Mk: a successor to *make*

Andrew Hume

ABSTRACT

Mk is an efficient general tool for describing and maintaining dependencies between files or programs. *Mk* is styled on, and largely compatible with, the UNIX† tool *make*. The major advantages of *mk* over *make* are executing recipes in parallel, using pattern-matching metarules rather than suffix transformation rules, and deriving dependencies by transitive closure on all rules. *Mk* runs anywhere from 2 to 30 times faster than *make*.

This report describes *mk* by means of an evolving example. Other sections summarize the differences between *mk* and *make* and discuss the principles underlying *mk*'s design.

1. Introduction

A large fraction of computer activity consists of repeated application of tools (special or general purpose programs) to input files to produce output files. The most obvious example is programming, but other no less important examples range from simple document-processing pipelines to the generation of a circuit board or integrated circuit involving hundreds of files. Common to all these activities are file dependencies, where changing a file requires that other files be remade. *Mk* reads a dependency description (called a *mkfile*) and does the minimal work necessary to bring a target file up to date.

Mk owes much to *make*, written by Stu Feldman, which has been doing a similar job on UNIX systems since 1976. The version of *make* referred to throughout this report is Feldman's research version distributed with Research UNIX, Eighth Edition and is substantially more advanced than the versions found found in System V or Berkeley UNIX systems.

The next section is rather long. It follows the gradual development of a somewhat complicated mkfile describing how to build a C program. It is followed by a section on fancy uses of *mk*. The fourth section summarizes the differences between *mk* and *make* and includes a comparison of execution times. The fifth section highlights the principles underlying *mk*. The appendix documents the predefined or builtin variables and rules for *mk*.

2. An Extended Example

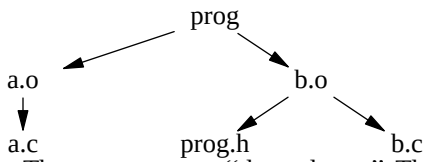
This section describes *mk* in the context of building C programs. This is for the reader's comfort; *mk* knows nothing special about C programs. The example starts off small and simple and is extended throughout the section. Sometimes, *mk*'s behavior is best demonstrated by excerpts from a terminal session. These will be shown as

```
$ date
Fri Feb 20 20:06:03 EST 1987
$
```

where **$** is the prompt for the next command. Comments will be shown in *italics*.

† UNIX is a trademark of Bell Laboratories.

Initially, our program is called **prog** and is made from **a.o** and **b.o**, which are made by compiling **a.c** and **b.c** respectively. In addition, **b.c** includes a header file **prog.h**. We represent these relationships pictorially below



The arrow means "depends on." Thus, **prog** depends on **a.o** and **b.o** and if **a.o** or **b.o** is modified, then **prog** needs to be rebuilt. Similarly, **a.o** depends on **a.c** and **b.o** depends on **b.c** and **prog.h**.

The textual description of how **prog** is built is kept in a *mkfile* and looks like

```
prog:   a.o b.o
        cc -o prog a.o b.o
a.o:    a.c
        cc -c a.c
b.o:    b.c prog.h
        cc -c b.c
```

The mkfile is a sequence of *rules*. Each rule defines a target (say **prog**) that depends on some prerequisites (**a.o** and **b.o**) and the commands (a shell script called the *recipe*) to bring the target up to date. *Mk* takes this description from a file named **mkfile** and builds the given targets. If no targets are given on the command line, the first target in the mkfile is built. For example, if we start with just the source files in our directory, *mk* creates **prog** by compiling **a.c** and **b.c**.

```
$ mk
cc -c a.c
cc -c b.c
cc -o prog a.o b.o
$
```

Executing *mk* again does nothing, as **prog** is now up to date.

```
$ mk
mk: 'prog' is up to date
$
```

If we change a source file, *mk* rebuilds only the files that are out of date:

```
modify a.c
$ mk
cc -c a.c
cc -o prog a.o b.o
$
```

Mk will explain why it is rebuilding a file if we use the **-e** option. For example,

```
modify prog.h
$ mk -e
b.o(540869437) < prog.h(540869535)
cc -c b.c
prog(540869493) < b.o(540869546)
cc -o prog a.o b.o
$
```

Thus, **b.o** was out of date with respect to **prog.h**. After **b.o** was remade, **prog** was found to be out of date with respect to **b.o** and was then rebuilt. The numbers are the actual time stamps of the files: the values are not as important as the difference between them. A time stamp of zero indicates a non-existent file.

