Make It Simple – An Empirical Analysis of GNU Make Feature Use in Open Source Projects

Douglas H. Martin
Queen's University
Kingston, Ontario, Canada
Email: doug@cs.queensu.ca

James R. Cordy
Queen's University
Kingston, Ontario, Canada
Email: cordy@cs.queensu.ca

Bram Adams
École Polytechnique
Montréal, Québec, Canada
Email: bram.adams@polymtl.ca

Giulio Antoniol
École Polytechnique
Montréal, Québec, Canada
Email: antoniol@ieee.org

***Abstract*—Make is one of the oldest build technologies and is still widely used today, whether by manually writing Makefiles, or by generating them using tools like Autotools and CMake. Despite its conceptual simplicity, modern Make implementations such as GNU Make have become very complex languages, featuring functions, macros, lazy variable assignments and (in GNU Make 4.0) the Guile embedded scripting language. Since we are interested in understanding how widespread such complex language features are, this paper studies the use of Make features in almost 20,000 Makefiles, comprised of over 8.4 million lines, from more than 350 different open source projects. We look at the popularity of features and the difference between hand-written Makefiles and those generated using various tools. We find that generated Makefiles use only a core set of features and that more advanced features (such as function calls) are used very little, and almost exclusively in hand-written Makefiles.**

I. Introduction

Build automation tools, or simply build systems, are tools, or sets of tools, that are responsible for transforming source code into executable code and thus are a vital part of most large software development projects. These tools are used by every developer of the system in some way, yet are notorious for being difficult to understand and modify by non-experts.

The most widely used build automation tool, Make, was introduced by Stuart Feldman in 1977 [1] and has since had many implementations and improvements. The most popular implementation is GNU Make, which is included in most Linux distributions and continues to evolve every few years. For example, version 4.0 was released in October of 2013, adding two new operators and two new functions. One of those functions is the "guile" function, which allows Makefiles to define and run GNU Guile extensions – a whole new scripting language embedded inside of Make.

Make is often criticized for its difficulty to understand and general lack of debugging facilities [2], [3]. Although this difficulty has been studied by measuring the size of Makefiles [4], coupling of Makefile changes to source code changes [5] and analysis of the kinds of changes made to build files [6], Make has not been studied yet from the program comprehension point of view. Which language features are used the most? Is there a common set of features that GNU Make can be reduced to without losing functionality? By knowing how Makefiles are used, we can help make decisions about future versions and implementations, such as what features to add, remove, or just make easier to use.

In this work, we use TXL [7] – a specialized language for software analysis and transformation – with a custom-written Makefile grammar to extract and count instances of Make features. From an extensive and detailed inventory, we address the following research questions:

1) How frequently are Makefile features used?: What are the features that are absolutely essential to Makefiles? What are the least used or unused features? Are there features that could be removed from Make to make it less bloated?

2) Are features used differently in generated Makefiles?: How do Makefiles generated by Automake, CMake and QMake differ from those written by hand? Are generators using more advanced features?

3) To what extent are bad practices, specifically obsolete features and recursion, still in use?: The GNU Make manual specifies a number of still supported, but obsolete features. How often are they still used? Calling Make within a Makefile is considered harmful. How many Makefiles do this?

II. The GNU Make Language

To understand feature use in Make, we must first know how it works and the features it offers. There have been many variations of Make since its invention, but we will be focussing on the most popular variant, GNU Make.

GNU Make reads Makefiles, which contain a set of instructions that describe how to build a particular software project. This is done using rules, which specify targets (files) and how they should be built or rebuilt, as well as on which other files they depend. Make will build only what is necessary, which means only the target files whose prerequisites have changed will be rebuilt.

Make is run by using the `make` command in the directory with a Makefile. With no arguments, GNU Make will look for a file named GNUMakefile, makefile, or Makefile (in that order) and execute the first rule. By convention, that will be a rule to build the whole system. Of course, there are many command line arguments that can be used to customize the execution. For example, `make foo` will look for a rule to build foo, and `make -f build/Rules.mk` will execute the given Makefile instead of the default Makefile.

```
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 1. Example Makefile

The rest of this section describes the features we will be measuring and discussing in the remainder of this paper. We group the features based on the syntax of the grammar and their intention. Figure 1 shows an example Makefile that will be used for illustration.

A. Readability

1) Comments: Comments in Make are denoted by a "#" and can occur on their own line or at the end of a line (e.g., line 1 of Figure 1). There are no multi-line comments in Make.

2) Continuations: Continuations provide a way of breaking up long lines of Makefile text by placing a "\" at the end of a line to denote that it continues on the following line. For example, lines 7-10 of Figure 1 use continuations to break up the declaration of the OBJS variable onto multiple lines.

B. Rules

Rules are the heart of the Makefile. They tell Make what should be made (targets), when they should be made (prerequisites), and how to make them (recipes). They take the form:

