Jam — Make(1) Redux

Christopher Seiwald
INGRES Corporation
Seiwald@Ingres.Com
March 11, 1994

Abstract

Despite the progress of UNIX, the basic mechanism by which developers build their programs — *make*(1) — has remained at its core unimproved since its inception. Most notably, the *make* language has seen few improvements. Jam is a *make* replacement that uses an extensible, expressive language for describing ways in which files relate. This new language simplifies the description of systems, both small and large, and renders extending Jam's functionality not only possible but easy.

Jam exists now and runs on many UNIX platforms, VMS, and NT. It is freely available in the comp.sources.unix archives. As proof of concept, it has been used to build a very large commercial product, generating in a single invocation 1,000 deliverable files from 12,000 source files.

1. Introduction

The UNIX *make*(1) program [Feldman, 1986], which automates the building of targets from their source files, is widely used. Together with its compatible successors (*dmake* [Vadura, 1990], GNU *make* [Stallman, 1991], NET2 *make* [BSD, 1991], SunOS *make* [Sun, 1989]) and mutations (*cake* [Somogyi, 1987], *cook* [Miller, 1993], *nmake* [Fowler, 1985], Plan 9 *mk* [Flandrena], *mms* on VMS, etc.), *make* enjoys world domination in its capacity.

As *make*'s author, Stuart Feldman, noted, *make* itself is not suited for describing huge programs. This is arguably because the *make* language has one useful statement: the expression of a direct dependency among files. This makes for a clumsy, bottom-up description of how to build a system — describing large systems this way is unmanageable. *make*'s successors try to overcome this difficulty with sundry tricks: dependencies with wild-carded names matched against directory contents; parsing command output as Makefile syntax; macro expansions with and without the help of the C preprocessor, etc.

Jam is an attempt to replace *make*'s rule system, with its bottom-up language and wayward implicit rules, with an expressive language that makes it possible to describe explicitly and cogently the compilation of programs. The current practice of building source simply because it matched wildcards is unreliable in the face of debris left around by a careless programmer. A robust product should be built explicitly, and to make this palatable, Jam makes it easy to be explicit.

A typical sample of Jam's language as seen by end-users will serve to anchor its description:

```
MAIN prog : prog.c ;
LIBS prog : libaux.a ;
LIBRARY libaux.a : compile.c gram.y scan.c ;
```

This example invokes three rules that instruct Jam to build an archive from three source files, to compile a fourth source file, and to link it against the archive. All three rules, as well as all other rules given as examples in this paper, are stock ones that come with Jam (see *Jambase*(5) [Seiwald, 1993]).

2. The Jam Language

As with *make*, the most important statement in the Jam language is the expression of a relationship among files. With *make*, the relationship is a direct dependency; with Jam, the relationship is user-defined. The expression of such a relationship is:

```
<RULE> <targets> [ : <sources> ] ;
```

This statement is referred to as a rule invocation, with the name of the rule leading the statement. Except for a handful of built-in rules, the definition of a rule is user-defined. The <sources> are optional. Each of the three lines in the example above are rule invocations.

Because rules invoke each other, the expression of a user-defined relationship can result in other user-defined relationships being made among the same or different files. In the case of the example given, the MAIN rule will invoke the rules:

```
CC prog.o : prog.c ;
LINK prog : prog.o ;
```

These rules (presumably) handle the cases of compiling an object module from a C source file and linking that object module into an executable.

2.1. Rule Definition

A rule is defined in two parts: the Jam statements to execute when the rule is invoked (essentially a procedure call); and the actions (*sh*(1) commands) to execute in order to update the targets of the rule. A rule may have a procedure definition, actions, or both.

A rule's procedure definition is given with:

```
rule <RULE> { <statements> }
```

This statement causes <statements> to be interpreted by Jam whenever <RULE> is invoked. The <targets> and <sources> given at rule invocation are available as the special variables $(<) and $(>) in the <statements> defining the rule. <statements> may be any of the Jam statements listed in this document. Carrying on the CC example, a definition for the CC rule might be:

```
rule CC
{
      DEPENDS $(<) : $(>) ;
}
```

This particular rule definition simply arranges for the targets to depend on the sources, using the built-in rule DEPENDS (described below).

