It's Elementary, Dear Watson: Applying Logic Programming To Convergent System Management Processes

Dr. Alva L. Couch and Michael Gilfix – Tufts University

ABSTRACT

In an ideal world, the system administrator would simply specify a complete model of system requirements and the system would automatically fulfill them. If requirements changed, or if the system deviated from requirements, the system would change itself to converge with requirements. Current specialized tools for convergent system administration already provide some ability to do this, but are limited by specification languages that cannot adequately represent all possible sets of requirements. We take the opposite approach of starting with a general-purpose logic programming language intended for specifying requirements and analyzing system state, and adapting that language for system administration. Using Prolog with appropriate extensions, one can specify complex system requirements and convergent processes involving multiple information domains, including information about files, filesystems, users, and processes, as well as information from databases. By hiding unimportant details, Prolog allows a simple relationship between requirements and the scripts that implement them. We illustrate these observations by use of a simple proof-of-concept prototype.

Introduction

Lately, the task of system configuration has been greatly eased by tools that automatically enforce compliance with a model of proper system operation and health via 'convergent processes' that detect and correct deviations from the model. System management then becomes a matter of crafting the model of appropriate behavior or configuration. This model contains 'rules' that specify proper behavior and configuration together with 'actions' that specify what to do to correct any discovered lack of compliance with a given rule.

Unfortunately, crafting such a model is difficult due to the number of different kinds of rules and actions involved in creating a complete model. These range from high-level operating policies to specification of dynamic operating behavior. First we must specify *operating policy*, a high-level description of how the system should behave and what services should be offered. We must then translate this high-level behavioral description to a description of the contents and disposition of system files that will insure this behavior. We can call this description the *static configuration* of the system, because the requirements it describes should not change over time, and typically we can insure these requirements are met with a single script or scripts, executed once. The system then begins operation and interprets these files in order to operate, so that other content does change over time. We can call the things that change over time part of the *dynamic configuration* of the system. To conform this to our requirements, we can craft *convergent processes* that observe the dynamic configuration of the system and modify system performance to match our models.

Static Configuration

There are now an endless variety of tools available for incrementally assuring desired static configuration of a system or network, beginning with the legacy of make [22] and rdist [8], both of which control file state based upon incremental generation and copying rules. The ideas in these tools are now pervasive and have made their way into almost all tools for configuration management. Package managers such as the RedHat Package Manager (RPM) [2] and Depot [7, 21, 28] only install requested software packages if those packages are not already present. Our own tool Slink [9, 10] and its relatives, including GNU Stow [14], incrementally modify a symbolic link tree to conform to a desired structure, while our own tool Distr [11] allows 'push' and 'pull' convergent file distribution, utilizing filters to translate file formats for differing platforms.

Cfengine [3, 4, 5] makes it possible to define and converge to very complex and expressive models of system state. Cfengine provides a powerful configuration language with built-in operations that act on files, links, directories, mounts, and even processes. Extensive built-in stream editing commands allow us to incrementally edit system files to conform with requirements, freeing us from having to store file prototypes on a master server.

All these static configuration tools share the same strengths and limits. Configuration files are relatively simple and easy to construct. The process by which one assures conformance with a configuration file is obvious and automatic. However, with the exception of a small number of database-driven prototypes, these tools specify system configuration at a fairly low level of abstraction, telling what to do to specific file contents.

One would like, instead, to simply list desired services and have the tool determine what to place in each file to implement each desired service. An ideal tool would query a distributed database or directory service, such as the Lightweight Directory Access Protocol (LDAP), for a high-level description of the services required on the system in question. Based upon the list of services to be offered, the tool would then proceed to modify all files requiring changes in order to provide that service. This kind of configuration power would require that the configuration tool know the mapping between services and file contents for each target operating system. This, in turn, requires maintenance of rather simple, detailed databases of system information that have little or nothing to do with operating policy: where files are, how configuration files are structured, etc.

Dynamic configuration

Historically, dynamic configuration has been preserved and enforced by a completely different set of tools than those used for managing static configuration. While static configuration tools rely heavily on databases, lists, and other declarative mechanisms for specifying configuration, dynamic configuration tools have relied upon user-crafted scripts. To use a tool, the administrator specifies behavioral patterns to detect and scripts to execute when each pattern is detected.

Current dynamic configuration tools allow monitoring of dynamic state, including processes, logfiles, and filesystems. Early tools were system log monitors, such as Swatch [15] and the more recent LogSurfer [18], which can page operators or run other scripts when potentially harmful events are posted in the system logs. These simple monitors have evolved into powerful tools that can monitor global system state, including TripWire [16, 17], which checks whole filesystems for compliance with a previous recorded state, and SyncTree, which can restore previous states of a system even if they are changed maliciously by a hacker [19].

Many system administrators resign themselves to writing custom scripts to monitor and correct problems in UNIX networks. These scripts interact with the same UNIX commands, and perform the same tasks, but must be customized for each site and platform, leading to massive duplication of effort. PIKT [24] (pronounced 'picket') greatly reduces the effort in writing scripts for multiple platforms and tasks, by providing a class mechanism for determining applicability of script parts and a powerful set of built-in parsing primitives (reminiscent of command parsing available in Tcl/TK [23]) for accessing the text output of UNIX status commands such as ls and netstat. Similar scripts for different platforms can be organized into a single script with class qualifiers, where appropriate lines in the script will be utilized for each target platform.

PIKT works well but, like the custom scripts it allows one to catalog, there is an uncomfortable distance between the scripts that implement policy and the policies they implement. A policy that is relatively simple to describe, such as "delete all core files more than three days old" might be written as the find command:

```
find /home -name core -mtime 3 \
        -exec rm -rf {} -print;
```

or something even more esoteric. It can be quite difficult to work backward from an arbitrary script to its meaning.

Ideally, we should be able to document operating policy and automatically translate the documentation into scripts that implement the policy. Ironically, the typical administrator in a hurry will document only the scripts. To determine what operating policies are, one must read and interpret what the scripts mean. Most administrators are not paid to write scripts, but to insure quality of service, so that script writing is done in great haste and with no attention to readability or potential software life-cycle. So documenting the actual operating policy for a network requires reverse-engineering the operating policy from the scripts on an ongoing basis.

Again, we need some way to automatically translate between a high level description of operating policy and the script that implements it, so that we no longer have to read and understand a script to understand the policy it implements. There must be a way to craft scripts that is in some way 'closer' to the natural way we would describe policy: a language closer to specifying what we want rather than how to accomplish it.

Databases

Many administrators have come to rely on databases [12], both normal and directory-based (such as LDAP or NIS+), to describe static network state. A database is a structured data storage and retrieval method, consisting of tables of information, where each table is organized into rows and columns. Database information is usually manipulated and accessed by use of Sequential Query Language (SQL), which specifies how to access individual rows and columns, and how to create new tables whose rows and columns can then be accessed.

Databases have several advantages over plain files. They can be accessed from anywhere in a network using standardized network access methods. Isolated parts of a database table can be incrementally modified with no chance of corrupting other parts of the table. When information is volatile, but must be modified and accessed in small chunks, databases provide more reliability than unstructured files.

