OMake: Designing a Scalable Build Process*

Jason Hickey and Aleksey Nogin

Computer Science Department,
California Institute of Technology,
{jyh, nogin}@cs.caltech.edu

Abstract. Modern software codebases are frequently large, heterogeneous, and constantly evolving. The languages and tools for software construction, including code builds and configuration management, have not been well-studied. Developers are often faced with using 1) older tools (like make) that do not scale well, 2) custom build scripts that tend to be fragile, or 3) proprietary tools that are not portable.

In this paper, we study the build issue as a domain-specific programming problem. There are a number of challenges that are unique to the domain of build systems. We argue that a central goal is compositionality—that is, it should be possible to specify a software component in isolation and add it to a project with an assurance that the global specification will not be compromised. The next important goal is to cover the full range of complexity—from allowing very concise specifications for the most common cases to providing the flexibility to encompass projects with unusual needs. Dependency analysis, which is a prerequisite for incremental builds, must be automated in order to achieve compositionality an reliability; it also spans the full range of complexity.

We develop a language for describing software builds and configuration. We also develop an implementation (called OMake), that addresses all the above challenges efficiently and portably. It also provides a number of features that help streamline the edit/compile development cycle.

OMake is freely available under the GNU General Public License, and is actively being used in several large projects.

1 Introduction and Problem Definition

The general objective of a build system is to automate the construction of a software product from a set of inputs. For example, the product might be an application executable, where the inputs are the source files; in this case, the executable is usually constructed by compiling and linking the source files. The software product might also have several parts, for example it might be a web site that is to be constructed from a set of source scripts and document files. The process of generating the product from the inputs is called *building* the product; each run is called a *build*; and the tool used to manage the build is called a *build system.*

* An extended version of this paper is available as a California Institute of Technology Technical Report CaltechCSTR:2006.001.

L. Baresi and R. Heckel (Eds.): FASE 2006, LNCS 3922, pp. 63–78, 2006.

1.1 Specification of the Build Process

In general, we will assume that both the inputs and the results of a build are represented as files. A complete build is usually composed of several steps, including actions like 1) compiling source files, 2) linking object files to construct libraries or executables, 3) generating documentation, 4) and packaging the results. In the interest of modularity, and also to allow incremental builds, we would like to specify a build in terms of steps, where each individual step involves executing a script or application, such as a compiler, to generate a set of output files from a set of input files. We call the output files *targets*; the input files are called *dependencies*. In some cases we also refer to named tasks as targets.

We assume that each step can be specified as a *build rule* with the following parts.

- a set of *targets* to be built,
- a set of *dependencies*,
- a set of files, called *side-effects*, that may be modified during execution of the rule; the targets are always side-effects of the rule,
- a function or script, called the *build commands*, that may be called to construct the targets from the dependencies. We say that a rule is *executed* when its build commands are executed.

For the purpose of incremental builds, smaller steps are often preferable.

We further classify the dependencies: *explicit* dependencies are part of the rule specification, and *implicit* dependencies are all other factors that may affect the outcome of a rule execution. For example, if a dependency file.c contains a line #include "file.h", then file.h is an implicit dependency of the rule if it is not already explicit. Strictly speaking, the compiler binary is also a dependency.

A *build specification* for a project defines a set of build rules that form a *dependency graph*. The leaves of the graph are the files that do not appear as targets; they correspond to source files. A target is considered *up-to-date* if all of the following hold:

1. it has been built at least once, and the most recent rule execution was successful,
2. all of its dependencies are up-to-date,
3. the dependencies and commands have not changed since the previous time it was built,
4. none of the effects (the side-effects and the target itself) have changed since the previous time the rule was executed.

Leaf files are always up-to-date, if they exist. The task of a build system is to bring the desired targets up-to-date by executing a (preferably minimal) set of rules.

This definition of "up-of-date" has several noteworthy properties. First, it does not refer to such unreliable properties as file timestamps; even a file with a very recent timestamp may be considered out-of-date if it was produced by something external to the build system and the build system has no way of

knowing for sure whether it was produced correctly. Second, the definition explicitly states that the target has to be rebuilt when the corresponding command changes. This means that when a user updates the build configuration (for example, the compiler flags), the build system will be required to rebuild all the targets that have to be built differently under the new configuration (again, regardless of how fresh the timestamps are). Finally, it allows for not propagating the changes that do not affect the outcome. For example, if a program myprog depends on file.o, which in turn depends on file.c, then insignificant changes to file.c (such as a white space or comment change) will cause only file.o to be rebuilt, but myprog does not have to be rebuilt unless file.o changes too.