A rule's updating actions are given with:

```
actions <RULE> { <string> }
```

This causes the *sh* script <string> to be associated with the <targets> in any invocation of <RULE>. Later, if Jam determines that the <targets> are out-of-date, it will pass <string> to *sh* for execution. Jam expands $(<) and $(>) in <string>, but $(<) and $(>) in this case refer to the <targets> and <sources> after they have been bound to real file path names (see Binding, below). Finishing out the CC example, a definition of CC actions would be:

```
actions CC
{
        cc -c $(CCFLAGS) -I$(HDRS) $(>)
}
```

2.2. Rule Effects

Rule invocations have no outputs or return values and, instead, do their job through three distinct types of side-effects. The first is when a rule's procedure invokes built-in rules to modify the target dependency graph. These built-ins will be discussed shortly. The second is when a rule's procedure sets variables. The third is the association of the updating actions with the targets, which occurs whenever a rule with updating actions is invoked.

2.3. Built-in Rules

There are six built-in rules, five of which modify the target dependency graph. None of these rules have updating actions. The built-in rules are:

```
DEPENDS, INCLUDES, ECHO, TEMPORARY, NOTIME, NOCARE
```

DEPENDS and INCLUDES take <targets> and <sources>. ECHO takes only <targets>. TEMPORARY, NOTIME, and NOCARE take only <targets> and mark them with attributes to indicate special handling when descending the dependency graph.

DEPENDS
: The basic builder of the dependency graph: it makes <sources> dependencies of <targets>, just like the simple *make* ":" dependency. If <sources> are newer than <targets> (using file update times for comparison), or if <sources> are being updated, then the updating actions of <targets> will be executed.

INCLUDES
: A variation on DEPENDS: it makes <sources> dependencies of any targets of which <targets> are dependencies. This example makes both foo.c and foo.h dependencies of foo.o:

```
DEPENDS foo.o : foo.c ;
INCLUDES foo.c : foo.h ;
```

ECHO
: Just echoes its targets to the standard output, as a means for communicating with the user. Jam knows no fatal error, so the message emitted by ECHO can only be advisory.

TEMPORARY
: Allows for intermediate targets to be missing and not updated if the final target is up-to-date. If a target marked TEMPORARY is not present, then it simply inherits its parent's time-stamp. TEMPORARY can be used for any temporary target, such as the short-lived object module that is to be part of a library archive.

NOTIME
: Indicates that the target is not really a file and therefore doesn't have a time-stamp. Any updating actions are only executed if the target's dependencies were updated, rather than on the basis of time-stamp comparisons. NOTIME is used for pseudo targets such as "all" or "install", which have dependencies but don't actually get built themselves.

NOCARE
: Indicates that the target may both be non-existent and not have any updating actions. This loophole is used to make up for the sloppiness of the header-file scanning.

2.4. Jam Variables

Part of Jam's programmability lies in its treatment of variables. As with *make* and *sh*, Jam variables are lists of strings, with zero or more elements. But unique to Jam, the result of variable expansion is the product of the variable values and literal constants in the token being expanded. An example helps here:

```
$(X) -> a b c
t$(X) -> ta tb tc
$(X)$(X) -> aa ab ac ba bb bc ca cb cc
```

This approach makes quick work of many normal variable manipulations: prepending a path name to a list of file names, prepending "-l" to a list of library names, appending a ",v" to RCS file names, etc.

Jam has a modicum of variable editing options to replace components of a path name and to subselect members of a list. These options are discussed in usable detail in Jam's manual page [Seiwald, 1993].

Unlike *make*, Jam does not defer expansion of variables. When a variable is referenced, even to assign a new variable, the value is expanded at that time.

Jam variables have two scopes: global and target-specific. Global variables behave much as one might expect, holding their value until reassigned. Target-specific variables take precedence over global variables when the specified target is being bound (see below) or updated. The distinction between global and target-specific variables is made when the variables are assigned. The syntax for setting the two types is, respectively:

```
<var> = <value> ;
<var> on <targets> = <value> ;
```

Target-specific variables have several uses. A simple one is to permit different compiler flag settings for different source files. In this way, the actions of the CC rule may be used to compile any C source file, with various flags (HDRS, CCFLAGS) being adjusted per-target. Other uses of target-specific variables will be discussed shortly.

2.5. Flow-of-Control

In addition to statements for defining and invoking rules and setting variables, the Jam language contains statements for flow-of-control and file inclusion. The statements are:

```
if <condition> { <statements> } [ else { <statements> } ]

for <var> in <value> { <statements> }

switch <value> { case <value> : <statements> ; ... }

include <file> ;
```

The "if" statement does the obvious; the <condition> is the usual mix of comparison and logical operators applied to variables.