```
targets : prerequisites
    [TAB] recipe
```

We will discuss each part in further detail, as well as some other aspects of Make rules, in the following sections.

1) Targets: Targets represent the artifacts to be built. Most of the time, that is the name of a file; however, a phony target, containing a string of characters not associated with a file, can be used to execute commands on request. Phony targets can be used to represent subsystems with one root "all" target, typically representing the build of the the whole system (lines 19-20 of Figure 1), or to represent a common build task, such as a "clean" target to remove all previously built files (lines 30-31).

2) Prerequisites: Prerequisites (or dependencies, as they are sometimes called) are other targets or names of files that are required to build the target(s) of the rule in which they are specified. The prerequisite list of a target T serves 2 purposes: First, the rule for each prerequisite of T is found and the corresponding recipe is executed if the prerequisite is out-of-date; second, if any of those prerequisites was found to be out-of-date, the recipe of T itself is executed to update the target(s). In certain cases, a target may not need to be updated if one of the prerequisites was found to be out-of-date. In such cases, these prerequisites – called order-only prerequisites – can be listed at the end of the prerequisite list after a pipe symbol ("|").

3) Recipe: A recipe is a list of commands to be executed. These commands, in theory, build the rule target(s) using the prerequisite files and should only update the target(s) and nothing else, but this is not guaranteed. For example, compiler commands or scripts can be invoked from inside a recipe, as in lines 20, 23, and 27 of Figure 1.

4) Special Targets: Make also defines a set of special targets that have special meanings. They all begin with a period (".") and are conventionally written in all caps. For example, the Makefile in Figure 1 contains 2 rules with the .PHONY target (line 18 and line 29). A rule with the .PHONY target means that the prerequisite(s) should not be considered as a filename, but rather as a subsystem or build phase. Without these rules, if there are files named "all" or "clean" in the same directory, Make would assume that those files are up-to-date and do not need to be rebuilt using a rule. In such a case, the "clean" target would therefore never be executed.

5) Recipe Flags: There are 3 prefixes that may be used in front of recipe commands. A minus sign ("-") indicates that Make should ignore any errors that the command may yield. A plus sign ("+") indicates that Make should execute this command even during a dry run, when the user has specified that it should not execute any commands (useful for debugging). And an at sign ("@") indicates that Make should not print the command itself to the standard output, only its programmed output. In the example in Figure 1, the recipe command on line 26 begins with a "+" to indicate that the echo command should always print the name of the file to be compiled, even when the user has specified otherwise.

6) Single vs. Double Colon: Ordinarily, a target should appear in only one rule. If not, Make will execute the recipe of the last rule and print an error message. However, there are some circumstances where a different recipe should be executed depending on which prerequisite has changed. In this

case, a double-colon is used in each rule rather than a single colon.

7) Pattern Rules: Pattern rules are used to define implicit rules in Make. Implicit rules describe a recipe for making certain types of files that are all built the same way. Make includes many predefined implicit pattern rules, such as the rule to compile .c files into .o files. A pattern rule looks the same as an ordinary rule, but it contains a target with a percent symbol ("%"), which matches any non-empty string. The "%" can then be used in the prerequisites list to match the same string. For example, lines 25-27 in Figure 1 show a pattern rule with target "%.o" to match all object files, while its prerequisite ("%.c") matches the corresponding C file.

8) Static Pattern Rules: Static pattern rules are the same as normal pattern rules, but operate on a static list of targets that come before the target pattern. Instead of searching the file system for targets matching the pattern, Make applies the pattern to the list of specified targets to extract the stem ("%") and find the prerequisites. If, for example, we only wanted the pattern rule on line 25 of Figure 1 to apply to foo.o and bar.o, we could add "foo.o bar.o:" before "%.o : ...".

9) Suffix Rules: Suffix rules are the older, now obsolete, way of defining implicit rules. They contain no prerequisites, and only a single target specifying one or two file suffixes. A single suffix rule contains one suffix and is a general rule applied to all targets with that suffix. A double suffix rule contains two suffixes together and apply to all targets that match the second suffix, with prerequisites that match the first suffix. For example, the pattern rule on line 25 of Figure 1 could be turned into an equivalent suffix rule by changing "%.o: %.c" to ".c.o:".

10) Recursive Make: A common but discouraged technique of creating a build system with Make is to split the system into multiple Makefiles, each responsible for building their own subsystem, that invoke each other as separate Make processes. These Makefiles are usually put into separate folders with the source code on which they operate, with a root Makefile at the top that calls these Makefiles (in recipe commands) to build the whole system. Invoking another Makefile within a Makefile like this is referred to as recursive Make, and it can have unwanted effects [8]. The problem with this is that, because separate Make processes are used, each Makefile has no knowledge of what is happening in the other Makefiles, and therefore may not rebuild all of the targets that require it. For example, if two of the Makefiles refer to the same physical file, it is possible that the second one rebuilds that file, making a new compiled version, while the first one has already made decisions based on the previously compiled version of that file, thus not rebuilding the targets on which it depends.

There are a number of solutions to fix this. The easy solution is to invoke the root Make more than once, thus giving the first Makefile the opportunity to notice changes made by the second. However, in general the number of times you would have to invoke Make to ensure everything was up-to-date varies depending on how many subsystems there are, not to mention the overhead involved (which can be huge for a large system). The best solution is to write one large Makefile or, to break it up, to use the include directive (described later) to put the subsystem Makefiles directly in the root Makefile. This method ensures that Make can find all of the required dependencies and rules it needs to build a complete and up-to-date system.

C. Variables

Like many languages, Make provides variables that can be referenced in target/prerequisite lists, recipes, or even other variables and functions. There are two flavours of variables: recursively expanded, which are evaluated as they are needed (i.e., when a reference is read), and simply expanded, which are evaluated when they are defined.

1) Assignments: Recursively expanded variables are defined using the "=" operator, or the define directive (see Multi-line Variables). Simply expanded variables are assigned with the ":=" or "::=" operators. There are also three special types of assignment operators: the "?=" operator will only set the variable if it has not already been set; the "+=" operator appends the value to the end of the variable; and the "!=" operator is used to assign a variable with the result of a shell command (e.g., "var != ls *.c"). In the example in Figure 1, the simply expanded OBJS variable is assigned on lines 7-10.

2) Variable References: Variables can be referenced anywhere in the Makefile (targets, prerequisites, recipes, variable assignments, etc.) by putting the variable name inside brackets with a "$" sign before it (e.g., "$(VAR)"). In the example in Figure 1, the OBJS variable is referenced in the prerequisite list and the recipe of the rule on line 19-20.

3) Multi-line Variables (Macros): Normal variables cannot contain newlines, but sometimes they are needed (for example, to define a common recipe segment). To do this, the define directive is used. They are defined in much the same way as regular variables but with the "define" keyword before the variable name, and "endef" after the value. For example, the definition below could be used to define the recipe on lines 26-27 of the example in Figure 1.

