SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 16(9), 809–820 (SEPTEMBER 1986)

Side-effects in Automatic File Updating

WEBB MILLER AND EUGENE W. MYERS
Department of Computer Science, The University of Arizona, Tucson, AZ 85721, U.S.A.

SUMMARY

File update programs, such as the UNIX make command are invaluable, but enigmatic. They are difficult to write because the repercussions of an algorithmic variation can be elusive. Even experienced users often specify file relationships incorrectly, and are occasionally suprised by the tool's behaviour. This paper develops a rigorous conceptual model and employs it to answer some of the questions facing writers and users of file update tools. Coupled with empirical data about makefiles, these results motivate a conceptually simple update program.

KEY WORDS File updating Program maintenance

INTRODUCTION

File update tools[1, 2] help manage the parts of a program. After some of the source files have been changed, such systems automatically generate the proper sequence of commands to derive the executable program. For example with the UNIX make utility the user specifies dependency relations and updating commands in a *makefile*, such as

```
prog:     file1.o file2.o
          cc file1.o file2.o-o prog
file1.o:  file1.c
          cc-c file1.c
file2.o:  file2.c
          cc-c file2.c
```

The first line declares that prog depends on file1.o and file2.o, and the second line gives the command for regenerating prog. The next two lines list the dependency for file1.o (file1.c) and its updating command. The last two lines give the analogous information for file2.o. make reads the *makefile* and executes the updating commands for out-of-date files.

Such tools have applications beyond compiling programs. One of the simplest and most popular uses is to subsume an arbitrary command file. A set of commands can be given a name that is not an existing file, e.g. commands to remove unneeded files might be called cleanup. If the update rule executes a file's updating commands

0038–0644/86/090809–12$06.00

Received 26 June 1985
Revised 26 November 1985 and
24 February 1986

whenever the file does not exist, the command:

```
update cleanup
```

generates the file-removal commands. Thus the 'updating commands' associated with a file name need not actually update the file, but may instead alter other files.

This flexibility of file update systems comes at a price. Even experienced users often specify file relationships incorrectly[3] and are occasionally suprised by the tool's behaviour. Moreover, seemingly minor variations in the tool's update rule can have subtle ramifications, as explored below. However, this phenomenon appears only with 'on-line' updating, i.e. when execution of the updating commands is interleaved with execution of the file update software. It is shown that off-line updating is insensitive to certain alterations in the update rule.

Users of file update systems often alternate between off-line and on-line updating. The system is first asked to list the commands that it deems proper. If the user agrees with this decision, the system is invoked a second time to execute the commands. It is not uncommon for the tool to then execute a command sequence different from the one it listed. The source of the problem is the tool's generality, i.e. that the commands associated with a file may instead alter other files.

An experiment was conducted to determine the frequency with which these algorithmic variations affected the practice of file updating. The experiment was conducted by studying *makefiles*. One conclusion is that varying the update algorithm has little effect in practice, so a simple algorithm can be chosen.

In most applications, informal reasoning about the update process is adequate. However, a more formal approach was needed for writing a make-like utility[2] because 'obvious' statements about program behaviour routinely proved to be false. The approach developed for reasoning about subtle tool behaviour is sometimes helpful for tool users. In addition to describing the approach, this paper provides a precise statement of make's update algorithm that exposes previously undocumented capabilities and implementation details.

THE MODEL

Consider a collection of related files, perhaps the source and binary files of a program. Associated with each file f are an ordered list of zero or more *precursors* of f and an ordered list of zero or more commands. Intuitively, p is a precursor of f if the contents of f are a function of the contents of p, and f's commands regenerate the contents of f. However, users need not adhere to either of these intuitive principles. Bringing a file up to date is defined recursively as follows. First, bring all of the file's precursors up to date. If the file is now older than one or more of its precursors, or if it does not yet exist, then execute the list of commands associated with the file.

