A Make Abstract Machine

Glenn Fowler

gsf@research.att.com

Abstract

Many programming languages rely on abstractions to speed the implementation or to share common components with other languages. For example, some C++ compilers produce C as an intermediate step [ATT89], some PROLOG compilers produce WAM (Warren Abstract Machine) code [Warr83], and some C, C++, and FORTRAN compilers generate intermediate code for common back ends that do optimization and machine code generation. Abstractions allow complex language constructs to be implemented once and well.

So why an abstraction for *make*? The concept behind *make* is simple, but the abundance of incompatible implementations (*make* [Feld79], *augmented make* [SV84], *build* [EP84], *nmake* [Fowl85][Fowl90], *mk* [Hume87], *SunPro make* [Palk87], *gnumake* [SM89], *parallel make* [Baal88], *POSIX make* [POSI93], *Microsoft NMAKE* [MICR90], *imake* [DuBo93], *4.4 bsdmake* [BSD93]) makes it difficult to determine the exact semantics for a given makefile. Although makefiles specify the files under *make's* control, it is only the *make* execution that reveals the actions to be taken. Traditionally this is where *make* programming support has been weak. The usual dynamic output, a shell command trace, reveals little about the relationships that triggered them.

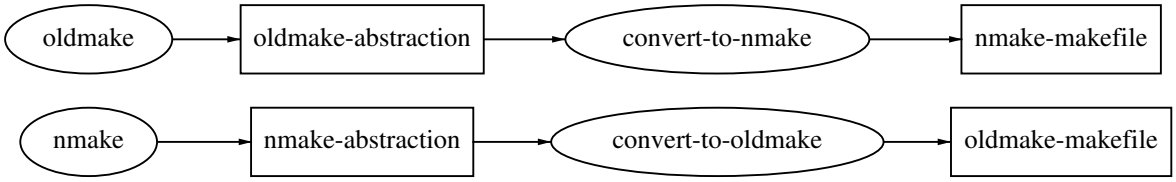
The *make abstract machine* encompasses the static and dynamic nature of *make*. It has a simple instruction set that has been used to abstract both *oldmake* (System V, BSD, GNU) and *nmake*. *make* abstractions form the basis for makefile conversion tools, makefile porting, regression testing, and makefile analysis.

1 Background

MAM (make abstract machine) was born of necessity. I use *nmake* to control all my software. This poses no problem for personal work on my home platform, but makes porting and shipping to incompatible platforms a headache. Originally I maintained two sets of makefiles: *nmake* for local use and *oldmake* for porting. This duplication was particularly annoying during the early stages of software development when makefiles would change many times in a single day. To make matters worse, no amount of care could keep the makefiles from diverging. The only acceptable solution was to single source the makefile information and generate the makefiles as needed.

Makefile generation has been done in the past (*makedepend* [Leff83], [Wald84]) and is still popular (*imake* [DuBo93]), but separating make information from the make engine introduces yet another divergence point between source and generated files. Makefile generation is also expensive enough that it is rarely part of the *make* process (e.g., the makefiles are usually generated at the user's discretion, not *make's*). To avoid inventing yet another makefile generation language, I decided to use *nmake* makefiles as the generator input (influenced by a heavy investment in *nmake*). The temptation at this point was to code the entire generator within *nmake*; after all, it already had the machinery to convert makefiles into an internal tree-based data structure. A pass over this data structure could produce makefiles in any format. But rather than weigh down *nmake* with the moral equivalent of *cb* (C program beautifier), the internal data structure was instead dumped for a separate conversion program to consume.

At the same time there was considerable pressure from new *nmake* users to produce a program that converted *oldmake* makefiles to *nmake*. So a version of System V *make* was modified in a similar manner to *nmake* and a separate generator was coded:



The prospect of adding more formats and converters to handle *mk*, *gnumake*, etc., made clear the need for a common abstraction.

