On the Maintenance Complexity of Makefiles

Douglas H. Martin
School of Computing
Queen's University
Kingston, ON, CA
doug@cs.queensu.ca

James R. Cordy
School of Computing
Queen's University
Kingston, ON, CA
cordy@cs.queensu.ca

ABSTRACT

Build systems, the tools responsible for compiling, testing, and packaging software systems, play a vital role in the software development process. It is therefore important that they be maintained and kept up-to-date, which has been shown to be required for up to 27% of source code changes. Make, one such build tool, uses a declarative language written in Makefiles to interpret the build and so is not amenable to traditional complexity metrics. Because of this, most research into the complexity has focused on simple measures like the number of lines, targets, or dependencies. In this paper, we take a different approach and observe that a large component of maintenance is about understanding. Since that understanding is done by following links and searching for related parts, we propose a new complexity metric based on the number of indirections (i.e. any instance of a feature that requires the reader to look somewhere else). We present an empirical study of the indirection complexity of a set of almost 20k Makefiles from more than 150 open source projects.

CCS Concepts

•General and reference → Metrics; •Software and its engineering → Software maintenance tools; •Human-centered computing → *Open source software;*

Keywords

build systems, software metrics, software maintenance

1. INTRODUCTION

Build automation tools, or simply *build systems*, are the backbone of every software project. They compile the source code submitted by developers into an executable program. They run tests to assure that the code is correct. They compile documentation for manuals. And they package everything into a distributable application. Given the vital role these systems play in the development process, maintaining them is a high priority because without them the whole project grinds to a halt. Moreover, even minor errors in the build process can lead major difficulties in a software release. They must also constantly evolve and adapt to changes and additions to the source code, tests and target platform of the software.

As the size and complexity of build systems grows, the overhead of build system maintenance becomes an increasingly large part of the overall software maintenance task. A recent study by McIntosh et al. [17] shows that the build system can account to up to 31% of code files in a project, and that up to 27% source code changes in a C project require corresponding updates to build artifacts. Adams et al. studied the Linux build system through its many refactorings since inception [3], showing that the build system has grown to be a large and complex software system of its own that has grown to represent a large part of the Linux code base. Thus the maintenance of build systems is an increasingly important and difficult part of the overall software effort [4, 17].

Build languages such as Make [9] and Ant [7] have also been shown to be notoriously difficult to understand and modify [18, 22]. This is partly because they were designed for much smaller systems [9], and partly because of their lack of higher level abstraction features that are appropriate for the million-line software systems they have grown to be [17].

In this work, we propose a new theory of software complexity based on program understanding effort, and a metric to characterize it based on indirect references. We then apply this metric to a large set of Makefiles and compare it to traditional measures of complexity.

2. COMPLEXITY OF SOFTWARE MAINTENANCE

Existing measures of code complexity, such as McCabe's Cyclomatic Complexity [15], Halstead's metrics [10], and Function Point Analysis [6], are primarily aimed at development effort and at traditional algorithmic code. While they have been used to estimate predicated maintenance effort [5], they are not designed for that purpose and do not apply well to declarative languages like those used in build systems. When they have been modified to apply to build systems, they have been found to be indistinguishable from simply the number of lines [16]. As a result, efforts as estimating complexity of build systems have relied primarily on simple metrics such as number of files, lines, targets, or

WETSoM'16, May 16 2016, Austin, TX, USA

DOI: http://dx.doi.org/10.1145/2897695.2897703

```
objs/comprul.o : UNIX/cinterface comprul.c
    cc -c -O comprul.c; mv comprul.o objs/comprul.o

objs/compdef.o : UNIX/cinterface compdef.c
    cc -c -O compdef.c; mv compdef.o objs/compdef.o

objs/boot.o : UNIX/cinterface boot.c
    cc -c -O boot.c; mv boot.o objs/boot.o

objs/loadstor.o : UNIX/cinterface loadstor.c
    cc -c -O loadstor.c; mv loadstor.o objs/loadstor.o

objs/ident.o : UNIX/cinterface ident.c
    cc -c -O ident.c; mv ident.o objs/ident.o

objs/scan.o : UNIX/cinterface scan.c
    cc -c -O scan.c; mv scan.o objs/scan.o

objs/parse.o : UNIX/cinterface parse.c
    cc -c -O parse.c; mv parse.o objs/parse.o

objs/parsepf.o : UNIX/cinterface parsepf.c
    cc -c -O parsepf.c; mv parsepf.o objs/parsepf.o

objs/trees.o : UNIX/cinterface trees.c
    cc -c -O trees.c; mv trees.o objs/trees.o

objs/xform.o : UNIX/cinterface xform.c
    cc -c -O xform.c; mv xform.o objs/xform.o

objs/xformpf.o : UNIX/cinterface xformpf.c
    cc -c -O xformpf.c; mv xformpf.o objs/xformpf.o

(...)
```

