Cake: a fifth generation version of make

Zoltan Somogyi
Department of Computer Science
University of Melbourne
Parkville, 3052 Victoria, Australia

UUCP: {uunet,mcvax,ukc}!munnari.oz!zs
ARPA: zs%munnari.oz@uunet.uu.net
CSNET: zs%munnari.oz@australia

Abstract

Make is a standard Unix[1] utility for maintaining computer programs. Cake is a rewrite of make from the ground up. The main difference is one of attitude: cake is considerably more general and flexible, and can be extended and customized to a much greater extent. It is applicable to a wide range of domains, not just program development.

1. Introduction

The Unix utility make [Feld79] was written to automate the compilation and recompilation of C programs. People have found make so successful in this domain that they do not wish to be without its services even when they are working in other domains. Since make was not designed with these domains in mind (some of which, e.g. VLSI design, did not even exist when make was written), this causes problems and complaints. Nevertheless, implied in these complaints is an enormous compliment to the designers of make; one does not hear many grumbles about programs with only a few users.

The version of make described in [Feld79] is the standard utility. AT&T modified it in several respects for distribution with System V under the name augmented make [Augm84]. We know of two complete rewrites: enhanced make [Hirg83] and fourth generation make [Fowl85]. All these versions remain oriented towards program maintenance[2].

Here at Melbourne we wanted something we could use for text processing. We had access only to standard make and spent a lot of time wrestling with makefiles that kept on getting bigger and bigger. For a while we thought about modifying the make source, but then decided to write something completely new. The basic problem was the inflexibility of make's search algorithm, and this algorithm is too embedded in the make source to be changed easily.

The name cake is a historical accident. Cake follows two other programs whose names were also puns on make. One was bake, a variant of make with built-in rules for VLSI designs instead of C programs [Gedy84]. The other was David Morley's shell script fake. Written at a time when disc space on our machine was extremely scarce, and full file systems frequently caused write failures, it copied the contents of a directory to /tmp and invoked make there.

---

[1] Unix is a trademark of AT&T Bell Laboratories.

[2] Since this paper was written, two other rewrites have come along: mk [Hume87] and nmake.

The structure of the paper is as follows. Section 2 shows how `cake` solves the main problems with `make`, while section 3 describes the most important new features of `cake`. The topics of section 4 are portability and efficiency.

The paper assumes that you have some knowledge of `make`.

2. The problems with `make`

`Make` has three principal problems. These are:

(1) It supports only suffix-based rules.

(2) Its search algorithm is not flexible enough.

(3) It has no provisions for the sharing of new `make` rules.

These problems are built deep into `make`. To solve them we had to start again from scratch. We had to abandon backward compatibility because the `make` syntax is not rich enough to represent the complex relationships among the components of large systems. Nevertheless, the `cake` user interface is deliberately based on `make's`; this helps users to transfer their skills from `make` to `cake`. The *functionalities* of the two systems are sufficiently different that the risk of confusion is minimal[3].

Probably the biggest single difference between `make` and `cake` lies in their general attitudes. `Make` is focused on one domain: the maintenance of compiled programs. It has a lot of code specific to this domain (especially the later versions). And it crams all its functionality into some tight syntax that treats all sorts of special things (e.g. `.SUFFIXES`) as if they were files.

`Cake`, on the other hand, uses different syntax for different things, and keeps the number of its mechanisms to the minimum consistent with generality and flexibility. This attitude throws a lot of the functionality of `make` over the fence into the provinces of other programs. For example, where `make` has its own macro processor, `cake` uses the C preprocessor; and where `make` has special code to handle archives, `cake` has a general mechanism that *just happens* to be able to do substantially the same job.

2.1. Only suffix-based rules

All entries in a `makefile` have the same syntax. They do not, however, have the same semantics. The main division is between entries which describe simple dependencies (how to make file `a` from file `b`), and those which describe rules (how to make files with suffix `.x` from files with suffix `.y`)[4]. `Make` distinguishes the two cases by treating as a rule any dependency whose target is a concatenation of two suffixes.

