideal state. This can either be a fixed value or a rule that describes the ideal value in a variety of situations.

Using this simplified model of script behavior, we can describe correct and compliant script behavior as well as possible deviant script behaviors:

- A script that is aware can read all the configuration attributes that it sets to check for success or failure.
- A script that is convergent only changes attributes that are "out of sync" with desired parameter settings. This means that it reads every attribute and writes it only if it does not satisfy parameter requirements or rules.
- A script that is homogeneous agrees with other scripts upon parameter values. This can be accomplished either by synchronizing the contents of multiple parameter sources or by depending upon a single source for parameter and policy information.

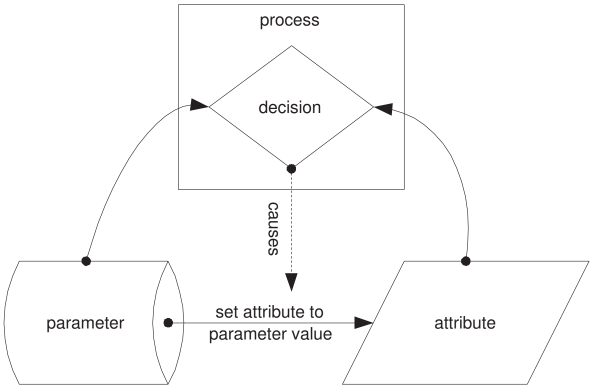

Figure 2: A healthy transaction copies a parameter into an attribute only if needed.

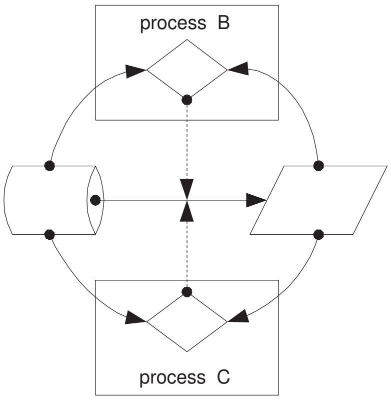

Figure 3: Two homogeneous scripts using the same parameter source.

Graphically, the relationships between scripts, attributes, and parameters may be depicted by representing scripts as rectangles, decisions as diamonds, parameters (or policies) as storage drums, and device attributes to be controlled as parallelograms. An arrow between two objects means that the first affects the second, even when one or more of the objects being affected are themselves arrows. An arrow between a parameter and an attribute means that this parameter affects that attribute by setting it to a compliant value. An arrow from a script to the prior *arrow* means that the script in question enforces that relationship.

For example, in Figure 2, we have one healthy configuration process that controls one attribute with respect to one parameter. The process reads both the attribute and its controlling parameter, and the attribute is only set if needed. This is indicated by a decision node whose inputs are the values to compare, and which controls the (data flow) arrow between the parameter and the attribute in question.

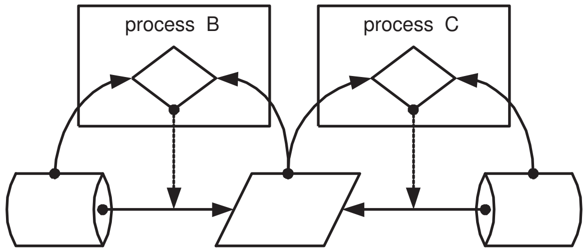

Figure 4: Two inhomogeneous scripts using different parameter sources.

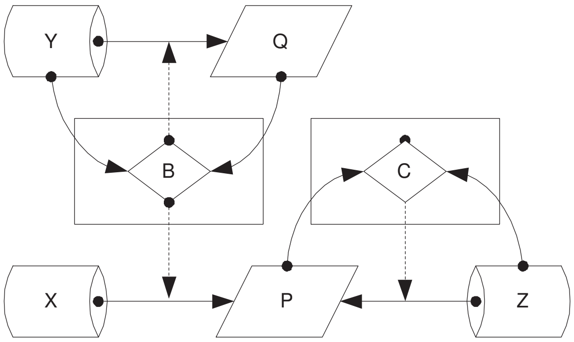

Figure 5: Latent inhomogeneity masked by a single environmental side-effect.