Variables

Suppose we now need to compile the source files with the **-g** flag so that we can use the debugger. We can of course simply edit each rule to change **cc** into **cc -g**:

```
prog:   a.o b.o
        cc -g -o prog a.o b.o
a.o:    a.c
        cc -g -c a.c
b.o:    b.c prog.h
        cc -g -c b.c
```

A better solution is to use a *variable*. A *mk* variable has a similar form and use to a shell variable. A suitable (mnemonic) name is **CFLAGS**. The new mkfile looks like this:

```
CFLAGS=-g
prog:   a.o b.o
        cc $CFLAGS -o prog a.o b.o
a.o:    a.c
        cc $CFLAGS -c a.c
b.o:    b.c prog.h
        cc $CFLAGS -c b.c
```

Now, if we want to profile **prog** (which means compiling everything with the **-p** option), we need only change the first line to

```
CFLAGS=-g -p
```

and recompile all the object files. The easiest way to recompile everything is with **mk -a** which says to always make every target regardless of time stamps.

Some variables are supplied by *mk* for use by the recipe. One is **prereq** whose value is all the prerequisites for this rule. We can rewrite the first rule like this:

```
prog:   a.o b.o
        cc $CFLAGS -o prog $prereq
```

This guarantees that the lists of object files (the prerequisite line and the cc line) are the same. It is now easy to incorporate a new object file **c.o** by adding the new name just once:

```
CFLAGS=-g -p
prog:   a.o b.o c.o
        cc $CFLAGS -o prog $prereq
a.o:    a.c
        cc $CFLAGS -c a.c
b.o:    b.c prog.h
        cc $CFLAGS -c b.c
c.o:    c.c prog.h
        cc $CFLAGS -c c.c
```

Metarules

The preceding rules for the **.o** files are very similar. *Mk* supports *metarules*, that is, rules that apply to a class of targets, rather than just one specific target. The class of targets is defined by pattern matching, with the symbol **%** (called the stem) equivalent to the regular expression **.***. For example, the normal rule for compiling C source files is

```
%.o:    %.c
        $CC $CFLAGS -c $stem.c
```

The variable **stem** in the recipe is the string matched by the **%**. The **CC** variable is good planning; a different compiler can be used very easily. Using this metarule, our mkfile becomes shorter:

```
CC=cc
CFLAGS=-g -p
prog:   a.o b.o c.o
        $CC $CFLAGS -o prog $prereq
b.o:    prog.h
c.o:    prog.h
%.o:    %.c
        $CC $CFLAGS -c $stem.c
```

Notice that the prerequisites for a target can be spread across many rules. Two rules apply to **b.o**, the specific rule with **prog.h** and the metarule for **.o**'s. Only one of the rules should have a recipe. If there is more than one recipe, *mk* complains that the way to make the target is ambiguous.

The **%** can appear anywhere in the target or prerequisite, not just at the beginning.

Mk has some predefined variables and rules listed in Appendix 1. Because our rule for **%.o** and the value for **CC** are the same as the predefined rules and variables, we can omit them for a shorter mkfile:

```
CFLAGS=-g -p
prog:   a.o b.o c.o
        $CC $CFLAGS -o prog $prereq
b.o:    prog.h
c.o:    prog.h
```

Any non-metarule takes precedence over a metarule. Thus, metarules for generating **.o**'s (say) do not conflict with any rule for generating a specific **.o**.