Databases become very useful in maintaining information about the *external* world outside the systems being maintained, such as information on each user's true identity and function. A typical application would be to record information about each user in a database, and then use that information to compute appropriate filesystem and mail quotas for the user.

Unfortunately, normal database access methods such as SQL do *not* allow one to specify how to *act* based upon database contents. One must call SQL from another language empowered to take action. So to use databases, we are forced to learn both SQL *and* a scripting language (such as Perl [26]) for crafting actions based upon SQL queries. How, then, can we utilize databases without having to simultaneously write in two scripting languages: one for database access and another to react to content?

Toward a 'Glue Language'

We seek to fulfill Burgess' dream of 'Computer Immunology' [5], in which a description of computer 'health' empowers computers to 'immunize' themselves against poor function, thus becoming self-repairing and correcting. We began the work of this paper by searching for a common language that one could use for specifying both static and dynamic configuration. If we could find such a 'glue language,' we would be one step closer to being able to write scripts that describe operating policies with true platform independence. The language had to be able to provide at least a superset of the combined capabilities of Cfengine and PIKT. As well, we desired a language that:

- allows both static and dynamic requirements, limits, and convergent processes to be specified with the same syntax.
- is extensible to provide interfaces to all conceivable kinds of data and actions.
- allows specification of high-level rules that codify all steps in providing one user service, so that users can simply ask for the service rather than describing its low-level modifications.
- interoperates well with structured forms of information storage such as databases and directory services.

We came to a surprising conclusion, even for us, that the closest existing language fitting that description is Prolog!

Prolog As a Database Query Language

Prolog [6] is a much misunderstood language with an somewhat undeserved reputation for inefficiency and difficulty of programming. In reality, Prolog is one of the most efficient mechanisms for making *queries into databases* and writing action scripts based upon database queries. As well, Prolog has unique *implicit* properties that make programs shorter and easier to read, by omitting details that the language can handle by itself. For example, both conditional statements and loops are implicit in Prolog, and their use is determined by context.

We explored the powers of this language by constructing a prototype interface between Prolog and the operating system on a single host, with the intent of creating Prolog utilities that duplicate the functionality of Cfengine and PIKT. Then, we experimented with the prototype to determine its strengths and weaknesses.

Prolog syntax

Programming in Prolog is very different from programming in a normal scripting language. Rather than saying what should happen, one declares what *should be true*. The Prolog interpreter translates those declarations into actions to perform. This is called *declarative programming*.

A Prolog 'program' consists of *facts* and *rules*. A *fact* can be thought of as a line entry in a table in a database. The fact:

```
login(couch).
```

says that there is a user whose login name is couch. It has a *functor name* of login and a single *argument* couch.

Facts can be pre-recorded in Prolog's databases, or can be computed by external functions written in C or other languages. For example, in our prototype, we compute facts of the form

```
passwd(couch,
  '3hit2839482912',
  1000,
  40,
  'Alva L. Couch',
  '/home/couch',
  '/usr/bin/tcsh').
```

with an external function (written in C) that scans the password table (as an NIS+ map) and reports its contents. This function implements the *functor* passwd of *arity seven* (seven arguments). This functor is named passwd/7 to distinguish it from other functors with the same name and differing numbers of arguments, which need not be related to it.

A Prolog *rule* tells how to make more complex facts from simpler ones. For example, the rule:

```
pig(Login):-
  passwd(Login,_,_,_,_,Home,_),
  du(Home,Usage),
  Usage>20000.
```

says that "Login is a pig if Login is a login name with home directory Home, Usage is the disk usage for that directory, and the disk usage is greater than 20 megabytes (approximately)."

A rule has a left hand and right hand side separated by :-. The left-hand side specifies the goal of the rule, which in this case is to find a value for the variable Login. The right-hand side consists of subgoals needed to accomplish a goal. The symbol :- is read 'if', and commas between terms on the right hand side represent 'and'. Login, Home, and Usage are *variables* because they begin with capital letters. The special symbol "_" (the *anonymous variable*) is a place holder that indicates that a value in a query should be ignored. In this rule, for example, we can ignore the user's password, uid, gid, name, and shell.

Queries

In order to get Prolog to actually do anything, one has to execute a *query*. This is a request to compute values of variables based upon known facts and rules. For example, to request a list of pigs, from the rule above, we could type this in the Prolog interpreter:

```
?- pig(X).
```

Prolog might respond:

```
X=couch ;
X=bgates ;
No.
```

The symbol ?- can be read as 'prove'. This query instructs Prolog to "find all X's such that pig(X) is true." Prolog responds with the first of these, couch. After each response, we type a ";" to tell Prolog to find the next value for X. The final No. indicates that there are no more matches.

Whenever Prolog needs to determine who is and who is not a pig, it uses the rule above to *compute* who all the pigs are. Prolog begins with no idea of who a pig is, and evaluates the subgoals in the rule on the right hand side of the 'if' from left to right. The first subgoal, passwd(Login,_,_,_,_,Home,_) sets Login to each login name in turn, and sets Home to the corresponding home directory of Login. Then the second term, du(Home,Usage) *computes* the disk usage for that directory (by scanning the home directory) and sets Usage to that value. The last subgoal, Usage>20000, checks the value Usage. If it is greater than 20000, the goal pig(Login) *succeeds*. This has the result of returning whatever Login value we found as the result of the query. In this case, we wanted X's, so each match is assigned to X and printed for us.

Backtracking

Queries are repeatedly satisfied through *backtracking*. To backtrack, Prolog backs up from right to left in the list of subgoals it is attempting to complete, and tries new values for variables. For example, we implemented passwd/7 so that when backtracking, passwd(Login,_,_,_,_,Home,_) will set Login and Home to the information for each user in the system, one per try. Through backtracking, the rule above can potentially check 2000 users for pigdom. After finding a new value for Login and Home, Prolog then continues trying to execute goals from left to right. Whenever it satisfies all subgoals of a goal, and gets to the end of the rule, Prolog returns a match.

In this way, Prolog enumerates all possible matches for each rule, as demonstrated above. Every Prolog goal potentially tries all reasonable values for each variable, so that one never has to write a 'for' loop in Prolog. This also means that the easiest program to write in Prolog is an infinite loop!

Backtracking is as tricky and dangerous as it is powerful. Suppose that instead of the preceding rule for pig/1, we wrote:

```
pig(Login):-
  du(Home,Usage),
  passwd(Login,_,_,_,_,Home,_),
  Usage>20000.
```

This never succeeds in finding any pigs, because Home and Usage are unbound when du(Home,Usage) gets called. The passwd/7 call must go first.

