SYMake: A Build Code Analysis and Refactoring Tool for Makefiles

Ahmed Tamrawi
atamrawi@iastate.edu

Hoan Anh Nguyen
hoan@iastate.edu

Hung Viet Nguyen
hungnv@iastate.edu

Tien N. Nguyen
tien@iastate.edu

Electrical and Computer Engineering Department
Iowa State University
Ames, IA 50011, USA

ABSTRACT

Software building is an important task during software development. However, program analysis support for build code is still limited, especially for build code written in a dynamic language such as Make. We introduce SYMake, a novel program analysis and refactoring tool for build code in Makefiles. SYMake is capable of detecting several types of code smells and errors such as cyclic dependencies, rule inclusion, duplicate prerequisites, recursive variable loops, etc. It also supports automatic build code refactoring, e.g. rule extraction/removal, target creation, target/variable renaming, prerequisite extraction, etc. In addition, SYMake provides the analysis on defined rules, targets, prerequisites, and associated information to help developers better understand build code in a Makefile and its included files.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement

General Terms

Algorithms, Languages, Management, Reliability

Keywords

Build Code Analysis, Maintenance, Refactoring, Code Smells

1. INTRODUCTION

In software development, software building is a crucial process to produce the deliverables, executable code, and/or documentations from source code and associated libraries. A building process is specified in *build files* which contain a set of *rules* that direct a build tool on how to derive the target programs from their corresponding sources. Among several build tools, Make [1], a build tool supporting build code written in make dynamic language, is very widely used. Despite its popularity, maintenance tool support for Make build code is still very limited. Due to the dynamic nature of Make's processing, it is challenging to build the analysis tools for tasks such as refactoring or code smell detection in Makefiles. Let us explain the challenges via an example.

Illustration Example. Figure 1 shows a Makefile that specifies the rules to build the main, sender and receiver programs from the corresponding code in either Java or C, and data files. To enable users to specify in a Makefile multiple building configurations for different environments, programming languages, or inputs, Make processes a Makefile in two phases. In the first phase, called the *evaluation phase*, it proceeds with the evaluation of all statements, variables, and rules in the Makefile based on the input command and the running environment. Make then resolves them into a set of concrete build rules. For example, Figure 2 displays the result of the evaluation phase when a command 'make -f myMakefile' is entered and the running machine has Java installed. Each *rule* typically contains a set of *targets*, (e.g. sender.jar at line 3), a set of *prerequisites*, (e.g. sender_src.java, sender_impl.java, sample.dat at line 3), and a *recipe* (e.g. line 4), which is a set of OS Shell commands to build the targets from the prerequisites. From that result, Make constructs a *concrete dependency graph* (CDG), in which nodes are targets, prerequisites, and recipes, and edges connect prerequisites to a recipe, or a recipe to targets. In the second phase, called the *execution phase*, based on the CDG, it executes the Shell commands to produce the target files from their prerequisite files, if the modification time of a prerequisite file is later than those of target files.

Let us explain the content of myMakefile and how Make's evaluation phase is performed. Line 1 in Figure 1 aims to check if the current machine has Java installed. The if statement at lines 3-11 is used to set the respective extensions for output files and source files, and the build commands for two languages, Java and C. Lines 13-18 define the variables, which are used to specify the names of target files and those of corresponding prerequisite files for both sender and receiver sides. Line 20 defines the target install with its prerequisites being defined via the variable $executables. The result of evaluating line 20 is line 1 of Figure 2. The value of $executables is in turn used to define a target for the rule at lines 28-29 whose results are two recipes for the sender and receiver at lines 4 and 7 in Figure 2. The foreach loop (line 26) is used to iterate over the values of the variable $executables (i.e. two target files for the sender and receiver), and to produce two building rules for them via the execution of the macro function at lines 22-24. For the case of Java, those