2 The *make* model

Unlike *COMMON LISP*, there probably won't ever be a *common make*; there are just too many variants. The differences are especially evident in makefile syntax and dynamic execution. Conditional makefiles, variable expansion syntax, dependency graph generation and traversal, shell interface, and concurrent target updates are typical divergence points. But the static semantics are a different story. Strip away the bells and whistles and syntactic goo and all *makes* become strikingly similar: execute a minimal (*minimal* is the ultimate goal) set of shell actions, in the correct order, to bring files up to date.

From a static viewpoint *make* manipulates two entities: rules and variables. Rules are the nodes in the dependency graph and their names usually associate one-to-one with file names. A dependency arc from rule `A` to rule `B` is asserted as:

```
A : B
```

and reads: if `B` is newer than `A` then `A` is out of date. An *action* may label a dependency arc:

```
A : B
    action
```

and reads: if `A` is out of date then execute *action* to bring `A` up to date.

Variables are used to parameterize actions and to avoid duplication in rule assertions. They can be assigned:

variable = *value*

and expanded:

`$(`*variable*`)`

make also supports *pattern metarules*. These are special rules that allow some assertions to be omitted. For example, the assertion:

```
main.o : main.c
    cc -c main.c
```

is not usually necessary because of the predefined `%.o : %.c` (or the obsolete `.c.o`) metarule. Metarules are a notational convenience; for any metarule based makefile there exists an equivalent metarule-free makefile. Conceptually, for all implementations, *make* expands makefile variables and metarules to generate a virtual makefile that contains only rule assertions and actions. This virtual makefile describes the complete dependency graph on which *make* operates.

3 The MAM Language

MAM provides a simple, concise notation for describing *make* entities (variables, rules, actions) and relationships (dependencies). The language has both a static and dynamic nature. Static MAM specifies the complete *make* dependency graph; this can be used by makefile display and conversion programs. Dynamic MAM traces *make* as it executes; this can be used for *make* regression testing or for communicating *make* actions to other tools, either in real time or as a logging technique.

At this point it may seem that MAM is shaping up to be just another *make*, but there is an important difference: MAM does not define the *make* algorithm. Although it specifies the dependency graph, it does not specify ''out of date'' or under what circumstances the actions are executed. The intent is not to replace *make* but rather to provide a common language that communicates ''make'' to other programs.

MAM syntax is akin to assembly: a sequence of instructions, one per line, read from top to bottom. The complete instruction set is listed in Figure 1. The rule and variable definition instructions support static makefile descriptions whereas the runtime instructions provide dynamic information as *make* executes.

rule definition

```
make rule [ attribute ... ]
done rule [ attribute ... ]
prev rule [ attribute ... ]
exec rule [ action-line ]
```

variable definition

```
setv variable [ value ]
```

runtime

```
info type [ value ... ]
bind rule [ binding ]
code rule exit-code time [ attribute ... ]
```

miscellaneous

```
note [ comment ]
```

Figure 1: MAM instruction set

A detailed description follows.

`make` *rule* [*attribute* ...]

`done` *rule* [*attribute* ...]

A `make/done` pair defines the target rule named *rule*. Nested `make/done` pairs define the dependencies: the enclosing *rule* is the parent and the enclosed *rules* are the prerequisites. The optional *attributes* classify the rule or dependency relationship (older *makes* may not emit any attributes):

`archive` An `archive` rule binds to a file that contains information on other files. `archive` usually marks libraries controlled by the *ar* command.

`dontcare` Marks files that need not exist. If the file exists then its time is checked and propagated, otherwise it is silently ignored.

`generated` Marks rules that are generated by a shell action.

`implicit` Marks the current rule as an implicit prerequisite of the enclosing parent rule. An implicit prerequisite can make the parent rule out of date without triggering the parent action. Implicit prerequisites usually correspond to `#include` prerequisites. For example, if `x.o` is generated from `x.c`, `x.c` is generated from `x.y`, and `x.c` includes `x.h`, then `x.h` is an implicit prerequisite of `x.c`. Touching `x.h` does not make `x.c` out of date but it does make `x.o` out of date. Implicit prerequisites are currently generated only by *nmake*.