For this scheme to work, `make` must assume three things. The first is that all interesting files have suffixes; the second is that suffixes always begin with a period; the third is that prefixes are not important. All three assumptions are violated in fairly common situations. Standard `make` cannot express the relationship between `file` and `file.c` (executable and source) because of assumption 1, between `file` and `file,v` (working file and RCS file) because of assumption 2, and between `file.o` and `../src/file.c` (object and source) because of assumption 3. `Enhanced make` and `fourth generation make` have special forms for some of these cases, but these cannot be considered solutions because special forms will always

---

[3] This problem, called cognitive dissonance, is discussed in Weinberg's delightful book [Wein71].

[4] For the moment we ignore entries whose targets are special entities like .IGNORE .PRECIOUS etc.

lag behind demand for them (they are embedded in the make source, and are therefore harder to change than even the built-in rules).

Cake's solution is to do away with make-style rules altogether and instead to allow ordinary dependencies to function as rules by permitting them to contain variables. For example, a possible rule for compiling C programs is

```
%.o:        %.c
            cc -c %.c
```

where the % is the variable symbol. This rule is actually a *template* for an infinite number of dependencies, each of which is obtained by consistently substituting a string for the variable %.

The way this works is as follows. First, as cake seeks to update a file, it matches the name of that file against all the targets in the description file. This matching process gives values to the variables in the target. These values are then substituted in the rest of the rule[5]. (The matching operation is a form of *unification*, the process at the heart of logic programming; this is the reason for the *fifth generation* bit in the title.)

Cake actually supports 11 variables: % and %0 to %9. A majority of rules in practice have only one variable (canonically called %), and most of the other rules have two (canonically called %1 and %2). These variables are local to their rules. Named variables are therefore not needed, though it would be easy to modify the cake source to allow them. If cake wanted to update prog.o, it would match prog.o against %.o, substitute prog for % throughout the entry, and then proceed as if the cakefile contained the entry

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

This arrangement has a number of advantages. One can write

```
%.o:        RCS/%.c,v
            co -u %.c
            cc -c %.c
```

without worrying about the fact that one of the files in the rule was in a different directory and that its suffix started with a nonstandard character. Another advantage is that rules are not restricted to having one source and one target file. This is useful in VLSI, where one frequently needs rules like

```
%.out:      %.in %.circuit
            simulator %.circuit < %.in > %.out
```

and it can also be useful to describe the full consequences of running yacc

---

[5] After this the rule should have no unexpanded variables in it. If it does, cake reports an error, as it has no way of finding out what the values of those variables should be.

```
%.c %.h:   %.y
           yacc -d %.y
           mv y.tab.c %.c
           mv y.tab.h %.h
```

2.2. Inflexible search algorithm

In trying to write a makefile for a domain other than program development, the biggest problem one faces is usually make's search algorithm. The basis of this algorithm is a special list of suffixes. When looking for ways to update a target file.x, make searches along this list from left to right. It uses the first suffix .y for which it has a rule .y.x and for which file.y exists.

The problem with this algorithm manifests itself when a problem divides naturally into a number of stages. Suppose that you have two rules .c.b and .b.a, that file.c exists and you want to issue the command make file.a. Make will tell you that it doesn't know how to make file.a. The problem is that for the suffix .b make has a rule but no file, while for .c it has a file but no rule. Make needs a *transitive rule* .c.a to go direct from file.c to file.a.

The number of transitive rules increases as the square of the number of processing stages. It therefore becomes significant for program development only when one adds processing stages on either side of compilers. Under Unix, these stages are typically the link editor ld and program generators like yacc and lex. Half of standard make's builtin rules are transitive ones, there to take care of these three programs. Even so, the builtin rules do not form a closure: some rare combinations of suffixes are missing (e.g. there is no rule for going from yacc source to assembler).

For builtin rules a slop factor of two may be acceptable. For rules supplied by the user it is not. A general-purpose makefile for text processing under Unix needs at least six processing stages to handle nroff/troff and their preprocessors lbl, bib, pic, tbl, and eqn, to mention only the ones in common use at Melbourne University.