1.2 Constraints and Requirements

Not every build graph defines a well-formed project. We impose the following constraints.

1. The dependency graph must be acyclic.
2. Each target is the target of exactly one rule.
3. If the transitive dependencies of a rule are up-to-date, then executing a rule successfully brings the targets of the rule up-to-date.
4. Rules may be executed in parallel if their side-effects do not intersect.

The acyclic requirement is used to help ensure termination of the build. If termination is not a concern, the acyclic requirement can be relaxed, and the build process becomes a fixpoint calculation. The second and third requirements are constraints on the programmer of the build. In order for the build specification to be robust and maintainable, there must be exactly one way to build each target, and the command to build it must be correct. The final requirement is for performance and is not strict. If interferences between rules are not specified accurately, the build user is limited to serial rule execution.

2 Design Requirements

As specified, the build system implementation might appear to be a straightforward task of providing a solver that takes a dependency graph and executes rules in some order to bring all targets up-to-date. Indeed, the solver is reasonably straightforward and algorithmically unsurprising. The interesting issues are on either side of it. First, how should the dependency graph be specified; and second, how may build commands be specified and executed portably? Before answering these questions, we introduce the design requirements.

- There should be a single build specification (perhaps in multiple files) for an entire project.
- Build specifications must be configurable. In other words, it should be possible to parameterize them by properties like project requirements and versions, by the availability of tools, by the target platform, and other properties of the build environment.

- The build specification should be stable relative to project evolution. That is, maintenance of the build specification should be insignificant relative to the project development and maintenance effort.
- Specifications must be compositional. That is, the sub-specifications for the components of a project may be expressed independently, and combined without interference.
- The build system should be general, not specific to a particular application domain.
- The system should not require that all dependencies be stated explicitly; instead it should provide an automated mechanism for discovery of implicit dependencies.

These requirements rule out some naive solutions. For example, requiring that the programmer provide the full, literal, dependency graph is not possible because for large projects the specification would be large, repetitive, and difficult to maintain; in addition the specification would not be parameterizable or configurable.

What is clear is that the language of the build should be general enough to support specification definitions that are both concise and configurable. One approach is to use a general-purpose scripting language to construct the dependency graph. This is the approach taken, for example, in both Cons [10] (which uses Perl) and SCons [8] (which uses Python). However, the expressive power of these languages often acts to tempt programmers away from simple declarative specifications. In particular, there is no guarantee of compositionality in these languages; the build specifications in different parts of a project may exhibit unforeseen interference unless programmers are strictly disciplined.

We take the opposite approach, working from the bottom up, including features in the language only when they satisfy the design requirements. In the next section, we begin the task with rule specifications, and work towards each of the design goals.

3 Language

3.1 Rule Specifications

The primary goal of a build specification is to define a dependency graph, which is a set of build rules. A build rule has a set of targets, dependencies, side-effects, and some build commands. For this purpose, the rule syntax used in the ubiquitous Unix make program[1] is ideal. A rule has the following form, where the targets, dependencies, and side-effects are lists of filenames, and the commands define a script to build the targets from the dependencies (we will use standard "command-line" syntax for the commands). The notation $[\cdots]$ indicates that the syntactic form is optional—the brackets are not part of the concrete syntax.

```
targets: dependencies [ :effects: side-effects ]
    commands
```

[1] Although indentation is not restricted to tabs.

For example, using a standard shell syntax for the commands, the following rule specifies how to construct a grammar implementation from its specification using the yacc application.

```
grammar.h grammar.c: grammar.y :effects: y.tab.c y.tab.h
    yacc grammar.y
    mv y.tab.c grammar.c
    mv y.tab.h grammar.h
```

We call these *explicit rules* because the targets, dependencies, and side-effects are all specified with explicit filenames.

