An Expectant Chat about Script Maturity

Dr. Alva L. Couch – Tufts University

ABSTRACT

Using scripts to automate common administrative tasks is a ubiquitous practice. Powerful scripting languages and approaches support seemingly 'efficient' scripting practices that actually compromise the robustness of our scripts, as well as indirectly detracting from the stability and maturity of our support infrastructure. This is especially true for scripts that automate complex interactive processes using the scripting tools Expect or Chat. I present a formal methodology for the design and implementation of interactive scripting that, with a little more effort than writing a simple Expect script, produces scripts with substantially improved robustness and permanence. My scripting tool Babble interprets a detailed structural description of an interactive session as a script. Using this declarative, fourth-generation language, one can craft interactive scripts that are easier to perfect, inherently more robust, easier to maintain over time, and self-documenting.

Introduction

The amazing powers of current rapid prototyping languages strongly entice us to ease our burdens by writing simple scripts to automate repetitive administrative tasks. But the nature of our profession also encourages us to cut corners on these scripts, writing in haste to satisfy often inadequately predefined needs. Our scripts are not subjected to rigorous software engineering process or testing. They are easy to write but almost completely undocumented and difficult for anyone but the author to understand and maintain. The process of script writing is evolutionary rather than planned, driven by expediency rather than coherent overall design. This accelerates the 'software rot' that is unavoidable in all software development processes [3].

But even when we employ the best accepted software engineering process, writing system administration automation scripts is actually more difficult than writing many other types of software. Administrative scripts have strong couplings to their operating environment and make substantive changes to their environment as they execute. They are highly embedded [2] systems with complex preconditions and requirements for script success. An administrative script can be faced with any pre-existing conditions, be required to modify anything, and be required to produce most any result. And errors in a script executed with administrative powers can have dire results.

We can most easily understand the perils in writing scripts by considering the target system configuration as a collection of global variables. No one writes 'normal' software using global variables anymore, because of the danger of creating code that makes undocumented and untraceable changes in unpredictable places. But we do the equivalent in writing privileged administrative scripts on a day-to-day basis.

Scripts and Organizational Maturity

Scripts are not simply passive tools that we can use or ignore on a whim. Once deployed, they have an active role in determining the maturity of our service organization as defined in the System Administration Maturity Model, or SAMM [18], based upon the Capabilities Maturity Model of software engineering [7, 29]. One goal of SAMM is to encourage stable organizational structures in which particular staff members are interchangeable and replaceable on a moment's notice. The scripts that we craft to ease our lives can violate this principle in a rather subtle way.

Ad-hoc scripts often possess hidden usability constraints and behaviors only known to the author. If they work when we use them, fine; else we page the author, who repairs the damage thus inflicted. It is easy for the rest of us to relax into complacency as long as the author responds to pages in a timely manner. But regardless of the benevolence and good intentions of the author, using such a script compromises the maturity of the whole service organization, because the author becomes an irreplaceable component and service bottleneck instead of being interchangeable with other staff.

Because of this effect, some site managers (who shall remain nameless) prohibit ad-hoc scripting and automation, so that anything that cannot be automated by high-quality, well-documented, industrial-strength automation tools is done entirely by hand, avoiding scripting wherever possible. I take the controversial stance that this seemingly strange decision is justifiable. By making this choice, all their staff remain interchangeable and replaceable on a moment's notice, thus increasing their support organization's stability and maturity, at the cost of reducing individual productivity.

I formed this controversial opinion from direct experience. I am not an average script writer. I have written over 30,000 lines of Perl in the ten years I have known the language, to achieve many different ends. But I have also had a unique opportunity to observe the impact of my own scripts upon operations in my absence. When I 'retired' two years ago from technical to managerial duties, I left all of my 'clever' administrative scripts in the hands of another highly qualified staff member. Little by little, over the course of two years, I had to make the administrative decision to 'retire' each of these scripts in order to make operations more efficient. Most of them were not crafted well enough to outlast my direct involvement in using them, so that I became a bottleneck in my own operations whenever they failed. The only exceptions were scripts I very heavily documented and widely distributed, at great personal effort.

Assessing Script Maturity

While we can assess the quality of scripts using traditional software quality metrics [25], *the relative importance of these metrics depends upon how script quality affects service organization maturity.* In this context, a script exhibits high quality when its use does not depend upon specialized and esoteric knowledge, so that any properly trained and authorized staff member can utilize it with predictable and helpful results. While traditional quality factors such as documentation, reliability, robustness, and maintainability remain important, the peculiar properties of the administrative environment in which we utilize these scripts suggest some new quality factors that are more relevant and focussed upon our mission:

1. *precondition awareness:* does the script understand the conditions under which it will function correctly?
   a *detection:* can the script detect conditions under which it will not function and avoid problems?
   b *assurance:* can the script change the system so that preconditions are satisfied?
2. *convergence:* does repeating the script produce the same effect?
   a *self-consistency:* does repeating the script produce the same results?
   b *non-intrusiveness:* does the script avoid repeating unnecessary intrusive actions that can potentially disrupt services?
3. *postcondition awareness:* does the script check upon what it should be accomplishing?
   a *verification:* does the script check whether it did what it intended to do?
   b *validation:* do script changes have appropriate external effects, as observed from another machine?
4. *atomicity:* does each script do related things as a unit, so that there are no partial effects that produce service failures?
   a *transaction control:* is there a mechanism whereby the script can detect partial completion?
   b *rollback:* is there a mechanism whereby the script can back out of changes made in the case of a failure?

These factors all concentrate upon assuring predictable script behavior that leaves the affected system in a predictable and hopefully usable state, regardless of the identity of the particular administrator using the script.

Avoiding Scripting

The simplest way to avoid quality pitfalls of scripting is to utilize an automation tool whose design exhibits the above quality factors. Cfengine [4, 5, 6] and its relatives provide pre-written configuration methods possessing convergent properties. All the administrator has to do is to describe what to accomplish, and Cfengine will accomplish it in the least intrusive way. Cfengine is highly aware of required preconditions and elegantly deals with their absence. It fails predictably if it cannot accomplish its tasks. Although it does not provide transaction control, a user can craft this through careful configuration [12]. In effect, Cfengine provides most of the control one can get from a script, and assumes responsibility itself for the quality of its actions.