```
define recipe :=
+echo "Compiling... " $(basename $@)
$(CC) -c $(CFLAGS) $(CPPFLAGS) $< -o $@
endef
```

4) Automatic Variables: When using patterns, function calls, or variable references in rule headers, it can be impossible to tell which target or prerequisite is being evaluated. For this reason, Make provides a set of Automatic Variables to refer to them inside a recipe (or prerequisite list). Table I lists the automatic variables offered by Make. In the example in Figure 1, the `$<` and `$@` variables are used in the rule on lines 25-27 to get the name of the prerequisite and target, respectively, since they are not known until runtime.

D. Statements

In addition to rules and variable assignments, there are a few special statements that can be used in a Makefile. We will discuss these in this section.

Table I
AUTOMATIC VARIABLES

`$@`	the filename of the target (in the case of multiple targets, the target that forced the rule to run)
`$%`	the member name of the target if the file is part of an archive
`$<`	the first prerequisite
`$?`	all prerequisites that are newer than the target
`$^`	all prerequisites, with duplicates removed
`$+`	all prerequisites, with duplicates
`$|`	the order-only prerequisites
`$*`	the stem ("%") of the pattern matched in a pattern rule

1) VPATH Variable and vpath Directive: By default, Make will look for target and prerequisite files in the current working directory; if the file is not there, an error has occurred. The VPATH variable and vpath directive allow the build engineer to tell Make where to look for files that cannot be found in the current directory. It allows a directory to be specified for a particular pattern, where the directory is searched if the target file meets some criteria (e.g., "*.c"). The example in Figure 1 contains both the variable and directive. Line 3 assigns "/usr/lib/" to the VPATH variables, meaning that Make should look there for any file that it cannot find in the current directory. And Line 4 uses the vpath directive to tell Make to look in "/usr/src/" for .c files that it cannot find in the current directory.

2) Includes: The include directive is used to tell Make to suspend reading the current Makefile and read some other specified Makefile(s). Once Make has finished reading the other Makefile(s), it continues reading after the include directive. This is used to separate the build into multiple subsystems, a safer method than using recursive Make [8]. The example in Figure 1 includes 2 external Makefiles called flags.mk (line 15) and flags_debug.mk (line 13), depending on whether the DEBUG variable is set to "yes" or not (more on conditionals in the next section). These Makefiles, as their names suggest, contain variable definitions for the CFLAGS and CPPFLAGS variables, used by the recipe command on line 27.

3) Conditionals: Like most languages, Make provides simple conditional if-statements that allow the build engineer to include parts of the Makefile only if certain conditions are met. Make evaluates conditionals, and replaces them with the text corresponding to the conditions that evaluate to true. The text can be rules or statements that could occur inside the Makefile, or recipe commands within a rule. For example, in Figure 1, lines 12-16 show a conditional statement that checks to see if the DEBUG variable is set to "yes" and includes an external Makefile if so.

E. Functions

Make includes a number of built-in functions for a variety of uses. Functions are called in much the same way as variables are referenced, using commas (and no spaces) to separate parameters, as shown below.

```
$(function param1,param2,param3 ... )
```

The example in Figure 1 uses the `basename` function on recipe line 26 to print the name (minus the .o) of the target file matching the rule. The remainder of this section briefly describes the other functions provided by GNU Make, categorized based on the manual [9].

1) String Functions: String functions operate on strings or lists (strings separated by spaces). These include: `findstring`, `subst`, `patsubst` for finding and replacing substrings; `strip` for removing trailing whitespace; and `filter`, `filter-out`, `sort`, `word`, `wordlist`, `words`, `firstword`, and `lastword` for operating on lists.

2) Filename Functions: Filename functions operate specifically on filenames and directories, or lists of filenames and directories. These include: `dir` and `notdir` for identifying directories; `basename` and `suffix` for stripping and retrieving file extensions (like .c), respectively; `addprefix`, and `addsuffix` for adding strings to the beginning and end of a path, respectively; `join` for concatenating lists; `wildcard`, for searching for files (with patterns); and `realpath` and `abspath` for returning non-relative paths (no "." or "..").

3) Conditional Functions: There are 3 conditional functions: `if`, `and`, and `or`. The `if` function differs from the if directive ("ifdef"/"ifeq") because it can be used in rule targets or prerequisites. The `and` and `or` functions are typical logical operators that return true or false when passed a series of conditions. They are normally used in conjunction with the `if` function.

4) Control Functions: Control functions can alter the way Makefiles are run. The `error` function will print a given error message and exit. The `warning` function will throw a warning with the given error message, but continue running. The `info` function simply prints a given message.

5) Variable Functions: The `value`, `flavor`, and `origin` functions are useful for determining the origin of a variable. The `value` function will return the current value of a variable, without expanding it (i.e., as text with variable names). The `flavor` function will return the flavour of a given variable (i.e., simple or recursive), while the `origin` function returns where a variable was defined (e.g., command line, automatic variable, environment variable, etc).

6) Other Functions: Make offers some other useful specialized functions. The `shell` function can be used to call a shell command. The `guile` function is used to run GNU Guile scripts. The `foreach` function iterates through a list of words and performs some other function using it. The `file` function allows a Makefile to write to a file by overwriting it or appending to it.

7) Custom Functions (The Call Function): If Make doesn't provide a function, or a particular sequence of function calls becomes too difficult to read, Make allows developers to write custom functions and invoke them using the `call` function. Invoking the call function is much like any other function, where the first parameter is the name of the function. Subsequent parameters are passed to the custom function and referred to using `$(1)`, `$(2)`, and so on.

Table II
OVERVIEW OF DATASET

Generator	# Projects	# Makefiles	Avg Lines	Total Lines
Automake	147	1704	1153	1964720
CMake	80	8672	135	1167300
QMake	2	2460	2031	4996408
Hand-written	129	6683	49	329224
All	**358**	**19519**	**433**	**8457652**

III. APPROACH – TOOLS AND DATASET

This section discusses our approach to studying the use of the GNU Make features discussed in the previous section.

A. Subject Systems

Our dataset contains almost 20,000 Makefiles spanning 358 different projects, comprising over 8.4 million lines of build scripts. We began with the latest snapshot of the Linux kernel (v3.19) and the most recent version of all GNU library projects updated within the past 5 years (since 2010)[1]. We chose these because they are among the most advanced users of GNU Make [10]. We were also interested in comparing Makefiles that were generated by the most popular Makefile generators – Automake, CMake, and QMake [11] – so we chose projects that were known to use those generators. Such generators help to automate the process of Makefile authoring, for example, by analyzing the project source to infer dependencies, or by generating customized builds from configuration files. The Linux kernel is known to contain almost 2000 hand-written Makefiles, while the GNU projects use a mixture of hand-written and Automake-generated Makefiles. For CMake, we chose the KDE library of applications and libraries because it is known to have switched to CMake [12]. Other CMake-based projects we included were CMake itself, MiKTeX, Ogre, Scribus, and Blender. QMake is developed by Qt and used mostly by Qt, so we included the Qt library and the Ruby Qt bindings. Each project was configured and, in the case of CMake and QMake, the Makefiles were generated.

To distinguish generated Makefiles from manually developed ones, we used a combination of knowledge drawn from previous experience with these generators, and the comments that each generator leaves in its generated Makefiles (e.g., "generated automatically by automake from Makefile.am"). Makefiles generated using CMake and QMake were easy to identifiy because they must be generated manually. However, Makefiles generated with Automake posed a challenge. Automake is a part of the Autotools family, along with Autoconf – a configuration tool that sets environment-specific values in Makefiles. Most, if not all, GNU projects use Autoconf, but only some use Automake. In addition, a system that uses Automake-generated Makefiles is also free to use hand-written Makefiles (Automake templates can even include hand-written Makefile code). Because of this we had to solely rely on the comments within the generated Makefiles, such that Makefiles with no such comment were assumed to be hand-written. The breakdown of the number of Makefiles for each generator and the average number of lines is shown in Table II.

[1] ftp://ftp.gnu.org/gnu/