`joint` Marks one of a group of rules that are built by a single shell action.

`metarule` A `metarule` rule provides predefined *make* information; its action and prerequisite actions are not executed, but are instead used to hold additional information. `metarule` is provided as an extension; no tools currently use it.

`state` Marks state variable rules.

`virtual` A `virtual` rule is not associated with any file. Some *makes* may optimize file system operations based on this.

`prev` *rule* [*attribute ...*]
Used to reference rules that have already been defined by `make/done`.

`exec` *rule* [*action-line*]
Appends *action-line* to the shell action for *rule*. The *rule* name − names the rule in the current `make/done` nesting. `exec` is not emitted until all prerequisite rules for *rule* have been defined.

`setv` *variable* [*value*]
Assigns *value* to *variable*. A `setv` is required for each variable referenced in the mamfile, even if its value is null, and it must occur before the first reference of the variable. This allows an analysis program to determine all variables used in the makefile and the proper assignment order.

`info` *type* [*value*]
The `info` instruction is a catch-all for *make* execution environment information:

`info mam` *MM/DD/YY* [*generator*]
This is always the first instruction. It identifies the MAM version (*YYYY-MM-DD*) and optional version information on the program that generated the mamfile (*generator*).

`info start` *time*
Shows the *make* start time in seconds since the epoch.

`info pwd` *path*
Shows the current working directory path.

`info view` *view*
Shows the current view information if either viewpathing or union directories are present.

`info finish` *time code*
Shows the *make* finish time and exit code.

`info error|warning|debug|panic|info` *message*
make messages of varying severity.

`bind` *rule time* [*binding*]
The `bind` instruction is emitted when *make* binds a rule to a file. *time* is the file modification time in seconds since the epoch. If *binding* is omitted then the file name is *rule*, otherwise *binding* is the path name of the corresponding file. Any of the *make* path search algorithms (VPATH, CPATH, .PATH* .SOURCE*) may cause *binding* to differ from *rule*.

`code` *rule exit-code*
This instruction reports the exit status *exit-code* when the action for *rule* completes.

`note` [*comment*]
This is the comment instruction and is otherwise ignored.

To support multiple *make* processes in a dynamic MAM trace, an optional process id number may prefix each instruction:

```
12345 info start 6789
54321 info finish 6790 0
...
```

4 MAM example

The makefile in Figure 2 will be used to illustrate MAM. The corresponding mamfile is listed in Figure 3, annotated with line numbers for easy reference. The first column contains mamfile line numbers and the second column contains line numbers from the makefile in Figure 2.

```
1    DEBUG = -g -DDEBUG=1
2    CCFLAGS = $(DEBUG)
3    cmd : cmd.o lib.o
4        $(CC) $(CCFLAGS) -o cmd cmd.o lib.o
5    cmd.o : cmd.h lib.h
6    lib.o : lib.h
```

Figure 2: example makefile

The `info mam` instruction (line 01) identifies the file as a mamfile, lists the MAM version, and also identifies the program that generated the mamfile. Variables are defined using `setv` (lines 02,03,04). Lines 02 and 03 come directly from the makefile and line 04 is inferred from the predefined rules. `exec` defines the rule actions (lines 13,19,21), but is not emitted until after all prerequisite rules have been defined. The rule name − is shorthand for the rule name in the current `make/done` nesting (lines 13,19,21).

Makefile variable references are converted to shell syntax in the mamfile (lines 03,13,19,21). This provides a common target language for actions and also makes it possible to analyze makefiles using the shell. Variable references may occur in all instructions, and are commonly used to parameterize *rule* names:

```
setv INSTALLROOT $HOME
....
make $INSTALLROOT/include/std.h
```

5 MAM analysis

From a human standpoint MAM is too verbose, but this is exactly what makes it an attractive input language for program analysis. The close connection between MAM and *shell* is also intentional; it allows most MAM analyzers to be written directly in *shell*. Also, the analysis scripts are short enough that a few are included in this paper.