The "for" statement iterates over the elements of <value>, assigning the (global) variable <var> to each element and executing the statement block.

The "switch" statement executes the statement block whose case <value> matches the switch's <value>.

The "include" statement sources another file containing Jam statements.

Jam neither needs nor desires a macro preprocessor. Making rule definitions and file inclusions normal statements obviates a macro preprocessor for conditional compilation, as these statements may appear within Jam conditionals. Further, preprocessing would require the Jam language to play "dodge-em" with the preprocessor semantics.

3. Binding Files

Jam can find source and target files in distant directories, much like the functionality of VPATH in GNU *make* and *dmake*.

By default, a target is located at the actual path of the target, relative to the directory of Jam's invocation. If the special variable $(LOCATE) is set to a directory name, Jam locates the target in that directory (correctly concatenating the value of $(LOCATE) and the target's path name). If $(LOCATE) is unset but the special variable $(SEARCH) is set to a directory list, Jam searches along the directory list for the target file (again, correctly concatenating the path names).

Jam makes available the bound target names by using them when expanding $(<) and $(>) for updating actions. Thus, a target can be referred to by a short, unrooted name when invoking a rule to define a relationship, but any shell commands manipulating the target see a path name usable from the current directory.

$(SEARCH) provides VPATH-like functionality, allowing Jam to be invoked in directories other than where the source lives, while $(LOCATE) liberates Jam from the directory tree altogether. With it, Jam can run anywhere.

By setting $(SEARCH) and $(LOCATE) properly, Jam can handle a variety of build environments. For example, read-only source trees can be handled by pointing $(SEARCH) at a read-only source code directory while pointing $(LOCATE) to a working directory. As another example, "sparse" source trees can be handled by having $(SEARCH) contain two directories: first the developer's own directory, which contains only the files he is editing, and then his group's directory, which contains the master copy of all source. Most importantly, much of any build environment can be encoded in the settings of $(SEARCH) and $(LOCATE), which leaves the file names used in rule invocations free from environment.

The power of $(SEARCH) and $(LOCATE) is realized when these variables are set per-target rather than just globally. Each individual target file can potentially be found along different search paths. In practice, related files will have the same search path, but Jam can efficiently accommodate the degenerate case of having these variables set per-target. In this way, Jam can build whole source trees, with source files scattered across directories.

4. Header-File Inclusion

Jam handles the incidental dependencies caused when source files include other source files. To find such dependencies, Jam scans source files for header-file inclusions, using a regular expression pattern match [Spencer, 1986]. The regular expression is given in the variable $(HDRSCAN). The result of the scan is not interpreted directly by Jam; to arrange the necessary relationship, Jam calls a user-defined rule named in the variable $(HDRRULE), with the scanned file as <targets> and the found header-files as <sources>. Usually, the definition of $(HDRRULE) includes a call to the built-in rule INCLUDES, which updates the dependency graph appropriately. An example HDRSCAN that works for C preprocessor includes is:

```
HDRSCAN = "^#[ \t]*include[ \t]*[<\"](.*)[\">].*$" ;
```

The combination of $(HDRSCAN) and $(HDRRULE), when set per-target, enables Jam to handle just about any include-file syntax or semantics. Unfortunately, this mechanism doesn't understand conditional includes (#include within #ifdef), and can produce bogus dependencies that must be crudely pasted over with the application of the built-in NOCARE rule.

5. Time-Stamps

Like *make* et. al., Jam uses time-stamps to determine when targets are out-of-date. Another possible design, a more forward-looking one, would have Jam taking file update cues from an integrated source management system. This was deferred for two reasons: first, it would require picking a source management system with which to work (or attempting to engineer a generic interface to source management systems); second, it would preclude using Jam as a drop-in replacement for existing uses of *make*.

The code in Jam that checks dependencies is isolated enough to be altered to work with a source management system. Internally, Jam already distinguishes between updates due to newer dependents and updates due to updated dependents.

6. The Base Rule Set

A collection of rules providing *make*–like functionality is supplied with Jam. Called Jambase, the file provides a dozen-odd rules for compiling and linking C source code. Different versions of Jambase exist for UNIX, VMS, and NT, all providing the same rule set.

Figure 1 lists the rules defined in the current Jambase (described comprehensively in *Jambase*(5)).

The last act of Jambase is to include a file called Jamfile from the invoking user's current directory. Using the rules defined in Jambase, the user's Jamfile enumerates the source files and their relationship to the targets to be built.