Cfengine is one of many tools available for avoiding scripting. My own Slink [9] solves the same problem for symbolic link tree hierarchies, and also provides a library of 'effective administrative abstractions' [10] with appropriate convergent properties for use in custom Perl scripts. Other file distribution methods that avoid or otherwise encapsulate scriptable actions include RPM (which supports scripts for custom actions) [1], rdist [8], and my own distr [11]. In my opinion, *whenever one can replace scripts with powerful, reliable, and well-documented management tools, one should.*

Unfortunately, there are many very common administrative problems that current high-quality tools do not address. While assuring appropriate contents for configuration files is relatively easy via file distribution (and interactive editing) approaches, controlling processes and other dynamic elements is a much more difficult task that usually requires some kind of custom scripting.

Short of avoiding scripts, we can better manage them and avoid writing too many. PIKT [22, 23] provides portability mechanisms for scripts that allow one script to function in a heterogeneous environment through preprocessing. Last year, we discussed how the logic programming language Prolog [12] subsumes the function of PIKT and supports script convergence, preconditions, and atomicity perhaps better than most scripting languages. But we concluded that coding in Prolog has its own unique difficulties and is not for everyone.

Then how should the mere mortals among us arrange to receive the benefits of scripting without the detriments? Providing scripts with the appropriate kinds of robustness is expensive in terms of coding labor, but utilizing naive scripts may be equally expensive in terms of administrative stability, because only people 'in the know' can deal with their deficiencies.

Interactive Scripting

To better understand the problems involved in scripting, I utilize a scripting example problem which, to my knowledge, presents almost all possible difficulties. Almost all network components have serial 'consoles' from which commands can be issued, and begin their lives in a state that requires some kind of manual configuration via interactive console commands. Routers, switches, and network appliances have to be assigned Internet addresses and networking information before I can utilize the Simple Network Management Protocol (SNMP) to finish the job. Typical UNIX and Linux servers must be built from the console before I can utilize automated methods to complete configuration. And if the network dies, then the only 'sure' method of interacting with potential culprits is still the trusty serial console.

Automating interaction with console interfaces poses many problems above and beyond just knowing how to write scripts. A human performing administrative actions must read reams of documentation, understand the meanings of commands, and adapt to messages from the console to determine future commands. A script trying to mimic these actions begins execution unaware of the device's current state, meanings of commands, or history of changes. It must discover these by parsing dialogs as it executes.

The easy way to configure a device is through 'invasive' scripting that erases the whole device configuration and starts over each time. This gives the script complete initial knowledge of the state of the device by clearing all data before making changes, which in turn makes writing the script a simpler task. For example, to add a new user, one can erase all users and then create them all again, including the new ones, much to the dismay of people currently using the device!

A 'convergent' script [12] changes the device from an unknown initial state to a desired one, without unnecessarily interrupting concurrent use of the device. The script must discover that state, compare it with what is desired, and craft a minimal set of actions that will accomplish needed changes. This process is much more complex than simple 'invasive' scripting, but much more desirable because it will not interrupt the function of correctly configured devices. Most devices support this kind of interaction rather poorly. In fact,

> *Convergence is not a property of the device, or of its configuration, but of our 'best practices' in managing it while maintaining an appropriate level of service for others.*

This makes crafting convergent scripts both particularly difficult and particularly important, as they embody all aspects of human interaction, including device knowledge, configuration requirements, and management policy.

My Goals

I began the work of this paper by looking for a better way to write Expect [19] or Chat scripts that will enable 'convergent' bootstrapping and administration of console-scriptable network nodes. I needed something like this in order to be able to reliably recover from errors made by students in building experimental networks. Left to herself, and given full reign over a network device, a student can unknowingly break SNMP (or other) management control over network devices, thus making 'front door' recovery techniques unreliable.

My second motivation was to bring the process of scripting closer to the 'best practices' I already understand. The tightest coupling I can make is to relate automated scripts to the commands I would have to issue myself in order to accomplish the same task. I consider this a much tighter 'semantic coupling' than, say, SNMP requests to accomplish the same changes: SNMP requests look very different indeed from the administrative commands to which they correspond.

The Lightwave ConsoleServer 3200

My example application is to create a convergent script that will automatically maintain the configuration of a LightWave ConsoleServer 3200 [30]. This is a serial console switch that allows access to any one of up to 32 serial consoles from up to 16 simultaneous incoming telnet sessions. It is remarkably easy to configure, but configuration involves setting many parameters, and these may only be set by hand using a serial command-line interface. There is no SNMP interface available and management functions are not network-accessible by any means. This device, once configured, also allows script access to all other consoles in my site, via telnet within a dedicated private (RFC1918) administrative subnet.

I wish to use the ConsoleServer in college coursework in order to give students access to remote Linux consoles, so that they may practice configuring Linux systems that are physically located in a protected location. This means that the configuration of the ConsoleServer will be changing frequently in order to allow new students access to the consoles. I already maintain databases of the students who should be given access to particular consoles. The trick is to craft a convergent mechanism by which I can assure that the appropriate students have access to appropriate machines, by adding and deleting accounts for particular students as they rotate through lab exercises.

Easy or Impossible?

One might think that this project is easy until one understands the true complexity in the interactions. Let us consider the simple subtask of making sure the switch is accessible to the correct students, and that they possess appropriate privileges. To add a user, one participates in a dialog similar to this:

```
LCI3200>login admin
PLEASE ENTER PASSWORD ****
sys admin>adduser
Number of available user records: 196
Number of users defined: 4
Enter user id | USER ID > foo
Enter case sensitive password
| PASSWORD > ******
Re-enter case sensitive password
| PASSWORD > ******
0-17 | MAX CONCURRENT LOGINS: 1 > 1
Allowed devices example:
1-5,10 | DEVICES 0 > 1-11
Allowed listen devices example:
1-5,10 | DEVICES 0 > 0
Allow user to clear device buffer
(Y/N) | YES > N
Clear screen after a command
(Y/N) | YES > Y
Enter user id | USER ID >
sys admin>logout
LCI3200>
```

(in this paper, long lines in examples are folded to fit within columns). With Chat, one can craft this specific dialog, but employing variables in the dialog is awkward at best. With Expect, one can write a TCL [24] subroutine that performs this task, where all user inputs are variables. Then one can call the subroutine multiple times to add multiple users.