Rather than get bogged down in shell compatibility issues (and for brevity) the example scripts use *ksh93* [BK94], which provides builtin arithmetic and associative arrays. For portability the scripts have also been written in V7 shell [Bour78], but the V7 scripts are much larger (up to 10 times) and noticeably slower, mainly because external commands must be used as a substitute for *ksh93* features.

The first program generates a shell script that contains the variable definitions and actions that bring all mamfile targets up to date. Such a script could be used to bootstrap software on systems that have no *make*. This is how

```
01    -    info mam 01/01/94 oldmake
02    1    setv DEBUG -g -DDEBUG=1
03    2    setv CCFLAGS $DEBUG
04    -    setv CC cc
05    3    make cmd
06    3    make cmd.o
07    -    make cmd.c
08    -    done cmd.c
09    5    make cmd.h
10    5    done cmd.h
11    5    make lib.h
12    5    done lib.h
13    -    exec - $CC $CCFLAGS -c cmd.c
14    3    done cmd.o
15    3    make lib.o
16    -    make lib.c
17    -    done lib.c
18    6    prev lib.h
19    -    exec - $CC $CCFLAGS -c lib.c
20    6    done lib.o
21    4    exec - $CC $CCFLAGS -o cmd cmd.o lib.o
22    2    done cmd
```

Figure 3: example mamfile

nmake source was ported from 1985 to 1989. Given a mamfile, generating a bootstrap script is trivial: simply grab the `setv` and `exec` instructions and convert to shell syntax:

`setv` *variable* `[` *value* `]`

converts to

variable="*value*"

and

`exec` *rule* `[` *action-line* `]`

converts to

action-line

The variable assignments and commands are in the correct order because, by definition, `setv` instructions are emitted before the first variable reference and `exec` instructions are not emitted until all prerequisite rules have been defined. The script, *mamsh*, is listed in Figure 4.

```
sed -n -e 's/^setv \([^ ]*\) \(.*\)/\1="${\1-\2}"/p' -e 's/^exec [^ ]* //p'
```

Figure 4: *mamsh*: convert MAM to shell script

The resulting *mamsh* bootstrap script for the makefile in Figure 2 is:

```
DEBUG="-g -DDEBUG=1"
CCFLAGS="$DEBUG"
CC="cc"
$CC $CCFLAGS -c cmd.c
$CC $CCFLAGS -c lib.c
$CC $CCFLAGS -o cmd cmd.o lib.o
```

Generating a graphical representation of makefile dependencies is slightly more complicated. Figure 5 lists *mamdag*, a script that converts mamfile input into a *dag* [GNV88] specification. *dag* input consists of a header, a trailer, and lists of the form `"A" "B" "C" ... ;` that reads: an edge connects node `A` to node `B`, node `A` to `C`, etc. *mamdag* uses the `make` and `prev` instructions to build the prerequisite lists (lines 07,11) and the `done` instruction to print the non-empty lists (line 13). It follows typical MAM script form:

- initialize (lines 01-04)
- check options and arguments (none in this example)
- read each mamfile line and `case` on each instruction (lines 05-17), where `make` and `done` correspond to push and pop operations (lines 07,13)
- clean up (line 18)

The *dag* specification generated from the example mamfile is:

```
.GR 7.50 10.0
draw nodes as Box ;
"cmd.o" "cmd.c" "cmd.h" "lib.h" ;
"lib.o" "lib.c" "lib.h" ;
"cmd" "cmd.o" "lib.o" ;
.GE
```

```
01  integer level=0
02  list[0]=all
03  print ".GR 7.50 10.0"
04  print "draw nodes as Box ;"
05  while read op arg junk
06  do    case $op in
07        make) list[level]="${list[level]} \"$arg\""
08              level=level+1
09              list[level]=
10              ;;
11        prev) list[level]="${list[level]} \"$arg\""
12              ;;
13        done) [[ ${list[level]} ]] && print \"$arg\" ${list[level]} ';'
14              level=level-1
15              ;;
16        esac
17  done
18  print ".GE"
```