(a)

```
all:
    cp -r ../generic/* .
    cp -r machdep/* .
    make -f Makefile_Generic SYS=$(SYS) AS="cc -c -m32"
        CC="cc -m32" LD="cc -m32"
clean:
    make -f Makefile_Generic clean
    rm -r -f TL* Makefile_Generic main.ch
```

(b)

Figure 1: These examples illustrate how current metrics used to measure complexity in Makefiles, like number of lines or dependencies, can be misleading. The example in a) has more dependencies and lines, but the example in b) hides its complexity in another Makefile that it calls recursively.

dependencies, and historical metrics such as build file churn [17, 3, 16].

We propose a new approach, designed to predict maintenance effort more directly. Studies of software maintenance indicate that the majority of maintenance effort is spent trying to understand the system [21]. While novices tend to read code locally and linearly, experts are more likely to follow links and related parts [11], and both kinds of maintainers spend most of their time navigating the software and searching for related items [20, 19, 13]. Many tools and interfaces have been designed to assist in these tasks, perhaps most successfully Hipikat [8] and Mylar [12], which has been adopted into Eclipse as Mylyn [23].

The bottom line is that software maintenance is all about the effort of code exploration and understanding, not so much about size or change. A very large Makefile can be very easy to understand (Figure 1a), and a very small file can be very difficult (Figure 1b). The difference between these two files is that the first is repetitive and self-contained – the programmer need look nowhere else to understand what is being built and under what conditions. By contrast, the build script in Figure 1b is dependent on a great deal of external information – in order to understand what is being built and when, the programmer must look into two different parts of the file system, must look to see how the Makefile is invoked elsewhere and with what parameters, must trace a recursive Make invocation, and must find and substitute overridden variables in the recursive call.

Each of these actions requires the programmer to look elsewhere to understand the meaning of the section of code at which they are looking. Every such reference disturbs the process and increases the difficulty of understanding when the script is being maintained. We call these references *indirections*, and theorize that as the number of them increases in a build script, the difficulty of understanding increases, and thus maintenance effort increases. More generally, every time a maintainer must look somewhere that is not where they are currently focussed in order to understand what they are doing, the cognitive overhead of understanding is increased.

Based on these observations, we propose a new complexity measure we call *indirection complexity*, which is calculated as the sum of all instances of such indirections. In a maintained software artifact, such as a build script, this can be approximated by identifying and counting occurrences of language features that result in indirection (that is, that require the programmer to look elsewhere in order to understand what is in front of them).

To put it formally, indirection complexity,IC, can be computed as

$$IC = \sum_{x=1}^{n} w_x i_x$$

for n indirect features, where i_x is the number of instance of indirect feature x and w_x is a weight associated with indirect feature n. For this work, we have kept all weights at 1. Further discussion and analysis would be necessary to find the ideal weights.

This measure has several advantages: it can be applied to any kind of programming artifact, including requirements, design documents, build systems, and source code; it is independent of programming language and paradigm; and it is focussed directly on estimating software maintenance effort rather than logical content.

In the remainder of this paper, we explore the application of the indirection complexity measure to Makefiles. We begin with a short reminder of the features of the Make-based build systems.

3. A BRIEF OVERVIEW OF MAKE

Make is one of the oldest and most commonly used build automation tools. There have been many implementations since Feldman's original proposal in 1979 [9], each with their own unique features and improvements, but our work focusses on GNU Make [2] because it is the most widely used.