This all seems relatively straightforward until we consider what can go wrong during the script. The user could have been created already, so that all we need to do is to modify settings and privileges. This requires executing a *different command* `edituser`. The answers to any of the above questions, as specified by our script user, could be inappropriate. In this case the device *repeats the question*:

```
Allow user to clear device buffer
(Y/N) | YES > No
Allow user to clear device buffer
(Y/N) | YES >
```

The device does not accept 'No', just 'N'. Improper answers leave the device asking the same question over and over again, so that the script must issue a control-C to reset the interface after any script failure. We must include one 'if' statement in our script to catch each possible failure.

Making the script 'convergent' requires much more work. We must teach the script how to gather data on existing users, and how to determine users that have not yet been added. Modifying users so that they have new privileges is a matter of reading each user's profile, comparing it with desired data, and changing it if necessary. Each of these processes is more complex than the example above. The net result is that the 'convergent' version of the script has an enormous number of possible execution paths, depending upon what goes wrong. This high 'branching complexity' [20] will cause the script to be very expensive to write, debug, and maintain.

If I write this script under pressure, I am obviously *not* going to have the time to do this correctly and will miss some case. So to use my script, I have to watch for failures, correct the values of parameters, and run the script again, perhaps after cleaning up after what it did the previous time. If I become impatient enough, I will modify the script to better handle some of the failures, in order of annoyance to me.

As I address the annoyances, I create another problem. As my script becomes increasingly clever, it is evolving with abandon, without functional boundaries or documentation. As it grows, it becomes likewise increasingly clever and increasingly unmaintainable. Any time the device changes, the script fails and I am the only hope of repairing it. I am well on the way to owning an irreplaceably valuable script that renders its author irreplaceable as well. This is what we call 'job security.'

It is for this reason that I used to consider the goal of creating 'convergent' interactive scripts (that apply minimally intrusive changes in order to assure device state) practically impossible.

Jackson System Design

In software engineering practice, one manages project complexity and avoids this kind of developmental 'script rot' by applying a formal design methodology that controls development in order to keep scripts both understandable and reusable. Fortunately, the methodology we need was well understood during the *punched card era* of computing, and only needs to be resurrected and reapplied!

Jackson [15, 16] claimed that the way to properly design a program for processing punched card stacks is to *link the structure of the program with the structure of the stack that it processes.* He created a simplified structural model that replaces program flowcharts with 'Jackson Diagrams' that are the same for the program and its input. Each diagram depicts containment and sequence of inputs or program parts, utilizing nodes for parts and undirected edges for relationships, reading from top to bottom and left to right. The diagram:

```
    A
  / | \
 B  C  D
```

represents a thing 'A' that consists of subparts 'B', 'C', and 'D' in that order. This thing can either be a stack of cards or the program that processes them in sequential order. Loops and branches in the program are indicated by annotating the diagram. Repeated items during a loop are starred, and optional items are annotated with a '0'.

For example, the diagram below refers to a deck of cards containing a structure A, which consists of structures B, C, and D, where B consists of multiple copies of E, and D might contain either J or K.

```
       A
     / | \
    B  C  D
    |    / \
    E*  J0 K0
   / \
  F   G
```

Jackson's key idea was to interpret this diagram also as the structure of a program to process the cards:

```
begin A;
 begin B;
  for (some cases)
   begin E;
    begin F; end F;
    begin G; end G;
   end E;
 end B;
 begin C; end C;
 begin D;
  if (some condition)
   begin J; end J;
  if (some condition)
   begin K; end K;
 end D;
end A;
```

The program *must* look something like this if the cards are structured the same way.

Applying Jackson's Principle

I began this project 'expecting' to utilize an enhanced version of Expect to address state awareness and convergence problems in traditional scripts. I initially added enhanced handling of parsing and variable binding, similar to that in PIKT, in an effort to make scripts shorter and easier to understand. My approach to this problem changed dramatically in mid-project, however, when I realized that Jackson's methodology applies to the structure of the input/output streams with which we as administrators control the device. The structure of these streams (of prompts and commands) is predetermined by device design and one's intent as an operator. This pattern can be mimicked by a script in order to accomplish the same intent:

> *The structure of a fully functional interactive script is exactly parallel to the branching and looping structure of the device interactions in which it must engage.*

This observation would have come to naught if I had used Jackson's diagrams as above, for they become unwieldy when used to describe interactions of this complexity. Fortunately, I did not need to utilize these, because one can use a variant of the Extensible Markup Language (XML) [13, 26] to perform the same function. For this, I employ the XML tags:

1. <repeat>: the equivalent of Jackson's star; indicates repetition of patterns within an I/O stream.
2. <branch>: the equivalent of Jackson's '0'; indicates that something is one of many possible options.

For example, the Jackson diagram above can be represented as:

```
<A>
 <B><E>
  <repeat><F/><G/></repeat>
 </E></B>
 <C/>
 <D>
  <junction>
   <branch><J/></branch>
   <branch></branch>
  </junction>
  <junction>
   <branch><K/></branch>
   <branch></branch>
  </junction>
 </D>
</A>
```

where

- <X> marks the beginning of X.
- </X> marks the end of X.
- <X/> marks the beginning and end of X. This is equivalent with <X></X>.

With this model in mind, I realized that my so-called 'high-quality scripts' looked much more like structural declarations than scripts, and that the non-structural imperative commands were obfuscating my understanding of the structure that the scripts documented. I was worrying about 'which variables to set' in the scripts, while I should have been worrying about the *structure of interactions.*

My obvious next step was to split each script into two parts: one part that documents structure and another that acts upon that structure. In the beginning, I considered the ability to separate structure from action as an extra 'toy' capability of my scripting tool. But eventually, as my structural markup language evolved in its ability to express detailed structure, I realized that:

> *For most purposes, crafting of individual interactive scripts can be replaced by an intelligent scripting engine that parses detailed structural specifications of the interface and its desired configuration, and then proceeds to assure that configuration by exploiting documented interface structure.*

At first glance this method of scripting may seem ridiculously awkward, but in practice it is much easier, faster, and more reliable than scripting. Once the scripting engine is written, all one must do is to document the streams that it controls. This involves specifying the sequence, variant content, conditional structure, and topology of the commands that the interface understands. One accomplishes this by collecting and annotating example sessions. If one can do something manually, and react to all possible responses, one can script the process. The 'script' becomes *documentation* describing what varies in the sessions, what options or branches there are in the process, and the 'causal intent' of each interaction, e.g., creating a user.