Rules with no prerequisites

Rules need not actually build their targets. Some rules are simply shell scripts embedded in the mkfile for convenience. For example, most mkfiles have the target **clean**:

```
clean:
        rm -f *.o prog core
```

Note that **clean** is intended as a label, not a file. Unfortunately, if a file named **clean** exists, the recipe will not be executed, since **clean** is up to date (because no prerequisite has caused it to be out of date). We want to avoid any such inadvertent interactions with the file system. *Mk* allows a label to have an attribute of *virtual*, which means that it is distinct from a file of the same name. Targets can be marked as virtual by appending a **V:** to the colon separator between targets and prerequisites:

```
clean:V:
        rm -f *.o prog core
```

Other attributes are described below.

Rules with multiple targets

The rules relating **b.o** and **c.o** to **prog.h** can be combined into one rule with two targets.

```
CFLAGS=-g -p
prog:   a.o b.o c.o
        $CC $CFLAGS -o prog $prereq
b.o c.o:  prog.h
clean:V:
        rm -f *.o prog core
```

If a rule with multiple targets has no recipe, it is simply a shorthand notation for all the simple rules with one target. A rule with multiple targets and a recipe has subtle implications described below. To motivate the subtleties, we digress to describe the *yacc* parser generator.

Yacc takes a file describing a grammar and produces the source for a C routine that will parse input according to the given grammar. The source is put in the file **y.tab.c**. *Yacc* also produces a header file called **y.tab.h** that links the parser to a lexical analyzer. The grammar file also contains semantic action code. Typically, changes to the grammar file do not change the header **y.tab.h**, but only the semantic routines.

Let us add a grammar and a lexical analyzer to **prog***:

```
prog:       a.o b.o c.o y.tab.o lex.o
            $CC $CFLAGS -o prog $prereq
b.o c.o:    prog.h
lex.o:      y.tab.h
y.tab.c y.tab.h:      gram.y
            yacc -d gram.y
```

The grammar is kept in **gram.y** (the conventional suffix for *yacc* input is **.y**). The **-d** option to *yacc* produces **y.tab.h**. Unfortunately, this mkfile does too much work in the normal case. Every time the grammar file is changed, a new **y.tab.h** is made and thus **lex.o** will always be out of date even though the contents of **y.tab.h** may not have been changed. The best solution maintains another header file (say **x.tab.h**) that only changes when necessary, that is, when the contents of **y.tab.h** actually change. The new mkfile is

```
prog:       a.o b.o c.o y.tab.o lex.o
            $CC $CFLAGS -o prog $prereq
b.o c.o:    prog.h
lex.o:      x.tab.h
x.tab.h:    y.tab.h
            cmp -s x.tab.h y.tab.h || cp y.tab.h x.tab.h
y.tab.c y.tab.h:      gram.y
            yacc -d gram.y
```

The recipe for **x.tab.h** is a conditional shell construct; if the command **cmp -s x.tab.h y.tab.h** returns with an error (the files are different), then execute the command **cp y.tab.h x.tab.h** to copy **y.tab.h** onto **x.tab.h**. In the case where **y.tab.h** doesn't change, the action is straightforward:

```
$ mk -e
y.tab.c(541051073) < gram.y(541051092)
y.tab.h(541051072) < gram.y(541051092)
yacc -d gram.y
y.tab.o(541051082) < y.tab.c(541051100)
cc  -c y.tab.c
x.tab.h(541042236) < y.tab.h(541051099)
cmp -s x.tab.h y.tab.h || cp y.tab.h x.tab.h
cp not done
prog(541051087) < y.tab.o(541051109)
cc  -o prog a.o b.o c.o y.tab.o lex.o
$
```

If we now change the grammar so that the header file does change:

