A Case for *make*

Glenn Fowler
gsf@research.att.com

AT&T Labs Research

Abstract

The *make* command has been a central part of the UNIX* programming environment for over fifteen years. An excellent example of a UNIX system software tool, it has a simple model and delegates most of its work to other commands. By dealing with general relationships between files and commands, *make* easily adapts to diverse applications. This generality, however, has become a handicap when compared with specialized integrated programming environments. Integrated environments are collections of tightly coupled (*seamless*) programs that can take advantage of programming language details not available to the loosely coupled (*tool-based*) *make* model.

There are limitations to both approaches, but it would seem that the *make* model, at least for software construction, is reaching the breaking point. *make* can be revitalized by abandoning restrictive implementation details and by extending the basic model to meet modern software construction demands. This paper explores these demands and changes and their affects on the UNIX system tool-based programming style.

1 Background

make is a tool-based software configuration management command [Feld]. A user asserts relationships between files and commands in a *makefile* and runs *make* to fulfill the assertions. An assertion

```
target  :  [ prerequisite ... ]
        action
```

specifies the action (shell command script) that generates a target file from zero or more prerequisite files. A target is *out-of-date* if it has no prerequisites or if one or more of its prerequisites is *newer*, where *newer* is determined by comparing system supplied file modification times. Executing the action brings the target *up-to-date* with all its prerequisites and fulfills the assertion. Assertions are recursive in that a prerequisite may appear as the target in another assertion.

The set of all assertions forms a directed graph. *make* traverses this graph from a given main target node, and recursively brings each out-of-date parent target node up-to-date with its child prerequisite nodes by executing the appropriate actions.

The *make* model joins declarative and sequential programming styles into a single language. Action ordering is implied by the target-prerequisite relationships: a target action executes only after all prerequisites have been brought up-to-date. Commands within actions execute sequentially, as determined by the shell.

This simple model allows much leeway in *make* implementations. Some implementation decisions, however, needlessly hamper *make* usage.

---

* UNIX is a trademark of AT&T.

2 Programming

How does *make* fit into a complete programming environment? Consider the basic program development cycle:

edit → **compile** → **execute** → **query**

Much is omitted here, such as the research, design and documentation stages that may precede the first **edit** and the enhancement, bug report, packaging and marketing stages that may follow the first successful **execution**. However, these stages capture the typical *programmer* development cycle.

A user **edits** source based on some design, **compiles** the source if necessary, and **executes** to test results. Testing and debugging involves information **queries** to files and commands in all cycle stages. This brings about design changes and another round of the cycle.

make most often handles the compile stage, but tie-ins with execution and queries are also possible. For example, a language interpreter may use makefile information to combine compilation and execution, or special makefile targets may generate *lint* or static language analysis information for queries. The problem is, such tie-ins require extra work by the makefile programmer.

An integrated programming environment (IPE) combines the development stages into a seamless environment [WDK]. The user rarely executes individual commands or manipulates inter-stage information directly. In contrast, the tool-based UNIX system environment has many commands for each stage, with user accessible boundaries (seams) between each.

An IPE controls the entire development cycle and can readily maintain internal information to be shared among the stages. For a given programming language and system the tool-based approach is clearly at a disadvantage. Communication between commands is limited to the individual command interfaces. It may not be possible to share or even extract the desired information. For example, an editor can easily determine incremental source file changes, but without an incremental compiler this information is useless.

Tool-based development, however, has the advantage of flexibility. If all commands follow a similar, general interface convention, then different commands may be substituted at any stage. For example, one changes the C compiler by adding the argument **CC=***mycc* to the *make* command line. Changing the compiler for an IPE, on the other hand, would often require compiler interface changes.

make can enhance the development cycle by providing the thread to close these tool-based seams. To do this with minimal user overhead, however, requires some basic changes to the *make* command. Such changes have been going on for years, but in a somewhat haphazard way [CF, EP, Hume, Palk, SM, SV1]. The following interrelated goals give direction to these changes. *make* must:

- easily **scale** from one to many users
- do **accurate** change propagation
- have sensible **information** distribution
- run with reasonable **speed**
- provide **portable** configuration management

These goals are discussed in the following sections.