Implicit rules. One of the main issues with explicit rules is that they are overly verbose and repetitive. For example, a project might have many C source files that are all the be compiled with the cc compiler, and it is inefficient to define a separate rule for each file. *Implicit rules* address the issue by defining *rule patterns*. In an implicit rule, the % character represents a "wildcard" pattern that stands for an arbitrary string of text. All occurrences of the wildcard represent the same string throughout a rule (in other words, the wildcard is universally quantified).

With implicit rules, generic rules can be specified by pattern matching. For the example in the previous paragraph, the following implicit rule specifies that the cc compiler may be executed to compile any file with suffix .c, producing a file with a .o suffix.

```
# Use cc to compile a .c file, producing a .o file
%.o: %.c
    cc -c -o %.o %.c
```

Rule selection. One of the requirements of the dependency graph is that there be exactly one rule for each target. For explicit rules, it is an error for a file to occur as the target of more than one explicit rule. However, with implicit rules, it is desirable to allow multiple potential matches (although at most one rule may be selected for use in the dependency graph). For example, one might define an implicit rule that specifies a "default" build action, and then define explicit rules for any special cases where the default is inadequate.

We define rule selection by policy. Given a specific target with name T,

- if T is the target of an explicit rule, that rule is used,
- otherwise, if T matches an implicit rule in scope (we define the concept of scope in the next section), then the most recently defined implicit rule that matches T is chosen,
- otherwise, the source file T must exist, and it is a leaf in the dependency graph.

3.2 Variables, Scoping, Compositionality, and Parameterization

Even with implicit rules, there is a great deal of duplication. Many related rules (such as compiling and linking rules) will want to ensure that they are constructed by the same application or compiler, with the same options. In addition,

it is usually desirable to allow the specification to be easily reconfigured. The obvious solution here is to introduce variables that represent values that may be used in multiple rules. Once again, we adopt the standard syntax, using the notation $\$(\cdots)$ for variable references, and single-line definitions using =. For example, the following two rules specify that program p is generated by compiling and linking two files x.c and y.c. The compiler is defined with the variable CC, and the options are defined with the CFLAGS variable.

```
CC = gcc
CFLAGS = -g
%.o: %.c
    $(CC) $(CFLAGS) -c -o %.o %.c
p: x.o y.o
    $(CC) $(CFLAGS) -o $@ $+
```

For convenience, within the rule commands, the variable $@ is defined as the target of the rule, and $+ are its explicit dependencies.

Scoping and compositionality. Before the introduction of variables, build specifications were purely declarative and compositional. That is, suppose two developers had defined build specifications for two sub-projects. We could combine their build specifications simply by concatenating them. As long as the two did not share targets (which might violate target uniqueness), the combined specification would be valid and correct relative to the sub-project specifications.

With variables, the situation changes. Suppose the concatenated specification happens to share variables, as follows.

```
# Developer 1
CC = cc
CFLAGS = -g
file1.o: file1.c
    $(CC) $(CFLAGS) -c file1.c
```

```
# Developer 2
CC = gcc
CFLAGS = -O6 # (unsafe in general)
file2.o: file2.c
    $(CC) $(CFLAGS) -c file2.c
```

In the concatenated specification, the CC and CFLAGS variables are defined twice—which value is the right one? Furthermore, it is known that the -O6 option is a bit dangerous with gcc. If the values defined by developer 2 "win," then the code for developer 1 might be compromised.

So far, we have more-or-less adhered to tradition. The GNU version of make [9] includes explicit rules, implicit rules, and variables in the form we have described. However, at this point we take a radical departure. In make/GNUmake, one of the values would "win," and for example, CFLAGS would be either -g (debug mode) or -O6 (unsafe-optimizing mode) for the entire project, with unintended consequences for the other developer.

Our approach is radical to some and natural to others. As we see it, both developers are right, and the correct interpretation is the pure one ("purity" in the sense of functional programming). That is, the definition of a variable, like CFLAGS = -O6, is a *definition*, not an *assignment*. Each rule is a *closure* (another

concept from functional programming) that pairs the rule with its environment. The file file1.c is compiled with cc -g, and the file file2.c is compiled with gcc -O6.