*Some unimportant detail has been removed from the mkfile.

```
$ mk -e
y.tab.c(541051100) < gram.y(541051148)
y.tab.h(541051099) < gram.y(541051148)
yacc -d gram.y
y.tab.o(541051109) < y.tab.c(541051155)
cc  -c y.tab.c
x.tab.h(541042236) < y.tab.h(541051154)
cmp -s x.tab.h y.tab.h || cp y.tab.h x.tab.h
cp done; x.tab.h updated
lex.o(541042267) < x.tab.h(541051165)
cc  -c lex.c
prog(541051114) < y.tab.o(541051163)
prog(541051114) < lex.o(541051169)
cc  -o prog a.o b.o c.o y.tab.o lex.o
$
```

The subtleties are twofold. The first is that the time stamps for files are only examined when the file is initially referenced or when it is the target of a rule. If **y.tab.h** had not been a target for the *yacc* rule, then *mk* would assume that **y.tab.h** had not been updated. The second subtlety is that the rule for **x.tab.h** need not change **x.tab.h**. If it does not, then **lex.o** need not be recompiled.

Aggregates

Some of the things we would like to maintain with *mk* are actually collections or *aggregates* of entities, such as UNIX object libraries (archives maintained by *ar*). Other (unsupported as yet) examples are *cpio* and SCCS files. The type of aggregate is determined by the file's ''magic number.'' Each type has support code within *mk* to get the time stamp of a member and to ''touch'' (see below) a member. The notation **a(m)** refers to member **m** of aggregate **a**. For example, consider an archive **lib.a** made up of **a.o**, **b.o**, and **c.o**. The mkfile looks like

```
lib.a:N:   lib.a(a.o) lib.a(b.o) lib.a(c.o)
lib.a(%.o):          %.o
        ar r lib.a $stem.o
```

As each new **.o** file is generated, it is put into **lib.a**. This is straightforward and correct but inefficient: an **ar** command is executed for every out of date object file. A better way is to generate all the **.o** files and then do the **ar**. The new mkfile relies on a shell script called *membername*:

```
lib.a:      lib.a(a.o) lib.a(b.o) lib.a(c.o)
        ar r lib.a ‘membername $newprereq‘
lib.a(%.o):N:       %.o
```

N attribute stops *mk* from complaining that there is no recipe to execute in order to build a target. In general, this would be an error but in this case, we update the target in another recipe. *Membername* takes aggregate notation and extracts the member names. For example,

```
$ membername ’lib.a(a.o)’ ’lib.a(b.o)’ ’lib.a(c.o)’
a.o b.o c.o
$
```

The quotes are to stop the shell from interpreting the **()**. We use the variable **newprereq** (supplied by *mk*) because we only need to replace the object files that have changed.

Parallel processing

Mk executes recipes by continually traversing the dependency graph looking for targets that can be made. For example, in our mkfile:

```
prog:       a.o b.o c.o y.tab.o lex.o
            $CC $CFLAGS -o prog $prereq
b.o c.o:    prog.h
lex.o:      x.tab.h
x.tab.h:    y.tab.h
            cmp -s x.tab.h y.tab.h || cp y.tab.h x.tab.h
y.tab.c y.tab.h:    gram.y
            yacc -d gram.y
```

the target **a.o** can be made immediately, while the target **y.tab.o** has to wait for **y.tab.c** to be made. When *mk* finds a recipe it can execute, it puts the recipe on a queue. When the recipe terminates, *mk* updates the dependency graph. The number of recipes executing simultaneously is the value of the variable **NPROC**, which is initially one. On multi-processor machines, *mk* goes faster with higher values; most mkfiles on our 12 processor machine have **NPROC** between 6 and 10. In most situations, increasing **NPROC** beyond a certain limit gains almost nothing. The other way to speed up parallel builds is to ensure that as many recipes as possible are executing; that is, order the sub-targets such that the slowest are done first. While *mk* gives no guarantees about the order of builds, generally prerequisites are built in left-to-right order as in the mkfile.

