The Fourth Generation Make

Glenn Fowler
gsf@research.att.com

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

Abstract

Make is a program that is widely used to maintain and update programs and libraries on UNIX* systems. This paper introduces the *Fourth Generation Make* which embodies major semantic and syntactic enhancements to the standard *make* program. The enhancements include support for source files distributed among many directories, an efficient *shell* interface that allows concurrent execution of update commands, dynamic dependency generation, dependencies on conditional compilation symbols and a powerful new meta language for constructing default rules. A complete rewrite of the standard make code has resulted in a unified software construction program that also provides improved functionality and performance. Improved performance includes an average five to ten times decrease in makfile size and an average two to five times decrease in execution time. It is assumed that the reader is familiar with the features and operation of the standard *make*.

1 INTRODUCTION

Since it first appeared in 1976[1], *make* has changed little, and most of these changes have been syntactic in nature[2]. However, the *build* variant of *make*[3] introduced a major semantic change. *Build* has *viewpaths* that allow source files to reside in more than one directory. *Viewpaths* support the sharing of source files between many users and help eliminate the proliferation of source copies. The semantics of multiple directories, however, conflict with the flat, single directory approach of old-make (*old-make* refers to the *make* distributed with the System 5 UNIX system and *new-make* refers to the Fourth Generation *make*). *Build* must *link* or *copy* files into the current directory to comply with the single directory nature of *old-make*. Without *build*, multiple directories are usually handled by using recursive *make*s. Because of the large data area (which must be copied on *fork*) and the time consumed by the startup mechanisms (parsing builtin rules, scanning directories, etc.), recursive *make* calls are inefficient.

The Fourth Generation *make* is the result of a research effort to improve the execution and semantics of *old-make* and to update the model of *make* to be more consistent with such concepts as multiple source directories and concurrent execution of update commands.

The main features of the Fourth Generation *make* are that it:

- Efficiently supports both source and targets in multiple directories.
- Provides a more efficient interface to the *shell*.
- Decreases redundancies inherent in old makefiles.
- Refines the granularity from the *file* to the *variable* level.
- Provides mechanisms for writing portable makefiles.
- Provides a powerful metalanguage for constructing builtin rules.

---

* UNIX is a trademark of AT&T Bell Laboratories.

- Supports program configuration and generation for many UNIX system environments.
- Compiles makefiles into binary form for reduced startup time.
- Provides an abbreviated syntax to accommodate the most common uses of *make*.
- Provides optional dynamic dependency generation.
- Provides a natural interface for object library maintenance.
- Provides a natural interface to SCCS.

This paper describes the motivation and design of the new features of the Fourth Generation *make* and also discusses the advantages of using *new-make*. Some of these advantages are:

- **Decreased makefile size** – most *new* makefiles consist of one or two lines of *source* file dependencies. Conversion of the BSD and 3B2 kernel makefiles shows a factor of ten decrease in makefile size.
- **Consolidation of makefiles** – usually, recursive configurations of old makefiles can be replaced by one makefile. This occurred in the conversion of BSD and 3B2 kernel makefiles.
- **Improved consistency checking** – duplicate source and header files are reported. Additionally, missing header files are reported before compilation takes place.
- **User definable builtin rules** – the default builtin rules are just a compiled makefile binary, so there is no loss of efficiency when other default rules are used. The default rules can either be completely bypassed or augmented, depending on the input options to *make*.
- **Decreased execution time** – experiments on large makefiles (when all targets are up to date) show a factor of ten average decrease in execution time. The shell interface also eliminates one *fork* and one *exec* for each makefile command line executed. In contrast to *old-make*, which *forks* for each command line, *new-make* *forks* only once to initialize the *shell* co-process. After this initialization the shell is the only *forked* process.
- **Efficient file sharing** – software development teams can easily share single copies of source and object files.
- **Improved software organization** – the multiple directory nature of *new-make* encourages the organization of large software project files into more than one directory. Unlike *build*, *new-make* allows files to be distributed over arbitrary directory structures.