3 Scale

A single user often generates many versions of a given source file. When many users share source configured by a common makefile the number of file versions and versioning methods may become unmanageable. Some users save old versions using complete backup copies, others use some form of source code version control. Past attempts to integrate version control and *make* have resulted in increased *make* complexity, mostly by forcing *make* to do the actual version extraction. A more elegant solution provides a unified concept of version file that is available to all commands. This concept, called *viewpathing* [EP], forms the basis of the *3D Filesystem* [KK].

A *viewpath* is an ordered sequence of directory hierarchies in which files occurring earlier in the sequence take precedence over files occurring later. It provides a logical hierarchy that is a layering of many physical hierarchies. Many users can share common source by creating local *private* views that overlay the common *official* source view. All changes are done in private views that contain only those files that are different from the corresponding files in the official view. Since many users may be involved, it is important that official view updates be a controlled operation. Figure 1 shows the physical directory hierarchies for both private (*prv*) and official (**ofc**) source. Figure 2 shows the logical view provided by viewpathing the private view on top of the official view, where the italicized file names are those found in the physical private view.

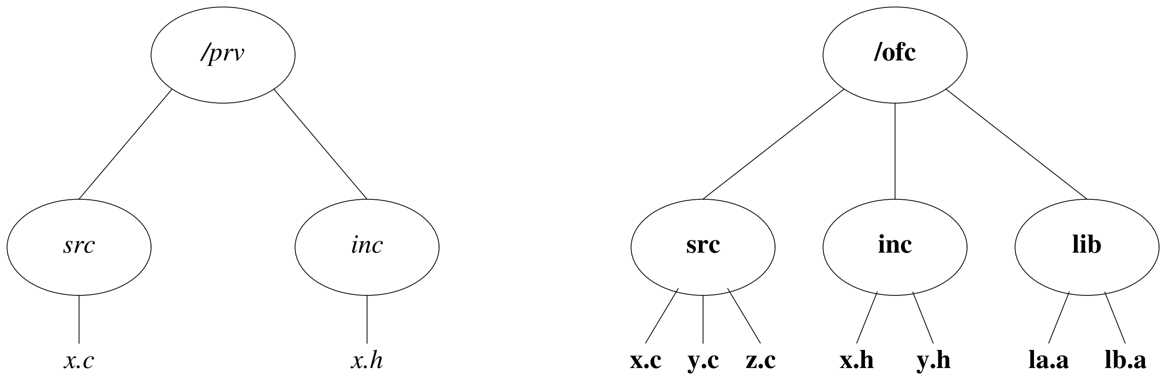

Figure 1. Physical view

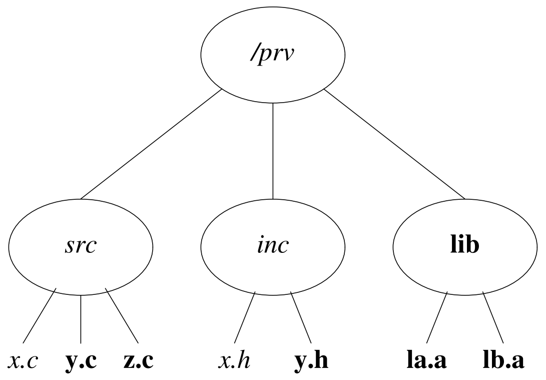

Figure 2. Logical view with viewpathing

Not only does viewpathing avoid the problem of each user having a complete copy of official source, the physical views clearly show the official files that need to be changed.