Cake's solution is simple: if file1 can be made from file2 but file2 does not exist, cake will try to *create* file2. Perhaps file2 can be made from file3, which can be made from file4, and so on, until we come to a file which does exist. Cake will give up only when there is *absolutely no way* for it to generate a feasible update path.

Both the standard and later versions of make consider missing files to be out of date. So if file1 depends on file2 which depends on file3, and file2 is missing, then make will remake first file2 and then file1, even if file1 is more recent than file3.

When using yacc, we frequently remove generated sources to prevent duplicate matches when we run egrep ... *.[chyl]. If cake adopted make's approach to missing files, it would do a lot of unnecessary work, running yacc and cc to generate the same parser object again and again[6].

Cake solves this problem by associating dates even with missing files. The *theoretical update time* of an existing file is its modify time (as given by stat(2)); the theoretical update time

[6] In this case make is rescued from this unnecessary work by its built-in transitive rules, but as shown above this should not be considered a *general* solution.

of a missing file is the theoretical update time of its youngest ancestor. Suppose the `yacc` source `parser.y` is older than the parser object `parser.o`, and `parser.c` is missing. `Cake` will figure that if it recreated `parser.c` it would get a `parser.c` which *theoretically* was last modified at the same time as `parser.y` was, and since `parser.o` is younger than `parser.y`, theoretically it is younger than `parser.c` as well, and therefore up-to-date.

2.3. No provisions for sharing rules

Imagine that you have just written a program that would normally be invoked from a `make` rule, such as a compiler for a new language. You want to make both the program and the `make` rule widely available. With standard `make`, you have two choices. You can hand out copies of the rules and get users to include it in their individual `makefiles`; or you can modify the `make` source, specifically, the file containing the built-in rules. The first way is error-prone and quite inconvenient (all those rules cluttering up your `makefile` when you should never need to even look at them). The second way can be impractical; in the development stage because the rules can change frequently and after that because you want to distribute your program to sites that may lack the `make` source. And of course two such modifications may conflict with one another.

Logically, your rules belong in a place that is less permanent than the `make` source but not as transitory as individual `makefiles`. A library file is such a place. The obvious way to access the contents of library files is with `#include`, so `cake` filters every `cakefile` through the C preprocessor.

`Cake` relies on this mechanism to the extent of not having *any* built-in rules at all. The standard `cake` rules live in files in a library directory (usually `/usr/lib/cake`). Each of these files contains rules about one tool or group of tools. Most user `cakefiles` `#define` some macros and then include some of these files. Given that the source for program `prog` is distributed among `prog.c`, `aux1.c`, `aux2.c`, and `parser.y`, all of which depend on `def.h`, the following would be a suitable `cakefile`:

```
#define    MAIN       prog
#define    FILES      prog aux1 aux2 parser
#define    HDR        def

#include   <Yacc>
#include   <C>
#include   <Main>
```

The standard `cakefiles` `Yacc` and `C`, as might be expected, contain rules that invoke `yacc` and `cc` respectively. They also provide some definitions for the standard `cakefile` `Main`. This file contains rules about programs in general, and is adaptable to all compiled languages (e.g. it can handle NU-Prolog programs). One entry in `Main` links the object files together, another prints out all the sources, a third creates a `tags` file if the language has a command equivalent to `ctags`, and so on.

`Make` needs a specialized macro processor; without one it cannot substitute the proper filenames in rule bodies. `Fourth generation make` has not solved this problem but it still wants the extra functionality of the C preprocessor, so it grinds its `makefiles` through both macro processors ! `Cake` solves the problem in another way, and can thus rely on the C preprocessor exclusively.

Standard `make`'s macro facilities are quite rudimentary, as admitted by [Feld79]. Unfortunately, the C preprocessor is not without flaws either. The most annoying is that the bodies of macro definitions may begin with blanks, and will if the body is separated from the macro name and any parameters by more than one blank (whether space or tab). `Cake` is distributed with a fix to this problem in the form of a one-line change to the preprocessor source, but this change probably will not work on all versions of Unix and definitely will not work for binary-only sites.