Figure 5: *mamdag*: convert MAM to *dag* input

The resulting *dag* layout is:

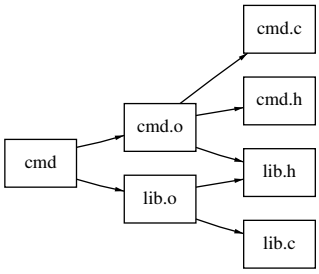


Figure 6 lists *mamexec*, a script that implements *make* with state using mamfiles rather than makefiles as input. A script similar to this (except written in ~300 lines of V7 shell) is a major component of a software distribution system that has been used to ship *ksh* and *nmake* to a wide variety of systems and requires only a V7 *sh*, a C compiler, *ls*, *comm*, *sort* and *sed*.

A common reaction to *mamexec* (especially to its V7 counterpart) is: Why would anyone want *make* in shell? The answer is portability:

- The V7 *mamexec* works the same on all UNIX® system variants that support the shell. It has even been used to bootstrap *ksh* and *nmake* on Microsoft Windows NT† (Microsoft *NMAKE* is completely different from *oldmake* and AT&T *nmake*).
- *mamexec* is small enough that it can be shipped along with the source to be built.
- *mamexec* avoids *make* implementation differences on recipient systems. Such differences are often difficult to debug, especially when no login is available.
- By operating from mamfiles, *mamexec* also avoids *make* implementation differences on the sending system. These are difficult to debug because features in the home environment are easy to take for granted. MAM inherently freezes *make* features into a common language.
- Unlike *oldmake*, *mamexec* handles actions with embedded shell *here documents* and multi-line `case` statements.

To be fair, *mamexec* does not implement full *make* semantics. It only has four options:

`-A`	Accept current files as up to date.
`-F`	Force all files to be out of date.
`-f` *mamfile*	The input mamfile is *mamfile* instead of the default `Mamfile`.
`-n`	Show actions but do not execute.

mamexec does not have target selection: all targets that are out of date are built (i.e., it cannot just make `main.o`; adding this would cost about 10 lines). To its advantage, however, *mamexec* implements a state algorithm for testing ‘‘out of date.’’ In *oldmake* a target is out of date if its modification time is older than the modification time of any of its prerequisites. In the state algorithm the modification time for each target is saved in a *statefile* just before *make* exits. A target is out of date if its current modification time is different from its state time (saved in the statefile) *or* if any of its prerequisites are out of date. The state algorithm detects changes that occur when old files