Viewpathing should be a system wide service available to all existing commands. A transparent operating system or shared library implementation would be ideal. Alternatively, commands of interest could be re-compiled with a special viewpathing library that captures all pathname system calls. Finally, if source and relocatable object files are not available, then viewpathing can be implemented entirely within *make*. This last solution, however, requires that *make* pass physical file pathnames, and possibly additional options, to all commands that it executes.

As with most UNIX system abstractions, there are a few commands that must manipulate the internal details. For viewpathing, *make* must distinguish between the physical views to detect files that may change when the viewpath changes. This is not possible, however, without *state*, a feature that will be explored in more detail in the next section.

4 Accuracy

make must accurately propagate source file changes to the generated targets. Given the importance of this issue it is surprising to find failings in the current algorithm. The first relates to the monotonic out-of-date test and the other relates to makefile specifications and implicit prerequisites.

Recall that a target file is out of date if its modification time is older than any prerequisite time. Other than the system maintained file time, this is a *stateless* algorithm. The algorithm works well based on the following time assumptions:

(1) A file time is updated (to the current time) whenever the file contents change.

(2) All file times are relative to a common system clock.

(3) Action execution updates the target time to the current time.

Until the arrival of remote file systems, **(1)** and **(2)** were almost always true. Even when restricted to local file systems, violation of **(1)** could happen if archive files were restored with original times (presumably from the past), and violation of **(2)** could happen if the system were rebooted with an inaccurate system clock setting. All the conditions are easily violated when remote and local files system clocks differ. One drawback to viewpathing is that **(1)** may be violated whenever viewpath layers are changed. For instance, removing a middle view might uncover prerequisite files that are different but older than the current targets. These files will be considered up-to-date and the corresponding targets will not be updated.

Time inconsistencies may allow some changed files to go undetected, or they may cause some targets to be always out of date. This is unacceptable. Assuming an accurate local system clock **(1)**, **(2)**, and **(3)** can be met by adding state to the out-of-date algorithm.

Associate with each file *f* three times:

time.previous (*f*)
time.event (*f*)
time.current (*f*)

where **time.previous(*f*)** is the file time from a previous *make*, **time.event(*f*)** is the time when **time.previous(*f*)** was observed to have changed, and **time.current(*f*)** is the file time during the current *make*. Notice that **time.event(*f*)** may be later than the time *f* actually changed. A time value of 0 indicates that the file doesn't exist (**previous**,**current**) or that the event has not been observed yet (**event**). **time.previous(*f*)** and **time.event(*f*)** represent the state of *f* and must be saved between *make* invocations. The *statefull* out-of-date algorithm can now be stated in terms of these times.

A target file *f* is out-of-date if any of the following conditions are true:

(a) **time.previous(*f*) == 0**

(b) **time.previous(*f*)** != **time.current(*f*)**

(c) for any prerequisite file *p*, **MAX time.event(*p*)** > **time.event(*f*)**

A target *f* is brought up-to-date by:

(d) execute *f*'s action

(e) evaluate **time.current(*f*)** when the action completes

(f) set **time.previous(*f*)** to **time.current(*f*)**

(g) set **time.event(*f*)** to the current system clock

The time assumptions **(1)**, **(2)**, and **(3)** are met because only the *local* system clock is used in monotonic time comparisons*. Observe that since **time.previous(*f*)** and **time.current(*f*)** are only involved in strict inequalities the values need not be restricted to time stamps. For instance, makefile prerequisite, variable and action changes, ignored by most implementations, can be handled by adding the target prerequisite list and action contents to the **previous** and **current** values. Other items, such as file checksums are also accommodated. By retaining the time component, file-related state components (file-attributes) need only be recomputed when **time.previous(***f***)** != **time.current(***f***)**. In other words, **time.current(***f***)** == **time.previous(***f***)** implies *file-attribute***.current(***f***)** == *file-attribute***.previous(***f***).**

The other accuracy failing lies in makefile specifications. Most common makefile errors involve improper (or missing) implicit file prerequisite assertions. Explicit prerequisite files must be specified to the target action commands or the commands will fail, whereas implicit prerequisite files are determined by the individual commands and need not be specified for proper command execution. This is a difficult problem to attack since an incomplete makefile often appears to be correct, even after several *make* invocations.

