Under 10 Flags

(not always smooth sailing)

David Tilbrook

Sietec Open Systems Division

ABSTRACT

Creating and managing source that can be installed successfully across a wide range of systems and machines is not easy!

This paper discusses some of the issues and problems that arise when managing a moderately large body of software on a wide variety of machines and environments, and presents some of the strategies used to overcome the many differences among systems.

Finally it registers some complaints and challenges to the reader.

The objective in writing this paper is to provide background information for the closing presentation at the 1989 Usenix Software Management Workshop. The author and Barry Shein of Encore are the co-chairs for this workshop.

1. Introduction

Despite my use of UNIX® since 1975, I cannot really call myself a UNIX user. The system I use is the sixth edition (circa 1977), plus a source control system from PWB/Unix (the SCCS system), the SIGSTOP signal from [24].?bsd, five of the settings provided by termcap or terminfo databases (I don't care which), the DBM database of the seventh edition, a small subset of *csh*(1) (modified to incorporate my cursor line editor), and some networking. However, I do supplement this somewhat modest subset with 250 other programs that I refer to as the D-Tree.

I am quite satisfied with this approach to UNIX, but I appreciate that the reader might think that this is due to the lack of alternatives.

This is not the case.

At this very moment I have a choice of five different flavours of UNIX and at least four (maybe more) window managers. As I write this document, there are seven graphic workstations within hitting distance, all unoccupied. I have made several serious attempts to use publicly available window managers, but, each time, I decided that I could not afford the aggravation and the considerable loss of performance and effectiveness.

But this paper is not to proselytize my approach to UNIX. It is to discuss how I deal with a problem that I have, given that approach.

That problem is that I need to install the D-Tree on every machine that I use or plan to use, and porting 8 megabytes of source (including documentation) is not as easy as you might think. There are little differences among the various flavours of UNIX that introduce some minor[1] impediments.

But I am relatively confident when I state that I have evolved a strategy and its supporting tools that can be

[1] "minor" when compared to the English Channel.

applied to create and install software on virtually any UNIX system[2].

In support of this claim, during a four hour meeting on Jan 31st, 1989, I installed the D-Tree on three different environments. This required creating a 30 line configuration file for each machine (by editing a copy of the prototype) the fixing of a sign extension problem reported to me by another user, and the modification of four other sections of code.

All of the latter modifications were required to compensate for the host's non-conformance with the base system (e.g., 4.{1,2,3}bsd unix5.{1,2,3}) and could be attributed to the schizophrenia of dichotomous[3] UNIX systems. (An example of one of these problems will be given later.) All of the constructions and installations used the same source files, via a distributed file system, without copying, linking, or changing the kernel's *namei* routine.

However, for me to claim success based solely on my experience, should be considered in the same manner as one would consider an Athena project leader saying he had no problem installing X11.

There are a number of other testimonials[4] including one from an relatively inexperienced D-Tree user (Nick Kosche at Sybase Inc.), who performed multiple first-time installations in three days, with some phone consultation with Pat Place of the SEI. One of his installations was on a new (for the D-Tree) machine, a new operating system, and a new version of C[5] yet the installation gave rise to only three previously unencountered problems: token pasting, modification of constant strings such as *mktemp*("/tmp/xXXXXXX"); and C compliance over entire files. (For example, even the conditionally suppressed sections have to be tokenized.) Pat has conveyed to me that Nick has very a positive attitude towards the experience. I find that encouraging.

1.1. Warning - reading the following may be upsetting

One of the recently encountered problems merits presentation due to its being a particularly good example of the kind of problem one faces in trying to make portable code:

On a system, that shall remain nameless, the header file <sys/stat.h> contained

```
#define S_IFIFO S_IFLOCK /* for sys5.3 compatibility */
```

Unfortunately one of my programs, which had been successfully ported to some 35 different flavours of Unix, contained:

```
#include <sys/stat.h>
...
        switch (...) {

#       ifdef S_IFIFO
        case S_IFIFO:
        ...
#       endif

#       ifdef S_IFSOCK
        case S_IFSOCK:
        ...
#       endif
        }
```

which raises the extremely useful and informative

[2] The D-Tree strategy and toolset are being used on non-UNIX systems, but not by me, so I cannot eliminate the qualification.