ASE'12, September 3–7, 2012, Essen, Germany

```
1  javaComp := $(shell which java)
2
3  ifneq ($(javaComp), "")
4      ext = .jar
5      srcExt = .java
6      cmd = javac
7  else
8      ext = .o
9      srcExt = .c
10     cmd = gcc
11 endif
12
13 sender := sender$(ext)
14 receiver := receiver$(ext)
15 executables := $(sender) $(receiver)
16
17 $(sender)_src= sender_src$(srcExt) sender_impl$(srcExt) $(wildcard *.
        dat)
18 $(receiver)_src=main_rcv$(srcExt) socket$(srcExt) receiver_src$(srcExt)
19
20 install : $(executables)
21
22 define ProgramMacro =
23     $(1) : $$($(1)_src)
24 endef
25
26 $(foreach exec,$(executables),$(eval $(call ProgramMacro,$(exec))))
27
28 $(executables):
29     $(cmd) $^ −o $@
30
31 %.dat : %$(ext)
32     getData $^ −o $@
33
34 ifneq ($(javaComp), '')
35     main.jar : main.java javaConf.dat
36         installJava $^ −o $@
37 else
38     main.o : main.c ccConf.data
39         installCC $^ −o $@
40 endif
```

Figure 1: myMakefile: An example of build code

```
1  install : sender.jar receiver.jar
2
3  sender.jar : sender_src.java sender_impl.java sample.dat
4      javac sender_src.java sender_impl.java sample.dat −o sender.jar
5
6  receiver.jar : main_rcv.java socket.java receiver_src.java
7      javac main_rcv.java socket.java receiver_src.java −o receiver.jar
8
9  javaConf.dat : javaConf.jar
10     getData javaConf.jar −o javaConf.dat
11
12 sample.dat : sample.jar
13     getData sample.jar −o sample.dat
14
15 main.jar : main.java javaConf.dat
16     installJava main.java javaConf.dat −o main.jar
```

Figure 2: Result after the evaluation phase on build code in Figure 1: 'make -f myMakefile'

two resulting rules are at lines 3 and 6 of Figure 2 after they are combined with lines 4 and 7. Lines 31-32 define an implicit rule in Make. It is used for building any file that ends with '.dat'. In this example, the result after applying that implicit rule is two concrete rules at lines 9-13 of Figure 2. Lines 34-40 define the rules for building main.jar and main.o.

Challenges in Build Code Maintenance. The key challenge in supporting build code analysis is the *dynamic nature* of Make's evaluation. The reason is twofold.

Firstly, the *analysis for the names* of variables or targets, and *automatic renaming* for them is not trivial. Since Make is dynamic, the name of a variable (i.e. an identifier) can be the result of the evaluation of other variables. For example, at lines 17 and 18, the prefixes of the variables on the left-hand sides are defined based on the values of the variables $sender and $receiver. A regular text search tool also cannot distinguish between the identifiers for variables and the string values in Make code. For example, at line 13 (sender := sender$(ext)), the variable sender is defined as a concatenation of the literal sender and the value of the variable ext.

The variable at line 17 illustrates another challenge. Here, the identifier of the variable $(sender)_src is composed of multiple sub-strings. If a user wants to rename the suffix src, a tool must rename all of the three locations: $(sender)_src (line 17), $(receiver)_src (line 18), and $$($(1)_src) (line 23). The reason is that, when executing foreach (line 26), at the first iteration, $$($(1)_src) (line 23) will be resolved to the name of the variable $(sender)_src (line 17), and at the second iteration to the name of $(receiver)_src (line 18). Therefore, $$($(1)_src) (line 23) affects both the variables at lines 17 and 18, and the texts at all three locations must be renamed consistently.