Make processes what are known as Makefiles, which contain rules that tell Make what should be made (targets), when they should be made (dependencies), and how to make them (recipes). They take the form:

```
targets : dependencies
    [TAB] recipe
```

```
1 #This is an example of a Makefile
2
3 VPATH=/usr/lib/
4 vpath *.c /usr/src/
5
6 DEBUG=yes
7 OBJS := \
8 bin/main.o \
9 bin/foo.o \
10 bin/bar.o
11
12 ifeq ($(DEBUG),yes)
13 include build/flags_debug.mk
14 else
15 include build/flags.mk
16 endif
17
18 .PHONY: all
19 all: $(OBJS)
20     cc -o $(OBJS)
21
22 foo.o: foo.c foo.h
23     cc -c foo.c
24
25 %.o : %.c
26     +echo "Compiling... " $(basename $@)
27     $(CC) -c $(CFLAGS) $(CPPFLAGS) $< -o $@
28
29 .PHONY: clean
30 clean:
31     rm bin/*
```

Figure 2: An example Makefile.

Each target may refer to a file or, alternatively, may be simply an identifier such as **all**, **clean**, or **debug**. When a target is an identifier like this, it is called a phony target, and may be referred to elsewhere in the Makefile as a dependency of some other target. Targets are followed by an optional set of dependencies, which are other targets or files on which the target depends. If any of the dependencies has been updated since the target was last made, or if the target does not exist, then it must be remade. If there are no dependencies, the target is never out of date and therefore only made when called explicitly. Following the target and dependencies is a list of commands to update the target. The commands may be simple calls to a compiler, other shell commands, or external scripts.

As a more concrete example, we present the Makefile in Figure 2. It contains six rules on lines 18, 19, 22, 25, 29, and 30. The rule on line 22 builds `foo.o` if `foo.c` or `foo.h` change (i.e. if one of their timestamps is newer than `foo.o`). The shell command on line 23 is the recipe that is used to build `foo.o` The rules on lines 18 and 29 are special rule that inform Make that **all** and **clean** are phony targets and should not be considered as object files.

By default, Make will pick the first target in the Makefile to build automatically, if it is not explicitly given one. This target, usually a target called **all** as in Figure 2, becomes the default goal. To make the default goal, Make must first resolve its dependencies by searching for the rule associated with each. If the dependency is up-to-date, then it can proceed; but if it is not, it must resolve this new target first. It proceeds in this manner until the default goal can be built.

Make provides a number of other features like variables, functions, and conditionals to give developers more flexibility to write rules. We will look at some of those in more detail in the next section.

4. INDIRECT FEATURES OF GNU MAKE

Before we can calculate the indirection complexity of Makefiles, we must first determine which features should be considered indirections. In this section, we describe the features we consider to be indirections and how we arrived at that conclusion.

4.1 Dependencies

The number of *dependencies* is a common metric for measuring build complexity, and we include it in our metric as well. This is because each dependency specified in a rule represents another rule that must be found and, therefore, an indirection. Since we include this in our metric, it can dominate the complexity score and appear to be no different than only counting the dependencies. However, as we will see later, our metric provides more nuance, especially for Makefiles with no dependencies at all.

4.2 vpath, directory change (cd), paths

Makefiles depend on knowing the state the of the filesystem, or at least the portions of the filesystem relevant to the software project being built, in order to know when files exist or when they are out of date (i.e. their timestamp is older than that of the files on which they depend). When a user is reading a Makefile and trying to understand it, or why it does not work, they must be aware of where Make is looking for these target files.

Make's *vpath* directive (and less versatile *VPATH variable*) allow the Makefile author to select directories where Make should look for target files or dependencies. For example, they may specify where third party library files can be found on the user's system, or they may specify a folder in the project directory with common files so as to avoid having to use paths. In any case, the reader must be aware of these directories and redirect their attention to them when they are referenced. The example in Figure 2 uses both the `VPATH` variable (on line 3) and the vpath directive on line 4).

A similar argument can be made for directory changes and paths in filenames. When the working directory is changed, the reader must be aware of the files in the new directory. And when a path is specified with a target or dependency, the reader must redirect to this directory to check the state of the file. We also count paths in variable assignments because they are often referenced in target and dependency lists. The example in Figure 2 does not change directories in any of its recipes, but it does contain paths in the definition of the `OBJS` variable on line 7. This means the reader must add the "bin" directory to the list of places that must be monitored in addition to the "lib" and "src" directories specified in the vpath variable and directive.