Performance is seriously affected by the way in which we write a Prolog rule. The original pig/1 rule finds all the pigs in a system roughly as quickly as a Perl script written to do the same thing. But suppose we instead write:

```
pig(Login):-
  passwd(Login,_,_,_,_,_,_),
  passwd(Login,_,_,_,_,Home,_),
  du(Home,Usage),
  Usage>20000.
```

This *means* the same exact thing as the original rule; the first subgoal requires that Login be a login name, while the second requires that Home be the corresponding home. Depending upon the cleverness with which we implement passwd/7, however, executing this query can take between twice and thousands of times as much time to execute, compared to the original rule.

Our first version of passwd/7 read and cached the whole NIS+ password table before returning each entry. This meant that the rule above did that *twice*. First, it backtracked through all values for Login, and then checked them against all values for Login in order to find a matching one and determine its Home. Since we have about 1000 users, the rule thus checked 1000 entries 1000 times in backtracking to satisfy both subgoals. This took forever.

We then re-implemented passwd/7 so that if Login is initially bound to a value, passwd/7 uses getpwnam rather than getpwent to match information instantly. The second subgoal passwd(Login,_,_,_,_,Home,_) thus does *not* try all combinations, but instead instantly returns the appropriate home directory. This change made the above example execute several thousand times faster, but it still takes about twice the time of the original, more efficient rule with one subgoal for passwd/7.

Unification

In converging toward system health, we wish to force our idea of what should be true to actually be true. In Prolog nomenclature, we wish to *unify* our idea of what reality should be with the state of a particular machine.

In a Prolog rule, variables begin their lives having no value whatsoever, and are called *unbound*. Variables become bound, or set, by being *unified* with constants or other variables. The way this works depends upon how built-in and external functions are designed, and functions can behave differently depending upon whether variables given to the function are bound or unbound. In evaluating the goal:

```
passwd(Login,_,_,_,_,Home,_)
```

much depends upon the *prior state* of the variables Login and Home before the goal executes.

1. if Login and Home are unbound, then the query tries to bind Login and Home to each valid pair in the password table (in our case, via NIS+ naming service).
2. If Login is bound but Home is unbound, then Home is unified with the corresponding home directory, if any.
3. If Home is bound but Login is unbound, then Login is unified with each login name with that home directory in turn (and there may be more than one)!
4. If Login and Home are both bound, then the query 'succeeds' if they are a valid pair, and 'fails' if they are not paired correctly.

The result is that after any 'success', Login and Home form a valid pair, regardless of how that pair arose. A general-purpose goal like this can be used in many contexts, with variables known and unknown, and will adapt to the context and respond appropriately. Given a small amount of information, of any kind, this rule determines the rest if possible. Prolog goals (the left hand sides of rules) are called *functors* (rather than functions) precisely because they are capable of setting variables in a very flexible way, and using the same variables as both input and output.

Goals That Modify Configuration

In using SQL to manipulate tables of configuration information, one must use a different language for actions than for queries. Not so in Prolog. There is no reason that a goal cannot modify the external world in order to satisfy a query. Consider the simple example of a goal that returns the owner and group of a file specified as a pathname:

```
path_owner(Path,Owner,Group).
```

This could be implemented in Prolog in several ways. For simplicity, and to avoid implementing too many external functions, we chose to implement the goal so that unbound attributes for a path are read from the filesystem, while the filesystem is *modified* to match bound attributes if possible. The query

```
?- path_owner('/etc/motd',
              Owner,Group).
```

unifies Owner and Group with the true owner and group of the file /etc/motd, and reports them, while the query

```
?- path_owner('/etc/motd',0,0).
```

attempts to *change* the file's owner and group to 0 (root) if they are not already both 0! The same function serves a dual purpose: it can *query* system state or *modify* that state to *unify it with specifications.* Thus unification need not only concern Prolog variables, but can be extended to modify the environment in which Prolog executes!

The ambiguity between goals and actions in Prolog can be exploited to construct many rules that implement both at once. If we write a goal copy(Source,Target) that insures that the file Source is identical with node Target, then goal succeeds if it *can make the files identical* and fails if it cannot. Then we can write:

```
?- copy('/Master/etc/motd',
        '/etc/motd').
```

to check and perhaps make a copy of a master file. Goals like this, which blur the distinction between doing something and checking it, are ideal for use in creating a configuration engine.

Such lingual power comes with a price. Structured programmers should cringe, because we have all been taught that programs should not contain 'side-effects' that change things other than program variables. In the above, we are executing 'ambiguous' goals that check and assure system states *solely for their side effects!* This can make Prolog programs difficult to interpret and maintain.

Careless implementation and use of convergent Prolog goals can lead to disaster. Suppose that we implemented the passwd/7 functor so that it was able to *set* any field to a desired value, as a side effect, as well as iterating over all password records. Then the query:

```
?- passwd(_,_,30,_,_,_,_).
```

would set everyone's user ID to 30! For our peace of mind, we have refrained from writing functors that are both iterators and convergent modifiers. Each of our custom functors either enumerates options or tries to assure individual conditions, not both!

Implicit Iteration

Suppose we want to tell all pigs what we think of them. In Prolog, iterating over all matches for a variable is *implicit*. One never has to write a for or foreach loop; any query will search through all possible options. For example, if we type:

```
?- pig(Login),
   email(Login,
         'you are a pig!',
         'oink!'),
   fail.
```

Prolog will mail the message 'you are a pig!' (with subject 'oink!') to all possible pigs, as determined by the rule above! First the goal pig(Login) tries to find a pig.

When one is found, the goal email mails a message to that pig. The magical thing here is the built-in Prolog goal fail. This goal *forces Prolog to backtrack* through all possible solutions to the preceeding goals. Thus *all pigs* get the message!

Implicit execution is both a curse and a blessing. The good news is that one never needs to write a 'for' loop again. But interpreting what Prolog code actually does can be difficult. A safe way to interpret a Prolog program is to mentally put the phrase "for all values of variables so that" in front of every goal.

Controlling Implicit Loops

Sometimes we may wish to *prohibit* this behavior, e.g., make an example of *one* pig. The special goal ! (the *cut*) tells Prolog *not* to try to backtrack to the goals on its left-hand side. We could type:

```
?- pig(Login),!,
   email(Login,
         'you are a pig!',
         'oink!'),
   fail.
```

This would find one pig, but the fail would not cause backtracking to find others. The cut's general meaning is actually a bit more subtle: when backtracking over a cut, the *parent goal* fails. The cut operates on the *problem* Prolog is trying to solve (in this case, the whole query), not the sequence of goals being utilized to solve the problem.

Prolog and Cfengine

So far we have shown only how to construct specific queries that have particular effects, by manually interacting with the Prolog interpreter. How, then, does one automate the process of configuring a complete system? As with Cfengine, we must craft a set of rules that describe when the system is healthy. We can learn much from how Cfengine rules accomplish the same task.

Cfengine is 'almost as powerful' as a real Prolog interpreter, and only falls short in its handling of variables and extensibility. To control whether and how to do something with a particular machine, Cfengine uses 'classes'. A 'class' is a variable that is either true or false, depending upon the machine in question. In configuring Cfengine, one qualifies each action with the conditions under which it should occur, expressed as a boolean expression of classes. For example, in the Cfengine code:

```
links:
  solaris.victim::
    /etc/sendmail.cf
      ->! mail/sendmail.cf
```

the link is only made if the classes solaris and victim are both true ('.' represents logical 'and').