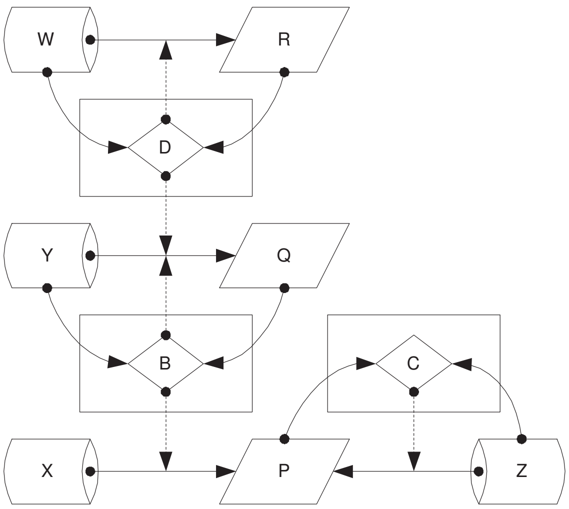

Figure 6: Latent inhomogeneity hidden by multiple environmental side-effects.

This notation allows us to notate many things that can go wrong (or right) when scripts try to cooperate. Homogeneity of scripts means that when they control the same attribute, they utilize the same source

(Figure 3). Inhomogeneity results from two processes that try to force the same attribute into conformance with conflicting parameters or policies (Figure 4).

It would seem from the above example that inhomogeneity is easy to observe and detect, but it can be easily masked and obscured by relatively common scripting errors so that it is only observable under specific environmental conditions.

For example, in Figure 5, script B changes an attribute P to a parameter value X depending upon conformance of a distinct attribute Q to a parameter Y. If another script C sets P according to another parameter rule Z, then the result of the sequence BC will always give P the value from Z, while the sequence CB will give P an inhomogeneous value *only if* Q *is initially out of conformance with* Y. Thus inhomogeneity of B and C *is not observable* except under certain (typically unknown) environmental preconditions.

One might think that this condition would be rare in traditional practice, but we actually find it very common in *our own* scripts! For example, we might want to update someone's password on a machine or network. This password is stored, with other information – such as the user's print name – that might be changed inadvertently at the same time as the password is changed.

Side-effects can make inhomogeneity arbitrarily difficult to detect. In Figure 6, three processes B, C, and D have homogeneous effects except when parameter W matches attribute R, parameter Y does not match attribute Q, and B and C are executed after script D. In this case alone, scripts B and C are inhomogeneous and the orders DBC and DCB will set attribute P according to conflicting parameters Z and X, respectively.

Thus inhomogeneity is very difficult to detect in general, even when the scripts being tested work properly in isolation and are fully convergent and aware of their effects. Using a generalization of the above construction, one can create a set of scripts for which every permutation produces the same result except within one particular environment, wherein two permutations produce inconsistent results.

Assuring Determinism

So far, whenever we have been unable to predefine something we need, we have consistently tried to observe it instead. When observing, we found that *the original abstract concept that we tried to describe is not what we need, and that something less descriptive and more observable suffices.* While we initially searched for a way to define precedences for tasks, we only truly need an order in which the tasks will function properly. This order can be discovered dynamically, even if the precedences remain hidden.

Applying this approach to homogeneity of a set of scripts, it is the observable results of homogeneity that we desire, rather than the property itself. Minimally, the execution of every set of scripts should have a deterministic outcome so there are 'no surprises.' If this outcome is inappropriate, we know that our scripts are incorrect. If a script has an inhomogeneity that does not affect the outcome, we will never know about it, and perhaps need not even care.