[3] Very close to "dicey" in the dictionary.

[4] provided on request

[5] ANSI C is a dialect of C, just as English is a dialect of German.

```
duplicate case label 49152 in switch
```

diagnostic. The solution is rebarbative and not foolproof (never underestimate the ingenuity of a fool) but amusing.

```
/* On some systems S_IFIFO is defined as S_IFSOCK
**      so undefine it
** Let us hope that no one ever
**      defines S_IFSOCK as S_IFIFO!
*/

#if defined(S_IFIFO) && defined(S_IFSOCK)
#       if (S_IFIFO == S_IFSOCK)

#               undef S_IFIFO

#       endif
#endif
```

Problems such as the above are exasperatingly common. In the following sections, I will try to explain some of the D-Tree philosophy (and the approaches employed) to ensure that installation continues to be relatively painless.

However, the philosophy does depend on the availability of tools beyond those offered by vanilla UNIX. To describe the construction suite in full is beyond the scope of this paper. There is a brief outline of some of the major goals for the system in the next section, followed by a listing of some of the more important aspects of the approach.

2. Quod <Erat|Est|Erit> Faciendum (qef)

qef is the most recent of a series of my UNIX construction systems, starting with work done with Tom Duff at the University of Toronto in 1976. *qef* (a.k.a., *qed*++) is primarily a driver used to control software construction and installation. Although *qef* has evolved and changed dramatically as it has been applied to more and more applications and systems, the primary objective of the research remains unchanged, and this was, and is, the creation of an approach to software construction and installation that meets the following criteria:

Constructions and Installation without Change to Source

The construction system should support the initial installations of large collections of software, on a variety of machines, at a variety of sites, without requiring the modification of **any** source, using a trivial configuration mechanism (e.g., fill in the blanks) and a single command (e.g., ‘‘qef Install Man Post[6]’’).

source is defined to be **all** the information that is created manually or must be provided by the supplier. This, by definition, includes **all** the information used to control construction and installation, such as *make*(1) files.

Ease of Distribution Upgrade or Modification

The system should support the upgrading of a previously installed body of software by the simple addition of new source files and/or the removal or replacement of old source files, and the issuing of a single command (e.g., ‘‘qef Install’’). Furthermore, the addition or removal of source files should rarely, if ever, require the modification of the construction control files.

---

[6] The *qef* system permits the division of installation into three phases: software installation; installation of documentation; and post installation procedures.

Controlling Information Specification and Use

The system should provide mechanisms to specify and/or retrieve required control information (e.g., the target location of the installation) from one and only one place, or through one and only one interface. In other words, user supplied information should never have to be expressed more than once. A user should be able to feel confident that all aspects of the construction system that need such information, should retrieve it in a manner that ensures consistency across the entire system and all uses. Furthermore, it should be possible to ensure that correct and consistent values are being used no matter what part of the system is invoked. That is to say, the information should be the same when retrieved during a full or partial construction, or the invocation of some subset of the construction system.

An extension, or almost a prerequisite, of this objective is that the system should support the trivial addition of new pieces of construction control information. For example, if the system needs to support *widget* compilation, there should be a simple mechanism to specify the name of the *widget* processor and its normal flags through a unique and universally accessible interface.

Finally, if the default value of a piece of control information needs to be changed or overridden within some subset of a source distribution, it must be ensured that the modification is effective over the entire subset.

Reduce Required User Supplied Information to Minimum

The system has to provide mechanisms to express construction specifications as succinctly as possible.

For example, in a directory of source that is compiled and archived into an object library, it should be sufficient to state: ‘‘build and install a library called X’’. From this statement alone, the system should be able to generate all the required information that will perform that task.