The choice of pure specifications has two major consequences. As a benefit, specifications are always compositional, because there is no way for one part of a project to interfere with another by side-effect. In consequence, there is no easy way in general for a programmer to collect global information through side-effects on a global variable. As an aside, the pure *vs.* impure debate has been present for many decades, but we argue that for build specifications in particular, impure programming is more often by accident than by intention, and the benefits of compositionality far outweigh the benefits of shared, mutable state.

3.3 Programming and Configurability

At this point, we have explicit rules, implicit rules, and variables. While this might encompass many applications, it is still insufficient. For example, while implicit rules can be used to describe a great deal of build procedures, they cannot describe rules in which the dependency names are not literally textually related to the targets. In addition, build specifications are not easily configurable, because there is no way to state that the set of build rules depends on compile-time configuration parameters.

To address these issues, we introduce a simple core programming language with functions, function application, and conditionals.[2] The syntax of our language is shown in Figure 1. It is still modeled on the language for GNU make, with user-defined functions. Here, we use braces $\{p\}$ to represent block structure as defined by indentation—that is, the program p must be indented from the enclosing context; the braces do not appear in the concrete syntax.

The structure of the language is quite simple. A program is a sequence of statements, and each statement is either a command line to be executed by the shell, or a language directive such as a variable/function definition, function application, conditional expression, or rule definition. A rule definition may include options, like the :effects: we have seen earlier. There are other options as well, including :value: for dependencies on computed values, rather than files.

Functions, simplification, and configuration. The use of functions can often significantly simplify build specifications. As an example, let's consider the problem of building a static library from a set of object files. The rules to do this differ slightly between Win32 and Unix platforms, so we would like to define a function that computes the appropriate build rule based on the platform. Consider the following program.

[2] Expressivity is a double-edged sword. The traditional make/GNUmake programs do not include user-defined functions, most likely because such languages are Turing complete—even termination is not decidable. However, the loss of completeness in these languages has a heavy cost, leading many programmers to resort to meta-programming, such as imake [2], autoconf/automake [6, 7], or other build specification generators.

e ::=		expressions
	text	text
|	$\$(v)$	variables
|	$\$(v\ e_1, \ldots, e_n)$	function application
|	$e_1 e_2$	concatenation
s ::=		statements
|	e	shell commands
|	$v = \text{e}$	variable definitions
|	$v(v_1, \ldots, v_n) = \{p\}$	function definitions
|	$v(e_1, \ldots, e_n)$	function applications
|	$\textbf{if} e\{p_1\}\,\textbf{else}\{p_2\}$	conditionals
|	$\textbf{section}\{p\} \mid \textbf{export}$	scoping directives
|	$e_{targs} : e_{deps}\,(\textit{:option:}\ e_{opt})^*\ \{p\}$	rule definitions
p ::=		programs
|	ϵ	empty program
|	$\begin{matrix} p \\ s \end{matrix}$	sequencing (line endings act as sequence separators)

Fig. 1. The build programming language

```
# Platform-independent library construction
StaticLibrary(target, deps) =
    if $(equal $(OSTYPE), Win32)
        ofiles = $(addsuffix .obj, $(deps))
        $(target).lib: $(ofiles)
            lib /Fo$(target).lib $(ofiles)
    else
        ofiles = $(addsuffix .o, $(deps))
        $(target).a: $(ofiles)
            rm -f $(target).a
            ar cq $(target).a $(ofiles)
# An example library with 3 object files
StaticLibrary(mylib, file1 file2 file3)
```

The StaticLibrary function takes two arguments. The target is the name of the static library, and the deps are the object files to be included. Both arguments are provided without suffixes, since the actual file suffixes depend on the platform. The first step in the function is to determine the platform using a conditional. The OSTYPE variable defines the name of the platform, and the builtin equal function is used to determine if the platform is Win32. If so, the library has the .lib suffix, the object files have the .obj suffix, and the application for constructing the library is called lib. The addsuffix function is used to append the suffix to each of the names in the deps argument. The other case is similar.

The construction of a static library is now reduced to a single function call that specifies only the name of the library and its dependencies, with the usual benefits. The platform-dependent configuration is now located within a single

function, and the remainder of the build specification can be significantly simplified and need not be cluttered with platform tests. By defining a general set of such functions in a shared standard library, we obtain very concise specifications for simple projects.