Stream-structured Design

My variant of Jackson's method documents the structure of I/O streams through a series of simple and straightforward steps. Each step requires modifying an example script of a user session by adding XML-like markup tags. These tags form a Stream-Structure Markup Language, or SSML. When this process is completed, I feed the sum total of all sessions I have recorded, all appropriately marked and annotated, to a 'scripting engine' that I call 'Babble'. This engine utilizes structural documentation to decide how to interact with the device.

For example, let us first consider how one documents the process that deletes a user from the Lightwave 3200. The first step is to collect an example dialog that accomplishes this, using the script and cu commands upon a connected UNIX host. This produces a file:

```
Script started on Sat Sep 16 16:29:13 2000
% cu -l cua/b -b 8^M^M
Connected^G^M

LCI3200>login admin^M
PLEASE ENTER PASSWORD ****^M
sys admin>deletu^H ^Heuser foo^M
Delete user:foo Yes or No (N):Y^M
sys admin>logout^M
LCI3200>~.^M
Disconnected^G^M
% exit^M
script done on Sat Sep 16 16:30:29 2000
```

where the prefix ^ indicates invisible control characters.

Standardize Input

My second step is to remove the header and trailer, together with chaff such as ^H that indicates a backspace (together with characters I backspaced over). I convert &, <, and > to their XML equivalents &, <, and > so that they will not conflict with the XML tags I will add in the next step. I convert the remaining special characters to their Perl escape-string equivalents for easy readability and editing.

```
\r\n
LCI3200&gt;login\sadmin\r\n
PLEASE\sENTER\sPASSWORD\s****\r\n
sys\sadmin&gt;deleteuser\sfoo\r\n
Delete\suser:foo\sYes\sor\sNo\s(N):Y\r\n
sys\sadmin&gt;logout\r\n
LCI3200&gt;
```

In the above, \r represents return and \n represents line-feed, \s represents space, and \007 represents bell (control-G). After this transformation, *whitespace is ignored* in all further steps and may be used to indent for clarity.

Mark Input and Output

The next step is to annotate the remaining text so that I know what is input and what is output. There is not yet an automated way to do this, so I manually insert get and put tags to distinguish things the interface sent from those I typed. At the end of this, untagged text will be ignored.

```
<brook name="delete">
 <put>\r</put>\n
 <get>LCI3200&gt;</get>
 <put>login\sadmin\r</put>\n
 <get>PLEASE\sENTER\sPASSWOR\s</get>
 <put>****\r</put>\n
 <get>sys\sadmin&gt;</get>
 <put>deleteuser\sfoo\r</put>\n
 <get>Delete\suser:foo\sYes
                 \sor\sNo\s(N):</get>
 <put>Y\r</put>\n
 <get>sys\sadmin&gt;</get>
 <put>logout\r</put>\n
 <get>LCI3200&gt;</get>
</brook>
```

I call a little part of an I/O stream a *brook* (!). One subtlety is that since the \r's are typed by us but the \n's are typed by the responding system, the \n are *outside* the respective put's. If I instead place the \n inside the put, the scripting engine will add an extra line-feed to every command, perhaps with problematic results.

Identify Variants

The next step is to document which strings vary in the stream, depending upon what I wish to accomplish with this script. I call these strings *variants* to distinguish them from traditional variables, with which they share only a superficial resemblance. In my example, only the administrative password and the name of the user to delete may vary. All else is always the same. I mark and name variants where appropriate, inserting (ignored) line breaks and indentation for readability:

```
<brook name="delete">
 <put>\r</put>\n
 <get>LCI3200&gt;</get>
 <put>login\sadmin\r</put>\n
 <get>PLEASE\sENTER\sPASSWORD\s</get>
 <put>
  <var name="adminpass">****</var>\r
 </put>\n
 <get>sys\sadmin]&gt;</get>
 <put>deleteuser\s
  <var name="username">foo</var>\r
 </put>\r
 <get>Delete\suser:
  <var pattern="[a-zA-Z0-9]+">foo</var>
 \sYes\sor\sNo\s(N):</get>
 <put>Y\r</put>\n
 <get>sys\sadmin&gt;</get>
 <put>logout\r</put>\n
 <get>LCI3200&gt;</get>
</brook>
```

There are two kinds of variants. A *named variant* is something to be placed into put commands or discovered during get commands. There are two of these: adminpass and username. A named variant has to be assigned a value in order to be put, but *acquires* a value after a get. An *unnamed variant* is only valid within a get and represents variant input to be matched and skipped over, documented by a Perl regular expression pattern.

In Expect, at this point, I would have to assign these values to variables outside the realm of the device language, in TCL. In SSML, these variable bindings are accomplished by comparing this description(of where variables appear) with a different XML database of appropriate values for each variant. In this way specifics of configuration are kept separate from the process by which one configures a thing.

Classify Echo Types

The next step is to carefully classify variant output into one of several classes. There are three classes of output, corresponding to echo options: normal echo (full duplex, the default), no echo, and starred echo (for passwords). These are indicated by tags that contain output within a put:

```
<put><stars>
 <var name="adminpass">****</var>
</stars>\r</put>\n
```

The default is that what I type shows up in the output in full-duplex mode. Placing output inside a stars tag documents that stars are displayed instead of what I type, while a noecho environment indicates that there is no echo at all, as in typical password dialogs.

Document Conditional Behavior

The next step is to indicate any branching or conditional behavior in the overall flow of the script. First it is possible that I will already be logged in when the script starts. In this case, I wish to skip the administrator login, a simple branch:

```
<put>\r</put>\n
<junction>
 <branch>
  <get>LCI3200&gt;</get>
  <put>login\sadmin\r</put>\n
  <get>PLEASE\sENTER\sPASSWORD\s</get>
  <put><stars>
    <var name="adminpass">****</var>
  </stars>\r</put>\n
  <get>sys\sadmin&gt;</get>
 </branch>
 <branch>
  <get>sys\sadmin&gt;</get>
 </branch>
</junction>
```

Each branch starts with a get that is used to select which branch to execute. If I receive a non-administrative prompt, I log in, else I skip the process. Appearances can be deceptive: this is *not* a method, but *documentation*. The stream can take two paths, and I have now documented both of them.