Since the full functionality of *new-make* cannot be realized using the restricted model provided by *old-make*, *new-make* does not support old makefiles. This is because the research effort of *new-make* emphasizes increased execution speed and simplicity of use over issues of backward compatibility.

The remaining sections elaborate on the features and improvements introduced in *new-make*. This paper is not a tutorial and so does not contain a detailed description of specific command arguments, options or actions. All timing estimates are based on the user and sys times of programs running on a VAX/750*.

2 MAKEFILES

Makefiles are typically at the source of software portability problems. More often than not, makefiles are run in combination with shell scripts to determine the current environment parameters. In the worst cases, the user is interactively prompted for the desired information.

A major goal of *new-make* is to encapsulate as much information as possible into a single makefile. A new shell interface and makefile preprocessor make this encapsulation possible. As a result it is possible for the same

---

* VAX is a trademark of Digital Equipment Corporation.

makefile to run on different machines and even on different versions of the UNIX system.

2.1 Preprocessing

Each makefile is preprocessed by the C preprocessor *cpp*, making the full flexibility of include files and macro definitions available to makefile users. The version of *cpp* distributed with the *new-make* also contains some changes that improve makefile portability. **#if** statements may use the following builtin tests:

exists(*file*)
: Returns 1 if *file* can be found using the **#include** search rules. *File* may optionally be enclosed in **<>**, just as in the **#include** statement. This test can be used to tailor software generation for specific UNIX variants and environments.

identifiers(*file*,*id1*,...)
: Searches *file* for the null terminated identifiers *id1*... and returns the number of identifiers found. Initial _ characters in both *id1*... and identifiers in *file* are ignored. If *file* is an archive with a symbol directory then only the symbol directory is searched. Therefore, this test can be used to determine if a function or global symbol is present in an object library.

Additional function predicates are defined by **#assert** statements in the include file **<default.h>** which is automatically included by *cpp* before the first input file is read. **#assert** *predicate*(*value*) makes assertions that can be tested in **#if** expressions. Such assertions are only recognized within **#if** expressions and do not conflict with **#define** variable expansions. Most assertions deal with the current machine environment:

system(*system-name*)
: Defines the operating system name. Example values for *system-name* are **unix** and **gcos**.

release(*system-release*)
: Defines the operating system release name. Example values for *system-release* are **apollo**, **bsd**, **research**, **sun**, **system5**, **uts**, and **venix**.

version(*release-version*)
: Defines the operating system release version. Example values for *release-version* are **4.1c** and **4.2** for **release(bsd)**, **7** and **8** for **release(research)** and **5.0** etc. for **release(system5)**.

model(*model-name*)
: Defines the hardware model name that also encompasses workstation names. Example values for *model-name* are **apollo**, **sun**, **ibm-pc** and **unix-pc**.

architecture(*architecture-name*)
: Defines the processor architecture name. Example values for *architecture-name* are **3b**, **68000**, **ibm**, **pdp11**, and **vax**.

machine(*architecture-version*)
: Defines the processor architecture version. Example values for *architecture-version* are **2**, **20** and **20s** for **architecture(3b)**, **70** etc. for **architecture(pdp11)** and **750**, **780** and **micro** for **architecture(vax)**.

2.2 Compilation

When all targets are up to date, *old-make* spends most of its time parsing the builtin rules and input makefiles, whereas *new-make* avoids this problem by compiling makefiles into binary make object files. The input makefile *x***.mk** is automatically compiled into the make object file *x***.mo** whenever the object file is out of date with the corresponding makefile.

Compilation takes place after the makefiles are read and parsed and involves writing the internal structures to a file in relocatable form. The time spent compiling is negligible compared to the parsing time and loading compiled makefiles saves an average of 2 seconds on program startup.