To assure determinism in the outcome of executing a set of scripts that are convergent and deterministic in isolation from one another, we can run them in a particular order on a *functioning* network. If they are homogeneous with one another, this will make no changes to the network. If they are inhomogeneous but convergent, a script that disagrees with another will make a conflicting change to the network. Even so, *the fact that the scripts were executed in a predetermined order guarantees a result dependent upon that order.* Once we have agreed upon this as a goal, detecting inhomogeneity becomes easier. If each script reports whether it changed anything, we can record whether there are any inhomogeneities apparent within the sequence in which we execute the scripts. This is much easier to determine than detecting inhomogeneities between all pairs of scripts, as we need only concern ourselves with anomalies in executing *one permutation* of the scripts, not all permutations (as would be required to detect the subtle examples of inhomogeneity presented above).

Suppose that in a *functioning* network, we run a sequence of scripts *twice*. We can conclude, in the absence of other external effects, that any script that reports changes twice during this process is inhomogeneous with at least one of the scripts invoked between its two invocations. For example, if the sequence of scripts is ABCDE, and we invoke them twice with the following results:

```
script: A B C D E A B C D E
change: n y n n y n y n n y
```

we can conclude that B and E are inhomogeneous with one another, while the other scripts are observably homogeneous. In more complex cases it may not be clear which scripts are inhomogeneous with others, as in

```
script: A B C D E A B C D E
change: n y y n y n y n n y
```

We can say little about the relationship between C and the others, as it only makes changes *once*. All that we can conclude from this is that E must follow B to ensure the result achieved by this run. It is unclear whether C is inhomogeneous with either B or E from this evidence.

Coping with Inhomogeneity

Just as homogeneity guarantees that the effect of a sequence of scripts is independent of its order, lack of homogeneity means that the effect of a sequence of scripts depends somewhat upon their ordering. While repeating the scripts in a functioning network in a predetermined order assures a deterministic result, we can restore some semblance of predictability with less effort by restricting ourselves to a subset of execution orders for the scripts during the troubleshooting process.

For example, consider scripts A, B, C, and D. Suppose that A is homogeneous with all others, D is homogeneous with all others, and B and C are not homogeneous with one another. This is a precise way of saying that BC (B success followed by C success) may perhaps exhibit a different result than CB. Thus there are two groups of permutations of the set of scripts that produce two distinct results:

```
result 1   result 2
BCAD       CBAD
BACD       CABD
BADC       CADB
ABCD       ACBD
ABDC       ACDB
ADBC       ADCB
BCDA       CBDA
BDCA       CDBA
BDAC       CDAB
DBCA       DCBA
DBAC       DCAB
DABC       DACB
```

Suppose that we wish to decide (perhaps arbitrarily) upon BC. Then we must execute C after B even if C has already succeeded. This is a less extreme solution than executing all of the scripts again in a functioning network, but in this case has the same effect. In the future, we will be able to assure this kind of behavior in Maelstrom by using extended precedence operators ::, :::, and perhaps ::::.

Latent Variables

In practice, however, homogeneity can be much more difficult to identify or refute, because other problems and processes can cause the same symptoms. Suppose for two otherwise homogeneous scripts B and C, there is *yet another unknown rogue process* interfering with B. In this case, the fact that B had to correct a problem twice does not indicate a sequencing problem with C, but is rather due to a hidden latent variable that has nothing to do with the performance of either script we are considering. Conversely, a successful test like the above can only indicate that homogeneity is preserved in one test case. Any such analysis only suggests that homogeneity *may* or *may* not be preserved, but does not indicate whether it *is* or *is not* preserved. Sadly, the latter are mathematical ideals that are meaningless in practice.

Phenomenology

In the above arguments, twice we have applied observation as a replacement for description, first in determining the order of troubleshooting scripts, and then in determining whether a particular set of scripts is capable of being ordered in that manner. These are both 'phenomenological' (or 'empirical') approaches that replace abstract descriptions with hard evidence, by *changing the problem to allow employing evidence instead of theory*.

In traditional approaches to troubleshooting, theory guides all actions. In the context of this paper, the precedences between tasks are theoretical. Interpreting a valid theory literally improves speed of troubleshooting by avoiding redundant tests, while an invalid theory may blind one to a potential problem by keeping one from executing tests in the order required by physical conditions.