The mechanism used to convert simple specifications into the full construction procedures must be applicable to as wide a range of applications as possible. It should be trivial for a user to add new procedures[7]. Furthermore, it must be possible to state possible variations to a procedure easily and tersely. Such variations should not require the total restatement of the construction rules[8].

Phased Construction and Distribution Subsets

The construction system must support partial or phased constructions such that they are absolutely equivalent to the same constructions taking place as part of a full construction. That is, the user should be able select parts or phases of the construction to be done and be assured that the behaviour is equivalent to those selected phases being done as part of a full construction.

Furthermore, there should be few (if any) modifications required if the source to be processed is a subset of the full distribution. For example, one should not have to modify the construction control information when parts of the full distribution are excluded for economic, licensing, or other reasons.

Consistency Between Incremental and Full Constructions

The system must guarantee the consistency of objects produced by an incremental construction with those that would be produced by a full reconstruction. Incremental reconstructions are defined to be those constructions that skip some construction steps if the object to be produced is determined to be up-to-date. The construction system should ensure that an object is reconstructed if there is any aspect of its construction that has changed, that might result in a significant change in the object itself.

---

[7] In *make*(1) one can add new suffix rules to an individual *make* file, however, that rule must be added to every file that might use it. To make it universally available normally requires recompiling the *make* program itself.

[8] Traditional *make*(1) provides suffix rules that are used to specify the default constructions for an object. But, if any variation from the normal rule is required, the entire construction, incorporating that variation has to be specified.

Support for Testing and Development

Finally, the system should facilitate testing and modification without endangering the production versions, whilst ensuring the almost trivial addition of those changes or extensions when completed. Experimentation with the source by a programmer should be easy to initiate and economic (i.e., not require large amounts of disk space). The system should ensure that the behaviour of a test system in a non-production environment is truly reflective of the behaviour of that system in the production environment.

2.1. Caveat

The above objectives are ambitious. The task of proper system construction and installation is a difficult one. Any approach that attempts to solve the problems of large scale software will itself be complicated and/or expensive. It will have to deal with a great many special cases (the ability of programmers to create new ways of complicating the lives of the software manager is unbounded).

The *qef* system is a tool-based approach. Instead of depending on a single tool (e.g., *make*(1)) and the willingness of people to spend their time (wasting their employer's money) manually creating *Makefiles*[9], *qef* depends on a disciplined approach to the organization of source trees, some adherence to naming conventions, and the avoidance of wildly divergent construction techniques within a single directory. If these criteria are met, then *qef* can apply its tool set to build far more accurate, flexible, and efficient constructions that can be done manually, and at huge savings. Decreases in the amount of manually generated information of three orders of magnitude are not uncommon, and as often as not, the conversion finds many errors in the original information.

But, one does have to learn about a dozen new tools[10] and two relatively simple languages, and has to adopt disciplines that might not conform with one's usual approach.

3. Parenthood

I have been fortunate that there have been a number of people[11] that have been willing to learn the tools and in turn made considerable contributions. Given Pat Place's continued participation and enthusiasm, as part of the preparation for this paper I asked him to help to enumerate the major aspects of the approach/philosophy. The following is that enumeration.

3.1. Constantly worry about portability

We have made a habit of assuring that any new code or modifications are tested on as many different environments as soon as is convenient. When a portability problem is discovered, a thorough search of all the source is done for any other occurrences of that problem immediately. Furthermore, if the problem can be avoided by adopting a stylistic change, we do so. Occasionally we will recheck the source for introduction of new instances of past problems.

One recent payoff for this approach was the ease of conversion to ANSI C. We were informed of the problem of having comments on “#else” or “#endif” statements some years ago and immediately found and fixed all instances, even though it was not necessary until last month.