3. The new features of `cake`

The above solutions to `make's` problems are useful, but they do not by themselves enable `cake` to handle new domains. For this `cake` employs two important new mechanisms: dynamic dependencies and conditional rules.

3.1. Dynamic dependencies

In some situations it is not convenient to list in advance the names of the files a target depends on. For example, an object file depends not only on the corresponding source file but also on the header files referenced in the source.

Standard `make` requires all these dependencies to be declared explicitly in the `makefile`. Since there can be rather a lot of these, most people either declare that all objects depend on all headers, which is wasteful, or declare a subset of the true dependencies, which is error-prone. A third alternative is to use a program (probably an `awk` script) to derive the dependencies and edit them into the `makefile`. [Wald84] describes one program that does both these things; there are others. These systems are usually called `makedepend` or some variation of this name.

The problems with this approach are that it is easy to alter the automatically-derived dependencies by mistake, and that if a new header dependency is added the programmer must remember to run `makedepend` again. The C preprocessor solves the first problem; the second, however, is the more important one. Its solution must involve scanning though the source file, checking if the programmer omitted to declare a header dependency. So why not use this scan to *find* the header dependencies in the first place ?

`Cake` attacks this point directly by allowing parts of rules to be specified at run-time. A command enclosed in double square brackets[7] may appear in a rule anywhere a filename or a list of filenames may appear. For the example of the C header files, the rule would be

```
%.o:        %.c [[ccincl %.c]]
            cc -c %.c
```

signifying that `x.o` depends on the files whose names are listed in the output of the command `ccincl x.c`[8], as well as on `x.c`. The matching process would convert this rule to

---

[7] Single square brackets (like most special characters) are meaningful to `csh`: they denote character classes. However, we are not aware of any legitimate contexts where two square brackets *must* appear together. The order of members in such classes is irrelevant, so if a bracket must be a member of such a class it can be positioned away from the offending boundary (unless the class is a singleton, in which case there is no need for the class in the first place).

[8] `Ccincl` prints out the names of the files that are `#included` in the file named by its argument. Since `ccincl` does not evaluate any of the C preprocessor's control lines, it may report a superset of the files actually included.

```
x.o:        x.c [[ccincl x.c]]
            cc -c x.c
```

which in turn would be *command expanded* to

```
x.o:        x.c hdr.h
            cc -c x.c
```

if `hdr.h` were the only header included in `x.c`.

Command patterns provide replacements for `fourth generation make's` directory searches and special macros. `[[find <dirs> -name <filename> -print]]` does as good a job as the special-purpose `make` code in looking up source files scattered among a number of directories. `[[basename <filename> <suffix>]]` can do an even better job: `make` cannot extract the base from the name of an RCS file.

A number of tools intended to be used in just such contexts are distributed together with `cake`. `Ccincl` is one. `Sub` is another: its purpose is to perform substitutions. Its arguments are two patterns and some strings: it matches each string against the first pattern, giving values to its variables; then it applies those values to the second pattern and prints out the result of this substitution. For example, in the example of section 2.3 the `cakefile` `Main` would invoke the command `[[sub X X.o FILES]]`[9], the value of `FILES` being `prog aux1 aux2 parser`, to find that the object files it must link together to create the executable `prog` are `prog.o aux1.o aux2.o parser.o`.

`Cake` allows commands to be nested inside one another. For example, the command `[[sub X.h X [[ccincl file.c]]]]` would strip the suffix `.h` from the names of the header files included in `file.c`[10].

3.2. Conditional rules

Sometimes it is natural to say that `file1` depends on `file2` *if* some condition holds. None of the `make` variants provide for this, but it was not too hard to incorporate conditional rules into `cake`.

A `cake` entry may have a condition associated with it. This condition, which is introduced by the reserved word `if`, is a boolean expression built up with the operators `and`, `or` and `not` from primitive conditions.