Modularity

The rules above for scripts allow scripts to be added freely to our debugging schema without any attention to precedences between scripts as long as they all remain homogeneous, convergent, and aware of their effects. These requirements allow a new form of 'phenomenological' software modularity; a modularity based upon effect rather than interface. If scripts can maintain agreement about their overlapping effects, they may be freely mixed and employed toward a common goal. Maelstrom's algorithm is nothing more than a 'phenomenological sort' of phenomenologically modular scripts into a total order within which each can function properly.

Traditional ideas of software modularity express modules as being defined by interfaces with specified preconditions and postconditions on the use of each interface. For Maelstrom, instead modularity is based solely upon effect and postconditions; modularity is a requirement of the postconditions of the network after the script finishes. Every script must check its preconditions itself and only proceed if they are appropriate, so all scripts are expected to function correctly for all possible sets of preconditions.

Of course, what has been omitted so far in this discussion is the difficulty of writing scripts with the appropriate properties of environmental awareness, convergence, and homogeneity. In general it is difficult to construct such scripts while assuring script quality and reliability [10]. It is this difficulty that justifies the use of complex tools such as Cfengine [2, 3, 4] to fulfill that purpose instead.

Assuring Convergence

The problems of constructing convergent scripts have already been discussed in [9, 10]. One approach to ensuring awareness and convergence is to create an intelligent interface for dealing with the environment [7]. This interface hides the details of convergence from the script writer by checking for each change before making it inside of subroutines that accomplish the changes. As long as one interacts only with the environment through the lens of these subroutines, the script is guaranteed to be convergent. Without this discipline, convergent scripts are more complex than their non-convergent counterparts. Checking every parameter before changing it leads to doubling of script size with a commensurate increase in the difficulty and cost of script maintainability.

Assuring Homogeneity

Even if we can solve these problems, the new problem of assuring that scripts agree globally on their effects (homogeneity) gives rise to even more script complexity. One part of enforcing homogeneity is easy. If two scripts configure or control distinct domains with no overlap in parameters, such as a router and a distinct switch, we say the scripts have *orthogonal domains of change*. Such scripts are automatically homogeneous once they are convergent.

If, however, two scripts affect the same environmental parameters, they must somehow agree upon the appropriate values (or rules) for those parameters or homogeneity will not be possible. Scripts that overlap in function should ideally gather their configuration details from the same source. In this way, the only requirement for the scripts themselves is that they be convergent, and homogeneity is guaranteed by convergence. Again, this calls for a common interface; a library that accesses the same database of desired traits of a network for all scripts.

With both of these ingredients – convergent interface libraries and globally consistent access to configuration information – *homogeneity is automatically guaranteed*. But almost none of the modern solutions to scripting have the second property of consistent access to configuration information.

A beginning for this kind of globally consistent access to parameters may be found in the configuration files and common configuration interface of the Arusha Project [17, 18]. Even with powerful mechanisms such as Arusha's XML-based parameter inheritance, assuring both of these ingredients also requires substantial discipline on the part of the programmer; *all* access to the network or to configuration information must be through this mechanism. Any sloppiness in code – such as embedding configuration parameters into code or using other methods for accessing the network or the parameters – will lead to non-compliant scripts.

The Myth of Causality

Can the Maelstrom approach be applied to the problem of inferring the 'causes' of network problems? Probably not. The question of what 'caused' a specific problem usually turns out to be ill-formed. This is mainly because the action that seemingly repaired a problem need not be directly related to the true cause of the problem.

Proposition 8: **Causality is an abstract ideal that is meaningless in practice.**

Foremost, the reason for this is that any claim that the Maelstrom process can make about causality is only true about the computing environment in which it was observed. This environment is constantly changing, and many changes are not observable, so no prior inference can be useful in predicting future behavior reliably. Theoretical 'cause/effect' relationships can be used to guide the troubleshooting process, but these cannot reliably be inferred from observations during that process.