Whereas *old-make* must parse its builtin rules on program startup, *new-make* loads the standard builtin rules from a compiled makefile. Since the standard builtin rules are placed in a compiled makefile, substituting local builtin rules for the standard ones results in no loss of performance. Local builtin rules can be specified either on the *make* command line or in the **MAKERULES** environment variable.

2.3 Operators

New-make supports (two character) operators that may be defined in the builtin rules file. The usage syntax of these operators is exactly like the **:** dependency operator, but the semantics depend on the specific operator definitions.

Only one new operator, **::**, is provided in the standard builtin rules. Unlike the **:** dependency operator, which typically specifies *object* file dependencies, **::** may be used to specify *source* file dependencies:

```
program : p1.o p2.o p3.o
	$(CC) -o program p1.o p2.o p3.o -lm
```

can be specified as

```
program :: program.mk p1.y p2.l p3.c -lm
```

Using **::** eliminates the duplication of file name lists and also provides other important features (see **Dependency Generation** and **Common Actions** below).

3 BINDING

A basic internal operation is the *binding* of rule names to objects (e.g., files and variables) in the system. *Binding* enables files to be placed in more than one directory and also allows targets to depend on variable definitions as well as files.

3.1 Files

Rule names are bound to file names using the dependencies of the special **.SOURCE** and **.SOURCE**.*x* rules. The dependencies of these rules are directories to be scanned when searching for files. The current directory is always scanned first. Then, files with suffix *.x* are searched for in the directories specified by **.SOURCE**.*x*. Finally, the **.SOURCE** directories are searched. The left to right **.SOURCE** dependency ordering is important; *make* warns when a file is found in more than one directory, but continues with the first file found.

3.2 State Variables

A *state variable* is a variable whose modify time and definition is stored from one invocation of *make* to the next. *State variables* have two basic forms: **(***variable***)** is a makefile or *environment* variable and *file*(*variable*) is a variable corresponding to a **#define** statement in *file*. For example:

```
x.o : header.h(DEBUG) (MACHINE)
```

specifies that *x.o* depends on the definition of the variable **DEBUG** in the file *header.h* and the definition of **MACHINE** from the current makefile. If either definition changes from one invocation to the next then *x.o* will become out of date and will be regenerated.

A file is scanned for **#define** definitions only if it has been modified since the last time it was scanned. The first **#define** definition for a variable is used and all other definitions are ignored.

The state variable definitions are stored in the state file *base***.ms** where *base* is the base name of the first makefile in the argument list. The state file is implemented as a compiled make object file, resulting in little maintenance overhead. Some dependency information is also stored in the state file (see **Dependency Generation** below).

4 VARIABLE EXPANSION

The basic actions of *new-make* are controlled by the builtin rules, with most of the expressive power concentrated in variable definitions and expansions. The addition of operators and editing to variable expansions has resulted in a powerful makefile metalanguage. As a testimony to the strength of this metalanguage, most new *make* features and ideas have resulted in changes to the standard builtin rules (written as a makefile) rather than in changes to the *make* C source files.

The metalanguage produces a more complex set of builtin rules, but in turn allows simple, concise and easy to maintain user makefiles.

4.1 Assignment

There are three variable definition operators:

variable **=** *value*	*Value* is assigned to *variable* without expansion.
variable **:=** *value*	*Value* is expanded and then assigned to *variable.* This allows a variable to contain all or portions of its previous value.
variable **+=** *value*	*Value* is expanded and appended to the current value of *variable*. This allows lists to be generated in variable values.

4.2 Expansion

Each occurrence of **$(***variable***)** in a makefile is replaced by the assigned *value* of *variable*. The substitution is recursive in that *value* is checked for other variable expansions before being substituted. **$(***variable***:***operator***)** causes *value* to be edited according to the specifications in *operator* before being substituted. **:** is used to separate multiple *operators*. The general form for *operator* is *op***[=***arg***]** where *op* is a single character operator name and *arg* is an optional operator argument. The operators are applied to each space separated token in the expanded variable value. The variable name itself is expanded before the value is determined, implementing primitive variable subscripting.