Missing prerequisites are not benign, however. They emerge as changed files that go undetected. Undetected prerequisites can cause insidious run-time bugs, such as data structure mismatches between routines compiled with different versions of a common header file.

C language **#include** files exhibit most of the implicit prerequisite problems. Library files implied by the C compiler **–l***library* options and text formatter macro files implied by the **–m***macro* options are other examples.

A common solution for the C language provides an implicit prerequisite scanner that recursively scans C source files for **#include** prerequisites. The scan results are then appended to the makefile as prerequisite assertions. Some systems have a C preprocessor with a **–M** option that generates makefiles assertions for individual source files. This is attractive from a completeness standpoint, since the same preprocessor is used as part of the C compiler.

Separate source file scans provide static analysis only; a complete scan must be re-evaluated whenever any **–D** macro definition option, **–I** include directory option, viewpath configuration, or implicit prerequisite changes. Static analysis fails when an **#include** of a generated file is encountered before the file has been generated. The major static analysis shortcoming, though, is that the analysis is usually at the user's discretion. This may be satisfactory for single user projects, but omitting the analysis in multi-user project invariably leads to the omitted prerequisite bugs mentioned above. For this reason most large projects periodically clobber all generated target files, do a new static analysis, and regenerate all targets to ensure that all changes have been incorporated.

Another approach modifies the compiler interface to provide dynamic implicit prerequisite information directly to *make*. This is an improvement, since the prerequisite analysis is done whenever the source or prerequisite files change. As with static analysis, generated files are still a problem; *make* must create the generated implicit prerequisites before the compiler is called, but it must call the compiler to generate the implicit prerequisite list. Several compiler passes may be required to generate all nested prerequisites. This method is impractical in the general case because of the nonstandard *make*-compiler interface.

Although it complicates the *make* model, source file scanning supported directly within *make* provides the most consistent prerequisite analysis. *make* scanning can be driven by patterns specified in the global *make* rules. An advantage to this form of scanning is that it can be incrementally applied to the source and prerequisite files. In this way only those files that have changed since the last scan need be rescanned. *make* scanning also correctly handles generated implicit prerequisites by bringing the prerequisites up to date as they are encountered during the scan. A *dontcare* attribute for non-existent implicit prerequisite files within conditionals allows *make* to continue as if the files existed (with a very old modify time).

* In all algorithms there is a slight chance that an invalid system clock setting could make **time.current(***f***)** == **time.previous(***f***)**, even though the file has changed.

```
#ifdef mymachine
#include <mymachine.h>          /* dontcare */
#else
#include <theirmachine.h>       /* dontcare */
#endif
```

In this conservative approach, a source file potentially depends on all its implicit prerequisite files, even though a only small subset of these may actually be referenced by the compiler. This is not a serious problem, though, since in the common case only one conditional choice is present for a particular system environment.

A drawback to *make* scanning is the complexity of include forms for the various language processors. For simplicity, constructs requiring language dependent macro evaluation may be ignored by a given implementation.

Efficient scanning implementations require state to save previous scan information. State times are also necessary to determine when prerequisite files must be rescanned.

5 Information

The *make* information distribution must accommodate various programming styles, ranging from individual, independent users to large, multi-user projects spanning many machines. *make* currently has builtin rules that define the local environment, and a makefile **include** statement for makefile sharing. The builtin rules are usually compiled into the *make* executable, and are sometimes difficult to change or override. This flat information model encourages overspecified makefiles that expose many implementation details.