The concept of causality is plagued by 'latent variables' that affect the function of a system without being recognized as having any effect. Most recently, months of carefully checking DHCP and router configurations for a DHCP problem ended when we observed a DHCP server answering an ARP request with an incorrect IP address. The 'latent variable' turned out to be a defective network driver on the DHCP server – something we had never considered. The idea that troubleshooting is often a search for 'latent variables' justifies the 'phenomenological modularity' that allows one to easily add tests to a decision tree.

Eells [12] points out that the latent variable problem is much more severe than this simple case study illustrates. He shows that by an appropriate ignorance of latent variables one can 'prove' that smoking *prevents* cancer. In his example, one latent variable is the location of the test subjects, either in large cities or on rural farms.

While network debugging by lower-level staff can be represented by decision trees, persistent problems referred to the highest level of triage almost always involve latent variables that are not part of the decision tree at all. Thus it becomes very important to be able to add tests to the debugging process as latent variables are discovered without affecting the global integrity of the debugging process. This need justifies use of 'phenomenological modularity' to assure interoperability between scripts, and the complete intractability of determining precedences between the various tests justifies the execution time that it takes to infer the precedences automatically.

The idea of causality cannot be even exhibited within scripts that are appropriately homogeneous and convergent. In fact:

Proposition 9: **Script convergence obscures causal relationships.**

Causal relationships take the form of scripts (or actions) A, B, and C, where C always succeeds after A and always fails after B. In this case, A and B cannot be homogeneously convergent with one another, because they sufficiently diverge in what they do to either assure or break C. To add to this quandry, it is seldom true that a script enables or disables another in *all possible conditions*. This is also a mathematical idea that never appears in practice.

Limitations

When scripts are convergent, this does not imply that they are non-disruptive. Convergence is a postcondition describing the *resulting* state of the network *after* the script completes. A convergent script can still make *arbitrary* changes *while it executes*, up to rebooting servers or even re-routing services to new servers. In this sense our definition is less stringent that that of Cfengine [2, 3, 4], which requires that a script avoid creating non-conforming equipment states while executing (which prevents it from rebooting servers).

This fact is the root of several limitations of the method. First, the method is inherently *serial*, because convergence and homogeneity are not otherwise guaranteed. *Maelstrom*, unlike make, cannot run scripts in parallel without presuming complete orthogonality of script domains of change. Future versions of *Maelstrom* may allow one to declare scripts as independent, to allow parallelism.

Perhaps the most severe hidden limit is the subtlety with which scripts must be developed in order to exhibit true and guaranteed homogeneity. The only reliable ways to assure this in practice are to use only scripts that administer non-overlapping domains and cannot behave inhomogeneously, or to force all scripts with overlapping domains to retrieve configuration information from a common and shared source.

Conclusions

Maelstrom is a relatively simple tool. But it is a result of complex and perhaps controversial changes in our thinking about the troubleshooting process. What we want is not necessarily what we need. It would be nice if we could predefine the script that accomplishes troubleshooting, by implementing a decision tree as a script. This does not work. It would be nice if we could predefine precedences between troubleshooting tasks. We cannot. It would be nice if the result of scripting was a description of the problem that was solved. As we often do not know the causes of problems when solving them as human troubleshooters, this is an idle dream.

The lesson of *Maelstrom* is that there are things we can do to automate troubleshooting without running into these roadblocks. We can employ convergence as a *replacement* for causal theory. We can concentrate on effects, and base the modularity of our automation upon consistency and agreement upon the effects we wish to achieve, and reliable reporting of effects, rather than agreement upon software interfaces or platforms. We can compensate for lack of agreement by sequencing.

But taking these steps also requires casting aside some commonly held values. We must remember that machine labor is cheap while staff labor is expensive, so that an inefficient machine process is sometimes preferable to an efficient one performed by a human. In utilizing machine labor, however, it is important to keep the automated process understandable and repeatable by a human, so that it can be verified, validated, and improved. *Maelstrom* does not use the most efficient strategy, but the most understandable one.