Another branch will be taken if the user name I choose does not exist. I can document this branch by collecting more data:

```
sys admin>deleteuser foo
User foo does not exist
```

To deal with this case, which is perfectly acceptable since I wanted to delete the user anyway, I can modify the master script by adding a branch describing the new response:

```
<put>deleteuser\s
 <var name="username">foo</var>\r
</put>\n
<junction>
 <branch>
  <get>Delete\suser:
   <var pattern="[a-zA-Z0-9]+">foo</var>
   \sYes\sor\sNo\s(N):
  </get>
  <put>Y\r</put>\n
 </branch>
 <branch>
  <get>User\s
   <var pattern="[a-zA-Z0-9]+">foo</var>
   \sdoes\snot\sexist</get>
  </get>
 </branch>
</junction>
<get>
  sys admin&gt;
</get>
```

If I receive the error message, I simply ignore it. The default action in SSML, if there is no match to a branch, is to fail with a script error.

Associating Values With Variants

The next step is to actually invoke a script engine upon the documentation in order to perform the documented function. This is a matter of binding variants to appropriate strings and calling the scripting engine to interpret the results. This in turn requires creating declarations of variants separate from – but in agreement with – the stream declarations I have made. For example, for my brook described above, I might declare:

```
<var name="adminpass">PASS</var>
<var name="username">couch</var>
```

to assign values of PASS and couch to adminpass and username, respectively. For simplicity, variant values are organized in a single global declaration in a separate file.

Repeating Commands

Most of the time, however, I do not wish to delete just one user. In SSML, I accomplish repeated tasks *implicitly* (as we did previously in Prolog [12]) by declaring sets of instances of variable values to use. I force an action to repeat by defining a variant that holds a set of instances, matched with a repeat markup that processes the instances. The structure of instances in the configuration data must be parallel to the structure of repeat tags in the markup. This process is aided by a simple but powerful name scoping mechanism.

For example, suppose that I wish to delete both users foo and bar. I create a brook to do both, by inserting the brook I have already created into a repeat context:

```
<brook name="expunge">
 <repeat instances="people">
  <insert brook="removeuser"/>
 </repeat>
</brook>
```

The variant people consists of two instances of data needed by removeuser:

```
<repeat name="people">
 <instance>
  <var name="username">foo</var>
  <var name="adminpass">PASS</var>
 </instance>
 <instance>
  <var name="username">bar</var>
  <var name="adminpass">PASS</var>
 </instance>
</repeat>
```

Each set of distinct values is called an *instance* of the process. During the repeat, each instance is processed in turn. During processing of a particular instance, the script engine *augments the top-level variant declarations* with new variable values for each case in turn, then invoking the brook with these new values. Variants declared outside the repeat clause keep their values unless shadowed by definitions inside an instance, so I could have accomplished the same effect through:

```
<var name="adminpass">PASS</var>
<repeat name="people">
 <instance>
  <var name="username">foo</var>
 </instance>
 <instance>
  <var name="username">bar</var>
 </instance>
</repeat>
```

As the variant adminpass occurs outside the block of instances, and is not shadowed within them, it is available to the contents of the repeat markup for each instance. Variant bindings during a repeat are *strongly typed* but *dynamically scoped*. The kind of variant (repeat or text) must exactly match its usage, but variants brought into scope by a repeat are available to any contained repeats or subprocess invocations. This allows one to declare multiply-dimensioned loops by defining two sets of instances with non-overlapping variant names, one set for each repeat.

Discovering Configuration

This variant binding scheme also works in *reverse* to allow us to inductively discover the values of variants for a set of instances. Suppose I want to get a list of all users. I know that the command for that is listusers:

```
sys admin>listusers
User id > COUCH
User id > FOO
User id > BAR
sys admin>
```

In my last use of repeat, the instances over which I iterated were all arguments to put commands. I can discover users by reversing this process, referring to the variants within a get command:

```
<brook name="listusers">
 <put>listusers\r</put>\n
 <while instances="people">
  <get>User\sid\s>\s
   <var name="username"
     pattern="[A-Z0-9]+">
       COUCH</var>\r\n
  </get>
 </while>
 <get>sys\sadmin></get>
</brook>
```

This creates several instances, all known under the name people, where each one contains the username of one user of the device.

The difference between repeat and while lies in their control over instances. repeat does something for a *fixed* number of instances, while while *inductively discovers* instances as they appear in the output, and creates a list of all of them. This has the effect of *updating variant space* for the discovered values, erasing any previous value of the structured variant people, where each instance contains a current username, mined out of the output by using the regular expression pattern expression pattern '[A-Z0-9]+'.

In this example, the while exits upon a timeout, after which instances discovered during each completed pass through the while process become instances of the repeat variant people, overwriting any previous values. At this time we know all the names of users, and could now, e.g., delete all of them by invoking the 'expunge' brook above.

Convergent Processes

Until the last example, I have not employed variants that are computed at runtime. The reader might ask what good it does to discover variable values if there is no script to utilize these values. The scripting engine itself, with appropriate guidance, can utilize this data to great advantage, or even print a new configuration file representing the *current configuration* of the device.

Assuring values of individual configuration parameters non-intrusively is fairly trivial. The engine reads them, and if they are incorrect, changes them accordingly. For it to do this, it is sufficient to instruct the engine on how to read and write particular configuration parameters, with appropriate branching to deal with different cases.

Difficulties arise, however, when one wishes to efficiently update a part of the configuration containing an unknown number of instances of a thing. For example, in assuring that my idea of current users agrees with that of the device, I start with two lists of users, one in hand and one already configured on the device. To update the users, the engine must read current user information from the device, note differences between current and desired users, and proceed to modify the configuration so that the desired information becomes current. To empower the engine to behave intelligently in this case, I must document a few processes that act on the same variants, including how to read the user list, how to read details on one user, and how to add, delete, and modify one user's data. All of this information together describes a *convergent process* that will update the user list to have desired contents.

So far, I have used SSML to describe more or less traditional program flow that is also representable using Expect. In this example Babble transcends Expect's capabilities by *responding intelligently to documentation.* Babble reads the user list and adds, deletes, or modifies users as needed, *checking its work* at every step by reading what it has written. In this way, five short declarations are used to synthesize one incredibly complicated action that would be impractical to code as a single declaration. This complexity thus migrates from the documentation into the script engine where it belongs.

Exceptions

Some devices have particularly annoying user interfaces. For example, the 3Com Corebuilder 9000 prints the contents of SNMP traps on the console while one is trying to configure it, sometimes in the middle of typing commands. To configure this device, one must ignore these alerts while issuing configuration commands. One can do this in SSML by declaring an 'exception' pattern to check for and discard if present. This arranges for the trap data to be discarded whenever it appears, regardless of context. This would be incredibly awkward to arrange in Expect, as the exception pattern would have to be included in *every pattern match* in the whole script!