```
define Assignment
   [WS] [PrivateExportOverride?] [WS] [Id] [WS]
                [AssignmentOp] [WS] [BSWS?] [Expr?]
      [AssignmentContinuation*] [EOL?]
end define

define AssignmentContinuation
   [EOL] [tabspace] [WS] [Expr]
end define

define AssignmentOp
   '+= | ':= | '?= | '=
end define
```

Figure 2. A Snippet from our TXL Grammar for Makefiles.

This small portion of the subgrammar for assignments demonstrates one of the many ambiguities of the Makefile language. Although the language manual says that a backslash should be used [BSWS], assignments are commonly continued over several lines [EOL] with no indication other than blank space [tabspace] that they are part of a continuation and no marker at the end of the statement.

B. Measuring Feature Usage

In order to accurately measure feature usage, we required an accurate and detailed identification of both the high level structure and the low level elements in Makefiles. For this purpose we created a new Makefile parser implemented in the TXL source transformation language [7]. Crafting a high quality parser for Makefiles is a particular challenge because there exists no formal documentation for its syntax, because Makefiles contain embedded languages such as bash shell and guile commands, because it uses a mix of strict lexical and free-form syntactic conventions to distinguish components, and because the interpretation of Makefiles is highly implementation- and version-dependent.

We began with the grammar previously published by Tamrawi et al. [13] as part of their work on Makefile analysis using symbolic dependency evaluation. While the Tamari grammar identifies the main structural components that those authors were interested in, it uses island grammar techniques to abstract lower level details that are not of interest in their application. To facilitate feature analysis, we require a much more detailed and precise parse. Beginning with a translation of their grammar to TXL, we spent the first two months of our project refining and extending the parser to provide a detailed and precise parse that accurately identifies both high level structures and low level detailed elements. Accurately parsing Makefiles is surprisingly difficult, and much of this time was spent discovering, researching and decoding the intended meaning of many little known and seemingly ambiguous combinations of features discovered in our test set. A snippet of our grammar is shown in Figure 2.

To insure its accuracy, we tested our new Makefile parser on our dataset of almost 20,000 Makefiles extracted from GNU, Linux, and other open source projects. The grammar was validated with respect to completeness by ensuring that every Makefile in the test set could be parsed. The accuracy of

```
% Grammar overrides to specialize special and suffix targets
redefine Target
      [SpecialTarget] [WS]
   |  [SuffixTarget] [WS]
   |  ...
end redefine

define SpecialTarget
      '.PHONY | 'phony | '.INTERMEDIATE | '.intermediate
   |  '.SECONDARY | '.secondary | '.PRECIOUS | '.precious
   |  '.DELETE_ON_ERROR | '.delete_on_error
   |  '.SUFFIXES | '.suffixes | '.NO_EXPORT | '.no_export
   |  '.MAKE | '.make | '.SILENT | '.silent
   |  '.IGNORE | '.ignore
end define

define SuffixTarget
   '. [id]
end define

   . . .

% Extract and count all targets in rules
construct Targets [Target*]
   _ [^ Rules]
construct _ [number]
   _ [length Targets] [putp "rule targets: %"]

% Extract and count special targets .PHONY .INTERMEDIATE ...
construct SpecialTargets [SpecialTarget*]
   _ [^ Targets]
construct _ [number]
   _ [length SpecialTargets]
     [putp "rule target special targets: %"]

% Extract and count function calls in targets of rules
construct TargetFuncs [FunctionCall*]
   _ [^ Targets]
construct NTargetFuncs [number]
   _ [length TargetFuncs]
     [putp "rule target function calls: %"]
```

Figure 3. Example TXL Grammar Specializations and Extraction Constructors to Count Feature Instances in Context

The TXL extract operator [^] allows us to extract a list of all occurrences of a given grammatical feature, such as rule Targets, from the parse. Cascaded extractions, such as the one for TargetFuncs above, extract from the results of a previous extraction, such as Targets, to identify features in the context of other features. Grammar specializations, such as [SpecialTarget] and [SuffixTarget] above, allow for more detailed analysis.

the parses was qualitatively validated using hand examination of the XML markup versions of the parses of a random sample of about 100 Makefiles, and quantitatively validated by comparing the extracted set of identified elements with those identified using independent lexical searches.

Several iterations of adapting, tuning and refining resulted in a reliable parser that yields an accurate, robust and highly detailed parse of GNU Makefiles and Makefile templates. This parser can serve not only for the high level feature analysis of this paper, but also for more detailed build system analysis. Figure 4 shows the XML markup version of the parse of the example Makefile shown in Figure 1.

To compute the metrics for each feature, we used a combination of agile parsing [14] to specialize our grammar and to distinguish elements and structures representing the features in which we are interested, and TXL type extraction [15] to extract and count instances of these features in each Makefile. An example of this specialization and extraction process in TXL is shown in Figure 3. By exploiting the structure of the

```
<Statement><Comment>
    #This is an example of a Makefile
</Comment></Statement>
<Statement><Assignment>
    VPATH=usr/lib/
</Assignment></Statement>
<Statement><Directive>
    vpath *c usr/src/
</Directive></Statement>
<Statement><Assignment>
    DEBUG=yes
</Assignment></Statement>
<Statement><Assignment>
    OBJS:= \
        bin/main.o \
        bin/foo.o \
        bin/bar.o
</Assignment></Statement>

<Statement><IfStatement>
ifeq( <Evaluation>$(DEBUG)</Evaluation> , yes )
    <Statement><Directive>
        include build/flags_debug.mk
    </Directive></Statement>
else
    <Statement><Directive>
        include build/flags.mk
    </Directive></Statement>
endif
</IfStatement></Statement>