Secondly, automatic *analysis for the dependencies* among prerequisites/targets is challenging. For instance, myMakefile has a subtle error that causes a cyclic dependency in the concrete dependency graph. If a user enters 'make -f myMakefile' on a machine with Java, Make builds its CDG from the code in Figure 2, and runs the rule install (line 1). It first updates install's prerequisites by running sender.jar (line 3) and receiver.jar rules (line 6). Then, it successfully produces sender.jar and receiver.jar. However, in a special situation, a cyclic dependency among files could occur. The target sender.jar depends on the files that are fetched from the current directory with $(wildcard *.dat) (line 17). The cycle occurs if there exists a file with the name sender.dat in the user directory because to build sender.dat, Make matches that file with the implicit rule at line 31, and adds the following rule:

```
sender.dat : sender.jar
    genData sender.jar −o sender.dat
```

The new line 3 in Figure 2 now specifies that sender.dat is a prerequisite for sender.jar. Thus, a cycle is now formed because sender.jar and sender.dat are prerequisites of each other. This causes an error in the execution phase. This bug is difficult to detect statically and even at run time, it is not likely to be detected because it depends on the input and the user environment/directories. Due to Make's dynamic nature, similar difficulties also exist when a tool wants to detect if a build rule is subsumed by an implicit rule (e.g. the rule at line 31).

2. SYMake APPROACH

To address those challenges, we have built SYMake [2], a tool to detect several types of code smells and errors in Makefiles. SYMake also supports Make code analysis and refactoring. There are two core techniques in SYMake. First, to support static analysis on Make code, we develop a symbolic evaluation algorithm [2] that analyzes a Makefile and produces a data structure called a *Symbolic Dependency Graph* (SDG). An SDG represents all possible build rules

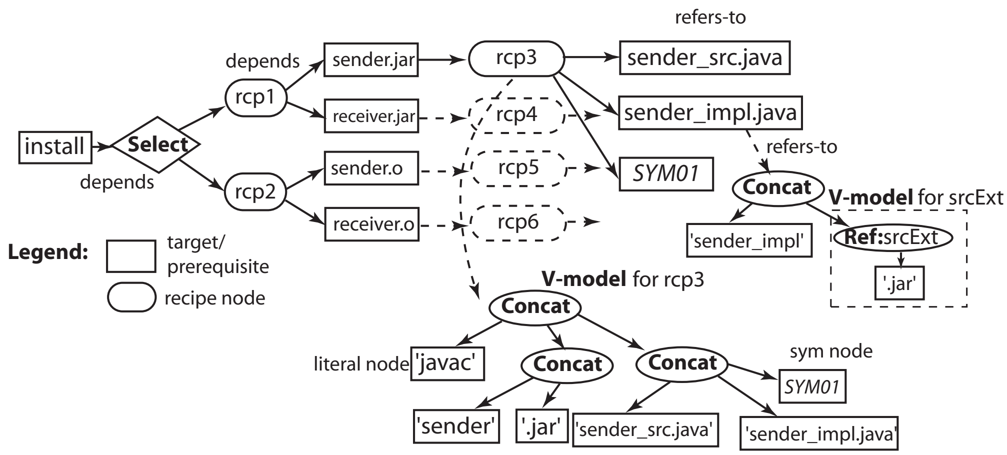

Figure 3: Symbolic Dependency Graph

and dependencies among targets and prerequisites via respective recipe commands. It takes into account all possible inputs and user environments by representing them via symbolic string values. The second technique is an algorithm to produce a symbolic evaluation trace, called a T-model, that represents how a string value in the Makefile is computed and manipulated via its program entities. A T-model is similar in spirit to an execution trace in a regular program execution and is used in the name analysis for a Makefile.

2.1 Symbolic Dependency Graph

An SDG has the following nodes: 1) target/prerequisite nodes, 2) recipe nodes, 3) **Select** node to represent alternative dependencies from a target to either of multiple recipes and prerequisites, and 4) a rule block contains all nodes/edges related to a rule. An SDG differs from a Make's CDG in that a component of a rule (target, prerequisite, or recipe) in an SDG might not be completely resolved into concrete strings due to user inputs or environment values (e.g. $(wildcard *.dat) in Figure 1). Instead, an SDG's node, representing a component of a rule, refers to a tree structure, called a *V-model*, which represents the symbolic string values for the component of the rule.

A V-model has two types of leaf nodes, literal and symbolic, to represent concrete and unresolved string values, respectively. There are three types of inner nodes. **Concat** and **Select** node represent a concatenated string value and a string value selected from the values corresponding to the sub-trees of that node, respectively. A V-model also contains a **Reference** node represents a reference to a variable. The child of the Reference node is a V-model representing the value of that variable.