Classes in Cfengine correspond roughly with Prolog facts of arity 0. Facts are present (and thus 'true') if they depict the current operating environment. When Cfengine starts executing, it discovers as many facts as it can about the system upon which it is executing. For a machine hillary, running Solaris 5.7, these include classes equivalent to the Prolog facts:

```
hillary.
solaris.
sunos_5_7.
```

as well as much other information about the machine and operating environment.

Cfengine variable assignments, such as

```
groups:
 victim = ( bill hillary monica )
```

appear to set victim to a string of names, but in actuality only determine whether the class victim is true or false. The class victim is true if one of the facts in the parens is true, false otherwise. These are not really assignment statements at all, but represent the Prolog rules:

```
victim:-bill.
victim:-hillary.
victim:-monica.
```

This says simply that anywhere victim appears, it is true if bill, hillary, or monica is true. Each of these is in turn true only if it represents the current machine name.

This simplicity gives Cfengine incredible speed, as it never needs to deal with classes with values other than true or false: true if a fact is true in this environment, false if not. Prolog is slower, but as a result of this slowness, gains the ability to customize administrative process far beyond Cfengine's capabilities.

Cfengine class qualifications correspond with qualification goals in Prolog rules, where all qualifications have arity 0. For example, the Cfengine rule:

```
links:
  solaris.victim::
    /etc/sendmail.cf
      ->! mail/sendmail.cf
    /etc/services
      ->! inet/services
```

(which makes a link from /etc/sendmail.cf to mail/sendmail.cf) corresponds with the Prolog rule:

```
links:-solaris,victim,
       link('mail/sendmail.cf',
            '/etc/sendmail.cf').
links:-solaris,victim,
       link('inet/services',
            '/etc/services').
```

Roughly translated: "If you want to do links, and you're executing under Solaris, and you're a victim, do these."

The control: section of Cfengine's configuration file specifies the sequence in which individual goals are assured. For example, the Cfengine configuration:

```
control:
  actionsequence = ( links copy )
```

(which says to do only symlinks and file copies, in that order) corresponds roughly to the Prolog code:

```
health:-links,fail.
health:-copy,fail.
?- health,fail.
```

We create an artificial goal health that represents the health of the whole computer. To assure health, we look at all aspects, including links and copy. The fail directives assure that we check all possible rules for each of these through backtracking.

Genericity

Cfengine has been a great inspiration to us, and is a crucial element in day-to-day operation of our site, but its limitations forced us to look elsewhere for a truly general-purpose language for configuring systems. Cfengine is fantastic for manipulating files, but is very difficult to use to create generic, platform-independent, reusable configuration instructions for implementing high-level services.

Using any tool, any time a system file can vary in location or format, one must construct a new special case rule or macro to handle the deviations. In Cfengine this process becomes unwieldy very quickly, as macros in Cfengine all inhabit one name space, and one must remember the meanings of all of them in order to write new rules.

For example, let us learn to deal with an inetd.conf file that moves between /etc/inetd.conf and /etc/inet/inetd.conf depending upon operating system. One can cope with this in Cfengine as follows:

```
editfiles:
 ftp.solaris::
  { /etc/inet/inetd.conf
    AppendIfNoSuchLine \
     "ftp stream tcp nowait root \
     /usr/sbin/in.ftpd in.ftpd"
  }
 ftp.osf::
  { /etc/inetd.conf
    AppendIfNoSuchLine \
    "ftp stream tcp nowait root \
     /usr/sbin/ftpd ftpd"
  }
```

To avoid unwieldy typesetting, we take some liberties with Cfengine examples; the \ is *not* recognized by Cfengine as a line break.

Using Cfengine macros, it is possible to code the same operation somewhat more neatly:

```
control:
 solaris::
  inetd = ( "/etc/inet/inetd.conf" )
  ftpd = ( "/usr/sbin/in.ftpd" )
  ftpd_base = ( "in.ftpd" )
 osf::
  inetd = ( "/etc/inetd.conf" )
  ftpd = ( "/usr/sbin/ftpd" )
  ftpd_base = ( "ftpd" )
editfiles:
 ftp::
  { $(inetd)
    AppendIfNoSuchLine \
     "ftp stream tcp nowait root \
     $(ftpd) $(ftpd_base)"
  }
```

The variables $(inetd), $(ftpd), and $(ftpd_base) represent varying quantities in an otherwise unvarying script.

This works fine, but has two significant drawbacks. These variables are macros, created by hand, and one cannot write a script to discover their values. Variables live in a flat name space, so that repeating this process leads to many variables and much to remember when writing configuration entries.

Using Prolog, one can accomplish the same task somewhat more neatly. First, we code a relation config_path(Name,OS,Path) into Prolog that relates the canonical name of a file with its location in the filesystem. This relation might start, e.g., with the tuples:

```
config_path(
  'inetd.conf', osf,
  '/etc/inetd.conf').
config_path(
  'inetd.conf', solaris,
  '/etc/inet/inetd.conf').
config_path(
  ftpd, osf,
  '/usr/sbin/ftpd').
config_path(
  ftpd, solaris,
  '/usr/sbin/in.ftpd').
```

This information concerns the nature of operating systems themselves, not their configuration, so that it *does not change with policies* and *should not vary with use*. By nature, thus, this information is fundamentally different than configuration information and should *not be present* in your policy description.

Then, using a functor os/1 that returns the generic name of the current operating system, one can write the rule:

```
editfiles:-
  os(Os),
  config_path('inetd.conf',Os,Path),
  config_path('ftpd',Os,Ftpd),
  file_base_name(Ftpd,FBase),
  appendIfNoSuchLine(Path,
    [ftp,stream,tcp,nowait,
     root,Ftpd,Fbase]).
```

In English, "If we know our operating system, and can determine the path of inetd.conf and ftpd, and can find the base name of ftpd, and can put a record into inetd.conf for it, we're done!" This series of goals queries for the correct location for inetd.conf and ftpd, computes the base name of the file from the ftpd location, dynamically constructs a line for the file by concatenation, and places that line into inetd.conf (after constructing the line from a Prolog list).

This does the exact same thing as the Cfengine example, but the Prolog code can be modified to do considerably more. Consider the rules in Listing 1. These rules compute the paths for these files by *actively probing the filesystem for their existence* This covers all cases *without* having to hand-code the locations for each operating system, using tests similar to those used by configure and autoconf.

Now let us go to an even higher level of abstraction: we should only do this when ftp is required. First, let's only use the rule when we need that service, by adding a 'guard clause' to the Prolog code:

```
editfiles:-
  service(ftp),
  os(Os),
  config_path('inetd.conf',Os,Path),
  config_path('ftpd',Os,Ftpd),
  file_base_name(Ftpd,FBase),
  appendIfNoSuchLine(Path,
    [ftp,stream,tcp,nowait,root,
                    Ftpd,Fbase]).
```

Then we write rules telling when a particular machine deserves the service:

```
service(ftp):-hostname(monica).
service(ftp):-os(osf).
```

This means that we should provide that service if our hostname is monica or our operating system is OSF! We have the *full power of Prolog* available for deciding which machines get the service. We could, e.g., write:

```
service(ftp):-not passwd(bgates,
                    _,_,_,_,_,_).
```

to install ftp only on machines where Mr. Bill does not have an account!