<Rule>
    <Target>.PHONY</Target> :
        <Dependency>all</Dependency>
    <Recipe></Recipe>
</Rule>

<Rule>
    <Target>all</Target> :
        <Dependency>
            <Evaluation>$(OBJS)</Evaluation>
        </Dependency>
    <Recipe>
        cc -o <Evaluation>$(OBJ)</Evaluation>
    </Recipe>
</Rule>

<Rule>
    <Target>foo.o</Target> :
        <Dependency>foo.c</Dependency>
        <Dependency>foo.h</Dependency>
    <Recipe>
        cc -c foo.c
    </Recipe>
</Rule>

<Rule>
    <Target>%.o</Target> :
        <Dependency>%.c</Dependency>
    <Recipe>
        +echo "Compiling... "
          <FunctionCall>$(basename
              <AutoEval>$@</AutoEval>)
          </FunctionCall>
        <Evaluation>$(CC)</Evaluation> -c
          <Evaluation>$(CFLAGS)</Evaluation>
          <Evaluation>$(CPPFLAGS)</Evaluation>
          <AutoEval>$<</AutoEval> -o
          <AutoEval>$@</AutoEval>
    </Recipe>
</Rule>

<Rule>
    <Target>.PHONY</Target> :
        <Dependency>clean</Dependency>
    <Recipe></Recipe>
</Rule>