The **-u** (utilization) option measures how many seconds (real time) are spent with so many recipes executing. For example, building **prog** with three simultaneous recipes yields

```
0: 1
1: 4
2: 7
3: 10
```

This means that the entire run took 22 seconds real time; 10 seconds with three recipes running, 7 with two and 4 with one. The time with zero recipes executing corresponds to *mk* reading the mkfile and building the dependency graph.

Parallel execution implies that recipes should not interact unnecessarily. For example, the first version of the library mkfile should not be run in parallel as simultaneous *ar* 's on the same archive interfere*. The second version can be run in parallel because only one *ar* is done, after all the object files are made.

Missing intermediates

In all the examples we have seen so far, *mk* has made all the targets ''between'' the file that changed and the main target. This is not always done. Any non-existent intermediate target (a target other than the root target with prerequisites) is treated specially. If pretending it existed with the time stamp of its most recent prerequisite would make all targets that depended on it be up to date, then it is not made. For example, in our mkfile:

```
$ mk -e
mk: 'prog' is up to date
remove a.o
$ mk -e
pretending a.o has time 540869454
mk: 'prog' is up to date
```

The intuition is that if we use the mkfile to build the targets, then removing the intermediates causes no harm. Of course, if we actually need the missing intermediates, *mk* builds them.

*Arguably, *mk* might protect against simultaneous updates of an aggregate but that is currently infeasible because it implies understanding what the recipe does.

change b.c
$ mk -e
pretending a.o has time 540869454
b.o(540869546) < b.c(541350226)
cc -c b.c
unpretending a.o because of prog because of b.o
a.o(0) < a.c(540869454)
cc -c a.c
prog(541104056) < a.o(541350255)
prog(541104056) < b.o(541350244)
cc -o prog a.o b.o c.o y.tab.o lex.o
$

The action is not too hard to follow: first *mk* sees that **a.o** is missing and pretends it is there. Then *mk* notices **b.o** is out of date and needs to be rebuilt. When **b.o** is finally built, it causes **prog** to become out of date and therefore *mk* no longer can pretend that **a.o** is up to date. It then builds **a.o** and then **prog**.

The major advantage of missing intermediates is avoiding multiple copies of files. For example, in our mkfile to maintain a library, we keep two copies of every object file. By using the notion of missing intermediates, we can keep one copy — the copy we need in the archive. To do so, simply remove the object files after they have been archived:

```
lib.a:      lib.a(a.o) lib.a(b.o) lib.a(c.o)
            names=‘membername $newprereq‘
            ar r lib.a $names && rm $names
lib.a(%.o):         %.o
```

We store the object files' names in the variable **names** to avoid executing *membername* twice. The **&&** is another conditional shell construct; we remove the files only if the archive command succeeds.

The special treatment of missing intermediates is suppressed by the **-i** option of *mk*.

Administrative

Mk provides an easy way to bring a target up to date without actually doing any work. For example, if we change **prog.h** in such a way that **b.o** or **c.o** won't change (such as adding a comment), we don't want to recompile the files. Instead, we can ask *mk* to modify the files' time stamps.

add something to prog.h
$ mk -t
touch(b.o)
touch(c.o)
touch(prog)
$

Mk lists the files it modified. This is a dangerous feature; use it carefully and sparingly. Virtual targets are not affected because *touch*ing only changes files.

Mk can also tell us what it would do without actually doing it. The option **-n** causes recipes to be printed rather than executed. There are two main problems. *Mk* assumes that every recipe will update all its targets. Normally this is true, but for our mkfile, **mk -n** would erroneously indicate that **lex.o** will always be remade. Thus, unnecessary work may be indicated. The second problem is that *mk* expands recognizable references to shell variables. It does this without parsing the shell script and can make mistakes with constructs like *for* loops. For example, with the mkfile (the **Q** attribute suppresses the normal recipe echo)

```
i=a b c
all:Q:
        for i in x y z
        do
                echo $i
        done
```

the difference between **mk** and **mk -n** is:

```
$ mk -n
for i in x y z
do
        echo a b c
done
$ mk
x
y
z
$
```

This latter problem applies to the normal recipe echo as well.