The following example, adapted from Reference 4, illustrates these ideas, as well as motivating subsequent discussions. An executable file named hoc is constructed by linking the four object files hoc.o, code.o, init.o and symbol.o. These four files are the precursors of hoc, and the commands associated with hoc are

```
link hoc.o code.o init.o symbol.o; name the executable file hoc
```

Each of code.o, init.o and symbol.o has its source file, distinguished by a .c suffix, as one of its precursors. A file x.h of macro definitions is included in each of those source files with a preprocessor, so x.h is a precursor of each of the three object files. (There is a natural tendency to say that the source files depend on x.h. However, it is the object files that depend on x.h, because changing x.h requires regenerating the object files, not the source files.) Each of the object files has a single associated command that compiles the source precursor. The only precursor of x.h is y.h, and the command for generating x.h is

```
if x.h ≠ y.h, then copy y.h to x.h
```

hoc.o has just one precursor, hoc.y; a sequence of several commands is needed to generate hoc.o from hoc.y. Dependencies among files can be modelled as a *dependency graph*, whose nodes are file names, as in Figure 1. (The distinction between solid and dashed edges is explained below.)

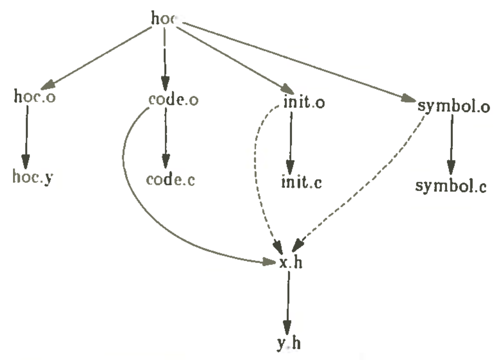


Figure 1. Dependency graph for the hoc *example*

The most interesting features of this example center around the commands associated with hoc.o and x.h. As a side-effect, hoc.o's commands always regenerate y.h. However, on most occasions, the contents of y.h are identical to its previous contents, though the modification time of y.h has changed. The point of the command associated with x.h is that the modification time of x.h will change only when its contents change. Regeneration of hoc.o from hoc.y triggers recompilation of code.c, init.c and symbol.c only if x.h changes.

The recursive update process can be performed by Algorithm 1. It uses a node-marking strategy to avoid both multiple updates for files that are the precursor of several files and infinite looping on dependency graphs containing cycles. The node marks do not affect the actual files, but instead are recorded in internal records about the file's status. The algorithm assumes that all marks are initialized to 'neither active nor processed'. The algorithm requires a function modtime that returns the last-modification time of a file. If the file does not exist, modtime returns 0.

update performs a depth-first search[5] of the dependency graph. A node is marked active when first encountered and marked processed when the search backtracks from the node. Because of the node-marking strategy, the algorithm detects common precursors, i.e. files that are the precursor of several files. For example, in Figure 1, the edges init.o→x.h and symbol.o→x.h are not traversed by the search. Such non-traversed edges in the dependency graph are depicted by dashed edges. The remaining edges, i.e. those corresponding to update calls, form the *depth-first search tree*. This separation of dependency edges into solid and dashed edges depends on both the ordering of the

```
update(file)
{
     mark file as active
     for each of file's precursors (in order)
          if the precursor is neither active nor processed
               update(precursor)
     m_time = modtime(file)
     for each of file's precursors
          if precursor is not active and modtime(precursor) > m_time
               record that file is out-of-date
     if file is out-of-date or m_time = 0
          execute file's commands
     mark file as processed
}
```

Algorithm 1.

precursor lists and the start node for the depth-first search. A file's updating commands are executed after the search explores all edges leaving that node, so the ordering of files according to potential command execution gives a *postorder* listing of the depth-first search tree.

The distinction between active and processed nodes is drawn so that cycles in the dependency graph are treated properly. Suppose that prog is an executable program, source is the source file for prog, and versions contains archived versions of source that are maintained by a version control tool.[6, 7] Thus the precursor of prog is source, the precursor of source is versions (since source can be derived by running a program that extracts it from versions) and the precursor of versions is source (since versions is updated by executing a program that installs the contents of source). If prog and source do not exist, then a call to update(prog) yields the depth-first search

$$\begin{tikzcd}
\text{prog} \arrow[r] & \text{source} \arrow[r] & \text{versions} \arrow[l, bend right, dashed]
\end{tikzcd}$$

The call to update(prog) calls update(source), which calls update(versions). After versions is found to be current, source is extracted from versions and compiled to produce prog. If source is then edited, a second update of prog yields the same depth-first search, and the call to update(versions) again avoids calling update(source) because source is active. The critical point is that the modification times of source and versions are not compared during the update decision for versions, so the installation commands associated with versions are not executed. In brief, even though source may be younger than versions, versions is not updated if the depth-first search starts at prog or source. However, once the edit–compile–test iteration brings prog to the desired state, the command to update versions performs the search

$$\begin{tikzcd}
\text{versions} \arrow[r] & \text{source} \arrow[l, bend right, dashed]
\end{tikzcd}$$

and runs the commands that install the contents of source in versions. In general, the update decision for a file ignores *ancestral* precursors, i.e. precursors that are ancestors in the depth-first search tree.