4.3 Includes

Include statements provide a way for the author to break a Makefile into logical units (or illogical ones, depending on the author) and include them in other Makefiles, essentially inserting it at the point of the include statement. This creates a level of indirection where the reader must switch their attention to the included file. At worst, they must switch back and forth between files when searching for rules that update and outdated target. The example in Figure 2 contains two include statements on lines 13 and 15.

4.4 Conditionals (ifdef/ifeq)

Like most programming languages, Make provides a *conditional construct* to apply a portion of the code to be included or ignored based on some criteria (e.g. if a variable is defined, or if a variable has a particular value). Conditional statements are evaluated during a pre-processing step, which simplifies them to an extent. However, when reading them, the reader may still have to redirect their attention to somewhere else in the Makefile to continue reading. This could be the next line, or it could be halfway down the file. The example in Figure 2 contains a condition on line 12 that checks if the value of `$DEBUG` is "yes" and includes a different external Makefile depending on the outcome.

4.5 Variable References

In large software systems, files tend to have a large number of dependencies that need to be explicitly defined in the Makefile. One way for an author to manage this is to use *variables* that list common dependencies and reference those. This is a classic case of indirection because a reader must search for where the referenced variable was last assigned to decipher the meaning of the rule in which it appears. The rule on line 19 of the example in Figure 2 depends on a list of files defined in `$OBJS`. When reading that rule, the reader must either have remembered the list from earlier or go back and read it again. Either way this adds complexity to understanding the build.

Make also includes a set of built-in automatically assigned variables, or simply *automatic variables.* These variable change based on the context in which they are used. For example, the `$@` variable will always refer to the filename of the target currently being considered, the `$<` will always refer to the name of the first dependency, and the `$?` variable refers to the dependencies that are newer than the current target. An example of some of these can be seen in the rule on line 25 of Figure 2. These variables are different in that they are not assigned explicitly, and therefore do not require the reader to search for the line where they were assigned, but we still count them because they may require the reader to look at the rule header to remind them of what is currently being considered, and possible process lists of objects in their head (e.g. keep track of the current target or dependency being considered).

4.6 Function Calls

Function calls are another classic case of indirection. When a reader encounters a function call, they will likely have to search for the location of the function definition to continue tracing the code. While Make allows the author to write simple custom functions, most of the common operations have their own built-in function. For example, string functions (e.g. `findstring`, `patsubst`, `filter`) provide facilities for manipulating strings (including lists), and filename functions (e.g. `dir`, `addsuffix`, `wildcard`) provide facilities to manipulate filenames or query the file system. The example in Figure 2 contains a call to the `basename` function on line 26, which returns the name of the file it is given, without any extension.

While it is unlikely that the reader will need to consult the definition of any of these built-in functions to obtain the result, their attention may still be redirected. For example, the result of a string function like strip, which removes whitespace from a string, is trivial and does not affect the meaning of the Makefile. However, a function like wildcard, which can be used to search a directory for a list of files that match a pattern, can be unpredictable and require the reader to consult the file system. Despite this, we count all function calls as indirection in our complexity measure because it is difficult to determine whether or not some do cause indirection in every context. Also, as we have previously shown [14], functions are not used in many Makefiles anyway.

4.7 Recursive Make

One common, but discouraged, practice when writing Makefiles is to invoke Make with another Makefile from inside a rule. This allows the author to split the system into multiple smaller Makefiles for each subsystem and call them individually. Similar to include statements, this requires the reader to redirect their attention to a different file. As we have also seen in Figure 1, this can also be used to hide complexity.

5. DATASET AND METHODOLOGY

To see how our new complexity metric performs on Makefiles, we used the same analysis framework and dataset that we used in our previous study of Makefile features [14]. More details are available in that paper, but we provide an overview here.

Our dataset consists of almost 20k Makefiles spanning 270 different projects. These projects consist of the GNU library of applications modified between 2010 and 2015, the KDE library of applications, the Linux kernel, the Qt bindings for Ruby, and some other smaller systems. These projects were chosen based on how they generate Makefiles. Some systems, like the Linux kernel and some GNU projects, use Makefiles that were written manually by hand. Other GNU projects use Automake to automatically resolve dependencies and generate Makefiles using a template. KDE and others use the popular CMake tool to generate and configure build artifacts of any kind (in our case, we are only interested in how it generates Makefiles). Finally, the Qt project uses its own generator, QMake, to generate Makefiles, hence why we use the Qt Ruby bindings.