The rule that actually adds the appropriate line to inetd.conf is *not* part of operating policy. It is a *reusable method* that works in most cases. The actual policy is embodied, instead, by the rules for service/1. Ideally, we should be able to write the former once and never touch it again, then modify the latter to taste for each site and application.

This represents a major difference between using Prolog and other languages for configuration. Typically, when one gets to a high-enough lingual level to describe policy, low-level details (such as your user database!) become inaccessible. Prolog allows one to craft high-level, service-based policies that utilize data from all facets of the running system.

Configuring High-level Services

While Cfengine does a good job of implementing policies regarding individual files, it is quite awkward to describe how to implement high level services using Cfengine's syntax. Take, for example, the case of setting up an entire FTP server. Everyone knows that there are at least three actions involved in setting up a typical server within a UNIX-like operating system:

1. Add an appropriate line to inetd.conf.
2. Add appropriate port descriptions to services
3. Send a HUP signal to inetd.

In Cfengine, to define ftp on three machines bill, andhillary, monica, a mix of Solaris and OSF machines, these actions could be declared as follows:

```
control:
 solaris::
  inetd = ( "/etc/inet/inetd.conf" )
  ftpd = ( "/usr/sbin/in.ftpd" )
  ftpd_base = ( "in.ftpd" )
  services = ( "/etc/inet/services" )
 osf::
```

```
config_path(
 'inetd.conf', _, '/etc/inetd.conf'):-
  path_type('/etc/inet/inetd.conf',file),!.
config_path(
 'inetd.conf', _, '/etc/inetd.conf'):-
  path_type('/etc/inetd.conf',file),!.
config_path(
 ftpd, _, '/usr/sbin/ftpd'):-
  path_type('/usr/sbin/ftpd',file),!.
config_path(
  ftpd, _, '/usr/sbin/in.ftpd'):-
  path_type('/usr/sbin/in.ftpd',file),!.
config_path(
  ftpd, _, '/usr/etc/in.ftpd'):-
  path_type('/usr/etc/in.ftpd',file),!.
```

Listing 1: Rules to probe filesystem actively.

```
   inetd = ( "/etc/inetd.conf" )
   ftpd = ( "/usr/sbin/ftpd" )
   ftpd_base = ( "ftpd" )
   services = ( "/etc/services" )
 groups:
  ftp = ( bill hillary monica )
 editfiles:
  ftp::
  { $(inetd)
   AppendIfNoSuchLine \
     "ftp stream tcp nowait root \
      $(ftpd) $(ftpd_base)"
  }
  { $(services)
   AppendIfNoSuchLine "ftp-data 20/tcp"
   AppendIfNoSuchLine "ftp 21/tcp"
  }
 processes:
  ftp::
   "inetd" signal=hup
```

The control section describes locations of files for each platform. The groups section defines hosts that need to provide ftp as a 'logical macro' that is true if the current host is a member, false if not. The editfiles section tells what to do to inetd.conf and services for these hosts. Within this, there are variations depending upon whether the operating system for the host is Solaris or OSF. Finally, the processes section describes what to do to running processes, i.e., send a HUP signal to inetd.

This approach has several advantages. Each action is done at most once, even if needed in several cases, e.g., inetd is only HUP'd once even if several changes are made to inetd.conf. Once ftp configuration is described for each operating system, one need only list the machines that should support it.

But there are several problems with this approach from our perspective. It is difficult to specify parameters that modify implementation, e.g., using TCP wrappers for security [25], passing parameters to daemons, etc. Each variant requires that a new class and/or macro be created, and these exist in a global name space. Actions of each type are done 'all together' with no concept of installing 'one service at a time'. Thus there is no concept of transaction integrity or transaction rollback if necessary for any reason. Variables and classes have global scope, so in every section of the file, we must remember that solaris represents an operating system, while hillary represents a machine name, and we cannot name a machine solaris without serious problems!

How does the same complex example look in Prolog? First, let's list the hosts that should have ftp service in Prolog rules:

```
service(ftp):-hostname(bill).
service(ftp):-hostname(hillary).
service(ftp):-hostname(monica).
```

We add code to the above example specifying where extra files are:

```
config_path('services',solaris,
            '/etc/inet/services').
config_path('services',osf,
            '/etc/services').
```

Then we add a list of goals that implement ftp service, in the manner of the above:

```
ftp:-
 service(ftp),
 os(Os),
 config_path('inetd.conf',Os,Inetd),
 config_path('ftpd',Os,Ftpd),
 config_path('services',Os,Services),
 file_base_name(Ftpd,FBase),
 appendIfNoSuchLine(Inetd,
  [ftp,stream,tcp,nowait,root,
   Ftpd,Fbase]),
 appendIfNoSuchLine(Services,
  ['ftp-data','20/tcp']),
 appendIfNoSuchLine(Services,
  ['ftp','21/tcp']),
 kill(inetd,hup).
```

In English,
"If we have a host name;
and we know our operating system;
and we know where inetd.conf, services, and ftpd live;
and we can put a record into inetd.conf;
and we can put two records into services;
and we can send a hup to inetd;
then ftp is installed."

Each clause guards against poor installation, by making constant 'sanity checks' and aborting if any one check fails.

In this example, we have done something that is very difficult to accomplish in any current configuration tool. The actual script that implements the ftp service is *generic* and *independent of architecture*. It will work for any host provided that file location tables are kept up to date. Customization is only required for service(ftp), which must be changed to reflect current policies and desires.

We chose in the prototype to describe services and system attributes very differently from in the way they are described in Cfengine. The class hillary in Cfengine is the goal hostname(hillary) in Prolog, while Cfengine's fact solaris becomes os(solaris). Thus facts are no longer filed in a flat name space, and we can distinguish between solaris the operating system and solaris the host name, if any. This extra work sidesteps inherent ambiguities in the meaning of Cfengine's class names.

Atomicity and Rollback

Another significant cost to Cfengine's remarkable efficiency is that there is no provision for recovery from partial configuration failures. Let's craft a Prolog example that undoes a configuration if any part of it fails. This will have the effect of making the installation more of an *atomic* act, one indivisible thing in which several changes are coordinated to achieve one effect. If any change fails, a *rollback* script will undo the other changes so that system integrity is maintained.