<Rule>
    <Target>clean</Target> :
    <Recipe>
        rm bin/*
    </Recipe>
</Rule>
```

Figure 4. Example XML Markup of Makefile Parse (Note: Lower level markup detail not shown for readability)

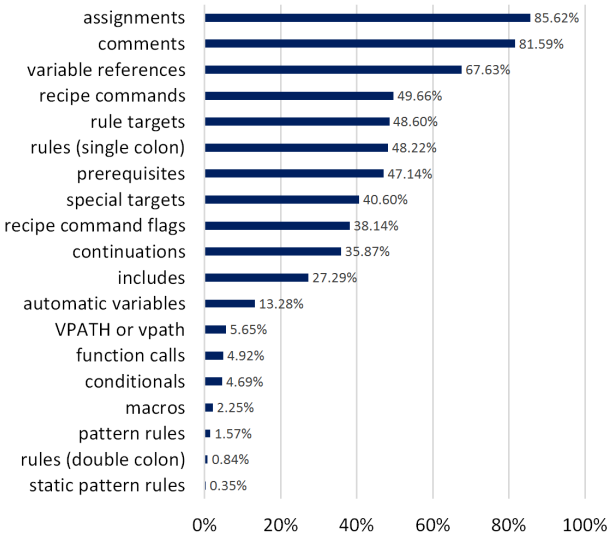

Figure 5. Feature Use (% of Makefiles)

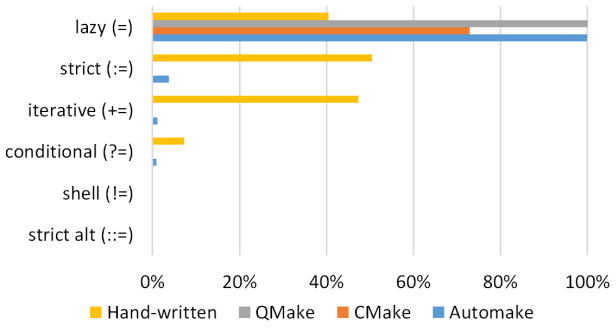

Figure 6. Assignment Use (% of Makefiles)

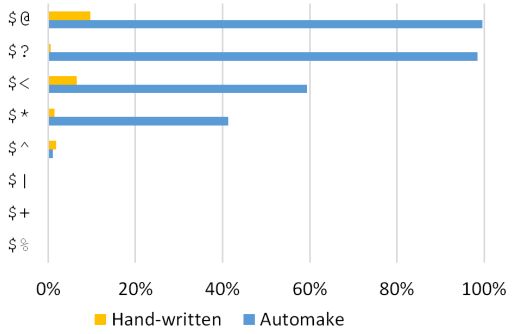

Figure 7. Automatic variables (% of Makefiles)

parse using multi-level TXL type extraction to isolate context, we are not only able to measure the overall total number of features, but also where they occur. For example, we can measure not only the total number of function calls, but also how many function calls occur in rule targets, in prerequisite lists, in assignment expressions, and so on.

As a concrete example, one of the parsed Evaluations of the OBJS variable in the "all" rule of Figure 4 is in a Dependency, while the other is in a Recipe. Thus we can count evaluations in dependencies in rules by separately extracting from the parse all Dependencies and then all the Evaluations in them, and all evaluations in recipes of rules by first extracting all Recipes and then all the Evaluations in them.

IV. Analysis

To help answer our research questions, we plotted several graphs illustrating feature use across the Makefiles in our dataset. Each graph shows the proportion of the dataset that uses the feature listed on the Y-axis at least once.

RQ1. How frequently are Makefile features used?

Figure 5 shows feature use of the whole dataset for the features listed. We group the features using the breakdown of Section II to simplify the chart. As we can see, Makefiles are frequently commented, with comments appearing in more than 80% of files. Assignments and variable references are the most used language features, at over 85% and 65% (respectively) of all Makefiles. Rules, targets, prerequisites, and recipes account for just under 50%. This indicates that Makefiles are split into two categories: Makefiles with rules, and Makefiles with only assignments that are included in the former. This is further supported by the number of Makefiles that use includes (just over 25%) and by browsing the Makefile source.

There are some features mentioned in the GNU Make manual that we did not find in our dataset. To see this, we have to unpack some of the categories of features we used. First, there are 6 different kinds of assignment operators: lazy ("="), strict (":=" and "::="), iterative ("+="), conditional ("?="), and shell ("!="). From the graph in Figure 6, we can see that the shell operator and the alternative strict operator (introduced to conform to the POSIX standard) are never used, which is not surprising considering that they were only introduced in GNU Make v4.0 (released October 2013). Lazy assignments are used the most, but what is interesting is the difference between hand-written and generated Makefiles. CMake and QMake use only lazy assignments, with Automake also using strict, iterative and conditional. It is a good time to note that, since it uses templates, Automake can have hand-written Make code inserted into the Makefiles it generates. Because of this, we see some similar features popping up in the feature composition of generated and hand-written Makefiles. For example, we can see that Automake uses other assignment operators in the same descending order as hand-written Makefiles.

In Figure 5, we can see that automatic variables are used in about 13% of Makefiles, but some are more common than others. Looking closer (Figure 7), we see that `$@` is the most common of these variables, followed by `$?`, `$<`, `$*`, and `$^`. `$|` and `$+` are also used to some extent, although they are not visible on the graph (0.01%), however, `$%` is never used. Automake is by far the biggest user of automatic variables, with almost every Makefile it generates using at least one, but they are also used in hand-written Makefiles.

Functions are another feature that is almost exclusively used in hand-written and Automake-generated Makefiles. Figure 5

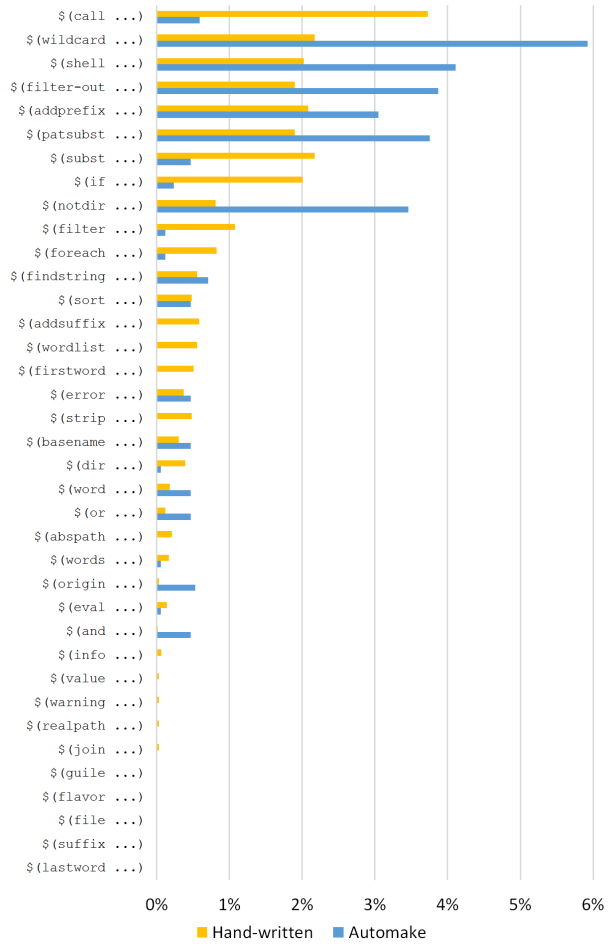

Figure 8. Function calls (% of Makefiles)

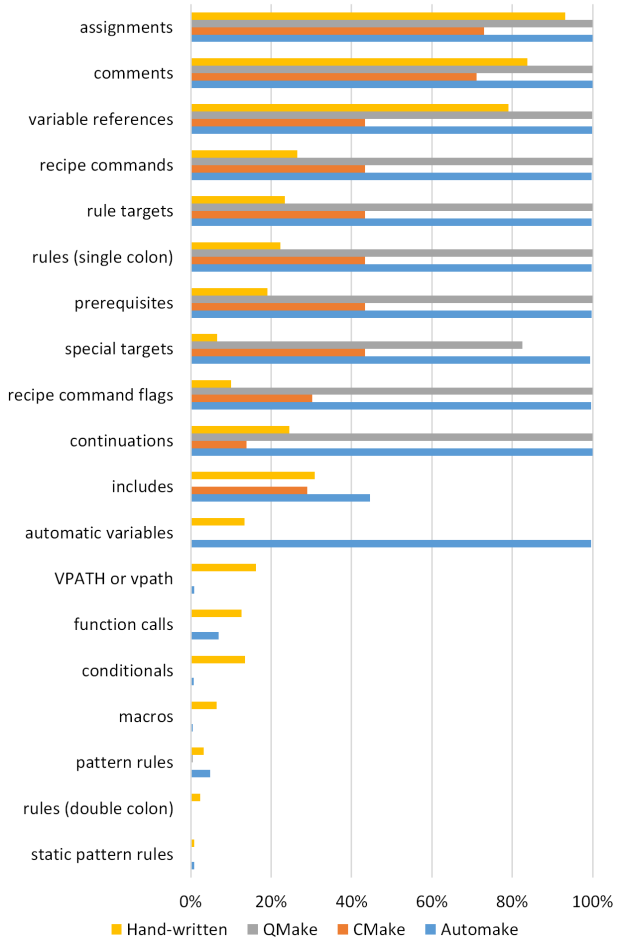

Figure 9. Feature Use By Generator (% of Makefiles)

indicates that they are only used in about 5% of Makefiles. Figure 8 shows the use of all the built-in functions listed in the Make manual. The `call` and `wildcard` functions are the most common with 1.8% and 1.7% use respectively, with the next highest function at 1.4%. On the other end of the list, there are a number of functions that are never used in our dataset. The `lastword`, `suffix`, `file`, `flavor`, and `guile` functions all have 0% usage. The `guile` and `file` functions were only recently added to GNU Make, which likely accounts for their absence.

RQ2. Are features used differently in generated Makefiles?

We have already seen how generators use assignment operators, automatic variables, and function calls differently, but Figure 9 shows the breakdown of feature use by all categories. A set of core Makefile features emerge that are used by all generated files, mainly variables (assignments, references) and single colon rules (targets, prerequisites, recipes). Other features such as special targets, continuations, and recipe command flags ("@", "-", and "+") are used by most. On the other hand, there are some features that seem designed specifically to make it easier for writing Makefiles by hand. Conditionals, functions, vpaths, double colon rules, pattern rules, and macros are all used only by hand-written or Automake Makefiles.

The graph also shows a set of features that are used very little in our dataset (i.e., $<$10%), and only by Makefiles with hand-written Make code (i.e., Automake-generated and hand-written Makefiles). Exceptions are pattern and double-colon rules, which are also used by a small number of QMake-generated Makefiles (i.e., $<$0.5%). Vpaths, functions, conditionals, macros, pattern rules, double colon rules, and – to some extent – automatic variables are what we consider to be the more advanced features of Make because they allow more general Makefiles to be written as opposed to a more verbose Makefile.

RQ3. To what extent are bad practices, specifically obsolete features and recursion, still in use?

The GNU Make manual gives 4 features that are "semi-obsolete" meaning that, while still supported, they are replaced by other features.

The first of these features are the .SILENT and .IGNORE special targets, which are obsolete because the "@" and "-" recipe command flags (respectively) offer a more fine-grained

Table III
OBSOLETE FEATURES IN MAKEFILES.

	All	Automake	CMake	QMake	Hand
.SILENT	25.76%	0.00%	43.38%	0.00%	0.03%
.IGNORE	0.00%	0.00%	0.00%	0.00%	0.00%
suffix rules	27.09%	98.77%	0.00%	83.50%	3.31%
old auto vars	0.17%	0.12%	0.00%	0.00%	0.34%

Table IV