Babble

Babble is a scripting engine that parses SSML specifications and performs desired configuration tasks. It is implemented as a set of cooperating Perl packages that parse both stream documentation and variant declarations, and allow one to selectively invoke individual `brooks` or convergent processes with desired parameters. It is implemented as a Perl library because of the many and varied forms in which I store configuration information, in the hope that any external specification of configuration policy can be translated into an appropriate set of variant declarations using Perl. Input to each invocation consists of an I/O stream with which to interact, a compiled SSML parse tree, a name of a branch to invoke, and a multi-level associative array describing variants and sets of instances to be processed. The output of each call is the modified variant array, modified to reflect any `gets` executed during the script.

This version of Babble is so new that the only application I have so far crafted is the one described herein. I can report from this that crafting Babble scripts to control the 3200's configuration was accomplished unbelievably quickly, because scripts almost always worked correctly the moment they had correct syntax according to the parser and builtin configuration tester included with Babble! Debugging was almost entirely a process of responding to complaints from Babble itself about mismatches of names, syntax errors, etc. The only specialized knowledge I needed was an understanding of how to accomplish specific things in Babble that I was used to accomplishing using scripting.

While I designed Babble specifically for serial console interaction, it can be used to automate any serial interactive process, including UNIX commands, telnet, etc, in the same manner as Expect. E.g., one can use it to parse the output of `ps` and then use the result to interact with the process table using explicit `kill` commands. Babble does not – and never will – have the ability to perform direct system calls. Employing solely console commands documents 'best practice' at the expense of script speed and resource efficiency, perhaps a proper decision.

Limitations

Babble of course refers to the tower of Babel from the Bible, because like the builders of the tower, it *speaks all languages, but without comprehension*. This lack of comprehension is the root of all of its permanent limitations. The commands Babble executes have no meaning to Babble itself, but are simply abstract patterns of interaction learned from experience. When that experience is somehow incomplete, it fails. When interacting with a complex device, this experience can never be complete and some failure is assured.

State Coherence

Babble's greatest weakness is nearly invisible in the example. Every script in the example requires hidden *preconditions* in order to function, and scripts must be strung together so that *the postconditions of each script satisfy the preconditions of the next.* The most important precondition is interface state, which is not even representable in the current markup language. The state of the interface indicates, e.g., whether the interface is in unprivileged or privileged mode, and whether the process is at a command prompt or within an interactive dialog. These hidden preconditions and postconditions affect the success of every script, in particular, *every subbrook of a convergent process must start from – and end within – the same exact interface state.*

Version Control

Another serious deficiency of Babble is that it cannot explicitly encode version information to assist it in dealing with identical hardware devices running different software or firmware. Each device revision requires a completely independent Babble script. This deficiency, alas, is completely intentional.

Babble's 'topological algebra' of structural tags is designed for *automated* merging of brooks that represent special cases of the same task on the same device. This will be accomplished in the future via 'parallel tree walks' through the descriptions to be combined, in which one combined description emerges with appropriate junctions and branches inserted. So far, all attempts I have made to combine this feature with version control have compromised this automated merging capability, by making the algorithms for automated merging unnecessarily awkward or perhaps even impossible.

Branching in Babble is a *temporal* phenomenon while versioning is *spatial* in character. Problems arise in temporal merging when spatial merging has been done first; one does not have enough information to complete the merge unless the streams being merged have identical spatial structure. I consider automated merging and temporal coherence more important than version control, and believe that version control may have to be handled by a completely different tool, much as PIKT provides a metalayer for managing versions of normal shell scripts.

Paranoia

Babble's run-time checking borders on paranoia. Unlike Expect scripts, which check only for specific cues in the input stream, Babble checks for full-duplex echo of output, as well as compliance of input with all markups one specifies. Scripts abort on any deviation. Babble also frequently 'checks its work' by reading parameters it has modified.

If one wishes even *more* paranoia, Babble allows one to craft scripts that *validate* behavior rather than merely *verify*. Unlike verification, which simply checks that parameters are being set correctly, validation checks that parameters have the appropriate *external effects*. For example, after enabling telnet on a device, one can telnet into it to check that it works; after creating a user, one can login as that user from another device. These detailed sanity checks would be impractical to craft via traditional methods, but are relatively easy to craft in SSML because the engine is handling most of the details of error detection and branching.

Awkwardnesses

Alas, Babble's documentation format was driven by many expediencies. I used XML syntax because I was used to writing XML parsers. This was *not* the optimal match. Because of this, I had to escape all special characters in an awkward way so that they would not interfere with XML parsing. In fact, the syntax is *not*, strictly speaking, fully XML compliant. To make it possible to drop into Perl during a script, I had to allow embedded Perl, but in true XML one would have to escape &, <, and > in Perl code, rendering it unreadable. Thus I allow regular Perl and preprocess the documentation file, escaping all special characters in Perl scripts *before* parsing the result as XML.

This awkwardness, however, gives me the ability to incorporate features into Babble that are difficult to code for any other base language. A Babble configuration is a 'literate program'[17] that represents several different facets of a process, including documentation, procedure, and policy, in one convenient package. The reason I chose XML for the base language of Babble was to enable me to render descriptions of these facets in HTML for viewing on any web browser. This is an invaluable debugging aid that will become a feature of Babble in the very near future. More important, *the interactions that Babble undertakes with a device can themselves be represented in XML,* so that each invocation can be described in HTML as well, with hyperlinks from the transcript of the invocation to parts of the configuration that determined its shape.

Critique

This approach is superior to writing scripts with Expect for several reasons. It avoids classical verification problems associated with script development, so that process refinement is much less dangerous than when writing real scripts. This strength is negated, however, if one employs regular programming as part of processing a stream. Babble also avoids one of the main difficulties in scripting with Expect: the need to craft complex regular expressions to parse input of irregular structure.

Avoiding Verification Problems

The greatest obstacle to using traditional scripting is that verifying the correct behavior of scripts is difficult. I avoid some of the difficulties by crafting *documentation of a pre-existing condition*, not a computer program. During the time I am tuning and perfecting documentation, the 'script engine' that utilizes the documentation remains unchanged. I thus simplify the problem of verifying my process into two problems: verifying the script engine itself as a program, and verifying the documentation as a description of external behavior. The engine need only be verified once. Verifying accuracy of its subsequent uses only requires checking its input for accuracy.