This example will contain two goals for ftp, one that installs it and one that removes it if anything goes wrong. First, we add a Prolog cut (!) as the last goal in the above installation script. This tells Prolog that when all subgoals are complete, the ftp goal itself is complete and no further work should be done on it. We then follow this rule with another that only gets executed if the cut is *not* encountered:

```
ftp:-
  service(ftp),
  os(Os),
  path('inetd.conf',Os,Inetd),
  path('ftpd',Os,Ftpd),
  path('services',Os,Services),
  file_base_name(Ftpd,FBase),
  deleteLinesContaining(Inetd,Ftpd),
  deleteLinesContaining(Services,
                        '20/tcp'),
  deleteLinesContaining(Services,
                        '21/tcp'),
  kill(inetd,hup).
```

When Prolog tries to do something, it tries every relevant rule in its database of rules, in the order in which they appear in its program. When asked to handle the rule for ftp, Prolog will begin by trying the initial rule we crafted. If that rule succeeds, then because of the ! (cut) at the end, Prolog will stop working on that goal. If the initial rule fails, it will try the next rule, which uninstalls ftp service.

This example shows the true power of implicit goal execution. As a script, this behavior would be a nightmare to describe, but in Prolog, it is a simple series of two rules, each of which is tried if the last one fails. Thus a rather complex logical chain of deduction is reduced to a relatively simple list of requirements.

Cfengine and Dynamic Policy

Cfengine only implements dynamic policy where the map from policy to process is relatively obvious, or the user is willing to allow Cfengine to make arbitrary decisions concerning the mapping. For example, if a user has temporary files that are too old, most everyone agrees that the obvious thing to do is to delete them, and Cfengine can do this easily. If, however, one wishes to archive them on tape or writeable CDROM, Cfengine cannot help very much, and one must write a custom script.

As a worst-case example, it is not so obvious that everyone should use the NFS disk management strategy imposed by Cfengine simply because Cfengine supports only that strategy. Unless one conforms somewhat precisely to a rather elaborate scheme, including a naming convention for network directories containing the name of the server, several features of Cfengine are unavailable. As we feel that mount points should be machine-independent (from bitter experience in moving user files and having to repair user scripts), we choose not to utilize these Cfengine features. Cfengine tries to impose a rather significant operating policy decision upon us – one we cannot afford to allow.

In implementing most dynamic policies, the map from policy to convergent process is so ill-defined that it becomes a policy decision itself. For example, at our site, users can gain access to a 'temporary storage' area that has no quota, for the purpose of doing things that require more storage than will fit into their home directories. But we would like to impose a time limit on peoples' use of that storage that is different from a normal quota. Suppose we find out that a user is using a large amount of temporary storage for too long. How do we 'correct' that state? We could:

1. Delete some files randomly from temporary storage and mail a message.
2. Mail a warning, wait a week, then lock the account until the user comes by to talk about the problem.
3. Invoke a temporary quota on the temporary storage area for this user, to force the user to clean up.
4. Write email to a system administrator describing the problem.

Clearly, the option we choose determines much about how users work and feel.

Prolog and PIKT

We chose Prolog as our prototyping language partially because of the intimate relationship between it and Cfengine. But we also wished to be able to control dynamic state, including manipulating user files, filesystems, and processes. The powers and ease of use of PIKT [24] inspired us to attempt to add those powers to the prototype without compromising Cfengine-like behavior.

There are remarkable similarities between Cfengine and PIKT. Both implement roughly the same idea of classes, but to slightly different ends. In Cfengine, a class is a guard mechanism that determines which rules apply. In PIKT, a class instead determines which lines are used in a script. In both, classes are primarily a portability mechanism. In Cfengine, classes insulate one from differences in file layout and location, while in PIKT, classes determine which scripts apply to which operating systems, and help one cope with differences in command output formats between operating systems. Cfengine's classes are logical variables, while in PIKT, classes are variables in the C preprocessor that become defined or undefined for each platform. Portability in PIKT's scripts is accomplished much like portability in C programs, by enclosing variants in preprocessor #if... #endif directives.

Like Cfengine, PIKT's configuration is separated into several distinct parts. While Cfengine operates on files, links, and processes, PIKT acts to detect alarm conditions and perform appropriate actions. In understanding how we can implement PIKT functions in Prolog, there are four parts to consider: classes, macros, alarms, and actions.

PIKT classes are specified much as in Cfengine. The class of three machines bill, hillary, and monica (from before) might be constructed as:

```
watch
   members bill hillary monica
```

However, PIKT interprets classes only on a master script server, not on the target machine being configured. This server uses class definitions and preprocessor directives to adapt generic master scripts to execute on the target machine, and then ships a class-free script to the target machine for actual execution.

The resulting scripts are very efficient, because many decisions have been made before shipping the script to the target machine. However, this also means that PIKT scripts cannot rely on any form of knowledge discovery in creating classes, as in Cfengine and Prolog. Classes that Cfengine and Prolog can discover automatically must be explicitly declared by hand in PIKT, including machine type and operating system version.

In PIKT, as in Cfengine, macros can be used to code file locations and other local dependencies. For example, in the PIKT file macros.cfg, one might write:

```
#if solaris
fstab    /etc/vfstab
#endif
#if osf
fstab    /etc/fstab
#endif
```

This makes the macro =fstab evaluate to /etc/vfstab when executing within Solaris and /etc/fstab within OSF.

Alarms in PIKT are specified by listing, for each host, a set of scripts to be run periodically to check for system problems. Each script is configured to execute regularly and take action if needed. This feature cannot be emulated by our prototype, but the Prolog interpreter can always be run periodically under control of the cron periodic execution daemon.

Scripts in PIKT operate on variables read from logfiles and the output of UNIX commands. For example, the script:

```
AnnoyBill
  init
    status active
    level critical
    task "Harass Bill"
    input proc "=w | =grep clinton"
    dat $tty 2
  rule
    exec wait =write \
          clinton $tty < =mesg
```

will check periodically whether Mr. Bill is logged in, and write an undisclosed message to each terminal on which he is working!

The init section describes conditions under which to do something. The input statement describes a filter that only generates input to which the rule should be applied. In this case, the rule will be applied whenever any line in the output of the w command contains the string clinton. The second field of that line (the tty field) will be assigned to the variable $tty before invoking the rule. The rule will call the program write to send a message to that tty, where the macros =write and =mesg must describe the locations of the write command and message file, respectively.

It is not surprising that such a rule-based execution is very easy to accomplish in Prolog. In our prototype, an equivalent rule looks like this:

```
annoyBill:-
  os(Os),
  command_path(w,Os,WPath),
  output_tail(WPath,1,Out),
  split(Out,'[ \t][ \t]*',[clinton,Tty|_]),
  command_path(write,Os,WritePath),
  file_path(message,Os,MessPath),
  concat_atom(
    [WritePath,clinton,Tty,
       '<',MessPath],' ',Command),
  system(Command).
?- annoyBill,fail.
```

The command_path/3 and file_path/3 goals compute where needed commands and files live. output_tail/3 executes a command and then binds Out successively to each line after the first during backtracking. split/3 splits this line into fields at spaces, assigning the second field to Tty only if the first field is clinton. When this happens, the write command is built and executed.