4.2.1 File Components Because of the multiple directory nature of *new-make* it is important to be able to separate a file name into its individual components. File names are divided into the following five basic components:

M *machine*	All characters up to and including the last **!**. *Null* if no **!** appears. The *machine* component is supported but not used in the current implementation.
D *directory*	All characters after the last **!** up to and including the last **/**. *Null* if no **/** appears.
P *prefix*	All characters after the last **/** up to and including the first **.**. *Null* if there are less than two **.**'s or if **.** is the first character.
B *base*	All characters after the first **.** up to but not including the last **.**.
S *suffix*	All characters from the last **.** to the end. *Null* if no **.** appears.

Here is an example using file name component editing:

```
FILES = a.y dir/s.x.c bozo!.profile

$(FILES)                    ->      a.y dir/s.x.c bozo!.profile
$(FILES:B:S=.o)             ->      a.o x.o .profile.o
$(FILES:DBS)                ->      a.y dir/x.c .profile
$(FILES:M)                  ->      bozo!
```

4.2.2 Matching Space separated tokens in variable values can be matched using the shell file pattern matching characters in the **:N=***pattern***: :N!=***pattern***:** edit operators. The matching occurs before component editing takes place. Using **FILES** as an example:

```
$(FILES:N=*.c)            ->      dir/s.x.c
$(FILES:N!=*.c)           ->      a.y bozo!.profile
$(FILES:N=*.[cy]:BS=.o)   ->      a.o x.o
```

4.2.3 Substitution Tokens in variable values can be substituted using the **:C**<*del*>*old*<*del*>*new*<*del*>**:** substitute edit command. <*del*> may be any character and **C/** may be abbreviated by **/**. For example:

```
$(FILES:/ /\:/)           ->      a.y:dir/s.x.c:bozo!.profile
$(FILES:C%/%--%)          ->      a.y dir--s.x.c bozo!.profile
```

Notice that the **:** must be escaped to distinguish it from the **:** edit operator.

4.2.4 Binding Variable value tokens may also be *bound* and selected by type using the **:T=***type***:** edit operator. The types are:

A Each token that can be bound to an *archive* is expanded.

D Each token that can be bound to a *state variable* is expanded using the *state variable* definition. The expanded definitions may be used as arguments to the *cc* command.

F Each token that can be *bound* to a file is expanded using the *bound* file name.

N If *variable* has a null value then the null string is expanded, otherwise # is expanded. This can be used to specify conditional makefile input.

O Each token that is bound neither to a file nor to a *state variable* is expanded.

S Each token that can be *bound* to a *state variable* is expanded.

V If *variable* has a non-null value then the null string is expanded, otherwise # is expanded.

For example:

```
FILES = x.c (DEBUG) (TEST) libc.a
TEST =
DEBUG = 1

$(FILES:T=D)              ->      -DDEBUG
$(FILES:T=F)              ->      x.c /lib/libc.a
$(FILES:T=S)              ->      (DEBUG) (TEST)
$(TEST:T=V)               ->      #
$(FILES:T=V)              ->
```

5 RULE ATTRIBUTES

New-make uses rule attributes to control the disposition of rules and targets. For example, the **.ARCHIVE** attribute specifies that a bound rule is to be treated as an object file archive. This attribute allows the following concise archive dependency specification:

```
lib.a :: a.c b.c c.y x.s
```

Attributes also allow *new-make* to assume some system dependent maintenance responsibilities. For example, on Berkeley variants of the UNIX system, **.ARCHIVE** targets are automatically updated using *ranlib*.

Some attributes are assigned automatically, some are determined by the suffix of the current target (**.a** files are given the **.ARCHIVE** attribute) and others may be assigned explicitly in the makefile.