3.2. Frequent testing of the construction process

In addition to the testing of the programs, the construction process itself must be exercised extensively. Complete reconstructions must be attempted on a regular basis. Any failure to complete necessitates the test be repeated in its totality until it succeeds. (Unfortunately this requirement has to be relaxed on those systems where the mean time between file system failures is measured using the hour hand instead of a

---

[9] The traditional use of 1000 line *make* scripts is not only time wasting, but the scripts themselves are inevitably incomplete, inflexible and, invariably, poorly maintained and wrong.

[10] The construction system is a relatively small part of the D-Tree.

[11] Calvin Delbarre, Robert Biddle, Adrian Pepper, Derek Thorsland and Mark Brader, all formerly of BNR, Andy Greener of Systems Designer Ltd. and Imperial Software Technology, Jim Oldroyd of the Instruction Set, Pat Place and Mike Day of IST and now the SEI, and Tom Lord of CMU, to list a few.

calendar).

3.3. Adopt a common coding style

Enforcing a common coding style inevitably results in long and bitter arguments, but, when one considers maintaining 2000 source files in co-operation with others, being able to confidently apply rules for searching and processing the text has substantial returns.

For example, **all** routines in D-Tree C source files have the following form:

```
<type>
name(arguments, ...)
	type arguments;
	...
{
	...
}
```

Consequently:

```
grep '^[a-zA-Z0-9_]+(' files
```

will find all routines in the system, and nothing but the routines. Going back one line gets the type. Going forward to the first line matching '^{' captures all the arguments.

Another simple but defendable rule is never using multiple declarations as in:

```
int i, j, k;
char *p, c, **p, carray[];
```

as doing so prevents one from easily changing the type for one of the variables and not the other.

3.4. Always All, Everywhere, Always, or Never - Never Sometimes

We attempt to ensure that special cases are avoided as much as possible. For example:

- for ALL D-Tree library source directories, ALL the files that have the same suffix, are treated in exactly the same way.
- ALL variables or routines that can be static, are static.
- ALL source files have a suffix or a specific name that is ALWAYS spelled in the same way (e.g., always “README”, never “ReadMe” or “Read_Me”).
- *cc*(1) “-D” flags are NEVER required. Any options or configuration parameters are ALWAYS put into header files.
- For EVERY object file (other than links) there is a deterministic method for finding its source or its source location. For example, for any installed library “libX.a” there is a directory called “libX”; for any “file.o” there is a source file called “file.X”. For every installed file “X”, there is a file in the source tree called “X” or “X.Y”, or “X” is a link to file “Z” for which this rule applies.
- ALL library routines contain an SCCS line of the following form:

  ```
  #if	!defined(lint) && !defined(NOSCCS)
  char	SIDfilename[] = {"%Z%%Q%(%M%)<tab>%I% - %E%"};
  #endif		/* not lint or NOSCCS */
  ```

  The “%Q%” is ALWAYS “-l<libname>”.

  This way:

  ```
  wot -o binaries
  ```

ALWAYS produces output that can be processed using simple stream editors to generate the list of files that were used to produce the program.

- ALWAYS use:

  ```
  #include    <file.h>
  ```

  NEVER use:

  ```
  #include    "file.h"
  ```

  The latter form prohibits creating a modified version of "file.h" in the current directory without copying into that directory all the files that might "#include" it.
- EVERY source file is "lint" free as much as is possible.
- EVERY library has a complete *lint*(1) library, automatically generated directly from information embedded in the source files for that library.
- EVERY program has a "-x" flag that outputs the program's brief description, synopsis and flag description.

There are many other rules that could be listed, however, the above should be sufficient to illustrate the principle. However, there will ALWAYS be someone who objects to one of the above restrictions. Some people are NEVER satisfied.

3.5. Separation of Object and Source

The D-Tree construction tools, due to contributions by Tom Lord, support the total separation of source and object files, using a source path mechanism. This is NOT done using virtual directories or links, but through full path names and regeneration of the construction script if a source file moves up or down the source path.

This does occasionally cause problems. Source path names can exceed the limits imposed by language processors (one version of *cc*(1) truncates files names in the stab entries at 50 characters). Many versions of *cc*(1) can only support eight "-I" flags which can easily be exceeded when there are three directories in the source path.