An overview of the dataset can be seen in Table 1. Individual Makefiles were sorted into generator categories by automatically detecting the comments added to the beginning of every generated Makefile (e.g. "# generated automatically by automake from Makefile.am...") or based on previous knowledge (e.g. we had to generate Makefiles ourselves for CMake and QMake), otherwise, we considered it to be hand-written. It should be noted that Automake templates allow hand-written Make commands to be included in them, meaning a project could have both kinds of Makefiles and we must rely on comments to tell us which is which. This also means that hand-written and Automake generated Makefiles share many similarities, as we saw in our previous study and will see in the following sections.

The core of our complexity analysis is a TXL grammar for GNU Makefiles. TXL is a source transformation language that allows us to define a grammar for any language, parse it, and manipulate that parsed data. This allows us to perform

Table 1: An overview of our dataset.

Generator	Projects	Makefiles	Avg Lines
Automake	147	1704	1153
CMake	80	8672	135
QMake	2	2460	2031
Hand	129	6683	49
All	**270**	**19519**	**433**

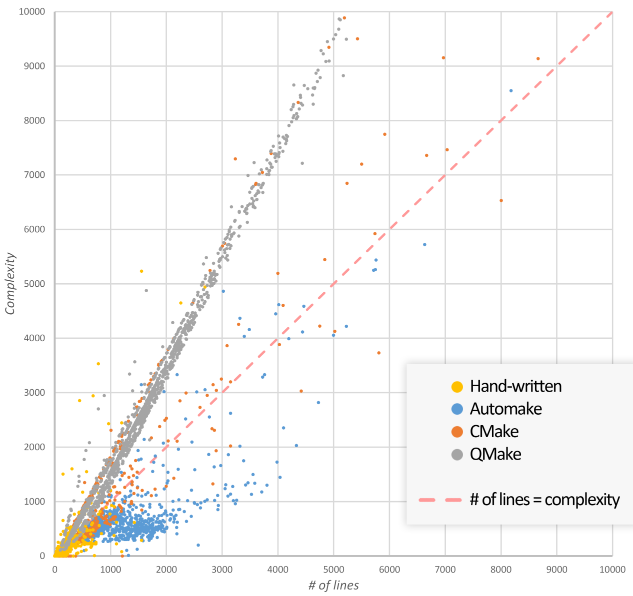


Figure 3: The indirection complexity of our dataset coloured by generator.

a static analysis of Makefiles – something not many others have done. The grammar is based on one from Tamrawi et al.[22] but yields a much more precise parse that allows us to do more.

After we parse the input Makefile, we extract and count the desired features then collect them in a spreadsheet. From there we add the feature counts together to get the complexity score. We present the results in the next section.

6. ANALYSIS

Since other complexity metrics for Makefiles have been shown to correlate with size, we graphed our results against the number of lines. This also gives us a way to normalize the data and lets us compare the complexity of files of different sizes (i.e. by dividing the total number of indirections by the number of lines, we get the average complexity per line, which we can see visually by plotting complexity vs the number of lines). Figure 3 shows the indirection complexity of our dataset with Makefiles less than 10k lines, coloured based on the generator used to create them.

In doing this, we can see that CMake and QMake Makefiles tend to be more complex, while Automake (Figure 4b) and hand-written (Figure 4a) Makefiles tend to be less complex. This observation is interesting when considering that we found that CMake and QMake use only the core features of Make (rules and variables) in our previous feature study. Only Makefiles written by hand or with Automake used some of Make's more advanced, complex features.

CMake and QMake also seemed to be more highly linearly correlated with the number of lines (R-squared of 0.87 and 0.99, respectively) than hand-written and Automake-generated Makefiles (both wirh an R-squared of 0.55), which makes sense since these generators use a limited set of templates that get duplicated as the size of the system grows. When we look closer, we can see bands of data points that roughly correspond to the 3 levels of Makefiles that CMake generates to recursively call one another [1]. QMake is less documented, but a similar design is likely responsible for the bands in it as well.