The Jambase and Jamfile files share the same language; only their purposes distinguish them. It is possible to write a special-purpose replacement Jambase that is totally self-contained and needs no directory-specific Jamfile. It is also possible to use any Jam syntax — including conditionals, rule definitions, etc. — in a Jamfile.

7. The Example

Returning to our earlier example:

```
MAIN prog : prog.c ;
LIBS prog : libaux.a ;
LIBRARY libaux.a : compile.c gram.y scan.c ;
```

This example invokes three rules that instruct Jam to build an archive from three source files, to compile a fourth source file, and to link it against the archive. All these rules are defined by the Jambase file and do most of their work by invoking other rules defined in the Jambase.

MAIN calls LINK to set up the relationship between **prog** and **prog.o**, then calls OBJECT to set up the relationship between **prog.o** and **prog.c**. OBJECT calls a rule specific to the file suffix, in this case, CC for **.c**. Along the way, the various rules invoke the built-in DEPENDS rule to set up the dependency graph.

```
MAIN image : source ;         link executable from compiled sources
LIBS image : libraries ;      link libraries onto a MAIN
UNDEFINES image : symbols ;   save undefs for linking
SETUID image ;                mark an executable SETUID
LIBRARY lib : source ;        archive library from compiled sources
OBJECT objname : source ;     compile object from source
HDRRULE source : headers ;    handle #includes
CC obj.o : source.c ;         .c -> .o
LEX source.c : source.l ;     .l -> .c
YACC source.c : source.y ;    .y -> .c
YYACC source.y : source.yy ;  .yy -> .y
BULK dir : files ;            populate directory with many files
FILE dest : source ;          copy file
SHELL exe : source ;          install a shell executable
RMTEMPS target : sources ;    remove temp sources after target made
INSTALLBIN sources ;          install binaries
INSTALLLIB sources ;          install files
INSTALLMAN source ;           install man pages
```

Figure 1 — Rules supplied with Jam

```
MAIN prog : prog.c ;
  DEPENDS exe : prog ;
  LINK prog : prog.o ;
    DEPENDS prog : prog.o ;
  OBJECT prog.o : prog.c ;
    CC prog.o : prog.c ;
      DEPENDS prog.o : prog.c ;

LIBS prog : libaux.a ;
  DEPENDS prog : libaux.a ;
  NEEDLIBS on prog = libaux.a ;

LIBRARY libaux.a : compile.c gram.y scan.c ;
  DEPENDS libaux.a : libaux.a(compile.o) libaux.a(gram.o) libaux.a(scan.o) ;
  DEPENDS libaux.a(compile.o) : compile.o ;
  OBJECT compile.o : compile.c ;
    CC compile.o : compile.c ;
      DEPENDS compile.o : compile.c
  DEPENDS libaux.a(gram.o) : gram.o ;
  OBJECT gram.o : gram.y ;
    CC gram.o : gram.c ;
      DEPENDS gram.o : gram.c
    YACC gram.c : gram.y ;
      DEPENDS gram.c gram.h : gram.y ;
      INCLUDES gram.c : gram.h ;
  DEPENDS libaux.a(scan.o) : scan.o ;
  OBJECT scan.o : scan.c ;
    CC scan.o : scan.c ;
      DEPENDS scan.o : scan.c
  ARCHIVE libaux.a : compile.o gram.o scan.o ;
  TEMPORARY compile.o gram.o scan.o ;
```

Figure 2 — Rule execution for the example rule invocations

LIBS is a rule that arranges for **libaux.a** to become a dependency of **prog**, and it sets the target-specific variable NEEDLIBS to let the actions of LINK know that **libaux.a** should be included on the link command line. LIBS has no actions of its own.

LIBRARY is a rule that sets up the (somewhat complicated) dependencies between the library **libaux.a**, its members, and the temporary object modules that are to be its members. It calls OBJECT to set up the relationship between each of the temporary object modules and their source files. It also calls the ARCHIVE rule to handle the archiving of the temporary object modules into **libaux.a**.

A more complete list of rule invocations seen by Jam for this example is given in Figure 2.

Probably lost in this litany of rules are some important features: the OBJECT rule, when presented with the task of making a ".o" file from a ".y" file, called both the CC and YACC rules. Note that this is considerably easier and more deterministic than *make*'s approach of making a ".o" from whatever happens to be available. Also, note that the YACC rule took advantage of the INCLUDES built-in to make sure the dependencies on the generated file are accurately registered.