Scoping and block structure. The introduction of block structure imposes a new twist on scoping. For example, in the previous section the StaticLibrary function defined the ofiles variable (twice). According to our scoping policy, these two definitions do not conflict since rules always use the most recent variable definitions in scope. The next question is whether the ofiles variable remains defined after the StaticLibrary function is called. Clearly, doing so would be undesirable because it would violate the abstraction provided by the function.

We adopt the usual scoping policy where each block in the program defines a scope, scopes are nested, and variables defined in inner scopes are not visible to outer scopes. Syntactically, blocks are determined by indentation, so for our example, the ofiles variable is *not* defined after the StaticLibrary function is called, because it is defined within an inner scope.

The **section** allows the introduction of a new nested block (with its corresponding scope), and it is frequently used to isolate variable definitions that are valid in only part of a project. For example, the following code fragment illustrates the common usage. The syntax CFLAGS += -g is equivalent to the expression CFLAGS = $(CFLAGS) -g, so the inner value of CFLAGS is "-O -g".[3]

```
CFLAGS = -O # Pass the "optimizing" flag to the C compiler
...
section
    CFLAGS += -g # Also add the "debugging" flag for the following targets
    ...rules...
# CFLAGS has the original value "-O"
...rules...
```

When combined, purity and strict scoping can be awkward. For example, consider the following program fragment, where the intent is to define the name of the C compiler and its default options on a platform-dependent basis.

```
# The compiler and flags are platform-dependent
if $(equal $(OSTYPE), Win32)
    CC = cl
    CFLAGS = /DWIN32
    OSUFFIX = .obj
else
    CC = gcc
    ...
```

[3] In our implementation, the system state including environment variables and the current directory are handled similarly—the extent of modifications is limited to the current scope.

Unfortunately, this program does not work as expected because the variable definitions are not visible outside the conditional. The **export** directive is designed to export variable definitions from an inner scope to its enclosing scope, canceling the nested scoping status of the block. In the example, the problem is solved by placing an **export** as the final statement in the branches of the conditional.

```
if $(equal $(OSTYPE), Win32)
    CC = cl
    CFLAGS = /DWIN32
    OSUFFIX = .obj
    export
else
...
```

3.4 Functions and Dynamic Scoping

One configuration pattern that arises often is to define the parameters of a project as variables, using the variables to define the rules that describe how to build the project. For an example, let's consider the rules for building applications in Objective Caml [5], which have the following general form.

```
OCAMLC = ocamlc # Byte-code compiler
OCAMLCFLAGS = # Compiler options (initially empty)
# Compile an OCaml file
%.cmo: %.ml
    $(OCAMLC) $(OCAMLCFLAGS) -c %.ml
# Link a program
OCamlProgram(target, deps) =
    cmofiles = $(addsuffix .cmo, $(deps))
    $(target): $(cmofiles)
        $(OCAMLC) $(OCAMLCFLAGS) -o $(target) $(cmofiles)
```

In this definition, we intend these rules to be the *default* rules. For example, even though the default compiler options OCAMLCFLAGS are empty, sub-projects should be able to redefine the variable if they require particular options. This presents a problem because, as we have stated, rules use the "most recent" definitions for variables, and these definitions are apparently fixed. Furthermore, the number of parameter variables can be quite large, and it would be unreasonable to require programmers to memorize them all.

The solution here is to use a definition of "most recent" as the most recent *dynamic* definition, not the most recent static one. That is, we adopt the use of dynamic scoping, rather than static scoping. With dynamic scoping, users of a build library need only be aware of the variables that need to be specialized. For example, the following code-fragment illustrates the temporary redefinition of the OCAMLFLAGS variable. In this case, the OCamlProgram is called in a context where the "-g" options has been added to OCAMLCFLAGS.

```
...
section
    # Compile the program with "-g" flag
    OCAMLCFLAGS += -g
    OCamlProgram(myprog, file1 file2 file3)
```

3.5 Automated Dependency Analysis

One of our design objectives is that it should be possible to automate the inference of implicit dependencies. There are several reasons, but the most important is that rule re-use becomes much more difficult when every rule is required to state all of its dependencies explicitly.