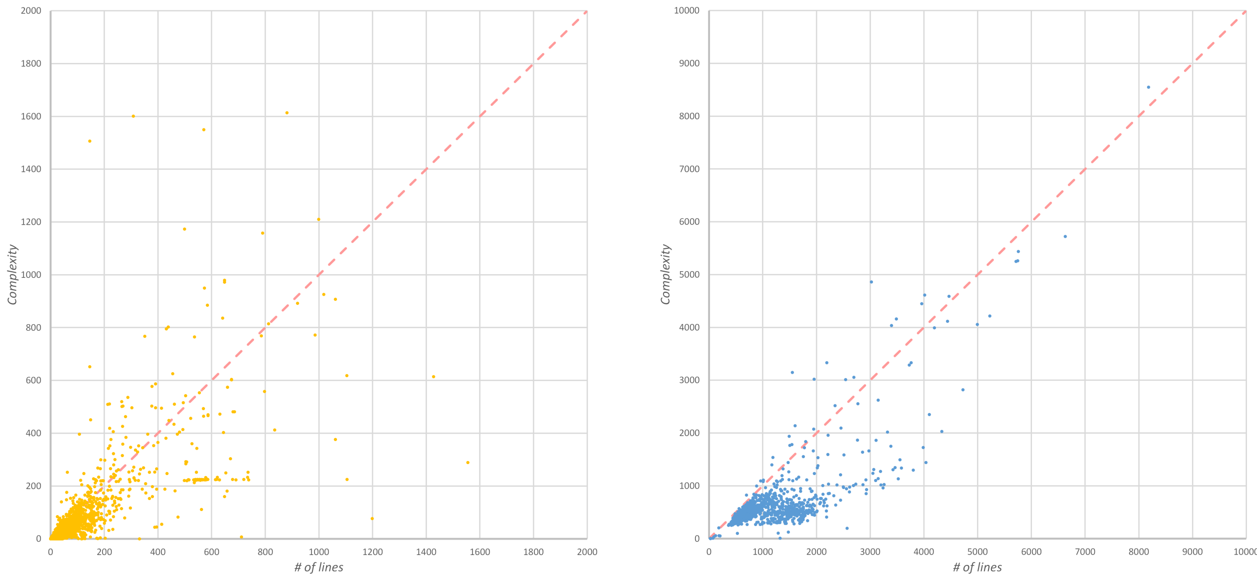
Generators like CMake and QMake are less interesting because the Makefiles they create are never meant to be read by developers. So the fact that their complexity can be accurately predicted by the number of lines is less meaningful. Our metric is better utilized on hand-written Makefiles that need to be debugged directly, as we can see when we graph their indirection complexity separately in Figure 4 where spread of data points is much greater and less predictable.

But the utility of indirection complexity is likely best seen through examples. Consider the Makefiles in Figure 5. Both are roughly the same size, but the one on the right is twice as complex (53 vs 23) according to the indirection complexity. And this is what we would expect when we read them. Figure 5a is a straight-forward test harness that runs a series of tests (i.e. `$PROGS`), while Figure 5b is a Makefile that is included in another Makefile several times to iterate through a list of items and output some variable assignments and rules. Figure 5a is quite easily understood. But, were it not for the the comments at the beginning, it would likely take a reader longer to fully understand Figure 5b due to all the references to variables that are not even defined in the same file. Note, however, that Figure 5a has more targets and dependencies, which would make it seem more complex under these measures.

Figure 5a also illustrates a potential weakness and threat to the validity of our approach. Because we use a static parse to count features, we would find that this Makefile contains 11 dependencies. However, looking at the rule to build `all`, you can see that it lists `${PROGS}` as a dependency and `${PROGS}` expands to include 22 files. So the Makefile actually contains 31 dependencies, not 11. In this way, we find that variables can be used to hide complexity from our analysis.

Another example of this can be seen in Figure 4a in the horizontal line of data points with a complexity of just over 200 and lines ranging from about 500 to 750. These Makefiles appear in a number of different GNU projects as part of building GNU `gettext` – a translation toolset for localizing software. Each of these Makefiles is configured from a seemingly hand-written template that add continuations (i.e. new lines) to some variable assignments. These assignments specify object files that become dependencies of the rules, but the build logic stays the same, as does the complexity.