Actually, the rule definitions include a few more machinations to give special variables sensible defaults. For source code, $(SEARCH) is set to $(SEARCH_SOURCE); for object files, $(LOCATE) is set to $(LOCATE_OBJECT); for C source files, $(HDRSCAN) is set to the example pattern mentioned above, and $(HDRRULE) is set to "HDRRULE", the generic header-handling rule defined in the Jambase file.

8. Implementation

The weight of Jam's implementation is evenly divided between its rule-processing subsystem (driven by a *yacc*(1) grammar), its recursive binding and scanning subsystem, and its recursive build subsystem.

The rule-processing subsystem is entirely system independent, only setting in-memory variables, building the dependency graph, and associating update actions with targets. The *yacc* grammar is less than 200 lines.

The recursive binding and scanning subsystem is mostly system independent, but calls system-dependent routines to time-stamp files and to manipulate file names.

The recursive build subsystem is mostly system independent, but calls system-specific routines to execute shell commands (which are system-specific as well).

The system dependencies are hidden through three interfaces: one to time-stamp files; one to manipulate file names; and one to execute shell commands.

The file time-stamp interface has two layers: a higher one that asks about individual files; and a lower one that scans directories and library archives whole. The latter is more efficient, and all current implementations (UNIX, VMS, NT) are coded against it.

The file name manipulation interface consists of two routines: one to break a file name down into its components and one to build a file name from its components. These are quite simple — except on VMS, where concatenating path names is black art.

The shell-command interface currently approximates the UNIX *system*(3) call interface, with an addition for catching interrupts.

Jam achieves its functionality while going sparingly on features. It has only four flags (mostly to do with debugging), six built-in rules (DEPENDS, INCLUDES, ECHO, NOTIME, NOCARE, TEMPORARY) and six special variables ($(>), $(<), $(SEARCH), $(LOCATE), $(HDRSCAN), $(HDRRULE)). The whole of Jam for UNIX is under 5,000 lines of code, exclusive of Henry Spencer's *regexp*(3) regular-expression code (about another 1,300 lines).

A design goal of Jam was portability, specifically so that the same mechanism could be used to build the same system on different platforms. Jam scores well in this category: the OS interface is constricted, leaving the bulk of the system dependencies in the Jambase file. Even the Jambase file is somewhat portable, with only the filename syntax and the actual update commands having to change between UNIX and VMS. Jamfile files themselves usually contain nothing system-specific.

9. Performance

Used to build from scratch a large commerical software system (the INGRES relational DBMS), lapse time for Jam breaks down as follows (on an HP9000/710):

```
parsing 5,000 lines of Jamfiles                  16 seconds
stat()'ing 12,000 source files                   1 minute
scanning 12,000 source files for headers         9 minutes
actual building (compiling, linking, etc)        12 hours
```

The simple conclusion is that Jam's performance in inconsequential. When everything is up-to-date, only few improvements could be made. *stat*()'ing files is essentially unavoidable without resorting to other techniques for determining outdated targets. Scanning source files could be avoided by caching header-file dependency information in state files. SunOS *make* and *nmake* use this approach. The only other recourse is to hammer on the *regexp* implementation.

The real performance limitation is in actual building time. Jam does not yet support parallel command execution, which on a large SMP system can reduce build time by a factor of 5 or more. This feature is anticipated.

10. Comparisons

Jam's per-target variables are a convenience approached only by SunOS *make*'s "target := macro = value" syntax. Both Jam and SunOS *make* make use of the value when updating the target, but Jam gets added mileage out of the facility by using the value during the binding and header-file scanning.

Jam's searching mechanism is superior to VPATH in two ways: first, it provides not only searching for existing targets, but also binding for new targets; second, Jam's SEARCH and LOCATE variables can be set per-target. GNU *make* allows VPATH to be set selectively, using patterns, and the patterns could be full file names, but GNU *make* handles the degenerate case of separate values per file poorly. Jam's SEARCH and LOCATE mechanism can make the invoker's directory irrelevant, which amounts to a complete solution.

Jam's pattern-scanning method of header-file scanning is faster than those that offload the problem to separate programs (*dmake*, *cake*, GNU *make*). It is not strictly correct, like SunOS *make* and GNU *make*, which use the C preprocessor. Jam's mechanism, driven by per-target variables and user-defined relationships is, however, quite flexible. It can handle languages that don't offer a separate preprocessor, as well as languages where the result of a file being included is more than just a simple dependency. For example, when a *yacc* file includes a C header-file, Jam can be made to understand that the generated C source file will include the generated C header-file. Jam supports these types of arrangements entirely in its language.