Most projects have makefile guidelines that require common variable definitions and target assertions. Consider a typical project makefile template:

```
COMMAND = cmd
SOURCES = cmda.c cmdb.c
OBJECTS = cmda.o cmdb.o

$(COMMAND) : $(OBJECTS)
      $(CC) $(CFLAGS) -o $(COMMAND) $(OBJECTS)

clean :
      rm -f $(OBJECTS)

clobber : clean
      rm -f $(COMMAND)

lint :
      lint $(SOURCES)

tar :
      tar cf $(COMMAND).tar makefile $(SOURCES)
```

Changing any of the common target assertions (`clean`, `clobber`, etc.) requires an edit of all project makefiles – not an encouraging prospect for large projects. A common solution, similar to the static prerequisite generators of the previous section, is to provide a local command that generates makefiles from project specific descriptions. As with static prerequisite generators, unless *make* usage is tightly controlled, the makefiles will frequently be out of sync with the specification files.

Another solution, more tolerant to change, is to use a project wide **include** file:

```
COMMAND = cmd
SOURCES = cmda.c cmdb.c
OBJECTS = cmda.o cmdb.o

include $(PROJECT)/command.mk
```

where the user makefiles define interface variables for the project makefile. The `$(PROJECT)/` directory prefix is required because *make* lacks the equivalent of the C compiler **–I***directory* include directory option. Given a well defined include file interface, all project makefiles can be enhanced by changing a single controlling makefile.

Now suppose a library target, rather than a command, is to be generated. Since *make* has no **if-else** programming support, a new include file interface is required:

```
TYPE = library
TARGET = lib.a
SOURCES = liba.c libb.c
OBJECTS = liba.o libb.o

include $(PROJECT)/$(TYPE).mk
```

These examples show that the key to manageable makefiles is high-level abstractions for low-level constructs. A few makefile language extensions allow for more concise abstractions. Notice that the last example is just a cumbersome emulation of a makefile procedure call and conditional test. An **if-else** construct would allow a single project makefile to test for target types:

```
if "$(TYPE)" == "command"
      /* command target assertions */
elif "$(TYPE)" == "library"
      /* library target assertions */
else
      /* error */
end
```

and a procedure call mechanism would eliminate the explicit project makefile interface variables:

```
include $(PROJECT)/rules.mk

cmd :source: cmda.c cmdb.c

lib.a :source: liba.c libb.c
```

Procedures, as presented here, are a simple extension of the *make* assertion style. In this example `:source:` is a programmable assertion operator that specifies the source files required to generate a target file. New operators may be defined using assertions:

```
source :operator:
      if "$(<)" == "*.a"
            /* library assertion */
      else
            /* command assertion */
      end
```

As opposed to shell scripts, operator actions are written in the makefile language. Operators are called with two parameter lists, accessed by the builtin variables; `$(<)` is the left hand side target list and `$(>)` is the right hand side prerequisite list.

Assertion operators encourage high level specifications. The operators break these down into normal assertions based on target and prerequisite attributes. A project can now define its *make* environment in terms of a few well defined operators. By providing enough abstraction, the same makefile style can adapt to a wide range of environments simply by changing the operator definitions.

Managing and accessing operator definitions now becomes a crucial issue. General operators and rules belong in the *make* base environment, traditionally provided by a set of builtin rules compiled into the *make* executable. Flexibility dictates that the builtin rules be moved to a more accessible part of *make*, so that they may be easily overridden or modified to suit local needs. This can be done by providing a base rules makefile that may be augmented by global makefiles specified either directly in the user makefiles or on the make command line. The latter accommodates portable user makefiles.

Programmable operators, combined with the base, global, and user makefile hierarchy offers the *make* user much flexibility. This flexibility allows generic makefiles to be used for applications not intended by the original makefile author. For the **:source:** operator above, source related common actions are easily added:

```
source :operator:
        target = $(<)
        sources = $(>)
        /* additional assertions ... */

print :
        /* print the sources */
        pr $(sources) | lp

tar :
        /* archive the sources */
        tar cf $(target).tar $(sources) makefile
```

By agreeing on a common set of base makefile operators, most user makefiles reduce to trivial assertions that associate target names with their source components:

```
make :source: make.h doit.c file.c main.c
```

State information, mentioned in the previous section, must also fit into the information hierarchy. It is most often kept in a file (*statefile*) in the directory where *make* is executed. State implementations must take care to handle cases where a directory contains more than one makefile. A reasonable solution is to use the makefile name as part of the statefile name. Viewpathing may also require a merge of statefile information from each view level.