It is unclear what the appropriate course of action is in this situation. One possibility is to weigh any dependencies containing variables against the length of the variable assignment or its complexity, thus making it more dynamic. This creates a whole new set of challenges around resolving variable references, which can be difficult to impossible. They may be assigned in multiple places, where they are overwrit-



(a) Hand

(b) Automake

Figure 4: The indirection complexity of hand-written and Automake-generated Makefiles.

```
noarg:
    $(MAKE) -C ../../

# The EBB handler is 64-bit code and everything
# links against it
CFLAGS += -m64

PROGS := reg_access_test event_attributes_test \
    cycles_test cycles_with_freeze_test \
    pmc56_overflow_test ebb_vs_cpu_event_test \
    cpu_event_vs_ebb_test \
    cpu_event_pinned_vs_ebb_test \
    task_event_vs_ebb_test \
    task_event_pinned_vs_ebb_test \
    multi_ebb_procs_test multi_counter_test \
    pmae_handling_test close_clears_pmcc_test \
    instruction_count_test fork_cleanup_test \
    ebb_on_child_test ebb_on_willing_child_test \
    back_to_back_ebbs_test lost_exception_test \
    no_handler_test cycles_with_mmcr2_test

all: $(PROGS)

$(PROGS): ../../harness.c ../event.c ../lib.c \
                ebb.c ebb_handler.S trace.c \
                busy_loop.S

instruction_count_test: ../loop.S

lost_exception_test: ../lib.c

run_tests: all
    -for PROG in $(PROGS); do \
        ./$$PROG; \
    done;

clean:
    rm -f $(PROGS)
```

(a)

```
# This file is included several times in a row,
# once for each element of $(iter-items).
# On each inclusion, we advance $o to the next element.
# $(iter-labels) and $(iter-from) and
# $(iter-to) are also advanced.

o := $(firstword $(iter-items))
iter-items := $(filter-out $o,$(iter-items))

$o-label := $(firstword $(iter-labels))
iter-labels := $(wordlist 2, \
                    $(words $(iter-labels)),$(iter-labels))

$o-from := $(firstword $(iter-from))
iter-from := $(wordlist 2,$(words $(iter-from)),$(iter-from))

$o-to := $(firstword $(iter-to))
iter-to := $(wordlist 2,$(words $(iter-to)),$(iter-to))

ifeq ($($o-from),$($o-to))
    $o-opt := -D$($o-from)_MODE
else
    $o-opt := -DFROM_$($o-from) -DTO_$($o-to)
endif

#$(info $o$(objext): -DL$($o-label) $($o-opt))

ifneq ($o,$(filter $o,$(LIB2FUNCS_EXCLUDE)))
    $o$(objext): %$(objext): $(srcdir)/fixed-bit.c
        $(gcc_compile) -DL$($*-label) $($*-opt) -c \
        $(srcdir)/fixed-bit.c $(vis_hide)

    ifeq ($(enable_shared),yes)
        $(o)_s$(objext): %_s$(objext): $(srcdir)/fixed-bit.c
            $(gcc_s_compile) -DL$($*-label) $($*-opt) \
                -c $(srcdir)/fixed-bit.c
    endif
endif
```

(b)

Figure 5: These examples from our dataset illustrate the advantages of indirection complexity over traditional metric like number of lines or dependencies. The Makefile in a) has more dependencies than the one in b) but is arguably less complex.

ten, conditionally assigned, or appended. Then there is the problem that arises from variables being assigned in other Makefiles, as is the case in Figure 5b.

7. CONCLUSION

Software maintenance is all about understanding code. Expert developers do this by following the indirections, which adds to the amount of information that they must keep track of and makes the code complex. We have measured this effort in Makefiles by calculating indirection complexity and have shown the advantages it has over other metrics that have been used, such as the number of lines and dependencies.

Our analysis was static, but a dynamic approach would be able to evaluate variables, functions, and other features of Make that would give a more accurate count of dependencies. It would also allow entire systems of Makefiles to be evaluated as a whole because it could resolve include statements that link them together. Another possibility to be explored is to assign weights to each indirect feature based on the amount of cognitive overhead. For example, a feature that requires the reader to look in a separate file may be weighted more than a feature that makes the reader look somewhere in the same file. This possibility was explored to some extent, but further work is needed to find the ideal weights.

We believe indirection complexity can be applied to other languages, but this remains to be seen. Even for Makefiles, user studies are needed to determine if our theory of indirection can actually predict maintenance effort or even perceived complexity.