The effects of updating commands are modelled as *strands*: if the updating commands for file u can alter the time stamp of file v, then a dotted arrow is drawn from u to v. The dependency graph with these added strands is an *update graph*. Figure 2 gives the update graph for the hoc example. It is important to remember that the solid and dashed edges in an update graph depict the precursor relations, whereas the dotted edges mirror the updating commands.

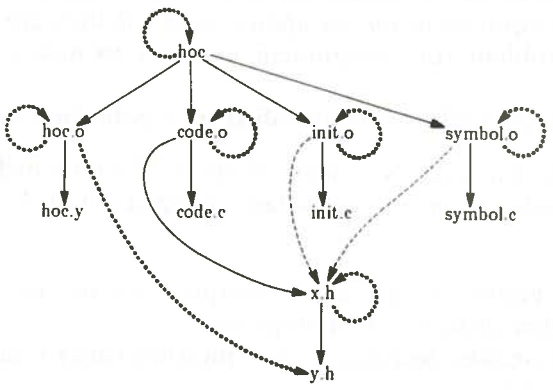


Figure 2. Update graph for hocexample

A strand from u to v can have one of three effects on a file's time stamp. First, u's commands might set v's time stamp to 0 with a remove command. Secondly, the commands might set v's time stamp to an essentially arbitrary time by assigning the name v to some other file with a rename command. Finally, strands such as those in the hoc example represent *touches*, that is cases where u's commands set v's modification time to the current time. In addition, strands can be of two varieties with regard to determinancy. Most strands, such as those from a file to itself, are of the *must* variety, i.e. successful execution of the commands will result in its time stamp being set. Occasional strands, such as the touch loop on x.h, are of the *may* variety, i.e. execution of the commands may or may not result in the time stamp being set, depending on factors not modelled by the update graph.

EQUIVALENCE OF UPDATE ALGORITHMS

The main results of this paper deal with 'equivalence' of update algorithms. In informal terms, this means that given any member of a certain class of update problems, the algorithms will generate identical command sequences. However, all factors affecting the update decisions must be fixed before it makes sense to ask for identical command sequences. Just as the contents of the files x.h and y.h affected the command sequence in the hoc example, so can factors such as the time of day or system load. For example, if the update program detaches the commands for a file so that they run in parallel, then the completion of those processes might affect the update decision for a postorder successor. Such subtleties make it difficult to treat the notion of 'equivalent update procedures' informally.

An *update problem* is an update graph in which each node and strand has been assigned to a non-negative integer, called a *time*. The integer assigned to a node represents the time that would be returned by a call to modtime on the file. If a node's time is 0, then the file does not exist. Note that only the relative magnitudes of the integers affect the problem. The integer assigned to a strand from u to v represents the effect of u's commands on v's time stamp. Specifically, if the commands update node u (i.e. determines that u's commands should be executed), then the time on each strand from u to v is attached to v, replacing v's former time. Formally, two update procedures are *equivalent for an update problem* if they update the same nodes. Two update procedures are *equivalent for an update graph* if they are equivalent for every conceivable update problem (i.e. assignment of times to nodes and strands) on the given update graph.

The rule for updating is informally specified in a published description of make.[1]

> To 'make' a particular node N, 'make' all the nodes on which it depends. If any has been modified since N was last changed, or if N does not exist, update N.

Although Algorithm 1 embodies a plausible interpretation of this statement, make uses an update algorithm that differs in two respects:

1. make bases each update decision on modification times that are sampled earlier in the update process.
2. make's update rule incorporates an *ad hoc* clause designed to support 'fictitious' files, such as cleanup, whose commands do not actually produce a file with that name.

The implications of these two differences are considered separately.

Algorithm 2 samples times in the same manner as make. Whereas Algorithm 1 samples file's modification time *after* processing all precursors, Algorithm 2 samples it *before* processing the precursors. Also, Algorithm 1 samples a precursor's modification time after processing *all* precursors, whereas Algorithm 2 samples the time either the instant that the precursor becomes active (if it is not updated) or just after the precursor is processed (if it is updated). An attribute remembered_time is associated with each node. The attribute's value is set just before the node's processing is complete, and it is subsequently used in place of modtime.