Another attribute **.USE** allows a single command update sequence to be shared by many targets. For example:

```
a.o : (DEBUG)
      $(CC) $(CCFLAGS) -S $(>)
      $(FIXUP) $(>:BS=.s)
      $(CC) -c $(>:BS=.s)
      $(RM) $(RMFLAGS) $(>:BS=.s)

b.o : (PROFILE)
      $(CC) $(CCFLAGS) -S $(>)
      $(FIXUP) $(>:BS=.s)
      $(CC) -c $(>:BS=.s)
      $(RM) $(RMFLAGS) $(>:BS=.s)
```

can be replaced by

```
a.o : .FIXUP (DEBUG)
b.o : .FIXUP (PROFILE)

.FIXUP : .USE
      $(CC) $(CCFLAGS) -S $(>)
      $(FIXUP) $(>:BS=.s)
      $(CC) -c $(>:BS=.s)
      $(RM) $(RMFLAGS) $(>:BS=.s)
```

A brief description of the attributes and special rules appears in the attached manual pages.

6 DEPENDENCY GENERATION

The main task of *make* is to verify that any changes made to source files are reflected in the corresponding object files. This verification, however, relies on the proper dependencies being placed in the controlling makefiles. A single omitted dependency can cause inconsistencies between the source and object files.

New-make eliminates some of these inconsistencies by dynamically generating file dependencies from a given set of source files. Such dependencies are automatically generated when the **::** dependency operator is used. The generated dependencies are stored in the state file along with the state variable definitions.

The file dependencies are determined by scanning the files for **#include** statements. A file is only scanned if it has been modified since the last time it was scanned. In the worst cases, automatic dependency generation only doubles the execution time of *make* (not including the time spent updating the targets).

#include dependencies within conditional **#if** constructs are given the **.DONTCARE** attribute that allows *make* to continue if the corresponding files cannot be found.

A minor drawback is that the files depend on *all* **#include** dependencies, even if some of the dependencies are from ‘‘compiled out’’ parts of the source files. However, for a given application the ‘‘compiled out’’ dependencies are rarely modified. A proposed alternative is to use *cpp* to produce the exact dependencies, and feasibility experiments are now underway. Initial results show that *cpp* scanning may be appropriate only after major changes to the source files, and that the basic **#include** scanning is sufficient for most applications.

In any event, even the basic dynamic **#include** file dependency generation provides a more consistent environment than statically generated dependencies using *old-make*.

7 SHELL INTERFACE

The execution of update commands has undergone major performance and semantic changes in *new-make*. All update commands are sent to a single copy of the shell, keeping the shell environment intact between command executions. This includes the effects of *cd* and shell parameter assignments.

Since only one copy of the shell is used, *new-make forks* just once to initialize the shell as a *co-process*. While commands are being executed by the shell, *new-make* continues by checking the dependencies of the next target. Thus the next update command is almost always determined by the time the current command completes.

An added advantage is that command *aliasing* and shell *functions* are preserved in update command blocks. *Old-make* uses either the *system* or *exec* call to execute each command line. In *old-make*, if a line contains shell metacharacters (**$** | ()><) then it is sent to the shell via *system*, otherwise the command is executed via a *fork* and *exec*. *Aliases* in the latter case are ignored by *old-make*.

7.1 Command Blocks

The new makefile structure supports natural shell command specification. Update command lines for a particular target are sent to the shell as a block, allowing shell **case**, **for**, **if** and **while** constructs to cross **newline** boundaries without intervening **backslash** and **semicolon** characters.

With this co-process arrangement makefiles can be viewed as *labeled* shell scripts. The labels (targets) merely determine when the corresponding command blocks are executed.

To avoid confusion between the use of **$** in *new-make* and the shell, only the **$(...)** forms are expanded by *new-make* (**$$(...)** expands to **$(...)**). **$** in any other context is passed untouched to the shell.