Implicit dependencies arise through many factors, including references to files in source code or applications, and they may change frequently as a project is developed. Traditionally, programmers have used ad-hoc meta-programming techniques, using a tool/compiler to generate the set of implicit dependencies, and then grafting them into the build specification textually.

In fact, the tools for dependency analysis already exist in many cases, and it takes very little for the build system to support them. To illustrate, the following program fragment specifies a rule for dependency analysis of C program files. The .SCANNER target is a directive, in this case indicating that the implicit dependencies of an object file can be extracted by compiling the C source file with the -MM option.

```
.SCANNER: scan-%.c: %.c
    $(CC) $(CFLAGS) -MM %.c
%.o: %.c :scanner: scan-%.c
    $(CC) $(CFLAGS) -c -o %.o %.c
```

The target of a scanner rule, in this case scan-%.c, is called a *scanner-target*, and represents the dependencies generated by the build commands. The command itself prints the dependencies (for one or more targets) to its standard output in make format.

Given a normal rule with targets targets and scanner dependencies deps, the complete set of dependencies is the union of the explicit dependencies, the dependencies generated by the scanner rules for each individual dependency, as well as the scanner targets themselves, or

$$\textit{explicit-dependencies} \quad \cup \quad \textsf{deps} \quad \cup \quad \bigcup_{d \in \textsf{deps}} \textit{scanned-dependencies}(\textsf{targets}, d).$$

3.6 Managing Subprojects

Our final design goal is that there must be a single build specification for an entire project. In most software projects, the software codebase is divided among several subdirectories, often, but not necessarily, along the lines of the software components. Similarly, it is impractical for the build specification to be placed

in a single file—instead it is more desirable to partition the specification along directory lines.

We adopt guidelines as follows. Each project must have a single directory, called the "root directory" (this is usually the root directory of the project). The root directory contains a file named OMakeroot that defines the build specification. Each build file may contain references to other directories of the project using a rule of the form .SUBDIRS: $dir_1, \ldots, dir_n$, where each subdirectory $dir_1, \ldots, dir_n$ defines its own build specification in a file named OMakefile.

Semantically, a .SUBDIRS directive acts as program inclusion in a nested scope. That is, variables and rule definitions are passed down to subdirectories, but definitions within a subdirectory are not propagated back to the parent. This prevents interference between the build specifications in separate subdirectories, and preserves compositionality. In addition, the ability to inherit values means that parent directories can define default behavior that can be specialized within the subdirectories.

3.7 Language Summary

At this point, it is worthwhile to revisit the design goals to see whether we have achieved our objectives. One of our primary objectives is compositionality, which we help ensure through the use of a pure language with well-defined scoping rules (even parts of the system state are treated purely). While it *is* possible for a programmer to achieve interference externally, for example through the filesystem, the risk of inadvertent interference is greatly reduced.

Another of our goals is configurability and generality. In this case, although the language is designed specifically for builds, it is general enough to cover a wide range of tasks. The expressivity and simplicity of the language also help in maintaining the build system. We have developed several large projects using a variety of build systems including GNU make, Cons, and SCons. In our experience, specifications based on the designs presented here are significantly more concise and easier to maintain. In addition, the .SUBDIRS approach to linking subprojects (also a feature of the Cons and SCons systems), has been enormously helpful for constructing simple, maintainable build specifications.

Finally, automated inference of implicit dependencies is important for ensuring consistency and accuracy of builds. Our approach allows the leveraging of existing dependency analyzers. In addition, the fact that the .SCANNER rules have the same properties as the normal build rules allows the use of the full build specification machinery, leading to concise, simple, yet flexible, and powerful dependency analysis.

4 Implementation: The OMake Build System

We have implemented a build tool, called OMake, that follows the design requirements stated in the previous section. OMake is freely available at the OMake home page `http://omake.metaprl.org/` under the GNU General Public License, and is actively being used in several large projects.

The OMake system has four major components. The first of them is a compiler that translates OMake specifications from the source language to an *intermediate representation* (IR). The second part is the interpreter capable of evaluating OMake programs in their intermediate representation form. The third part of the system is the build manager that keeps track of targets, dependencies and build rules (in their explicit and implicit forms). The build manager is also responsible for instantiating the implicit rules when needed and scheduling the build commands for execution. Finally, the fourth component of the build system is the shell interpreter that is responsible for executing individual build commands, spawning external processes as necessary, and passing the control back to the IR interpreter for commands that are to be executed internally.