8. REFERENCES

[1] CMake FAQ, http://www.cmake.org/wiki/cmake_faq.

[2] GNU make, www.gnu.org/software/make/manual/make.html.

[3] Adams, B., De Schutter, K., Tromp, H., and De Meuter, W. The evolution of the linux build system. *Electronic Communications of the EASST 8*, 0 (Oct. 2008).

[4] Adams, B., Tromp, H., De Schutter, K., and De Meuter, W. Design recovery and maintenance of build systems. In *IEEE International Conference on Software Maintenance, 2007. ICSM 2007* (2007), pp. 114–123.

[5] Ahn, Y., Suh, J., Kim, S., and Kim, H. The software maintenance project effort estimation model based on function points. *Journal of Software Maintenance and Evolution: Research and Practice 15*, 2 (2003), 71–85.

[6] Albrecht, A. J. Measuring application development productivity. In *Proceedings of the Joint SHARE/GUIDE/IBM Application Development Symposium* (1979), vol. 10, pp. 83–92.

[7] Apache Software Foundation. Apache ant, http://ant.apache.org, 2000.

[8] Cubranic, D., Murphy, G. C., Singer, J., and Booth, K. S. Hipikat: A project memory for software development. *IEEE Trans. Software Eng. 31*, 6 (2005), 446–465.

[9] Feldman, S. I. Make - a program for maintaining computer programs. *Software: Practice and Experience 9*, 4 (1979), 255–265.

[10] Halstead, M. H. *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc., 1977.

[11] Iio, K., Furuyama, T., and Arai, Y. Experimental analysis of the cognitive processes of program maintainers during software maintenance. *2013 IEEE International Conference on Software Maintenance 0* (1997), 242.

[12] Kersten, M., and Murphy, G. C. Mylar: a degree-of-interest model for ides. In *Proceedings of the 4th International Conference on Aspect-Oriented Software Development, AOSD 2005, Chicago, Illinois, USA, March 14-18, 2005* (2005), pp. 159–168.

[13] Ko, A. J., Myers, B. A., Coblenz, M. J., and Aung, H. H. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Trans. Software Eng. 32*, 12 (2006), 971–987.

[14] Martin, D., Cordy, J., Adams, B., and Antoniol, G. Make it simple - an empirical analysis of gnu make feature use in open source projects. In *Program Comprehension (ICPC), 2015 IEEE 23rd International Conference on* (May 2015), pp. 207–217.

[15] McCabe, T. A complexity measure. *Software Engineering, IEEE Transactions on SE-2*, 4 (Dec 1976), 308–320.

[16] McIntosh, S., Adams, B., and Hassan, A. The evolution of ANT build systems. In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR)* (2010), pp. 42–51.

[17] McIntosh, S., Adams, B., Nguyen, T. H., Kamei, Y., and Hassan, A. E. An empirical study of build maintenance effort. In *Proceedings of the 33rd International Conference on Software Engineering* (New York, NY, USA, 2011), ICSE '11, ACM, pp. 141–150.

[18] Miller, P. Recursive make considered harmful. *AUUGN Journal of AUUG Inc 19*, 1 (1998), 14–25.

[19] Singer, J. Practices of software maintenance. In *1998 International Conference on Software Maintenance, ICSM 1998, Bethesda, Maryland, USA, November 16-19, 1998* (1998), pp. 139–145.

[20] Singer, J., Lethbridge, T. C., Vinson, N. G., and Anquetil, N. An examination of software engineering work practices. In *Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative Research, November 10-13, 1997, Toronto, Ontario, Canada* (1997), p. 21.

[21] Soh, Z., Khomh, F., Gueheneuc, Y.-G., and Antoniol, G. Towards understanding how developers spend their effort during maintenance activities. In *Reverse Engineering (WCRE), 2013 20th Working Conference on* (Oct 2013), pp. 152–161.

[22] Tamrawi, A., Nguyen, H. A., Nguyen, H. V., and Nguyen, T. Build code analysis with symbolic evaluation. In *2012 34th International Conference on Software Engineering (ICSE)* (2012), pp. 650–660.

[23] Tasktop Technologies Inc. Mylyn, http://www.tasktop.com/mylyn, 2007.