Research is itself a convergent process, so it is not surprising that many others have faced the same problems in other areas and come to some of the same conclusions. The need to limit ourselves to cases that are observable is echoed in current work on testing of distributed software systems [6, 11]. *Safety* (freedom from undesirable states) and *liveness* (freedom from deadlocks) are goals of both software engineering and system administration. Safety and liveness are difficult to assure, even when one has a complete model of what proper operation should be – something a network administrator often lacks.

It is often claimed that learning to be a painter is not so much learning to paint as it is learning to *see*. Here we have the same effect; we need to learn to trust our senses rather than our theories, our results more than our models. Only then can we sort out the maelstrom that is troubleshooting, and can truly know that we can "paint what we see."

Availability

Maelstrom is freely available from http:://www.eecs.tufts.edu/˜couch/maelstrom . It is written in Perl-5 to be portable to almost every UNIX system in creation.

Acknowledgments

A work such as this is not the work of any individuals, but the product of an intellectual community of thinkers all willing to look beyond the obvious. We are indebted to every member of that community who stopped by for awhile to talk about the issues. Scott Pustay and John Hart were both willing ears as we initially developed these ideas. When things started taking shape, Michael Gilfix stepped in with many good ideas, applications, and even source code; the current neatness of *Maelstrom* is his doing. Our tireless system administrators Andy Davidoff and Shawn Doughty contributed both thought and problems we could try to solve. Ambrose Kofi Laing provided valuable references on combinatorics. David Krumme helped with last-minute checking of typesetting and technical results. Mark Burgess, Frode Eika Sandnes, and Adam Moskowitz all provided valuable feedback on the paper and its ideas. Max Ben-Aaron provided valuable suggestions for copy editing and prose. All of these people, not just ourselves, created and nurtured this idea – and we all draw strength from the community we have formed, together.

Author Biographies

Alva L. Couch was born in Winston-Salem, North Carolina where he attended the North Carolina School of the Arts as a high school major in bassoon and contrabassoon performance. He received an S.B. in Architecture from M. I. T. in 1978, after which he worked for four years as a systems analyst and administrator at Harvard Medical School. Returning to school, he received an M.S. in Mathematics from Tufts in 1987, and a Ph.D. in Mathematics from Tufts in 1988. He became a member of the faculty of Tufts

Department of Computer Science in the fall of 1988, and is currently an Associate Professor of Electrical Engineering and Computer Science at Tufts. Prof. Couch is the author of several software systems for visualization and system administration, including Seecube (1987), Seeplex (1990), Slink (1996) Distr (1997), and Babble (2000). In 1996 he also received the Leibner Award for excellence in teaching and advising from Tufts. He has assisted in maintaining the Tufts computer systems for Computer Science teaching and research since 1985, when he was a Ph.D. student. He can be reached by surface mail at the Department of Electrical Engineering and Computer Science, 161 College Avenue, Tufts University, Medford, MA 02155. He can be reached via electronic mail as couch@eecs.tufts.edu. His work phone is (617)627-3674.

A senior while this work was in progress, Noah Daniels received his BS in Computer Science, cum laude, from Tufts University in May 2001, and formerly studied Physics at Swarthmore College. He is currently a systems engineer for Analog Devices, Inc. in Wilmington, Massachusetts. His current position entails supporting a large network of SunOS and Solaris-based microchip testers in the backend manufacturing division of Analog. Prior to this, Noah was a system and networking consultant for Interliant, Inc. (formerly Net Daemons, Associates) in Woburn, Massachusetts. His interests within the realm of computer science include algorithms, system administration, and the Darwin kernel. When not in front of a keyboard, Noah enjoys playing the violin, running, and being around (and attempting to ride) horses. Noah can be reached via email at ndaniels@eecs.tufts.edu, and via postal mail at 509 Main St. Apt. B, Waltham, MA 02452.

References

[1] Adleman, L., "Short Permutation Strings," *Discrete Mathematics* **10**, pp. 197-200, 1974.