7.2 Communication

Pipes are used for communication between *new-make* and the shell. *Make* sends commands to the shell on one *pipe* and receives status information from the shell on a second *pipe*. The status information is organized into four message packets:

error exit-code
: Sent when a command returns a non-zero exit code.

exit
: Sent when a command block completes.

read make-command
: Sends *make-command* back to *make* to be parsed as if it originated in a makefile. This allows dynamic makefile generation, but this feature has not been used (or recommended) in any major applications.

start process-id
: This packet associates a command block with a subshell process id when concurrent execution is enabled. This allows *make* to wait for its children for proper time accounting.

It is sometimes desirable for *make* to continue with other targets even after an error has occurred. A *trap* on command error was added to the shell to handle this case in a co-process environment. In addition, since entire command blocks are sent to the shell, the **−x** execution trace flag of the shell is used to echo each individual command as it is executed. *New-make* places control commands between the command update blocks and constantly switches between the **−x** and **+x** options. So as not to clutter the execution trace with **set +x** commands, the shell suppresses the execution trace of **set +x** when **set −x** is in effect. Many thanks to Dave Korn for adding these two features to *ksh*[4]. Although *new-make* works best with KSH, it also runs with the Bourne shell[5].

7.3 Concurrent Execution

With the shell as a co-process it is a trivial matter to add concurrent command execution to *new-make*. Internally each command block is simply enclosed by { and }**&** and a *jobs* table is used to associate targets with process ids. The dependency graph specified by the input makefiles is then used to determine when *new-make* must wait for certain targets to complete.

The **–j***n* command line option is used to specify the maximum number of concurrent jobs. By default only one job is used, but if *n* is greater than 1 then each update command block is sent to a new subshell (background shell). Background shells inherit the environment of the main shell (foreground shell) and the foreground shell inherits the environment of *new-make*. It is important to note that no makefile changes are necessary to support concurrent execution.

Concurrent execution should have a major effect on multi–processor machines and on systems equipped with compiler ''black boxes.'' On such machines it would be possible for each command block to execute on different processors.

8 OPTION GENERATION

The builtin rules automatically generate the proper **–D**, **–I** and **–L** options of *cc* in the **$(CCFLAGS)** variable. The **–D** options are generated from *state variable* dependencies, the **–I** options are generated from the dependencies of the **.SOURCE.h** rule and the **–L** options are generated from the dependencies of the **.SOURCE.a** rule. *State variable* dependencies specified using the **::** operator apply to all dependencies of the corresponding target (global dependencies), otherwise the dependencies apply only to the individual target of each **:** operator.

9 COMMON ACTIONS

When the **::** operator is used several common action targets are automatically defined. The common action target *xxx* is defined as *.XXX* in the builtin rules. If *xxx* appears as a command line target and *xxx* has not been defined by the input *makefiles* then the target *.XXX* is made. The common actions are:

clean Deletes all object files corresponding to the current makefile.

clobber Executes the **clean** action and also deletes the target(s) corresponding to the current makefile.

cpio Creates a *cpio* archive of the source files listed after each **::** operator. The archive is placed in the file *main***.cpio** where *main* is the base name of the main target rule.

install Makes the main target and copies it to the directory **$(INSTALLDIR)**. By default, **$(INSTALLDIR)** is **$(ROOT|HOME)/bin** (use **$(ROOT)** if it is defined otherwise use **$(HOME)**) for executable targets and **$(ROOT|HOME)/lib** for object archive targets. The commands associated with the rule **.DOINSTALL** are used to do the copy.

lint Runs *lint* on the input source files. The proper **–D** and **–I** options are associated with each source file (see **Option Generation**). Any **.l** and **.y** source files are automatically preprocessed if necessary.

print The source files are printed by passing them through the filter **$(PR)** and listing them with **$(LP)**.

tar Creates a *tar* archive of the source files listed after each **::** operator. The archive is placed in the file *main***.tar** where *main* is the base name of the main target rule.

ucpio Same as **cpio** except that only those source files modified since the last **ucpio** are archived. If **$(UTIME)** is defined then it is taken to be a file name whose modify time is used to determine the files to be archived; only those files newer than this modify time are archived.

uprint Same as **print** except that only those source files modified since the last **uprint** are printed.

utar Same as **tar** except that only those source files modified since the last **utar** are archived.