However, there are very real advantages to the scheme that are not realized using links or virtual directories. Furthermore, our approach is cheap and works on all UNIX systems in the same way.

3.6. Reduce your dependence on others as much as possible

It is amazing, and somewhat dismaying, how unreliable that other guy's system is. I know your system is okay, but his *basename* does not work in certain cases.

Currently, the only tools that the D-Tree construction uses, that it does not provide itself, are:

> ar, as, awk, cat, cc, cmp, comm, cp, cpp, diff, echo, grep,
> lex, lorder, lint1, make, mkdir, mv, pwd, ranlib, rm,
> sed, sh, sort, strip, touch, tsort, & yacc[12]

Other tools can be added to this list relatively easily, but are done so in manner that permits a unique easily overridden specification and the location of all uses using *grep*(1).

In the D-Tree EVERY (boring isn't it) tool invocation in either C source or construction scripts is done using a macro beginning with "_T_" defined in either a *cc* or *qef* header file.

---

[12] *make*(1) is employed sparingly for very simple scripts. *sed*(1) only required to deal with *yacc*(1) deficiency, namely eliminating name clashes. *awk* is used once in the bootstrap to perform a relatively simple task that should probably be done using *ed*(1), since: "If you have a problem and you think *awk*(1) is the solution, then you have two problems." - David Tilbrook

3.7. E2BIG - U2SMALL

Many of the D-Tree tools support the facility to specify that the arguments to be processed are contained in the standard input or the argument files. The commands:

```
grep pattern `cat SourceList`

        or

xargs grep pattern <SourceList
```

fail or are unacceptably slow when *SourceList* contains thousands of files and tens of thousands of characters. Quick and efficient ways to process large lists of files are essential.

3.8. Cleanliness is near to Portliness

The removal or creation of a single file can be extremely damaging and can frequently invalidate any testing or construction. Tools to check the source, object, and installed trees are an important part of the D-Tree construction system, used to help manually maintain lists of the source, object and installed files. During periods of heavy activity this package is invoked on an almost hourly basis. It can be tedious, but it yielded a very high return on investment the day a file server silently misplaced 72 of my source administration files, along with a number of other files that could be reproduced. The file list package discovered the loss within an hour and provided all the information I required to create the current version of the software. Fortunately there was no file for which both the 'g-' and 's-files' had been lost.

Unfortunately, I was unable to reproduce the system as it had existed the previous day (thus preventing any regression testing) as that was also the day that it was discovered that the backup system had failed for the previous 5 nights.

4. System Configuration

This is the penultimate section, and as such is probably the last opportunity to present something concrete. In previous sections I referred to the configuration file and the unique source of all system dependent information.

The only difference between two D-Tree installations is the specification of the two arguments to the *SetUp* command, which is the first command to be executed. This command is used to establish ALL the environmental settings and controls. The "-x" flag to *SetUp* yields:

SetUp [-x] [-s SrcDir] Machine_file Object_dir

initial setup of a D-Tree distribution

-x	display this explanation
-s SrcDir	name to be used for source directory
machine_file	name of the machine file to be used
Object_dir	directory to contain objects (default ..)

Shell script builds the required files to in Object_dir/Magic to build the D-Tree system. The machine file is described in the Installation documentation.