Algorithm 1 calls modtime once for each node and once for each dependency edge to an inactive node, whereas Algorithm 2 calls it just once or twice per node. This fact and the relatively high cost of system calls to modtime help motivate Algorithm 2.

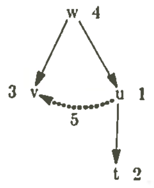
There exist update problems where Algorithms 1 and 2 produce different results. For example, consider the update problem

$$\begin{tikzcd} v\ 1 \arrow[d] \\ u\ 2 \arrow[d] \arrow[u, bend right, dotted, "4"'] \\ t\ 3 \end{tikzcd}$$

```
update(file)
{
    mark file as active
    m_time = modtime(file)
    for each of file's precursors {
        if the precursor is neither active nor processed
            update(precursor)
        if precursor is not active and remembered_time[precursor]>m_time
            record that file is out-of-date
    }
    if file is out-of-date or m_time = 0 {
        execute file's commands
        m_time = modtime(file)
    }
    remembered_time[file] = m_time
    mark file as processed
}
```

Algorithm 2.

Algorithm 1 updates u, which resets v's time to 4. Then modtime(v) is computed as 4, so v is not updated. However, Algorithm 2 computes modtime(v) before updating u, and hence updates both u and v. The difference is caused by the strand from u to a proper ancestor of u in the depth-first search tree. Similarly, with the update problem



Algorithm 1 updates u and w, whereas Algorithm 2 updates only u. The algorithms differ because there is a strand from u to a node, v, preceding u in postorder that has an antecedent, w, that follows u in postorder.

Algorithms 1 and 2 are equivalent for update graph G if and only if G has no strands from u to v such that either v is a proper ancestor of u in the depth-first search tree, or v precedes u in postorder and v is a precursor of some node that follows u in postorder. This fact is rigorously proved in Reference 8, as are all subsequent equivalence claims. When interpreting these statements, algorithms are equivalent for an update graph if their execution is the same for *all possible* assignments of time stamps to nodes and strands. For example, if all strands are of the *touch* variety then the conditions guarantee equivalence but are overly stringent. This is illustrated below, where equivalence is observed for a *makefile* not satisfying the conditions, but for which every strand is labelled with the current time.

The second difference between make's update algorithm and Algorithm 1, i.e. the difference between make's update algorithm and Algorithm 2, is explained informally in the *UNIX Programmer's Manual*:[9]

If the file exists after the commands are executed, its time of last modification is used in further decisions; otherwise the current time is used.

The rationale for this addition is to support rules, such as cleanup, with mnemonic names for commands that do not actually generate the named file. Without this provision, antecedents of such fictitious files will not be updated when the file's commands are executed. Algorithm 3, which is essentially make's update algorithm, adds the appropriate clause to Algorithm 2. To obtain Algorithm 3, the clause

```
if m_time = 0
    m_time = current_time()
```

is added just after the second assignment to m_time in Algorithm 2. In this algorithm, the values of remembered_time need not be actual file modification times. Algorithms 2 and 3 are equivalent for update graph G if and only if every node, except possibly the root, has a touch loop.[8]

OFF-LINE UPDATE ALGORITHMS

There are circumstances under which an update procedure will not actually execute the updating commands that it determines are necessary. First, users sometimes want the procedure to merely report what commands it thinks need to be executed. Visual inspection of the commands before they are run might avoid an expensive disaster. Secondly, there are computer systems where command execution cannot be interleaved with execution of the update procedure. Instead, commands are written to a file and executed later.