The most important primitive is a command enclosed in double curly braces. Whenever `cake` considers applying this rule, it will execute this command after matching, substitution and command expansion. The condition will return true if the command's exit status is zero. This runs counter to the intuition of C programmers, but it conforms to the Unix convention of commands returning zero status when no abnormal conditions arise. For example, `{{grep xyzzy file}}` returns zero (i.e. true) if xyzzy occurs in `file` and nonzero (false) otherwise.

---

[9] `Sub` uses `X` as the character denoting variables. It cannot use `%`, as all `%`'s in the command will have been substituted for by `cake` by the time `sub` is invoked.

[10] As the outputs of commands are substituted for the commands themselves, `cake` takes care not to scan the new text, lest it find new double square brackets and go into an infinite loop.

Conceptually, this one primitive is all one needs. However, it has considerable overhead, so `cake` includes other primitives to handle some special cases. These test whether a filename occurs in a list of filenames, whether a pattern matches another, and whether a file with a given name exists. Three others forms test the internal `cake` status of targets. This status is `ok` if the file was up-to-date when `cake` was invoked, `cando` if it wasn't but `cake` knows how to update it, and `noway` if `cake` does not know how to update it.

As an example, consider the rule for RCS.

```
%:          RCS/%,v          if exist RCS/%,v
            co -u %
```

Without the condition the rule would apply to all files, even ones which were not controlled by RCS, and even the RCS files themselves: there would be no way to stop the infinite recursion (`%` depends on `RCS/%,v` which depends on `RCS/RCS/%,v,v` ...).

Note that conditions are command expanded just like other parts of entries, so it is possible to write

```
%:          archive          if % in [[ar t archive]]
            ar x archive %
```

4. The implementation

4.1. Portability

`Cake` was developed on a Pyramid 90x under 4.2bsd. At Melbourne University it now runs on a VAX under 4.3bsd, various Sun-3's under SunOS 3.4, an Encore Multimax under Umax 4.2, a Perkin-Elmer 3240 and an ELXSI 6400 under 4.2bsd, and on the same ELXSI under System V. It has not been tested on either System III or version 7.

`Cake` is written in standard C, with (hopefully) all machine dependencies isolated in the makefile and a header file. In a number of places it uses `#ifdef` to choose between pieces of code appropriate to the AT&T and Berkeley variants of Unix (e.g. to choose between `time()` and `gettimeofday()`). In fact, the biggest hassle we have encountered in porting `cake` was caused by the standard header files. Some files had different locations on different machines (`/usr/include` vs. `/usr/include/sys`), and the some versions included other header files (typically `types.h`) while others did not.

As distributed `cake` is set up to work with `csh`, but it is a simple matter to specify another shell at installation time. (In any case, users may substitute their preferred shell by specifying a few options.) Some of the auxiliary commands are implemented as `csh` scripts, but these are small and it should be trivial to convert them to another shell if necessary.

4.2. Efficiency

`Fourth generation make` has a very effective optimization system. First, it forks and execs only once. It creates one shell, and thereafter, it pipes commands to be executed to this shell and gets back status information via another pipe. Second, it compiles its `makefiles` into internal form, avoiding parsing except when the compiled version is out of date with respect to the master.

The first of these optimizations is an absolute winner. `Cake` does not have it for the simple reason that it requires a shell which can transmit status information back to its parent process, and we don't have access to one (this feature is provided by neither of the standard shells, `sh` and `csh`).

`Cake` could possibly make use of the second optimization. It would involve keeping track of the files the C preprocessor includes, so that the `makefile` can be recompiled if one of them changes; this must be done by fourth generation make as well though [Fowl85] does not mention it. However, the idea is not as big a win for `cake` as it is for `make`. The reason is as follows.

The basic motivations for using `cake` rather than `make` is that it allows one to express more complex dependencies. This implies a bigger system, with more and slower commands than the ones `make` usually deals with. The times taken by `cake` and the preprocessor are insignificant when compared to the time taken by the programs it most often invokes at Melbourne. These programs, `ditroff` and `nc` (the NU-Prolog compiler that is itself written in NU-Prolog), are notorious CPU hogs.