Because each Prolog subgoal acts as a natural filter that limits further operations to valid data, Prolog easily emulates the function of the PIKT script while adding additional safeguards against erroneous operation. The PIKT script will misbehave if one forgets to define =w, for example, while the Prolog rule will stop executing in that case. Admittedly, this is much less efficient that computing system dependencies before running the script, as PIKT does, but PIKT functions can be emulated with some performance loss.

Unlike PIKT, however, it is easy to write this rule 'at a higher level of abstraction,' by hiding system dependencies in subgoals. Consider the rule:

```
w(User,Tty):-
  os(Os),
  command_path(w,Os,WPath),
  output_tail(WPath,1,Out),
  split(Out,'[ \t][ \t]*',[User,Tty|_]).
```

This rule tells how to execute a w command and present the results through backtracking. Once this is written, the above script can be written:

```
annoyBill:-
 w(clinton,Tty),
 os(Os),
 command_path(write,Os,WritePath),
 file_path(message,Os,MessPath),
 concat_atom(
   [WritePath,clinton,Tty,'<',
          MessPath],' ',Command),
 system(Command).
```

Just like writing a subroutine in a script, writing the w/2 subgoal allows one to forget about the details of running the w command and concentrate on the action to take. One can do the same with the action of writing the message to make the rule even more readable:

```
annoyBill:-
 w(clinton,Tty),
 file_path(message,Os,MessPath),
 write(clinton,Tty,MessPath).
```

The Prototype

Our prototype Prolog system administration interface is based upon SWI-Prolog 3.7.2 [27], a freely available interpreter that executes both under UNIX and NT. Its many features include built-in functions for manipulating files and the ability to call dynamically loaded C functions from within Prolog programs. This made it easy to adapt the language for administrative tasks.

We explored the potential of using logic programming for configuration control by writing rather simple 'interface' rules, like the ones in the above examples, that expose system configuration and dynamic state, and perform common actions. Then we tried to do the same things with Prolog that we would do with normal tools and scripts.

The prototype's system interface began as a very simple hack. When a kind of system fact was needed, the Prolog program executed a Perl utility called "glue." This utility queried the system and provided the results as Prolog facts. Whenever any fact of a certain kind was needed, all such facts were loaded, to minimize external system calls and avoid program execution overhead. This was a very rough approximation to what should really be done, and with great effort, we converted the prototype to use the shared library support built into the SWI-Prolog interpreter. This allows information to be transferred directly from system calls into the interpreter with no script execution overhead. The current prototype can access system information as quickly as a C program using the same system calls.

Alas, this is only a prototype and has several serious limitations. The prototype only compiles under Sun Solaris (which the bulk of our hosts utilize), and little effort has been made to port it to other architectures. There is no provision for file transfer between hosts, and file editing is extremely primitive by Cfengine standards. The interface implements very few of the capabilities built into Cfengine: just enough to perform some convincing tests, as above. Results of these tests, however, strongly encourage us to implement more features as time allows.

Performance

It may seem, from the preceeding examples, that programming in Prolog is easy. This is false! Our prototype simply hides the details of real programming from the administrator, by providing predefined rules one can use. These predefined rules are actually quite complex to craft with any kind of efficiency.

Prolog excels at implying complexity from form. Usually, a simple statement of what should happen is enough to make it happen: Prolog infers the process to do this from the needs one describes. It is very difficult, however, to make Prolog do something efficiently, because there are many valid descriptions of how to achieve the same effects, with great variations in performance.

In any high-level approach, we trade specificity and control for ease of use. One of the reasons Prolog is so attractive to us is that describing an instance suffices to operate on all instances. Because of this power, however, we lose the ability to easily control iteration in the way to which we are accustomed when writing scripts. In particular, there is no easy way to craft a nested loop that iterates over all pairs in the same set.

Here is a real life example of something difficult to do in Prolog. We have a directory full of electronic submissions of homework that we would like to check for similarity. To do this, we would like to run diff on all pairs and locate pairs with few differences. We could write:

```
diff:-
 expand_file_name('*',Nodes),
 member(File1, Nodes),
 member(File2, Nodes),
 concat_atom_chars(['diff',File1,
              File2],' ',Command),
 system(Command).
?- diff,fail.
```

In this code, we first obtain a Prolog list of all files in the current directory, then select File1 and File2 from this list and compare them. It is not so obvious that this does twice as much work as necessary, because the two implied loops for selecting files not only compare files against themselves, but also against all other files in both orders. This does more than twice as many comparisons as we need.

The most efficient implementation of this loop is:

```
pair([File1|Rest],File1,File2):-
 member(File2,Rest).
pair([_|Rest],File1,File2):-
 pair(Rest,File1,File2).
```

```
diff:-
 expand_file_name('*',Nodes),
 pair(Nodes,File1,File2),
 concat_atom_chars(
  ['diff',File1,File2],
  ' ',Command),
 system(Command).
?- diff,fail.
```

In English,
"To diff all files,
get all filenames in a list;
select pairs from that list;
and diff them."
The complexity here is in pair/3: "To select a pair, make it the first element of the list along with some other, or repeat that process with some suffix of the original list." Prolog efficiency is *not* an oxymoron, but it takes an expert Prolog programmer to consistently generate non-trivial and efficient Prolog code.

Conclusions

In no way is our prototype the equal of Cfengine or PIKT, but it does have very important capabilities not present in either. Using the prototype, one can concentrate on what should happen in each case, and leave scripting of the actual changes to Prolog itself. With a few more relatively simple extensions, Prolog can in principle accomplish anything that either of these tools can do. Prolog programs are not subject to either the assumptions built into Cfengine or the lack of dynamic probing capabilities in PIKT. Prolog scripts are roughly the same length as scripts in either Cfengine or PIKT, but much easier for an expert Prolog programmer to refine for readability and extend for new capabilities. The chain of logical deduction involved in Prolog execution is a close match with the way configuration processes should work. Goals to be used in that chain can be crafted to force the system into compliance with requirements, or to actively probe for system problems. Custom scripts to accomplish special purposes can be written in Prolog without recourse to external scripting languages.

The basic programming metaphor for Prolog, *unification*, matches exactly what we have to do in creating a convergent process: to *unify* the rules with the system so that both "describe the same thing." This is not an easy task in any sense, but the fact that the language in some sense "matches the problem" makes crafting complex processes easier than when using normal scripting languages or less flexible configuration languages.

Our prototype illustrates several important lessons about the problem of configuration and the power of language. It is possible to configure a system using a language in which describing a single instance of a problem suffices to repair all instances. It is possible to craft service installation scripts that are truly generic, so that the administrator need not program, but simply correctly populate databases describing the system and desired behavior. It is possible to separate data concerning the invariant system from data describing operating policies. It is possible to describe invariant system data once and reuse it for all similar cases. It is possible to describe both static and dynamic configuration issues with a single language.

The prototype also reiterates lessons we have learned whenever we attempt to operate 'at a higher level' than existing tools. Simplicity of a language reduces the ability to craft efficient programs. Language flexibility increases the potential for confusion in reading programs. Undisciplined use of a flexible language leads to unpredictable results.