Sometimes we would like to know what *mk* would do if some files were changed. The **-w***files*,... option supports this ‘‘what if’’ query by setting the time stamps internally for the named files to the current time. With our mkfile for **prog**, we can ask what would happen if we changed **prog.h**:

```
$ mk -n -wprog.h
cc  -c b.c
cc  -c c.c
cc  -o prog a.o b.o c.o y.tab.o lex.o
$
```

The advantage of **-w** is that neither the files nor their time stamps are changed. Of course, **-w** can be used without using **-n**. For example, to force *mk* to remake **b.o** we can say

```
$ mk -wb.c b.o
cc  -c b.c
$
```

Quoting

The quoting rules for assignment lines and rule header lines are intended to be the same as for *sh* (1) (the Bourne shell). As these rules are nowhere described clearly, we describe *mk*’s quoting rules below. The term *quoting a character* means making that character stand for itself, rather than any special, or meta, meaning. For example, **$a** stands for the value of the variable **a**, whereas **\$a** (the **$** is now quoted) stands for the two characters **$** and **a**.

Input is parsed until a newline without a preceding \ is seen. If during parsing a backquote **‘** is seen, input is collected until another backquote is seen. During this collection, \ quotes every character except **\n** which is deleted. The collected input is given as standard input to the shell and the standard output replaces the collected input and the two backquotes. After all the backquotes are processed, the resulting text is scanned for single quotes, double quotes and variable expansions. Text between single quotes is quoted. Text between double quotes is quoted after variable expansion is done and \ only quotes the characters **"’$**\.

The text is then broken into parts separated by unquoted white space and any part containing **[*?** as unquoted characters is then expanded as filenames as per *sh* (1).

More on metarules

There are actually two kinds of metarules; we have looked only at the kind that uses **%** to match arbitrary strings. The second kind uses full regular expressions as supported in *egrep* (1). The expression may include sub-expressions enclosed in **\(\)**; the values of the sub-expressions can be used in the prerequisites and the recipe. For example, consider the problem of making object files in sub-directories. That is, we wish to make **dir/a.o** from **dir/a.c**. The C compiler only generates object files in the current directory, so we need to break the target into two parts:

```
'(.*)/([^/]*)\.o':R:   '\1/\2.c'
        cd $stem1; $CC $CFLAGS -c $stem2.c
```

The **R** attribute for the rule means interpret the target(s) as regular expression(s). The different ways of referring to the sub-expressions (**$stem1** inside the recipe and **\1** on the rule header line) are regrettably a consequence of not processing the recipe. A warning: regular expression metarules are significantly slower than **%** metarules.

Regardless of which kind of metarule you use, certain metarules can lead to infinite dependency graphs. For example, the metarule

```
%:      %.z
        unpack $stem.z
```

gives this dependency graph

x → x.z → x.z.z → x.z.z.z → ...

The problem arises any time a metarule has a prerequisite that can be a target of the same rule. *Mk* handles this problem by restricting the number of times a metarule is used in generating prerequisites to the value of the variable **NREP**. This value is normally one; if set to 3, the dependency graph for **x** in our example is

x → x.z → x.z.z → x.z.z.z

Thus, setting **NREP** to greater than one is necessary if we have files that have been packed repeatedly.

3. Getting Fancy

namelists from a sibngle list

diatribe against nmake: cflags as a file.

(where do i talka bout environ var precedence?)

general P stuff; rewrite yacc rule

4. Differences between *make* and *mk*

The qualitative differences between *mk* and *make* can be summarized as

- *Make* builds targets when it needs them, allowing systematic use of side effects. *Mk* constructs the entire dependency graph before building any target.
- *Make* supports suffix rules and **%** metarules. *Mk* supports **%** and regular expression metarules.
- *Mk* performs transitive closure on metarules, *make* does not.
- *Make* supports cyclic dependencies, *mk* does not.
- *Make*'s recipes are collections of one-line shell commands, executed a line at a time. Variable values are passed by editing the recipe text before passing it through to the shell. *Mk*'s are simply shell scripts executed as one unit. Variable values are passed through environment variables.
- *Make* supports parallel execution of single line recipes when building the prerequisites for specified targets. *Mk* supports parallel execution of all recipes.
- *Make* uses special targets (beginning with a **.**) to indicate special processing. *Mk* uses attributes indicated by qualifiers after the **:** separator in a rule definition.