† Microsoft Windows and Microsoft Windows NT are trademarks of the Microsoft Corporation.

```
01  typeset beg="(set -ex;" end=") </dev/null" exec=eval mamfile=Mamfile
02  typeset -i accept=0 force=0 level=0
03  typeset -A list same
04  while getopts "AFf:[mamfile]n" opt
05  do    case ${opt#${opt%?}} in
06        A) accept=1 ;;
07        F) force=1 ;;
08        f) mamfile=$OPTARG ;;
09        n) exec=print beg= end= ;;
10        esac
11  done
12  [[ $opt == "?" ]] && exit 1
13  ml=$mamfile.ml ms=$mamfile.ms
14  [[ $force == 0 && -f $ml && -f $ms ]] &&
15  for i in $(ls -ld $(<$ml) 2>/dev/null|sort|comm -12 $ms -|sed 's/.* //')
16  do same[$i]=1; done
17  while read -r op arg data
18  do    case $op in
19        setv) eval val='$'$arg
20              [[ $level != 0 || $val ]] && eval $arg='$data </dev/null'
21              ;;
22        make) level=level+1
23              eval arg=$arg
24              [[ " $data " == *" metarule "* ]] && same[$arg]=1 && continue
25              name[level]=$arg cmds[level]= list[$arg]=1
26              ;;
27        prev) eval arg=$arg
28              [[ ! ${same[$arg]} ]] && same[${name[level]}]=
29              ;;
30        exec) [[ ! ${cmds[level]} ]] && cmds[level]=$data ||
31              cmds[level]=${cmds[level]}$'\n'$data
32              ;;
33        done) eval arg=$arg
34              [[ ! ${same[$arg]} ]] &&
35              {  [[ ${cmds[level]} ]] &&
36                 { $exec "$beg${cmds[level]}$end" || exit; }
37                 same[${name[level-1]}]=; }
38              level=level-1
39              ;;
40        esac
41  done <$mamfile
42  [[ $accept == 1 || $exec == eval ]] &&
43  { print ${!list[@]} > $ml; ls -ld ${!list[@]} 2>/dev/null|sort > $ms; }
```

Figure 6: *mamexec*: *make* with state

are restored, a situation that *oldmake* cannot handle. This may happen when *cpio* or *tar* archives are read into the current source tree, but also happens more frequently under viewpathing (using `VPATH` in *oldmake*, *gnumake*, and *nmake*, or *vpath* in *nDFS* [KK90] or *union mounts* in Plan 9 and 4.4 BSD) when top layer files are removed to expose older lower layer files. More importantly, the state algorithm allows an action to chose *not* to update its target file(s). Unlike *oldmake*, the state algorithm will not trigger such actions the next time. This supports smart actions that may determine ''out of date'' by mechanisms more complex than modification time.

The *mamexec* source is straightforward. Lines 01-13 initialize the local variables (lines 01-02), associative arrays (line 03), and options (lines 04-13). Lines 15-16 compare the current state with the previous state (if it exists). `same[`*rule*`]` is `1` if *rule's* current modification time is the same as the statefile modification time (*ls* determines the time, *comm* and *sort* determine entries that have not changed). A rule is out of date if `same[rule]` is null. The main loop (lines 17-38) collects prerequisites and actions and executes actions for out of date rules (line 33). Only builtin commands are executed in the main loop. `setv` assigns variable that have no previous value unless the `setv` occurs inside a `make/done` nesting (line 20). Lines 39-40 update the statefile before the script exits. The state consists of 2 files; *mamfile*`.ml` contains the list of all rules and *mamfile*`.ms` contains the state time and name for all rules.

	user	sys	user+sys
gnumake	0.35	0.30	0.65
oldmake	0.43	1.08	1.51
mamexec	1.23	0.40	1.65
nmake	0.85	0.95	1.80

Figure 7: comparative *make –n* times

Figure 7 is a table of user+sys times on an unloaded SPARC 2 for various *make –n* commands running on a makefile (or mamfile) with 97 up to date rules and 223 action lines. Since *mamexec* is a shell script and the other *makes* are compiled programs one would assume that *mamexec* would perform poorly, but this is not the case. There are a few reasons for this: the mamfile parse is simple compared to makefiles, and all rule bindings are precomputed in the mamfile (i.e., metarules and file path searches have already been applied by the mamfile generator). *nmake* is slightly slower in this example because it computes the implicit prerequisites that have already been asserted in the mamfile and *oldmake* makefiles, and the startup cost for this outweighs the small makefile size.

Makefile conversion, last mentioned in the introduction, hasn't been forgotten. Original plans were to describe an 800 line, 5 year old C program that converts MAM to *oldmake* makefiles, but after cleaning up the *mamdag* and *mamexec* scripts for publication a *mamold* script precipitated out. Figure 8 lists *mamold*, a MAM to *oldmake* makefile converter.