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

[3] Burgess, M., and R. Ralston, "Distributed Resource Administration Using Cfengine," *Software: practice and experience* **27**, 1997.

[4] Burgess, M., "Computer Immunology," *Proc. LISA-XII*, 1998.

[5] Cai, M., "A New Bound on the Length of the Shortest String Containing all r-Permutations," *Discrete Mathematics* **39**, 1982, pp. 329-330.

[6] Cheung, S. C., and J. Kramer, "Checking Subsystem Safety Properties in Compositional Reachability Analysis," *Proceedings of the International Conference on Software Engineering*, Berlin, Germany, March 25-29, 1996.

[7] Couch, A., "SLINK: Simple, Effective Filesystem Maintenance Abstractions for Community-Based Administration," *Proc. Lisa-X*, Usenix Assoc., 1996.

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

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

[10] Alva L. Couch, "An Expectant Chat About Script Maturity," *Proc. LISA-XIV*, Usenix Assoc., 2000.

[11] Duri, S., U. Buy, R. Devarapalli, and S. M. Shatz, "Application and Experimental Evaluation of State Space Reduction Methods for Deadlock Analysis in Ada," *ACM Transactions on Software Engineering and Methodology*, Vol. 3, No. 4, ACM Press, 1994.

[12] Eells, Ellery, *Probabilistic Causality*, Cambridge University Press, 1991.

[13] Galbiati, G., and F. P. Preparata, "On Permutation-Embedding Sequences" *SIAM Journal on Applied Mathematics*, Vol. 30, No. 3, pp. 421-423, May, 1976.

[14] Gilfix, M., and A. Couch, "Peep (The Network Auralizer): Monitoring Your Network with Sound," *Proc. LISA-XIV*, Usenix Assoc., 2000.

[15] Goldfarb, C., and P. Prescod, *The XML Handbook, Second Edition*, Prentice-Hall, Inc., 2000.

[16] Hansen, S. and T. Atkins, "Centralized System Monitoring with Swatch," *Proc. LISA VII*, Usenix Assoc., 1993.

[17] Holgate, Matt, and Will Partain, "The Arusha Project: A Framework for Collaborative Unix System Administration," *Proc. LISA XV*, USENIX Assoc., San Diego CA, 2001.

[18] Holgate, Matt, Will Partain, et al., "The Arusha Project Web Site," http://ark.sourceforge.net.

[19] Kleitman, D. J., and D. J. Kwiatkowski, "A Lower Bound on the Length of a Sequence Containing All Permutations as Subsequences," *Journal of Combinatorial Theory (A)* **21**, pp. 129-136, 1976.

[20] Libes, D., *Exploring Expect*, O'Reilly and Assoc., 1994.

[21] McCabe, T., "A Software Complexity Measure," *IEEE Trans. Software Engineering 2*, 1976.

[22] Mohanty, S. P., "Shortest String Containing All Permutations," *Discrete Mathematics,* Vol. 31, pp. 91-95, 1980.

[23] Oram, A., and S. Talbot, *Managing Projects with Make, Second Edition*, O'Reilly and Associates, 1991.

[24] Osterlund, R., "PIKT: Problem Informant/Killer Tool," *Proc. LISA-XIV*, Usenix Assoc., 2000.

[25] Sandnes, Frode Eika, "Scheduling Partially Ordered events in a Randomised Framework – Empirical Results and Implications for

Automatic Configuration Management,'' *Proc. LISA XV*, USENIX Assoc., San Diego CA, 2001.

[26] Savage, C., ''Short Strings Containing All k-Element Permutations,'' *Discrete Mathematics*, Vol. 42, pp. 281-285, 1982.

[27] Schaffer, A. A., ''Shortest Prefix Strings Containing All Subset Permutations,'' *Discrete Mathematics,* Vol. 64, pp. 239-252, 1987,

[28] Trocki, J., ''Mon, the Server Monitoring Daemon,'' http://www.kernel.org/software/mon .