Figure 3 shows part of the SDG and its V-models for Figure 1. The target node install (line 20) can have either of the two different sets of prerequisites and recipes depending on whether or not Java compiler is installed (lines 3-11): {sender.jar, receiver.jar} or {sender.o, receiver.o}. In turn, sender.jar depends on the recipe rcp3 whose string content is represented by its V-model. Also, rcp3 depends on the set of prerequisites including sender_src.java, sender_impl.java, and a symbolic node SYM01 representing the result returned from a call to wildcard to get the data files from the current directory (line 17).

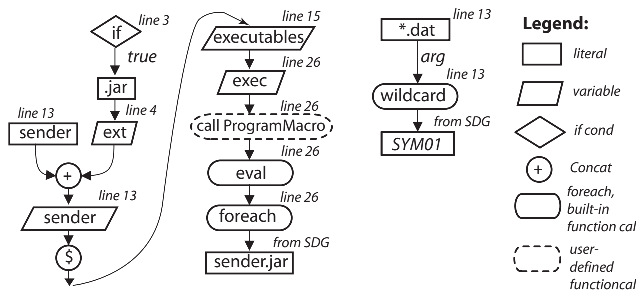

Figure 4: T-models for sender.jar and SYM01

2.2 Evaluation Trace Model

During symbolically evaluating a Makefile, for each resulting string value that represents a part of a rule or of a recipe in an SDG, SYMake provides a labeled acyclic graph, called T-model, to capture the construction of that string value via the program entities in the Makefile. A T-model contains three type of nodes: data, control, and operation/action nodes. A data node can be either a variable or literal node. A control node can be either an if or foreach node to represent branching or repetition points in the evaluation. To represent an operation/action, an T-model can contain 1) **Concat** nodes, 2) **Evaluation** nodes to represent variable evaluation, and 3) **Function Call** nodes. Figure 4 shows the T-model of sender.jar (left), and that of SYM01 (right).

From SDG, associated V-models, and T-models, we develop algorithms in SYMake to detect errors and code smells such as cyclic dependencies, loops of recursive variables, duplicate prerequisites, rule inclusions, etc. With T-models, SYMake also supports automatic refactoring, e.g. rule extraction/removal, target creation, target/variable renaming, prerequisite extraction, etc. More details are described in [2].

3. SYMake'S FUNCTIONALITY

The main functionalities of SYMake are: build rule analysis via symbolic evaluation and the SDG, renaming and refactoring support, and code smell detection in Makefiles.

Symbolic Evaluation. SYMake's GUI contains four main views: the Makefile view, the rules and variables view, and