This fact makes refining a description much easier than refining a true script. The form of documentation is sufficiently simple that traditional limits to script validation do not apply. One can use automated static verification tools to exhibit possible sub-paths, validate syntax, and check correspondence between stream and parameter declaration structure. Thus one can largely avoid the problems of 'software rot' that plague the maintainers of true scripts.

The trick of separating documentation from programming only works if one can avoid embedding real program code into the documentation. If one must, one loses most of the benefits of the approach. In order to avoid this, one must be able to compile a complete parameter space before one starts the engine. If one cannot, but must compute configuration parameters 'on the fly' during the script, simple process documentation no longer suffices. Then one must drop into Perl during a stream, so that my intended documentation now again assumes the role of a program. Of course, when one must do this, one compromises many of the strengths of the approach, in the very same way that employing embedded TCL weakens the maintainability of Expect.

Simplifying Regular Expression Syntax

While regular expression pattern matching is one of the most powerful features of Expect, TCL, and Perl, it is also one of the most dangerous and unwieldy. The parenthetic syntax for binding substrings to output variables is a source of constant confusion. The most common error is counting parentheses incorrectly so that variables are bound to incorrect values. In SSML, all parenthetic matching is *implicit* in the order of variables within a get, and parentheses are *not* enabled in the regular expression patterns. This makes the patterns much easier to craft and debug, and there is no danger of mismatched variables, as in parenthetic patterns or split statements.

This convention replaces one weakness with another (hopefully lesser) weakness. When crafting documentation, one must be careful to declare variants using appropriate regular expressions so that the pattern that the engine derives from your declarations is not ambiguous. For example, it is poor practice to declare two adjacent variants whose patterns create ambiguity in assigning values to the variants:

```
<var name="poor" pattern="[A-Z]+[0-9]+">
<var name="style" pattern="[0-9]*">
```

When the engine tries to match these two variants against the input ‘AB1234’, the first pattern match causes an early binding of the variant poor to ‘AB1’, after which the second pattern matches ‘234’. But matching these patterns could just as well have split the input into AB12’ and ‘34’, ‘AB123’ and ‘4’, or ‘AB1234’ and ‘’.

Thinking Declaratively

Several limitations of SSML are entirely intentional. One cannot negate a pattern in SSML, or use a regular ‘for’ loop. These are not simple oversights, but based upon fundamental limits of the theory of automatic program verification.

Verifying the correctness of regular scripts without exhaustive testing is impractical. The most common method of verification is called ‘weakest precondition analysis’ [14, 21, 28]. To use this method, one clearly documents preconditions and postconditions of the script, and then analyzes the script line by line from end to beginning, starting from desired postconditions and carefully computing the preconditions needed to assure those postconditions. A script is ‘correct’ if the preconditions actually required to assure postconditions are ‘weaker’ (i.e., less demanding) than the stated preconditions we document. This process is easy to perform automatically for scripts containing only linear code with no loops, but is equivalent in complexity to mathematical theorem proving for scripts containing loops in which the same variables are both set and used. Theorem proving takes far too much time to be practical.

In practice, this means that the only practical way to assure the quality of a script is to put it through a full ‘regression test’ after *any* change. Because of the complexity of the environment in which administrative scripts must run, this kind of testing is usually impractical or perhaps impossible. This allows incidental script bugs to remain hidden until they cause a crisis, perhaps until the original author has long ago moved on to other employment.

To be able to efficiently verify a program, one has to ‘weaken’ the scripting language so that limits to automatic verification do not apply. SSML documentation contains no loops that would present problems for a ‘weakest precondition’ verifier, unless one intentionally reads a variable and sets it inside a repeat scope *in the same brook*. Doing this in SSML constitutes a markup error that future versions of Babble will be able to detect and report.

The current implementation performs only limited verification, including reporting disagreement on names and types of variants between stream documentation and value declarations. The structure of SSML will allow future versions of Babble to locate more kinds of common programming errors, including overlap or ambiguity of regular expression patterns, as well as ambiguity of intent, such as writing data during a stream that should only be reading it, or vice versa.

Relearning Common Techniques

Writing SSML specifications requires that one learn new equivalents for common but less reliable scripting techniques. For example, it is very common for a traditional script to parse a line of input into an array with a split command. E.g., in Perl, one can write:

```
@parts = split(/\s+/,$line);
```

where /\s+/ is a regular expression, $line is the unparsed line, and @parts is an array of fields within the line. Babble does not allow this kind of matching in a straightforward way, but there are two equivalent constructions. First, one can name all parts of the line to be matched, and match them individually:

```
<var name="first" pattern="[^\s\n]+">
<var pattern="\s+">
<var name="second" pattern="[^\n]+">
<var pattern="\s+">
<var name="third" pattern="[^\s\n]+">
```

where the pattern [^\s\n]+ matches non-whitespace, while the pattern \s+ matches whitespace. This will bind first, second, and third to the next three whitespace-delimited fields. If there are a fixed number of fields, this is the best possible documentation on their structure.

If there are several fields for which one does not know a field count, such as a list of ports separated by commas, one can instead declare a repeating structure:

```
<while instances="ports">
 <junction>
  <branch>
   <var name="port"
        pattern="[^\s\n]+">
   </var>
  </branch>
  <branch>
   <var pattern=",\s+"></var>
   <var name="port"
        pattern="[^\s\n]+">
   </var>
  </branch>
 </junction>
</while>
```

The complexity here is more apparent than real. This declares a sequence of input in which there are repeated instances of a port, where each pair of port numbers are separated by whitespace and a comma. The branching structure indicates that it is possible that there is whitespace in front of each instance. If, e.g., the input is '2, 5', the data that this process binds to the variant 'ports' has the structure:

```
<repeat name="ports">
 <instance>
  <var name="port">2</var>
 </instance>
 <instance>
  <var name="port">5</var>
 </instance>
</repeat>
```

This data can in turn become the argument to a repeat markup if one wishes to do the same thing to each discovered port!

Automation

Many steps in this process can be automated or streamlined so that much less user input is required. For example, when capturing example sessions, a tool that would capture and correlate both input and output (using time stamps to determine relationships) could generate the direction and echo markups that I created by hand in the example.