When files are not modified by a call to update(precursor), the update procedure cannot subsequently call modtime to see whether precursor was modified. In effect, the procedure must work with just the dependency graph and the initial times on files; it will not have any information about the strands and their attached times. An *off-line version* of an update algorithm constructs a simple update problem from the given dependency graph and initial file times, then solves the simple problem by simulating on-line (i.e. normal) updating. Specifically, at every node u in the submitted dependency graph that has commands, the off-line procedure adds a *touch loop*, i.e. a strand from u to u whose attached time exceeds the time on every node. In the absence of strand information, this construction is plausible assuming that a file is regenerated by its associated commands.

make provides the option of off-line updating. In effect, make simulates on-line updating of a file by labelling a loop at that node with the current time, i.e. the time that would be stamped on the file if it were updated at the current instant. This approach is natural because the clause added to Algorithm 2 to give Algorithm 3 need only be changed to

```
if m_time = 0 or updating is off-line
    m_time = current_time()
```

With Algorithms 1 and 2, the use of current__time is not so natural, and simpler implementations of off-line updating are possible. For example, Algorithm 4 handles both the on- and off-line problems and was obtained by augmenting Algorithm 1 as shown by the italicized code fragments. It suffices to know only the initial time stamps for each file and whether or not a file has been updated. The time stamp of the file after the update is not relevant since it is known to be greater than the initial time stamp of every file. Thus the off-line variant only needs a single bit, changed, for each file. This bit is only set in the off-line case and hence does not affect on-line operation. This bit is distinct from a file's status (active, processed, neither active nor processed) and is assumed to be initially clear for each file.

```
update(file)
{
    mark file as active
    for each of file's precursors (in order)
        if the precursor is neither active nor processed
            update(precursor)
    m_time = modtime(file)
    for each of file's precursors
        if precursor is not active and
          (modtime(precursor) > m_time or precursor's change bit is set)
            record that file is out-of-date
    if file is out-of-date or m_time = 0 {
        if updating is on-line
            execute file's commands
        else if file has commands {
            print file's commands
            set file's change bit
        }
    }
    mark file as processed
}
```

Algorithm 4. An on/off-line extension of Algorithm 1

The off-line version of Algorithm 3 used by make makes a precursor look younger than an antecedent if that precursor has no commands and does not exist. (This is not totally out of the question — another file may have commands that create the precursor). For this reason, Algorithm 3 is not equivalent to Algorithms 1 and 2 in the off-line situation. However, if every file with precursors has commands, then every node has a touch-loop and all three off-line algorithms are equivalent.

OFF-LINE VS. ON-LINE UPDATING

In essence, off-line updating assumes that each file with commands has a must-touch loop and that side-effects are limited. The hoc example illustrates that failure of either assumption can cause off-line updating to diverge from on-line updating. First, consider update's assumption that commands always modify the associated file. Suppose that hoc is brought up to date, then y.h is touched without altering its contents. A subsequent off-line update will assume that x.h would be modified by its updating command, and will generate the report

```
if x.h ≠y.h, then copy y.h to x.h
compile code.c
compile init.c
compile symbol.c
link hoc.o code.o init.o symbol.o; name the executable file hoc
```

Only the first command would be executed by on-line updating.

This example also illustrates the problems caused by side-effects. Suppose that hoc is brought up to date, then hoc.y is modified. A subsequent off-line update discovers that hoc.y is younger than hoc.o and reports that hoc.o's commands should be executed. It is unable to anticipate that updating hoc.o has the side-effect of modifying y.h. Thus off-line updating generates just the commands for hoc.o and hoc. On-line updating also executes the command for x.h and may recompile code.c, init.c and symbol.c.

The on-line and off-line versions of Algorithms 1–3 are equivalent for update graph *G* if and only if *G* satisfies

(i) Every node with precursors has commands and a touch loop except possibly the root.

(ii) For every other strand from *u* to *v*, *v* precedes *u* in postorder and is not the precursor of a node that follows *u* in postorder.[8]

Taking *u* to be the node init.o in Figure 1, conditions (i) and (ii) disallow strands to any nodes except init.o, hoc.y, code.c, init.c and y.h. Except for the special case of a loop on init.o, strands to these nodes are permitted because their time stamps affect only update decisions that are made before init.o is updated. Other strands affect update decisions made after the side-effect occurs. For example, a strand from init.o to x.h would mean that init.o's commands can alter the time stamp on x.h, thereby affecting the update decision for symbol.o.

EXPERIENCE

make is the first and foremost tool of its kind. However, it has been ten years since make was introduced and there is now a larger reservoir of experience to draw upon when designing or examining update tools. This experience is recorded in

1. *makefiles*, which are the best indicators of users' needs, since they do not share the requirement of backward compatibility that propagates outdated or ill-chosen features in programs and descriptive documents
2. make itself, which can serve as the final authority for the fine structure of its update algorithm
3. documents describing make, which help interpret information from the other sources.