Compiler. When performing a build, the OMake systems begins by reading the specification files, starting with the OMakeroot file. Each file that is part of the project specification is parsed and the code is transformed into an *intermediate representation* (IR). The translation process is straightforward; the IR is a slightly simplified, slightly more explicit version of the source language; there is little difference between the two. For every specification file, the resulting IR is cached on disk; this allows skipping the parsing and compilation of that file on subsequent executions of OMake (provided the given file does not change, of course).

Interpreter. Once a specification file is read and its IR is generated (or loaded from the cache), the OMake interpreter *evaluates* the IR. For the most part, the interpreter implementation is fairly straightforward. One of the least trivial parts of the interpreter is its handling of the variable environments. The variable environment data structure is implemented as a functional immutable lookup table where updates operate by partially copying the table. Each time a rule (whether implicit or explicit) is encountered during evaluation, the interpreter passes the rule and the current variable environment to the build manager as a closure. The rule itself is not evaluated immediately. Thus, many versions of the environment will be saved by the build manager. This approach is made cost-effective through the use of functional data structures and extensive sharing.

Build manager. Once the build specification has been evaluated, control is passed to the build manager, which now has a complete collection of build rules. This is not yet a dependency graph because some of the rules are implicit, and automated dependency analysis has not yet been performed. The build manager is responsible for building the dependency tree and making sure that the goal targets[4] are brought up-to-date. Note that it is not always possible to fully discover the dependency tree before the build process starts—it may be the case that in order to discover the full set of implicit dependencies the build manager will need to execute a number of .SCANNER rules and those rules may in turn depend on targets that need to be built first. Because of this, the build manager constructs the dependency tree in parallel with the main build process.

[4] If the goal targets are not specified on the command line, the .DEFAULT target is the goal.

The build manager works by keeping a *worklist* of targets that need to be brought up-to-date, each marked with a *state* specifying how far along the build process for the particular target has progressed.

For each project, the build manager maintains a database where it stores information about each rule that was successfully executed, including the full set of dependencies, the commands text, side-effects, and targets. On successive runs, the database is used to determine which targets are already up-to-date.

The shell interpreter. For portability, and a degree of efficiency, OMake includes a built-in command interpreter (this *shell* can be used both as part of the build system and also as a standalone command interpreter). For the most part, this is a straightforward task, involving standard methods for process creation and management. However, one of our primary goals is to provide transparent portability—OMake should behave similarly on all platforms, and Win32 in particular is problematic. Among the difficulties are the lack of a fork system call, signals, process control, and terminal management. As a result, we developed a compatibility library that emulates most of these features. The use of functional, immutable data structures allows us to emulate the fork system call using threads without the need for address space duplication.

Built-in functions and standard library. OMake provides a broad set of functions for string and string arrays manipulation, input/output, including functions that mirror the Unix standard IO library.

In addition, OMake provides a set of higher-level functions that can be used in order to make a project work correctly on platforms like Win32 that do not provide Unix-style file processing tools. The toolset includes functions that mirror the core functionality of the Unix programs grep, sed, awk, and test.

As mentioned in Section 3.3, OMake includes a shared standard library of variables, functions and implicit rules that can be used to significantly simplify the build specifications for commonly used languages. Using the standard library, simple projects in languages like C, OCaml, and LaTeX can often be specified with just a few lines of code, sometimes as little as one line.

5 Related Work

On Unix systems, the make program [3], originally designed by Feldman in 1979, is the ubiquitous build system, especially for open source projects. There are at least two reasons why make retains its popularity. First, the model is extremely simple. Second, make does not require any particular project style, nor is it tied to any particular programming language.