Jam is missing a few features cherished by some *make* users: the ability to run update commands concurrently and fancy variable editing. These may appear in future versions of Jam.

11. Discussion

The comparison of Jam's language with *make*'s is somewhat subjective and complicated. As stated in the introduction, Jam is an attempt to replace *make*'s rule system with an expressive language that makes it possible to describe explicitly and cogently the compilation of programs.

In this respect, Jam is a success. For small systems, the Jamfile file is often not larger than the three lines that made up our example. For large systems, any added complexity can be centralized in the Jambase file, while the Jamfile file(s) in source directories remain simple.

Jam's rule semantics, that of expressing named relationships among files, is Jam's single biggest advantage over its contemporaries. Its power and economy of expression seem unmatched. There are two other approaches deserving mention. *nmake* and *dmake* allow new operators (replacements for the simple ":" dependency statement) to be defined as macros, and these can be used to create new relationship types. Unfortunately, the number of available "operator" characters is limited, and the coding of the macros would curl the eyebrows of even seasoned *sendmail* hackers. *cook*, *cake*, Plan 9 *mk*, and NET2 *make* promote a different approach: that of defining variables and then #include'ing "recipes" (other *make* files) that define the relationships. The recipes are the approximate equivalent of Jam rules, using pass-by-name variables. This scheme works, but it is an ugly ordeal to try to recover with a preprocessor the functionality that is lacking in a language.

The Jam language turns out to be fairly straightforward to program. With its reliance on keywords rather than special characters and its use of a ";" to terminate statements, it is easier reading than most *make* syntax.

Abandoning *make* syntax was an easy decision: even those new *make*s that understand traditional *make* syntax get their added functionality through incompatible syntax. If compatibility with *make* is the priority, users can just use *make*. If users want greater functionality, they can't use vanilla *make* anyway.

The proof of Jam is in the pudding (sorry...): it is worth mentioning that the timing information given above is for a single, non-recursive invocation of Jam to compile 12,000 source files scattered throughout 300 directories, producing 7,000 intermediate targets and 1,000 deliverable files. Each source directory contains a single Jamfile with an average of 1.5 words per source file (including the source file name). The author knows of no other *make* that can approach such completeness with such economy.

12. Availability

Jam is freely available in Volume 27 of the comp.sources.unix archives. It is known to compile and work on VMS (Alpha and VAX) and the following variants of UNIX: BSD/386, OSF/1, DGUX 5.4, HPUX 9.0, AIX, IRIX 5.0, PTX V2.1.0, SunOS 4, Solaris 2, Ultrix 4.2, and Linux. Support for NT is not part of that initial release but is available from the author.

13. Bibliography

[Brokken, 1994]
Frank B. Brokken and Karel Kubat, "ICMAKE — the Intelligent C-like MAKEr, or the ICce MAKE utility", Linux Sources, 1994

[BSD, 1991]
BSD NET2 make(1) manual page, BSD NET2 documentation, July 1991.

[Feldman, 1986]
S. I. Feldman, "Make — A Program for Maintaining Computer Programs", BSD NET2 documentation, April 1986 (revision).

[Flandrena]
R. Flandrena, "Plan 9 Mkfiles", available via anonymous FTP from plan9.att.com.

[Fowler, 1985]
Glenn Fowler, "The Fourth Generation Make", Proceedings of the USENIX Summer Conference, June 1985.

[Miller, 1993]
Peter Miller, "Cook — A File Construction Tool", Volume 26, comp.sources.unix archives, 1993.

[Seiwald, 1993]
Christopher Seiwald, Jam(1) and Jambase(5) manual pages, Volume 27, comp.sources.unix archives, 1993.

[Spencer, 1986]
Henry Spencer, Regexp code and comment, comp.sources.unix archives, 1986.

[Stallman, 1991]
Richard M. Stallman and Roland McGrath, "GNU Make — A Program for Directed Recompilation", Free Software Foundation, 1991

[Somogyi, 1987]
Zoltan Somogyi, "Cake, a Fifth Generation Version of Make", Australian Unix System User Group Newsletter, April 1987.

[Sun, 1989]
Sun Microssytems Corporation, SunOS make(1) manual page, SunOS 4.1.2 documentation, September 1989.

[Vadura, 1990]
Dennis Vadura, dmake(1) manual page, Volume 27, comp.sources.misc archives, 1990.