For consistency and portability, the information hierarchy precedence must be clearly stated. Ultimately, the *make* user must have complete control over predefined information. This means that assertions must be retractable, a feature not found in many implementations. A consistent information definition precedence order, from highest to lowest, is:

(1) command line

(2) makefile

(3) environment

(4) global makefile

(5) base makefile

Some implementations reverse **(2)** and **(3)**, enhancing environment related makefile bugs. Other implementations have the base rules compiled into the *make* executable and provide **(5)** at the cost of *make* startup time. The most flexible implementations have no builtin rules, allowing the *make* environment to be completely redefined.

6 Performance

As with most seasoned commands, changes and enhancements run the risk of degrading command performance. New features usually present performance/functionality tradeoffs. If the cost of a feature is too high then knowledgeable users will either circumvent the feature or abandon the command altogether. Performance effects can be reduced by amortizing computations over many *make* invocations or by precomputing repeated evaluations.

Viewpathing slows performance by adding to the number of files to be checked during prerequisite time analysis. File system pathname lookups are expensive, so a *make* file and directory cache is essential for viewpath efficiency.

Implicit prerequisite scanning represents a large initialization cost. In practice *make* time (excluding compile time) triples when all source files must be scanned. After the initial *make*, however, incremental file scans are insignificant when compared with compiles.

Providing statefiles and base and global makefiles can degrade *make* startup performance. An easily loaded format for makefiles and statefiles can speed up load time when compared with a full makefile parse.

Finally, sending entire actions to the shell, rather than the traditional line-by-line, not only makes actions more readable (no backslash-newlines for block constructs), it also reduces make (and shell) *fork-exec* overhead. Moreover, individual actions are the natural unit of granularity for parallel execution. Because of I/O and computation interleaving, parallel support is desirable, even for uniprocessor systems. As mentioned above, the target-prerequisite directed graph clearly defines action ordering and synchronization. However, an explicit synchronization primitive must also be provided to support mutual exclusion that cannot otherwise be asserted. One method presents virtual (non-file) semaphore prerequisites. Any two target actions having the same semaphore prerequisite cannot execute concurrently. Since the *yacc* command generates output in a fixed file (`y.tab.c`), it is a prime candidate for synchronization:

```
/* yacc pattern metarule */

.yacc.semaphore : .SEMAPHORE

%.c : %.y .yacc.semaphore /* only one at a time */
      yacc $(%).y
      mv y.tab.c $(%).c
```

7 Portability

Configuration management portability runs counter to the model presented by integrated programming environments. Rather than exploit features of a particular system or compiler, a portable *make* exploits compatibilities between systems and compilers. Although not as important as the other goals mentioned above, portability has practical implications. Many projects and companies will only use standard configuration management systems during development and testing. This ensures that customers will be able to build and maintain products with minimal help. Relying on a standard *make** also lightens the customer documentation, education, and maintenance load.

Even though standard makefiles may be required for external use, projects need not be stuck with old *make* technology for internal development. A makefile generator, similar to the static prerequisite generator mentioned above, can translate internal makefiles to external form as a part of the software packaging process. This is an interesting twist, as most makefile conversion commands translate in the *old* to *new* direction. Since all *make* derived commands eventually deal with file dependencies and shell actions, *new* to *old* translators are easy to build. Assuming the recipients will not develop the shipped product, the target makefile language can even be a monolithic

* Defining a standard *make* is left as an exercise to the reader.

shell script that simply does a one-time product build and installation.