- *Mk* allows the standard output of a recipe to be read as an additional mkfile while *mk* is running. This allows a mkfile to configure itself at run time.
- *Mk* supports *virtual* targets which exist only within an execution of *mk* and are independent of the underlying file system.
- *Mk* supports a general mechanism for deciding whether a file is out of date as well as the normal method of comparing file modification times.

In most situations, mkfiles and makefiles (the input for *make*) will have only minor syntactic differences. In practice, mkfiles often are significantly bigger because of embedded shell scripts or to make the most of underlying parallel hardware.

The most striking difference between *mk* and *make* is in speed of execution. There are three main factors involved. *Make* uses a linear list to access variables and rules; *mk* uses a hash table. *Mk* and *make* use time stamps in slightly different ways; *make* often has to measure a file's time stamp unnecessarily. If there are metarules, *mk* will typically create a much larger dependency graph than *make*. The graph gets pruned but at the cost of testing (for existence) a large number of files. In the examples given below, execution times are given (in seconds) as a sum of user time (a measure of how efficiently the dependency graph is built and executed) and system time (a measure of how many time stamps are measured). The times do not include times for recipe executions.

For mkfiles with no metarules, *mk* is always faster than *make* because of better accessing algorithms. For example, the mkfile to compile the operating system describes 83 object files. *Make* takes 19.8u+3.6s, *mk* takes 6.6u+3.6s. *Mk* is faster by a factor of 3 (user time) and 2.3 (user+sys).

For more normal mkfiles (that use the builtin metarules), *make* is somewhat faster than *mk* until about a dozen prerequisites are involved. *Mk* is much better for larger mkfiles. In most cases, *mk*'s performance can be improved by only using necessary metarules. For example, for a program made from 61 object files all compiled from **.c** files, we give the times for a normal mkfile and a mkfile that has only one metarule (generating **%.o** from **%.c**).

Command	Run Time	Relative Speed (user)	Relative Speed (user+sys)
make	12.0u+9.7s	1	1
mk (all metarules)	5.1u+4.0s	2.3	2.4
mk (one metarule)	3.9u+2.9s	3	3.2

Mk handles aggregates efficiently. The main C library has 242 members. *Make* takes 47.7u+10.9s, *mk* takes 6.3u+12.5s. *Mk* is faster by a factor of 7.6 (user time) and 3.1 (user+sys).

The final example comes from Ted Kowalski at AT&T Bell Laboratories. The mkfile is about 20,000 characters and describes an experimental workstation environment built from 238 **.c** files, 59 **.h** files, 7 **.y** files and 7 **.l** files. The mkfile makes heavy use of variables. *Make* takes 278.8u+16.2s, *mk* takes 8.4u+10.5s. *Mk* is faster by a factor of 33 (user time) and 15.6 (user+sys).

Despite the marked speed advantage of *mk* over *make*, the main reason users in our computing community use *mk* is its functionality, in particular, transitive closure on metarules, parallel execution of recipes, and the regular expression metarules.

Conversion from *make* to *mk*

Conversion of makefiles into mkfiles comes in two parts. The first is a mechanical process of syntax conversion (such as changing variable references) handled by the *sed*(1) script *mkconv*. It produces a mkfile on its standard output. For most makefiles, this is all that needs to be done.