There are subtle semantic difficulties in automating the task any further without human intervention. For example, it is not possible to reliably infer the positions of variant data – or the regular expressions that describe them – from a few examples. A person must mark these. But once input, output, and variants are distinguished, multiple example sessions exhibiting different branches for the same task can reliably be combined automatically by parsing them, fusing their parse trees, and then printing the result. A person must nonetheless identify which set of brooks all accomplish the same task and should be fused. Likewise, after I tell the engine which scripts read and write data, the engine can automatically determine whether write operations worked or not, by reading the results and checking those against my intent.

Conclusions

When I began this work, I was possessed by the traditional spirit that scripting can solve any problem, and that all I had to do was to make scripting easier. Even when applying the relatively declarative thinking required for logic programming, I retained the old script mentality and first tried to do 'everything I could do with Perl'. This attitude was the result of 28 years of conditioning, and it took a long while to question this thinking, and even longer to unlearn old habits in order to discover ways of doing without this 'expressive power'.

The quality of our work, as script writers, is controlled by fundamental theoretical limits known to Computer Science. Normal scripts are difficult if not impossible to validate and verify by any method short of exhaustive testing. The unique properties of the administrative environment make this testing impractical, while our lack of knowledge of the complete effects of our actions hampers top-down thinking and design. Babble cannot violate any of these limits, but can carefully work around them. It discourages unproductive practices and shifts responsibility for script quality – whenever possible – away from the script itself and into a reliable intermediary component that better interfaces desires with devices.

My journey has been a 'tale of power'. Sometimes apparent power is an illusion. This illusion can cost us much time and effort to avoid. It can sap strength from our infrastructure while superficially pleasing our egos. It can keep us from realizing its effects, 'trapping us in a lifestyle' of seeming opulence with an underlying and terrible cost. But the first step in avoiding a trap is knowing of its existence.

We each seek personal empowerment in our own ways, weaving a fabric of practices and tools that gives us the stability and security we all crave as human beings. Intrinsically we all know what the real 'best practices' are: those techniques that enhance our personal empowerment and security. We may be given these by a superior, or discover them ourselves via bitter experience, but the result is the same.

If we can document these practices so that they will outlast our attention and presence, we empower others in the same way. Thus we can move beyond the 'network of trust' to form a 'network of empowerment' in which our community of administrators is much stronger than the sum of its parts. This goal requires putting the community above one's self-interest, in order that the community become strong enough to protect us better than we can protect ourselves. It requires looking beyond 'job security', toward 'mission security'. It requires acting fairly within the 'social contract' that irrevocably binds us with our organization in a pact of mutual protection and shared mission.

Because true empowerment flows not from inside ourselves, nor from our technologies, but from caring community carefully woven around shared purpose, vision, and dreams.

Availability

Babble will be available soon in alpha release from http://www.eecs.tufts.edu/~couch/babble. While it is written entirely in Perl 5 and should be portable to all UNIX systems, the current version does not function properly in Linux due to a bug in the CPAN pseudo-tty module Ptty.pm – I am working on this.

Acknowledgements

I first wish to thank intrepid system administrator Andy Davidoff for putting up with me while I learned the hard way how to be a good manager. Tufts administrators Rich Papasian, Lesley Tolman, and Tony Sulprizio were all excellent examples to me in learning this lesson. Judy Jovanelly of Lightwave Communications, Inc. was most helpful in both suggesting the Lightwave 3200 for my application, and helping me repair a trivial bug in 3200 software that Babble's engine discovered through the engine's megalomania and paranoia. Max Ben-Aaron, Robert Osborn, and Steve Moshier dedicated two lunchtimes to discussing the paper and greatly improved its content. David Krumme and Remy Evard read the manuscript and provided helpful comments. Particular thanks to my student research group, including Michael Gilfix, Noah Daniels, John Hart, and Scott Pustay, for walking alongside me on this journey of discovery, and putting up with endless discussions of what Babble can do, before it could do it.

Author Biography

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. 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 +1 617-627-3674.

References

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

[2] B. Boehm, "Software Engineering Economics," *IEEE Trans. Software Eng.* **10**, No. 1, 1984.

[3] R. Brooks, *The Mythical Man-Month*, Addison-Wesley, Inc., 1982.

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

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

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

[7] K. Caputo, *CMM Implementation Guide: Choreographing Software Process Improvement*, Addison-Wesley-Longman, Inc, 1998.

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

[9] A. Couch and G. 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] A. Couch and M. Gilfix, "It's elementary, dear Watson: applying logic programming to convergent system management processes," Proc. Lisa-XIII, Usenix Assoc., 1999.

[13] C. Goldfarb and P. Prescod, *The XML Handbook, 2nd Edition*, Prentice-Hall, Inc., 2000.

[14] C. A. R. Hoare, "An axiomatic basis for computer programming," *Comm. ACM* **12**, pp. 576-581, 1969.

[15] M. A. Jackson, *Principles of Program Design*, Academic Press, 1975.

[16] M. A. Jackson, *System Development*, Prentice-Hall, 1983.

[17] D. Knuth, "Literate Programming," *Computer Journal* **27**, No. 2, 1984.

[18] C. Kubicki, "The System Administration Maturity Model – SAMM," *Proc. LISA-VII,* Usenix Assoc., 1993.

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

[20] T. McCabe, "A software complexity measure," *IEEE Trans. Software Engineering* **2**, 1976.

[21] B. Meyer, *Introduction to the Theory of Programming Languages*, Prentice-Hall, Inc, 1990. Chapter 9: "Axiomatic Semantics."

[22] R. Osterlund, "PIKT: Problem Informant/Killer Tool", to appear in *Proc. LISA-XIV*, 2000.

[23] R. Osterlund, ‘‘PIKT Web Site,’’ http://pikt.uchicago.edu/pikt.

[24] R. Ousterhout, *TCL and the TK Toolkit*, Addison-Wesley-Longman, Inc, 1994.

[25] R. Pressman *Software Engineering: A Practicioners' Approach*, Fifth Edition, Prentice-Hall, Inc., 2000.

[26] E. Ray with C. Maden, *Learning XML*, O'Reilly and Assoc., est. release Jan. 2001.

[27] L. Wall, T. Christiansen, and R. Schwartz, *Programming Perl,* 2nd edition, O'Reilly and Assoc., 1996.

[28] D. Watt, *Programming Language Syntax and Semantics*, Prentice-Hall, Inc., 1991.

[29] The Carnegie Mellon Software Engineering Institute, *The Capability Maturity Model: Guidelines for Improving the Software Process,* Addison-Wesley-Longman Inc, 1995.

[30] Lightwave Communications, Inc, http://www.lightwavecom.com .