A modest file update tool, update,[2] draws on this experience. In preference to Algorithm 3, update uses Algorithm 1 because it is easier to motivate informally and describe accurately. These qualities were particularly desirable since update's code is intended for study. However, it was necessary to assess the practical impact of using a different algorithm, so *makefiles* were inspected. The study had to be performed by hand because it is insufficient to, say, run two versions of make on every *makefile* and see if they differ. The extra effort is needed to determine equivalence on all conceivable update problems; testing a single situation does not suffice.

All the *makefiles* used for building the commands in the UNIX 4.2 Berkeley

Distribution were examined. The 10 *makefiles* contained a total of 721 lines, 521 of which were hand-written. The remaining 200 were simple dependency lines generated by a preprocessor.

No cycles were found in any of the dependency graphs specified by the *makefiles*. However, a robust tool must detect dependency cycles in order to avoid crashing if one is inadvertently introduced by a naîve user, and it takes no more code to handle them in a useful way than to treat them as a fatal error. It may be that an absence of literature discussing dependency cycles has contributed to their scarcity in actual *makefiles*.

The core of every *makefile* studied consisted of executable files that depended on object files, which in turn usually depended on one source file and a number of header files. Each executable and object file had a must-touch loop, some of which involved scripts of three or more commands. In addition, every *makefile* had two to five fictitious files used to encapsulate frequently executed command sequences. For example, cleanup removes all object files, print makes a listing of all the source and header files, install moves an executable program to a system area, get fetches the most recent version of a source file from a version control system and comp constructs a new executable program, compares it to the installed version, and then removes the files it made. The commands for these fictitious files induced either no strands (e.g. print, install), must-remove strands (e.g. cleanup, comp), or must-touch strands (e.g. get). However, these files had no antecedents, implying that they are either the root of a make or not visited at all. Hence the conditions for equivalence between off-line and on-line versions hold, and thus all algorithms and their off-line versions operate identicallly on such *makefiles*.

The investigation uncovered only one example where strands did not fit into the simple pattern above. The *makefile* is very similar to the hoc example. Its update graph contains one ancestral must-touch edge and one may-touch loop. The may-touch loop corresponds to the one on x.h and the ancestral strand results from reversing the order of hoc's precursors, which in turn forces one to add a dependency edge from y.h to hoc.o. The on-line and off-line algorithms can differ for this *makefile*. However, all the on-line algorithms are equivalent, even though the conditions for the equivalence of Algorithms 1 and 2 are violated. This is because the relevant strands are of the must-touch variety. This fact suggests that as a topic for further research, one should consider refining the equivalence conditions to consider the types (touch, remove or rename) of the strands in an update graph.

ACKNOWLEDGEMENTS

Stu Feldman kindly discussed several aspects of make's update algorithm. Use of the Kernighan–Pike example to illustrate differences between off-line and on-line updating was suggested by Dave Hanson. Chris Fraser made numerous useful suggestions. The referees' suggestions improved the paper. This work was supported in part by the U.S. National Science Foundation under Grant MCS82–10096.

REFERENCES

1. S. I. Feldman, 'Make—a program for maintaining computer programs', *Software—Practice and Experience*, **9**, (3), 255–265 (1979).
2. W. Miller, *A Software Tools Sampler*, Prentice-Hall, Englewood Cliffs, NJ, 1986.

3. K. Walden, 'Automatic generation of make dependencies', *Software—Practice and Experience,* **14**, (6), 575–585 (1984).
4. B. W. Kernighan and R. Pike, *The Unix Programming Environment*, Prentice-Hall, Englewood Cliffs, NJ, 1984.
5. A. V. Aho, J. E. Hopcroft and J. D. Ullman, *Data Structures and Algorithms*, Addison-Wesley, Reading, MA, 1983.
6. M. J. Rochkind, 'The source code control system', *IEEE Transactions on Software Engineering,* **1**, (4), 364–370 (1975).
7. W. Tichy, 'RCS—a system for version control', *Software—Practice and Experience,* **15**, (7), 637–654 (1984).
8. W. Miller and E. W. Myers, 'Side-effects in automatic file updating', *TR 85-12*, Department of Computer Science, University of Arizona, Tucson, AZ, June 1985.
9. S. I. Feldman, 'Make—a program for maintaining computer programs', *UNIX Programmer's Manual: Supplementary Documents (4.2BSD)*, University of California, Berkeley, 1984.