8 Experience

The ideas and features proposed here have been incorporated to a limited extent in the publicly available *make* implementations [Fowl, Hume, Palk, SM], and to a greater extent in commands based on different models [Clem, Wate]. The Bell Laboratories *nmake* command (internal version) provided the basis for the arguments in this paper, and so supports them all. Although *nmake* is not backwards compatible with *make*, it is used in hundreds of projects within AT&T. The *nmake* benefits most cited by users are:

- Makefiles are easier to maintain. Most projects report a factor of 10 makefile size reduction.
- Implicit prerequisite scanning practically eliminates weekend clobber-build cycles.
- The information distribution allows common project features to be controlled from a single point while still allowing user flexibility.
- Viewpathing is indispensable for multi-developer, multi-release projects. The view layers clearly delineate developer and release boundaries.
- State eliminates many bugs common to *make* in multi-developer and remote file system environments.
- Handles large projects (˜100 developers, ˜1000 makefiles, ˜8 viewpath levels) without coming apart at the seams.
- Multi-makefile applications often collapse down to a single makefile (System V Release 3 kernel: from ˜50 makefiles, ˜4000 lines down to one makefile, ˜500 lines).
- The same interface is presented on the popular UNIX system variants (System V, 4.[23] BSD, SunOS, Ultrix) and machine architectures (vax, 3b, sparc, i386, 68000, mips, cray, masscomp, pyramid).

nmake is also used to develop and package itself. The *nmake* source distribution contains standard makefile equivalents that are generated, using *nmake*, from the original *nmake* makefiles. Generated makefiles are then used to bootstrap the *nmake* executable. Once installed, *nmake* is run on itself to prime the statefile and implicit prerequisite scans. Other commands and libraries distributed with *nmake* are then built using *nmake*.

9 Conclusion

Enhancing *make* can strengthen its place in modern UNIX system software development environments. These changes can reduce makefile size, support multi-user development, and speed up the software configuration management cycle. Incompatible extensions can be handled by automatically generating standard makefiles for portability.

"References"

[CF] S. Cichinski and G. Fowler, *Product Administration through SABLE and nmake*, AT&T Technical Journal, **67**(4), 59-70 (July/August 1988).

[Clem] G. H. Clem, *The Odin System: An Object Manager for Extensible Software Environments*, Ph.D. thesis and Technical Report CU-CS-314-86, University of Colorado Department of Computer Science, February 1986.

[EP] B. Erickson and J. F. Pellegrin, *Build – A Software Construction Tool*, AT&T Bell Laboratories Technical Journal, **63**(6), 1049-1059 (July/August 1984).

[Feld] S. I. Feldman, *Make – A Program for Maintaining Computer Programs*, Software – Practice and Experience, **9**(4), 256-265 (April 1979).

[Fowl] G. S. Fowler, *The Fourth Generation Make*, USENIX Portland 1985 Summer Conference Proceedings, 159-174 (1985).

[Hume] A. G. Hume, *Mk: a successor to make*, USENIX Phoenix 1987 Summer Conference Proceedings, 445-457 (1987).

[KK] D. G. Korn and E. Krell, *A New Dimension for the UNIX Filesystem*, this issue.

[Palk] R. Palkovic, *SunPro: The Sun Programming Environment*, M. Hall (ed.), A Sun Technical Report, Sun Microsystems, Inc., 67-86 (1987).

[SM] R. M. Stallman and R. McGrath, *GNU Make – A Program for Directing Recompilation*, Edition 0.1 Beta, March 1989.

[SV1] *Augmented Version of Make*, UNIX System V – Release 2.0 Support Tools Guide, 3.1-3.19 (April 1984).

[Wate] R. Waters, *Automated Software Management*, Software – Practice and Experience, **19**(10), 931-955 (October 1989).

[WDK] *UNIX Needs A True Integrated Environment: CASE Closed*, M. Weiser, L. P. Deutsch, and P. B. Kessler, Xerox PARC technical report, CSL-89-4 (January 1989).