The "Machine_file" is the only place that system, machine or configuration information is specified. The following is a representative subset of the 28 lines of this machine's machine file.

```
ANSI_C 0   # __STDC__ not to be trusted
BOOTCFLAGS -g   # Cflags when booting
CFLAGS -O   # default Cflags when installing
CONTAXDIR %/contax   # default for the contax database
DESTDIR /usr/dtree   # default for target directory
```

```
DORANLIB 1   # 1 if do ranlib, 0 otherwise
DTREE /usr/dtree   # default for the installed dtree
_F_INSTAL -qaIs   # default instal(1) flags
INSTLOG @LclVarDir/Ilog   # instal(1) audit trail
MISSINGRTNS No_strchr No_strrchr No_mkstemp
        # missing routines to be provided by -ldtree
MUSTALIGN 1   # must pointers be on word boundaries
OPTIONS   # e.g., BOGUSCSH, BLIT, etc.
ORGANIZATION Sietec Open Systems Division
PATH /bin:/usr/bin:/usr/ucb   # default user path
SYSTEM pyramid_4.2bsd   # See below.
```

Note that some of the above values could be deduced by programs, but such processes can go wrong and when they do they are difficult to correct. The SetUp process creates a configuration file that is the Machine file plus the source directory name and the name of the machine file itself. This is copied into the ObjDir tree, which was also created by SetUp. The first phase of the construction process is to use the variable value pairs to process prototype header files. For example, the following is the prototype header file that contains the default D-Tree location:

```
#ifndef     DTREE
#define     DTREE     "@DTREE@"
#endif      /* not DTREE */
```

The ‘‘@DTREE@’’ is replaced by the DTREE value as assigned in the configuration file.

There a dozen files that are configured using the machine file and then installed in the required location as part of the boot strap process.

In addition to the above style of configuration, the SYSTEM setting is passed as an argument to *mksyshdr*(1), which builds a header file called ‘‘system.h’’. This header file is used throughout the system to conditionally select or suppress code or constructions. No only other mechanism used is to test for the definition of a macro in a header file such as was illustrated before. An example is given in the next section.

No other configuration is used or required, with the exception of the processing of the OPTIONS settings. This is done using *mkopts*(1) which creates a header file, as part of the construction of a system that depends on the absence or presence of an option.

I think that one of the major reasons that the D-Tree has been successfully and easily ported to so many systems is that the configuration mechanism as simple as it is.

4.1. Lets play “Hunt the Header”

One of the most frequent and irritating problems that one faces when building UNIX software that is to be ported to a wide range of systems, is the use of header files. Header files are supposed to help make porting easier. In many cases they make it more difficult.

For example, you might want to use ‘‘utmp.h’’ or ‘‘sys/param.h’’. Both these files define types and macros that are highly system dependent. But, on a system V system, the inclusion of either must be preceded by an inclusion of ‘‘sys/types.h’’. However, on 4.2bsd, both ‘‘utmp.h’’ and ‘‘sys/param.h’’ include ‘‘sys/types.h’’ and the latter is not idempotent.

To overcome this problem I use:

```
#include     <envir/sysVtypes.h>
```

prior to any inclusion of ‘‘utmp.h’’. This header file contains in part:

```
#ifndef     SYSVTYPES_H
```

```
#	define	SYSVTYPES_H

#	ifndef	SYSTEM_H
#		include	<envir/system.h>
#	endif	/* SYSTEM_H */


#	if	SY_U5x
#		include	<sys/types.h>
#	endif	/* SY_U5x */

#endif		/* SYSVTYPES_H */
```

I have totally done away with using ‘‘param.h’’ by creating a process that extracts all the manifests it needs and putting them into another header file. The headaches caused by using ‘‘sys/param.h’’ became unbearable and I only required five constants. I also added to this process the calibration of some other difficult to retrieve or use values (the maximum lengths of an archive member name or a user’s uid) thereby eliminating other porting problems.

The other major header file headaches are:

- Where is it (‘‘sys/time.h’’ vs. ‘‘time.h’’)?
- What is it called now (‘‘file.h’’ or ‘‘fcntl.h’’)?
- Which one do I use (‘‘sys/file.h’’, ‘‘unistd.h’’) and what do I do if there isn’t a file that provides what I need (e.g., ‘‘R_OK’’).