RECURSION IN MAKEFILES.

	All	Automake	CMake	QMake	Hand
make	44.62%	99.24%	28.29%	82.76%	5.06%
automake	11.51%	98.53%	0.00%	0.00%	0.04%
cmake	25.75%	0.00%	43.38%	0.00%	0.00%
qmake	16.84%	0.00%	0.00%	100.00%	0.00%

implementation. We can see from our data that .IGNORE is never used. We can also see that .SILENT is only used in a small number of hand-written Makefiles (two, actually), and more than 40% of CMake-generated Makefiles.

Another obsolete feature is suffix rules, the old way of expressing implicit rules, which were replaced by pattern rules. Our results show that these are still used often across most categories, appearing in almost every Automake-generated Makefile in or dataset, and in the majority of QMake-generated files. On the other hand, CMake doesn't use it at all. Interestingly, there are 100 Makefiles in our set – across multiple generators – that use a mixture of both suffix and pattern rules.

The last obsolete feature is a variation on automatic variables. Each automatic variable returns a file or list of files, which is in the form of a path. These paths can be reduced to filenames or directories only by appending an "F" or a "D" (respectively) to the automatic variable (e.g., "$(@F)" or "$(%D)"). This form is considered obsolete because the `dir` and `notdir` functions provide a similar output. When we look at the results, we can see this form used in less than 0.2% of all Makefiles (25 total), and only appear in Automake or hand-written Makefiles (the only Makefiles that use automatic variables).

As discussed in Section II, Make can be run recursively by calling itself within a recipe. This can be dangerous and can lead to unexpected results if not done carefully [8]. Generators can also be recursively invoked from a Makefile recipe, to report build progress back to the generator or to compile a report. As such, these are likely less harmful, but we measure them nonetheless.

Table IV shows the use of recursive calls across all categories of Makefiles. What we see is that generators rely heavily on recursive calls. Most Automake Makefiles use recursive Make and Automake calls (about 99%). The same goes for CMake and QMake: 43% of CMake files call CMake, all QMake files call QMake, and both use recursive Make calls (29% and 83%, respectively). Interestingly, only about 5% of the hand-written Makefiles use recursion. A tiny number of hand-written Makefiles seem to call Automake, possibly an indicator of misclassification for some Makefiles.

V. DISCUSSION

From our analysis, it appears that there is a core set of features that are essential to any Make-based build system, roughly corresponding to Feldman's original Make features. Assignments and variable references are by far the most common, appearing in almost every Makefile. This makes sense, considering that some Makefiles consist only of assignments that are then included in another Makefile (as a configuration technique). For example, the Linux kernel adopted this pattern to improve the life of driver developers, who no longer need to write their own rules but just assign the names of their source code and object files to the right variable. The Linux build system then automatically reads these variables and processes them using generic build rules.

Rules and their associated parts (targets, prerequisites, and recipe commands) form the other essential Make features. What is surprising is how many hand-written Makefiles contain no rules. It is possible that these files have been incorrectly classified as hand-written, when in fact they may have been generated by Autotools (and Automake). Autotools often produces smaller Makefiles containing only variable assignments, but doesn't produce any comment to say they were generated, so it is difficult to know for certain.

Special targets are used a lot by generators, but not often in hand-written Makefiles. We saw earlier that all CMake-generated Makefiles use the obsolete .SILENT target. This is because all CMake-generated Makefiles containing rules use a rule with a variable called VERBOSE. By default, VERBOSE is null and therefore an empty rule with only the .SILENT target is produced, which Make interprets to suppress all output from the Makefile. If VERBOSE is set to any non-null value, the target ceases to be the special .SILENT target and the rule is meaningless.

It may seem like include statements are not used that much, but it makes sense when one realizes that not every file is going to include another file. Some Makefiles will be be written for the purpose of being included in other Makefiles, and therefore probably won't include others.

All CMake-generated Makefiles that contain rules use recursive Make calls, which the developers argue is necessary in order to automatically scan implicit dependencies [16]. This works using 3 levels of Makefiles that call each other. The top-level Makefile is the one that is called from the command line. It then calls the second-level Makefile (Makefile2) with the appropriate target. The second-level Makefile calls the third-level Makefiles (build.make) that are within the sub-directories of the system. These third-level Makefiles get called twice: the first time with the "depend" target, which scans the system for implicit dependencies; and again with the "build" target, which actually builds the system. This is how it avoids the pitfalls of recursive Make described by Miller [8].

The lack of rule patterns in automatically generated Makefiles can be explained by the fact that generators specialize their Makefiles after scanning for the names of all source code files and their header file dependencies. Hence, they know all

the files by name during generation, and no generic rules are necessary. Apart from being faster, the use of only essential Make features makes these Makefiles more portable to other implementations such as BSD Make.

VI. Related Work

Adams et al. investigated evolution in the Linux build system [4] from version 0.01 to 2.6.21.5 using lines of code as a measurement. They showed that build systems evolve in tandem with the source code. They also asserted that Makefile complexity increases, based on the increasing number of targets, and thus the number of tasks, to be built.

McIntosh et al. [6] extended this analysis to show that Java systems using Ant evolved in the same way. They used static measurements, like the number of files, lines (SBLOC), targets, tasks (similar to Make recipes), and modified Halstead complexity metrics – measurements for estimating how much information someone reading the source code would have to absorb, and how much mental effort would be required. The Halstead metrics, which usually utilize operators and operands, were modified to use tasks, targets (operators) and the parameters passed to them (operands). They also used some dynamic measurements, like the length (total number of executed tasks/targets) and depth (the maximum number of tasks to create one target) of the build graph. They found a general growth in the number of lines of code in the build scripts, and a high correlation between the Halstead complexity and lines of code. As for the dynamic metrics, they found two patterns concerning the depth of the build graph: one where the depth remains mostly unchanged and one where the depth increases over time. Further analysis showed that the projects where the depth increased used a recursive design, whereas the others did not.