Since 1979, software projects have grown tremendously in size, and new versions of make have been developed, notably GNU make [9]. The usual model with large projects is to split a project into multiple subdirectories, each with its own Makefile describing how to build files in that subdirectory. Although this is adequate in many cases, there are several issues with regard to scalability. First, dependency analysis is based on timestamps. When a file is modified in any way,

it may cause large portions of the project to be rebuilt even if the modification was innocuous (for example, a change to a comment). Second, dependency information is local to each Makefile in each subdirectory. One result of this is that the subdirectories must be built in a specific order, and the graph of dependencies between subdirectories must be acyclic. A third problem is that each Makefile may have to duplicate a substantial portion of code also used in other Makefiles (for example, one of the main features of the imake utility [2] is automated code duplication).

A number of tools address some of these limitations. The Jam build system [11] addresses the problem of cross-directory dependencies, by performing a global dependency analysis and automating it so that dependency information is always up-to-date. The Odin system [1] provides a truly global environment by using a single cache for each user. In addition, Odin provides extensive support for build variants based on the concept of a *derived object* that couples properties with a file name.

The Cons tool [10], written in Perl, the SCons tool [8], written in Python, go further by adding configurable dependency scanners and adopting the use of MD5 digests instead of file timestamps. In both Cons and SCons, the build system is closely integrated with the implementation language. That is, in order to use these tools effectively, one must write build specifications in Perl (for Cons), or Python (for SCons). One advantage is that the use of a general-purpose programming language can reduce the amount of code duplication. However, there are also disadvantages of these tools when compared with the make model. In make, there is a clear separation between the language of Makefiles and the implementation language (C). The make language was designed specifically for specifying builds—it is clear and concise, it is widely used, and it is easy to understand. The make language is better suited for build specifications than the Perl or Python languages.

The Ant build system [4] takes another approach, where the build specification is written declaratively as an XML specification. The Ant system allows extensions written in Java.

We believe that one of the principal features that distinguishes OMake from all of the above systems is the use of a language that is Turing complete, yet preserves compositionality of build specifications. In addition, the language preserves the basic spirit, model, and syntax of make, preserving its strengths while addressing limitations of scalability and reliability.

6 Future Directions

While we are very satisfied with the convenience provided by the OMake tool, there are a number of further enhancements that we are hoping to explore in the future.

One of them is support for fixpoint builds (where certain dependency cycles are allowed). Examples of projects that could benefit from this feature include building self-hosting compilers (when a new version of a compiler is built using

an older binary, one often wants to arrive at a fixpoint) and LaTeX compilation (one may need to re-run latex several times if the .aux file is changing).

In addition, we are investigating the use of modular namespaces as a means of further improving scalability.

While OMake has some initial support for distributing builds over several different computers, it will probably require additional work before it is fully usable.

Acknowledgements

The authors would like to thank the OMake user community at omake@metaprl.org for discussion and support for the OMake project. The authors would also like to thank the anonymous reviewers for their comments and suggestions.

References

1. Geoffrey M. Clemm. *The Odin System*. Morristown, New Jersey, 1994.
2. Paul Dubois. *Software Portability with Imake, Second Edition*. O'Reilly, 1996.
3. Stuart I. Feldman. Make-a program for maintaining computer programs. *Software - Practice and Experience*, 9(4):255–265, 1979.
4. Steve Holzner. *Ant: The Definitive Guide*. O'Reilly, 2nd edition, 2005.
5. Xavier Leroy. *The Objective Caml system release 1.07*. INRIA, France, May 1997.
6. David MacKenzie, Ben Elliston, and Akim Demaille. *Autoconf: Creating Automatic Configuration Scripts*. Free Software Foundation, November 2003. http://www.gnu.org/software/autoconf/manual/index.html.
7. David MacKenzie and Tom Tromey. *GNU Automake*. Free Software Foundation, September 2003. http://www.gnu.org/software/automake/manual/index.html.
8. Scons: A software construction tool. Home page http://www.scons.org/.
9. Richard M. Stallman, Roland McGrath, and Paul Smith. *GNU Make: A Program for Directing Recompilation*. Free Software Foundation, July 2002. http://www.gnu.org/software/make/manual/index.html.
10. Rajesh Vaidheeswarran. Cons: A Make replacement. Home page http://www.dsmit.com/cons/.
11. Laura Wingerd and Christipher Seiwald. Constructing a large product with Jam. In *ICSE '97: Proceedings of the SCM-7 Workshop on System Configuration Management*, pages 36–48, London, UK, 1997. Springer-Verlag.