The second kind of changes that need to be made have to be done by hand. They involve the use of side-effects by *make*, such as the normal way *yacc* grammars are handled. The proper way to handle these grammars is described above; in other cases, the general rule is to tell the truth about dependencies and let the dynamic time measuring prevent unnecessary work. *Mk* has much support for the debugging for these cases, particularly where the makefile is complex or subtle. The most useful options are **-dg** (to find out the exact dependency graph), **-e** (to explain why *mk* thinks something is out of date), and **-n** and **-w** (to conduct

what if? experiments).

Availability of *mk*

There are three sources for *mk* depending on who wants it. AT&T Bell Laboratories employees can get it from TOAD. Commercial UNIX licensees can obtain *mk* from the AT&T Toolchest (1-800-828-UNIX to talk to a person; 1-201-522-6900, login **guest**, to browse and talk to a computer). Educational (and Administrative) UNIX licensees can get an electronic or magnetic tape distribution from

Judith L. Macor
Computing Information Service
AT&T Bell Laboratories
600 Mountain Avenue
Murray Hill, NJ 07974

5. The Principles

Mk's semantics and syntax were designed according to a few general principles or guidelines.

Use existing syntax and notions. The syntax of mkfiles is almost exactly the same as a makefile (used by *make*). (The only syntactic change for rules is the attribute marking.) *Mk*'s variables are exactly the same as shell variables. Recipes are written in *sh*(1), not a special purpose language. The regular expression syntax and semantics were adopted from existing tools (such as *egrep* and *ed*), trading some awkwardness for familiarity.

Generalize features. *Make*'s metarules (already a generalization of the early *make* suffix rules) were extended to full regular expressions. *Mk* performs the transitive closure on the target-prerequisite relations defined by all rules, including metarules. The primitive form of parallel processing supported by *make* has been generalized to allow parallel execution of any recipe. By constructing the entire dependency graph before executing any recipes, *mk* maximizes the benefits from parallel processing.

Removing special cases. *Make*'s variables and recipes were so close to being shell variables and scripts that the differences were removed in *mk*. Making recipes shell scripts had the further advantage that *mk* does not have to parse or process the recipes. The use of special target and prerequisite names (beginning with a dot) to indicate special actions has been dropped in favor of a more explicit notion of target *attributes*.

Mk is a general purpose tool. Recent versions of *make* (such as *nmake*) focus on the issues connected with building software and generally contain much builtin knowledge about C programming. *Mk*, on the other hand, is a tool for maintaining file dependencies, whether they be programs or circuit board descriptions. It offers general purpose and powerful mechanism for all users, not just help for programmers.

6. Appendix

The following variable definitions are made before processing the environment or any mkfiles.

```
AS=as
CC=cc
CFLAGS=
FC=f77
FFLAGS=
LDFLAGS=
LEX=lex
LFLAGS=
NPROC=1
NREP=1
YACC=yacc
YFLAGS=
```

The builtin rules are

```
%.o:    %.c
        $CC $CFLAGS -c $stem.c
%.o:    %.s
        $AS -o $stem.o $stem.s
%.o:    %.f
        $FC $FFLAGS -c $stem.c
%.o:    %.y
        $YACC $YFLAGS -o $stem.c $stem.y &&
        $CC $CFLAGS -c $stem.c && rm $stem.c
%.o:    %.l
        $LEX $LFLAGS -t $stem.l > $stem.c &&
        $CC $CFLAGS -c $stem.c && rm $stem.c
```

The environment for the recipe's shell is augmented by these variables:

alltarget all the targets for this rule.

newprereq the prerequisites that are more recent than the target.

nproc this is the process slot for this recipe. It is a number between zero and **$NPROC-1** inclusive. It is useful for parallel execution on a single CPU machine on a network.

pid the process id for the *mk* invoking this script. This is useful for communicating with other rules.

prereq all the prerequisites for this target. This may include prerequisites from several rules.

stem,... the value of **%** in a metarule. It is null for a non-metarule. The value of the *n* th subexpression in a regular expression metarule is put in the variable **stem***n*, for **n<10**. It is null otherwise.

target the targets being built for this rule.