Since it generates *oldmake* makefiles, *mamold* must differentiate between explicit (`prereqs[`*rule*`]`) and implicit (`implicit[`*rule*`]`) prerequisites for a given *rule* (lines 27-29). `setv` collects the variable names (line 23) so that the shell style variable references can be converted to *make* style by the `convert` function (lines 01-07). The assertions are emitted in mamfile order (lines 32,43-48), and the `closure` function emits the transitive closure of the explicit and implicit prerequisites.

Embedded newlines in actions have always been a problem for *oldmake* because it traditionally executes action lines one at a time, each in a separate shell. Some shell constructs, such as *here documents* and multiline `case` statements, cannot appear in *oldmake* actions. So *mamold* does not play as an important a role as *mamexec* in the software porting process.

6 Generating MAM

MAM deliberately mimics the *make* algorithm, so adding MAM to *make* is a simple task. The `make` and `done` instructions bracket the *make* inner loop, called *doname*() in the original and *update_file_1*() in *gnumake*. Often MAM instructions correspond to portions of the debug trace output, as happens with *gnumake* and System V

```
01  function convert {
02        typeset buf=$*; typeset -i i
03        set -s -A variable ${variable[@]}
04        for (( i = ${#variable[@]} - 1; i >= 0; i-- ))
05        do  buf=${buf//\$${variable[i]}/\$(${variable[i]})}; done
06        print -r -- "$buf"
07  }
08  function closure {
09        typeset i j
10        for i
11        do    [[ " $list " == *" $i "* ]] && continue
12              list="$list $i"
13              for j in ${implicit[$i]}
14              do  closure $j; done
15        done
16  }
17  typeset -A prereqs implicit action
18  typeset -i level=0 nvariables=0
19  typeset rule list order target variable
20  print "# # oldmake makefile generated by $0 # #"
21  while read -r op arg val
22  do    case $op in
23        setv) variable[nvariables++]=$arg
24              convert "$arg = $val"
25              ;;
26        make|prev) rule=${target[level]}
27              [[ " $val " == *" implicit "* ]] &&
28              implicit[$rule]="${implicit[$rule]} $arg" ||
29              prereqs[$rule]="${prereqs[$rule]} $arg"
30              [[ $op == prev ]] && continue
31              target[++level]=$arg
32              [[ " $order " != *" $arg "* ]] && order="$order $arg"
33              ;;
34        exec) [[ $arg == - ]] && arg=${target[level]}
35              [[ ${action[$arg]} ]] &&
36              action[$arg]=${action[$arg]}$'\n'$'\t'$val ||
37              action[$arg]=$'\t'$val
38              ;;
39        done) level=level-1
40              ;;
41        esac
42  done
43  for rule in $order
44  do    [[ ! ${prereqs[$rule]} && ! ${action[$rule]} ]] && continue
45        list=
46        closure ${prereqs[$rule]} && print && convert "$rule :$list"
47        [[ ${action[$rule]} ]] && convert "${action[$rule]}"
48  done
```

Figure 8: *mamold*: MAM to *oldmake* makefile converter

oldmake. The `exec` instruction is just an intercept of action execution. Actions may be parameterized by first emitting `setv` for each parameter variable:

```
setv CC cc
setv CCFLAGS
...
```

and then initializing the *make* variable values to expand to their shell counterparts:

```
CC = $$CC
CFLAGS = $$CCFLAGS
...
```

The `-M` option generates MAM in *nmake*. `-M` implies `-n` (show actions but not execute) and `-F` (force all targets to be out of date); this ensures that all rules and actions are represented in the output MAM. A separate option, `-o mamtrace=`*file*, outputs runtime MAM in *file* as *nmake* executes. `-M` and `-F` have also been added to an internal version of System V *oldmake*. The normal *make* arguments apply: if you want a mamfile of the `install` target then run `make -M install`; if you want a mamfile of the default target then run `make -M`.