Here are some statistics to back up this argument. The *overhead ratio* is given by the formula

$$\frac{\textit{cake process system time} + \textit{children user time} + \textit{children system time}}{\textit{cake process user time}}$$

This is justifiable given that the `cake` implementor has direct control only over the denominator; the kernel and the user's commands impose a lower limit on the numerator.

We have collected statistics on every `cake` run on two machines at Melbourne, mulga and munmurra[11]. These statistics show that the overhead ration on mulga is 11 while on munmurra it is 86. This suggests that the best way to lower total CPU time is not to tune `cake` itself but to reduce the number of child processes. To this end, `cake` caches the status returned by all condition commands `{{command}}` and the output of all command patterns `[[command]]`. The first cache has hit ratios of 42 and 54 percent on munmurra and mulga respectively, corresponding roughly to the typical practice in which a condition and its negation select one out of a pair of rules. The second cache has a hit ratio of about 80 percent on both machines; these hits are usually the second and later occurrences of macros whose values contain commands.

`Cake` also uses a second optimization. This one is borrowed from standard `make`: when an action contains no constructs requiring a shell, `cake` itself will parse the action and invoke it through exec. We have no statistics to show what percentage of actions benefit from this, but a quick examination of the standard `cakefiles` leads us to believe that it is over 50 percent.

Overall, `cake` can do a lot more than `make`, but on things which *can* be handled by `make`, `cake` is slightly slower than standard `make` and a lot slower than fourth generation make. Since the main goal of `cake` is generality, not efficiency, this is understandable. If efficiency is important, `make` or one of its other successors is always available as a fallback.

4.3. Availability

The `cake` distribution contains the `cake` source, some auxiliary programs and shell scripts (many useful in their own right), diffs for the `lex` driver and the C preprocessor, library cakefiles, manual entries, and an earlier version of this paper [Somo87]. It was posted to the Usenet newsgroup comp.sources.unix in October of 1987.

---

[11] On mulga (a Perkin-Elmer 3240), the main applications are text processing and the maintenance of a big bibliography (over 58000 references). On munmurra (an EXLSI 6400), the main application is NU-Prolog compilation.

5. Acknowledgements

John Shepherd, Paul Maisano, David Morley and Jeff Schultz helped me to locate bugs by being brave enough to use early versions of `cake`. I would like to thank John for his comments on drafts of this paper.

This research was supported by a Commonwealth Postgraduate Research Award, the Australian Computer Research Board, and Pyramid Australia.

6. References

[Augm84] 'Augmented version of make', in *Unix System V - release 2.0 support tools guide*, AT&T, April 1984.

[Feld79] Stuart I. Feldman, 'Make - a program for maintaining computer programs', *Software - Practice and Experience*, **vol. 9**, pages 255-265, (April 1979).

[Fowl85] Fowler, G. S., 'A fourth generation make', *Proceedings of the USENIX 1985 Summer Conference*, Portland, Oregon, pages 159-174, (June 1985).

[Gedy84] Gedye, D., *Cooking with CAD at UNSW*, Joint Microelectronics Research Center, University of New South Wales, Sydney, Australia, 1984.

[Hirg83] Hirgelt, E., 'Enhancing make or re-inventing a rounder wheel', *Proceedings of the USENIX 1983 Summer Conference*, Toronto, Ontario, Canada, pages 45-58, (July 1983).

[Hume87] Hume, A., 'Mk: a successor to make', *Proceedings of the USENIX 1987 Summer Conference*, Phoenix, Arizona, pages 445-457, (June 1987).

[Somo87] Zoltan Somogyi, 'Cake: a fifth generation version of make', *Australian Unix system User Group Newsletter*, **vol. 7**, pages 22-31, (April 1987).

[Wald84] Kim Walden, 'Automatic generation of make dependencies', *Software - Practice and Experience*, **vol. 14**, pages 575-585, (June 1984).

[Wein71] Weinberg, G., *The psychology of computer programming*, page 288, Van Nostrand Reinhold, New York, 1971.