To deal with the above I have created 31 extra header files, listed below, all of which reside in *$DTREE/hdrs/envir*.

access.h	what are values for access(2)'s 3rd argument
armagic.h	magic number for ar(1) files
canchown.h	can non-root user chown files?
company.h	site address and phone number.
doranlib.h	do ranlib(1)?
envuser.h	$LOGNAME, $USER, or something else?
file.h	all sorts of stuff
gethost.h	where am I?
gidget.h	what group(s) am I?
jerq.h	where's the jerq ioctl?
keepfids.h	what's the highest standard fid (2 or 3)?
localtime.h	where's time.h?
lock.h	how do I lock a file?
lseek.h	what are values for lseek(2)'s 3rd argument?
mailsys.h	where's incoming mail?
ndir.h	what does a directory look like?
open.h	what are values for open(2)'s 2nd argument?
options.h	what options?
param.h	5 magic numbers
printf_t.h	whats the sprintf(3)'s type?
readdir.h	see ndir.h
sccsbin.h	where is delta(1)?
siglist.h	what are the signals called?
sigrtn.h	what's the type returned by a signal catcher?
sysVtypes.h	see above!
syscall.h	where's syscall.h?
system.h	what system is this anyway?
termlib.h	termcap or terminfo?
uid_t.h	type of a user id.
toolpath.h	where is sort, chown, chgrp, ...?
wait.h	type of argument to wait(2)

By using the above I have been able to reduce the number of places that conditionals based on the system or configuration to a very small number and we have been able to quickly adapt to changing needs as required.

5. Challenges, Complaints and Prognostications.

5.1. File Systems and Software Management

I expect file systems to work. I expect backup systems to work[13]. Furthermore, if the file system is unreliable, I expect software produced on that system to be just as unreliable. I have yet to see a counterexample.

However, there are a number of things that a system can provide that can be used to circumvent bogus file systems:

- Standard tools that behave properly. For example, */bin/cp*(1) must succeed quietly or fail noisely. Using ''/usr/local/cp'' is not acceptable.
- A guaranteed (it works or fails immediately) inexpensive indivisible append to a file of a single line. Without the above there is no acceptable way of building an audit trail.
- Reasonable performance. When an unreliable file system is also unacceptably slow, one has to compensate by circumventing or not doing the ''proper'' thing.

[13] I did not believe these were unrealistic expectations since I cannot recall ever losing an irreproducible file that was more than a day old, up until 1987.

There are many other requirements that I could state, but these are the three that I see as most important.

5.2. Elimination of mutually exclusive processes

A number of process (e.g., *yacc*) can not be run in parallel as they produce or use a file name that is fixed. This must be fixed.

5.3. A lint that works

System providers continue to deliver lint libraries that are wrong, code that is unlintable, and lints that don't work properly. *lint* is an extremely useful tool. Unfortunately, far too often it fails to report some problems due to bugs or inaccurate lint libraries.

5.4. Documentation that is accurate

I would like to be able to prepare code for your system before I use your system. If your documentation of standard tools and system calls is missing or wrong I cannot do that.

5.5. Prognostications

I do not expect the porting task to get easier. Every new system introduces some new variation, and the old ones do not go away. But, there seem to be an ever increasing number of people concerned with the porting problem and that is encouraging.

References

1 T.Lord, *Polices and Tools for Hierarchically Managed Source Code Development*, San Francisco, 1988.

2 D.M.Tilbrook and Z.Stern, *Cleaning Up UNIX Source -or- Bringing Discipline to Anarchy.*, EUUG Dublin Conference Proceedings, September 1987.

3 D.M.Tilbrook and P.R.H.Place, *Tools for the Maintenance and Installation of a Large Software Distribution*, EUUG Florence Conference Proceedings, April 1986. Usenix Atlanta Conference Proceedings, June 1986.

4 D.M.Tilbrook, D.Thompson, and M.Lorence, *A New Look at Some Old Problems*, Unix Review, April, 1988.

5 L.Branagan and D.M.Tilbrook, *Installation Documentation Documentation*, EUUG Portugal Conference, Oct. 1988.

6 D.M.Tilbrook, *Quod Erat Faciendum*, unpublished.