10 SCCS INTERFACE

The SCCS[6] interface is enhanced by the addition of *prefix* rules. The prefix rule *p.* specifies how the file **x** is to be generated from the file *p.***x**. The following rules provide complete support for SCCS files:

```
.PREFIXES : s.

s. :
       $(GET) $(GETFLAGS) $(>) > $(<)
       ...
       $(UNGET) $(<)
```

The **...** separates the update blocks into *pre*−commands and *post*−commands. The *pre*−commands are executed to update the target, whereas the *post*−commands are stacked (first in first out) until the last target has been updated.

This implementation eliminates the proliferation of ~ rules found in *old-make* and *build*.

11 CONCLUSION

The Fourth Generation *make* is being used successfully by a growing community of users. The program performs well enough that users are not tempted to bypass *make* by manually issuing compile and *touch* commands.

After nearly five months of constant use the program has settled into a panic-free steady state and is ready for wider experimental distribution. The features and performance gains of this program should make it an attractive software construction tool for both individual users and large projects. The average tenfold decrease in makefile size should be of particular interest to large software project managers.

REFERENCES

1. S. I. Feldman, *Make – A Program for Maintaining Computer Programs*, Software – Practice and Experience, Vol. 9 No. 4, pp. 256-265, April 1979.

2. *Augmented Version of Make*, UNIX System V – Release 2.0 Support Tools Guide, pp. 3.1-3.19, April 1984.

3. V. B. Erickson and J. F. Pellegrin, *Build – A Software Construction Tool*, AT&T Bell Laboratories Technical Journal Vol. 63 No. 6 Part 2, pp. 1049-1059, July-August 1984.

4. D. G. Korn, *KSH – A Shell Programming Language*, USENIX Toronto 1983 Summer Conference Proceedings, pp. 191-202, 1983.

5. S. R. Bourne, *The UNIX Shell*, AT&T Bell Laboratories Technical Journal, Vol. 57 No. 6 Part 2, pp. 1971-1990, July-August 1978.

6. L. E. Bonnani and C. A. Salemi, *Source Code Control System User's Guide*, System V Programmer's Manual.

12 Appendix A: MAKEFILE EXAMPLE

```
#
# old makefile
#

DEBUG = -DDEBUG
MAX = 123

CFLAGS = -O -I$(HOME)/include $(DEBUG)

CFILES = dir/a.c b.c c.c d.c
HFILES = c.h
OFILES = a.o b.o c.o d.o

command : $(OFILES)
      $(CC) -o command $(OFILES) $(HOME)/lib/lib.a -lm

a.o : dir/a.c $(HOME)/include/a.h
      $(CC) $(CFLAGS) dir/a.c

b.o : $(HOME)/include/b.h c.h

c.o : c.c
      $(CC) $(CFLAGS) -DMAX=$(MAX) c.c

print :
      pr Makefile $(HFILES) $(CFILES) | lp

lint :
      lint $(CFLAGS) $(CFILES)



/*
 * new makefile
 * all files recompiled if DEBUG value changes
 * c.o recompiled if MAX value changes
 */

DEBUG = 1
MAX = 123

.SOURCE.h : $(HOME)/include
.SOURCE.a : $(HOME)/lib
.SOURCE : dir

command :: Makefile c.h a.c b.c c.c d.c lib.a -lm (DEBUG)

c.o : (MAX)
```

13 Appendix B: MANUAL PAGE