We do *not* suggest that every administrator should learn to program in Prolog! Prolog programs are difficult to write correctly and efficiently. Even in this simple prototype, one must often repeat code in several rules in order to emulate classes utilized in one or two lines of Cfengine or PIKT configuration. We do not view Prolog as a language for administrators to use directly, but as an *assembly language* into which even higher level descriptions can eventually be compiled. This common language for both static and dynamic configuration management, though cumbersome in its raw form, can be made much more friendly by some relatively simple syntactic translations.

Ideally, a true Prolog configuration tool would handle *all* the low level details and portability issues, leaving us to decide the high-level policies to implement. File and command location databases could be built with configure, to be used in generic implementation routines as needed. Configuring the system would consist of deciding which services to offer and which periodic configuration tests to enable. Custom Prolog programming would only be needed if an administrator wished to extend the tool's capabilities by adding new services or tests. Our prototype is nowhere near this ideal, but shows us that this ideal is possible to attain with effort.

Future Work

Our work on the prototype has only just begun. We have a long 'wish-list' of features to add, all of which are relatively straightforward to implement. This list includes giving the Prolog interpreter:

1. generic interfaces to SQL, LDAP, and NIS+ database services, both to request and modify data.
2. domain-specific interfaces to system files, both for scanning and updating file contents.
3. extensive, Cfengine-like file editing capabilities.
4. a generic interface to the Simple Network Management Protocol (SNMP) [20].
5. the ability to request master files from Cfengine configuration servers.

We are also involved in writing a preprocessor that will translate easier-to-use configuration instructions into Prolog, avoiding the need for anyone but the system designer to code in Prolog directly.

A Simple Dream

Everyone working in configuration management would like to find a way to simplify and perhaps obsolete this dreary job. The 'impossible dream' is that everyone writing configuration scripts can use the work of the whole community instead of re-inventing the wheel for each site and purpose. But so far, while configuration tools proliferate, it has been difficult to convince people to distribute and maintain reusable scripts for configuring systems and implementing common services. There seem to be "too many options," "too many system dependencies," and "too many site-specific assumptions."

Our work shows that using logic programming, one can break the problem of service installation into small steps so that the deliverables at each step will be maintainable. These steps can be coded in a single language with wide applicability. This language may not be Prolog, but all evidence suggests that it will have quite similar capabilities. We will know we have succeeded when 'unification' is something we can do to human effort as well as system behavior.

Acknowledgements

Many people inspired and otherwise contributed to this work. We are forever indebted to Mark Burgess, whose work on Cfengine showed us that our dreams were possible, and inspired us to look beyond the obvious for a better way. We thank Jan Weilemaker for providing SWI-Prolog, without which writing the prototype would have been difficult if not impossible. We especially thank him for timely and extremely helpful responses to our bug reports on SWI-Prolog and for putting up with our lack of understanding of how to write foreign extensions to Prolog. We thank Robert Osterlund for PIKT, and for showing us both the difficulty of dynamic monitoring, and the way to effectively handle script heterogeneity in a complex and varied environment. Finally, we thank the EECS systems staff, George Preble and Warren Gagosian, for being there when we needed them and keeping our systems running smoothly.

Author Information

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) and Distr (1997). 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. With a lot of help, Prof. Couch still maintains the largest independent departmental computer network at Tufts in the department of Electrical Engineering and Computer Science. 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.

Michael Gilfix was born in Winnepeg, Canada on Aug. 3rd, 1980. He presently resides in Montreal, Canada, where he attended high school at Lower Canada College. He is currently a sophomore at Tufts University, double-majoring in Electrical Engineering and Computer Science. His many interests include Jazz and Rock music, playing improvisational guitar, movies, and shooting pool. He can be reached by electronic mail as <mgilfix@eecs.tufts.edu>.

References

[1] P. Anderson, "Towards a High-Level Machine Configuration System" *Proc. LISA-VIII*, Usenix Assoc., 1994.

[2] E. Bailey, *Maximum RPM*, Red Hat Press, 1997.

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

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

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

[6] W. F. Clocksin and C. F. Mellish, *Programming in Prolog, Fourth Edition*, Springer-Verlag, Inc., 1994.

[7] Wallace Colyer and Walter Wong, "Depot: a Tool for Managing Software Environments," *Proc. LISA-VI*, Usenix Assoc., 1992.

[8] Michael Cooper, "Overhauling Rdist for the '90's," *Proc. LISA-VI*, Usenix Assoc., 1992.

[9] Alva Couch and Greg Owen, "Managing Large Software Repositories with SLINK," *Proc. SANS-95*, 1995.

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

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

[12] C. J. Date, *An Introduction to Database Systems, Sixth Edition*, Addison-Wesley, Inc., 1995.

[13] R. Evard, "An Analysis of UNIX Machine Configuration," *Proc. LISA-XI*, Usenix Assoc., 1997.
[14] B. Glickstein, "GNU Stow," http://www.gnu.org/software/sto w.
[15] S. Hansen and T. Atkins, "Centralized System Monitoring With Swatch," *Proc. LISA-VII*, Usenix Assoc., 1993.
[16] G. Kim and E. Spafford, "Monitoring File System Integrity on UNIX Platforms," InfoSecurity News **4** (4), July 1993.
[17] G. Kim and E. Spafford, "Experiences with TripWire: Using Integrity Checkers for Intrusion Detection," *Proc. System Administration, Networking, and Security-III*, Usenix Assoc., 1994.
[18] W. Ley, "LogSurfer Homepage," http://www.fwl.dfn.de/eng/logsurf/home.html .
[19] J. Lockard and J. Larke, "Synctree for Single Point Installation, Upgrades, and OS Patches," *Proc. LISA-XII*, Usenix Assoc., 1998.
[20] J. Murray, *Windows-NT SNMP: Simple Network Management Protocol*, O'Reilly and Assoc, 1997.
[21] K. Manheimer, B. Warsaw, S. Clark, and W. Rowe, "The Depot: a Framework for Sharing Software Installation Across Organizational and UNIX Platform Boundaries," *Proc. LISA-IV*, Usenix Assoc., 1990.
[22] A. Oram and S. Talbot, *Managing Projects with Make, 2nd Edition*, O'Reilly and Associates, 1991.
[23] J. Ousterhout, *TCL and the TK Toolkit*, Addison-Wesley, Inc., 1994.
[24] R. Osterlund, "PIKT Web Site," http://pikt.uchicago.edu/pikt ..
[25] W. Venema, "TCP WRAPPER, Network Monitoring, Access Control and Booby Traps," *Proc. UNIX Security Symposium III*, September 1992.
[26] L. Wall, T. Christiansen, and R. Schwartz, *Programming Perl, 2nd edition*, O'Reilly and Assoc., 1996.
[27] J. Weilemaker, "SWI Prolog Web Site," http://www.swi.psy.uva.nl/projects/SWI-Prolog .
[28] Walter C. Wong, "Local Disk Depot – Customizing the Software Environment" *Proc. LISA-VII*, Usenix Assoc., 1993.