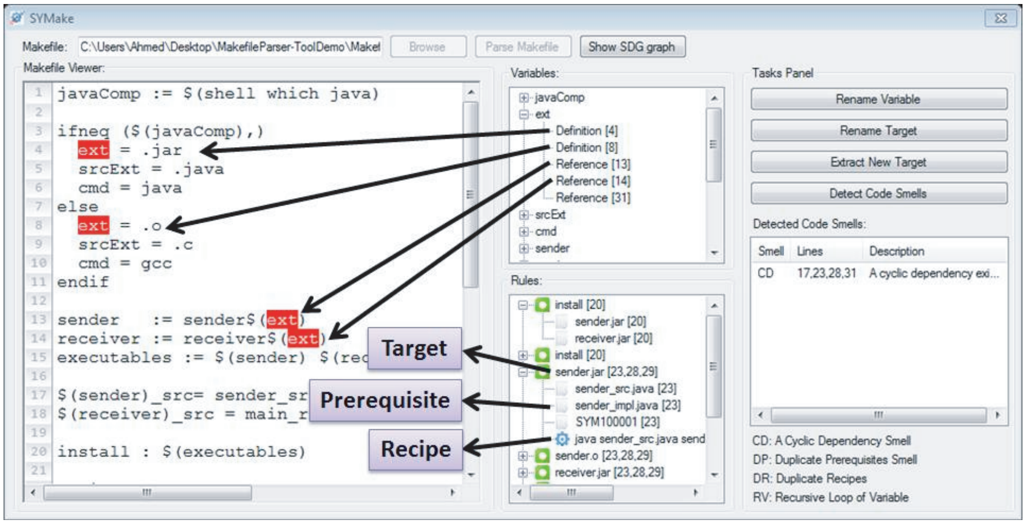

Figure 5: Analysis on Build Rules in a Makefile with SYMake

two task views for code smell detection and refactoring tasks. SYMake allows a user to load a Makefile for analysis and it will symbolically evaluate the loaded file. Figure 5 shows myMakefile in SYMake. The resulting SDG graph can be viewed via the *Show SDG graph* button.

SYMake displays also the views for variables and rules in the loaded Makefile. When a variable is selected from the variable's view, SYMake highlights all corresponding locations where the variable is initialized/referenced. Figure 5 shows SYMake as a user selects the variable ext. Similar to variables, if the user selects a rule, the corresponding references for that rule are highlighted in the Makefile view. For each rule, the sub-tree in the rules' view represents its prerequisites, recipe, and the respective code locations.

Refactoring Support. To rename, the user selects a variable and clicks on *Rename Variable*. A pop-up window will ask the user for the new name. Similar to renaming variables, *Rename Target* button is used to rename a target. For target extracting, the user first selects a set of prerequisites and then creates a new target for them. In addition to renaming, SYMake supports extracting new targets from existing prerequisites via *Extract New Target* button.

Code Smell Detection. To detect the types of code smells and errors listed in the previous section, a user can simply click on *Detect Code Smells* button. SYMake will display for each detected smell the corresponding smell type, source code locations involved in the smell, and a smell description showing all Makefile's elements involved in that smell/error.

4. RELATED WORK

Prior work has shown that the maintenance of build code causes a high percentage of overhead on general development efforts in a software process [3, 4]. Build code needs to be maintained and changed with a comparable normalized churn rate to that of source code and could contain as many defects due to that high rate [4].

A related work to SYMake is MAKAO [5]. It provides visualization and code smell detection support for Makefiles. There are key differences between SYMake and MAKAO. First, SYMake aims to provide program analysis on Make build code. MAKAO focuses more on visualization and reverse engineering for different views on the build architecture. Moreover, MAKAO can only work on *concrete dependency graph* for a Makefile, thus it cannot support renaming/extracting, and code smell detection for Make code as in SYMake. As seen in Section 1, due to Make's dynamic nature, program elements in a Makefile are not always fully exposed in build code (i.e. before the evaluation phase).

5. CONCLUSIONS

We introduce SYMake, a build code analysis tool for Makefiles that is based on symbolic evaluation to statically detect code smells/errors and supports Make code analysis and refactoring. We also performed an empirical evaluation on real-world Makefiles and the results showed that SYMake is accurate and efficient, and that with SYMake, users could detect code smells and refactor Makefiles more accurately.

6. ACKNOWLEDGMENTS

This project is funded in part by US National Science Foundation (NSF) CCF-1018600 grant. We would like to thank the ASE'12 reviewers for their constructive feedbacks.

7. REFERENCES

[1] S. Feldman, "Make: A program for maintaining computer programs," *Software Practice*, vol. 9, pp. 255–265, 1979.

[2] A. Tamrawi, H. A. Nguyen, H. V. Nguyen, and T. N. Nguyen, "Build Code Analysis with Symbolic Evaluation," ICSE'12. IEEE CS, 2012.

[3] L. Hochstein and Y. Jiao, "The cost of the build tax in scientific software," in *ESEM '11*. ACM, 2011.

[4] S. McIntosh, B. Adams, T. H. Nguyen, Y. Kamei, and A. E. Hassan, "An empirical study of build maintenance effort," in ICSE'11. ACM, 2011.

[5] B. Adams, H. Tromp, K. De Schutter, and W. De Meuter, "Design recovery and maintenance of build systems," in *ICSM'07*. IEEE CS, 2007.