Zadok proposes a different approach for measuring build system complexity (as it relates to portability) [17]. After plotting the number of lines of code and the number of CPP conditionals (e.g., #if, #ifdef, etc.), he notes their similarity. The packages with the fewest lines have the fewest CPP conditionals, and vice versa. He takes this to mean that neither is a good measure of complexity, and instead proposes the number of CPP conditionals per 1000 lines of code. He found that the biggest factor in complexity was related to the number of operating system calls and the portable C code that comes with it.

Tamrawi et al. [13], [18], created a tool called SYMake[2] that uses Symbolic Dependency Graphs (SDGs) to automatically detect build errors and smells, as well as to allow refactoring (e.g., rule removal, target renaming). Targets are represented as rectangles, with the initial target "install" at the far left. Recipes are represented as rounded rectangles, and diamond-shaped "Select" nodes represent a choice between alternatives. Each recipe is associated with a V-model, representing a string value. The grammar published in their work was a starting point for our TXL grammar.

Adams et al. recognized that the build system is used by almost all project stakeholders at some point, and that they suffer from understandability and maintainability issues [19], [20]. With this in mind, they set out to build the MAKAO tool (Makefile Architecture Kernel featuring Aspect Orientation)[3]. This tool is built on top of the GUESS graph exploration tool and presents a visualization of the whole build system that allows the user to explore by zooming. The user can also use Python to query, filter, refactor, and validate the system.

Shridhar et al. [21] manually analyzed the changes made to Java-based build systems. Corrective, adaptive and new functionality changes are the most common, and require the largest changes to a build system. The authors also analyzed who is responsible for making these changes. More invasive changes tended to be made by the build experts rather than regular developers.

VII. Conclusion and Future Work

By looking at the frequency of use of GNU Make features, we observed two things. First, there is a core set of features that are used in more than a third of all Makefiles. Second, there is a set of more advanced features that seem to be more suited to writing Makefiles by hand, and that are used almost exclusively in that context. We also found that recursive Make is used a lot – especially by generators – even though it has been considered bad practice for almost two decades, and that suffix rules are still used extensively despite being considered semi-obsolete.

While in this paper we only have room to present a summary of our feature use analysis, our entire raw analysis can be downloaded as a spreadsheet from *http://research.cs.queensu.ca/home/doug/ICPC15/*. We have only begun to scratch the surface of what can be analyzed in Makefiles. Our analysis infrastructure allows for much more than feature usage. For example, it can be used to measure other aspects of Makefiles such as their complexity. We are also interested in automatically detecting and analyzing design patterns in Make-based build systems.

References

[1] S. I. Feldman, "Make — a program for maintaining computer programs," *Software: Practice and Experience*, vol. 9, no. 4, pp. 255–265, 1979.

[2] "What's wrong with GNU make? - conifer systems." [Online]. Available: http://www.conifersystems.com/whitepapers/gnu-make/

[3] J. Graham-Cumming, "Debugging makefiles." [Online]. Available: http://www.drdobbs.com/tools/debugging-makefiles/197003338

[4] B. Adams, K. De Schutter, H. Tromp, and W. De Meuter, "The evolution of the linux build system," *Electronic Communications of the EASST*, vol. 8, no. 0, Oct. 2008.

[5] S. McIntosh, B. Adams, T. H. Nguyen, Y. Kamei, and A. E. Hassan, "An empirical study of build maintenance effort," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. New York, NY, USA: ACM, 2011, pp. 141–150. [Online]. Available: http://doi.acm.org/10.1145/1985793.1985813

[6] S. McIntosh, B. Adams, and A. E. Hassan, "The evolution of java build systems," *Empirical Software Engineering*, vol. 17, no. 4-5, pp. 578–608, Aug. 2012.

[7] J. R. Cordy, "The TXL source transformation language," *Sci. Comput. Program.*, vol. 61, no. 3, pp. 190–210, Aug. 2006.

[2] http://home.engineering.iastate.edu/~atamrawi/SYMake/

[3] http://mcis.polymtl.ca/~bram/makao/index.html

[8] P. Miller, "Recursive make considered harmful," *AUUGN Journal of AUUG Inc*, vol. 19, no. 1, pp. 14–25, 1998.

[9] "GNU make." [Online]. Available: http://www.gnu.org/savannah-checkouts/gnu/make/manual/make.html

[10] R. Mecklenburg, *Managing Projects with GNU Make*. O'Reilly Media, 2004. [Online]. Available: http://shop.oreilly.com/product/9780596006105.do

[11] P. S. PhD and P. Smith, *Software Build Systems: Principles and Experience*, 1st ed., Upper Saddle River, NJ, Mar. 2011.

[12] "Why the KDE project switched to CMake – and how (continued) [LWN.net]." [Online]. Available: http://lwn.net/Articles/188693/

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

[14] T. R. Dean, J. R. Cordy, A. J. Malton, and K. A. Schneider, "Agile parsing in TXL," *Automated Software Engg.*, vol. 10, no. 4, pp. 311–336, Oct. 2003.

[15] J. R. Cordy, "Excerpts from the TXL cookbook," *Lecture Notes in Computer Science*, vol. 6491, pp. 27–91, 2011.

[16] "CMake FAQ." [Online]. Available: http://www.cmake.org/Wiki/CMake_FAQ

[17] E. Zadok, "Overhauling amd for the '00s: A case study of GNU autotools," in *FREENIX Track: 2002 USENIX Annual Technical Conference*, Berkeley, CA, USA, 2002, pp. 287–297.

[18] A. Tamrawi, H. A. Nguyen, H. V. Nguyen, and T. N. Nguyen, "SYMake: a build code analysis and refactoring tool for makefiles," in *27th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2012, New York, NY, USA, 2012, pp. 366–369.

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

[20] B. Adams, "Co-evolution of source code and the build system: Impacton the introduction of AOSD in legacy systems," PhD dissertation, Ghent University, Belgium, May 2008.

[21] M. Shridhar, B. Adams, and F. Khomh, "A qualitative analysis of software build system changes and build ownership styles," in *8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '14, 2014, pp. 29:1–29:10.