7 MAM omissions

There is no MAM analogue for makefile conditionals:

```
# generic makefile conditional syntax
if  this is release x of system y with hardware z
    command : x_y_z.c
endif
```

Basically MAM describes makefiles relative to the current environment, just as the C preprocessing phase freezes `#if` and `#ifdef` conditionals and macro expansions for C programs. This is a drawback *if* MAM is viewed as a replacement makefile language.

8 Conclusion

Although MAM started out to support makefile conversion, it has inspired the development of several different programs that aid makefile analysis, generate shell scripts for software porting, report software updates as *make* executes, and even replace a generous subset of *make* itself. MAM is stable and has been used to port and analyze software within AT&T for over four years.

References

[ATT89] *UNIX System V AT&T C++ Language System: Release 2.1 Release Notes*, AT&T, Select Code 307-160, 1989.

[Ball88] Erik Ballbergen, *Design and Implementation of Parallel Make*, USENIX Computing Systems, 135-158, Spring 1988.

[BK89] Morris Bolsky and David G. Korn, *The KornShell Command and Programming Language*, Prentice Hall, 1989.

[BK94] Morris Bolsky and David G. Korn, a revised edition of [BK89], to be published.

[Bour78] S. R. Bourne, *The UNIX Shell*, AT&T Bell Laboratories Technical Journal, Vol. 57 No. 6 Part 2, pp. 1971-1990, July-August 1978.

[BSD93] *4.4 BSD make*, 4.4 BSD UNIX distribution.

[DuBo93] Paul DuBois, *Software Portability with imake*, O'Reilly and Associates, 1993.

[EP84] 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.

[Feld79] S. I. Feldman, *Make – A Program for Maintaining Computer Programs*, Software – Practice and Experience, Vol. 9 No. 4, pp. 256-265, April 1979.

[Fowl85] G. S. Fowler, *The Fourth Generation Make*, USENIX Portland 1985 Summer Conference Proceedings, pp. 159-174, 1985.

[Fowl90] G. S. Fowler, *A Case for make*, Software – Practice and Experience, Vol. 20 No. S1, pp. 35-46, June 1990.

[GNV88] E. R. Gansner, S. C. North and K. P. Vo, *DAG - A Program That Draws Directed Graphs*, Software – Practice and Experience, Vol. 17, No. 1, pp. 1047-1062, 1988.

[Hume87] A. G. Hume, *Mk: a successor to make*, USENIX Phoenix 1987 Summer Conference Proceedings, pp. 445-457, 1987.

[KK90] D. G. Korn and E. Krell, *A New Dimension for the Unix File System*, Software – Practice and Experience, Vol. 20 No. S1, pp. 19-33, June 1990.

[MICR90] *Microsoft C: Advanced Programming Techniques*, Microsoft Corporation, pp. 103-132, 1990.

[Leff84] Samuel J. Leffler, *Building 4.2BSD UNIX Systems with Config*, UNIX System Manager's Manual, University of California, Berkeley, July 1984.

[Palk87] R. Palkovic, *SunPro: The Sun Programming Environment*, M. Hall (ed.), A Sun Technical Report, Sun Microsystems, Inc., pp. 67-86, 1987.

[POSI93] IEEE Standard for Information Technology – Portable Operating System Interface (POSIX) – Part 2: Shell and Utilities (Volume 1), pp. 666-679, IEEE, 1993.

[SM89] R. M. Stallman and R. McGrath, *GNU Make – A Program for Directing Recompilation*, Edition 0.1 Beta, March 1989.

[SV84]*Augmented Version of Make*, UNIX System V – Release 2.0 Support Tools Guide, pp. 3.1-19, April 1984.

[Wald84] Kim Walden, *Automatic Generation of Make Dependencies*, Software – Practice and Experience, Vol. 14 No. 6, pp. 575-585, June 1984.

[Warr83] David H. D. Warren, *An abstract Prolog instruction set*, Technical note 309, SRI Project 4776, Stanford Research Institute International, Menlo Park, CA, October 1983.