SYMake: A Build Code Analysis Tool for Makefiles

Ahmed Tamrawi, Hoan Anh Nguyen, Hung Viet Nguyen, Tien N. Nguyen
Iowa State University
{atamrawi,hoan,hung,tien}@iastate.edu

Abstract—**Software building is an important task during software development. However, program analysis supports for build code are still limited, especially for build code written in a dynamic language such as Make. We propose SYMake, a novel program analysis tool for build code in Makefiles that is capable of detecting several types of code smells and errors and providing automatic supports in renaming of variables, targets, and extracting them into new ones. SYMake also provides the analysis on defined rules, targets, prerequisites, and associated information to help developers to better understand build code in a Makefile and its included ones.**

***Keywords*-Make; Symbolic Evaluation; Build Maintenance**

I. Introduction

A. Software Building with Make

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.

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. Make processes a Makefile in two phases. In the first phase, called *evaluation phase*, it proceeds with the evaluation of all statements, variables, and rules in the Makefile based on the input command, and the input and running environment, and then resolves them into a set of concrete build rules. The evaluation phase enables users to specify in a Makefile multiple building configurations for different environments or inputs. For example, Figure 2 displays the result of the evaluation phase when a command 'make -f myMakefile' is entered and the running machine has installed Java. Each *rule* typically contains a set of *targets*, a set of *prerequisites*, and a *recipe*, 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 *execution phase*, based on the CDG, Make executes the Shell commands to produce the target files from their prerequisite files, if the modifying 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 on it. Line 1 in Figure 1 aims to check if the current machine has installed Java. 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 is line 1 of Figure 2. That variable in turn defines a target for the rule at lines 28-29 whose results are 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 define/produce two building rules for them via the execution of the macro function at lines 22-24. For the case of Java, those 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.

B. Challenges in Build Code Maintenance

Despite its popularity, maintenance tool supports for Make build code are still very limited. Due to the dynamic nature in Make's evaluation, it is challenging to build the analysis tools such as for refactoring or code smell detection in Makefiles. The reason has twofolds. Firstly, the *analysis for the names* of variables or targets, and *automatic renaming* for them is not straightforward. 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 other variables $sender and $receiver. Moreover, a regular text search tool cannot distinguish between the identifiers for variables and the string values in build code. For example, at line 13 (sender := sender$(ext)), the variable sender is defined as a concatenation from the string literal sender and the value of variable ext.

Importantly, the variable at line 17 also illustrates another challenge. That is, the identifier of the variable $(sender)_src is composed of multiple sub-strings. Assume that, a user wants to rename the suffix src, a tool must rename all three

```
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. An example of build code in Make

```
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 myMakefile

following 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 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 all three texts at those locations must be renamed consistently.

Secondly, automatic *analysis for the dependencies* among prerequisites/targets is also challenging. For example, myMakefile code has a subtle error that causes a cyclic dependency in the concrete dependency graph. Assume that, when 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 programs. However, if Make is requested to build install, a cycle could occur. Rule 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 since Make matches that file with the implicit rule at line 31, and adds the following rule:

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

A cycle is 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 reveal at static time and even at run-time because it depends on the user environment/directory, and the input. Due to the dynamic nature of Make, similar difficulty also exists when a tool wants to detect if a build rule is subsumed by an implicit rule (e.g. the one at line 31).

II. SYMake Approach

To address those challenges, we build SYMake, a tool to detect several types of code smells and errors in Makefiles. SYMake also supports renaming for variables/targets whose names might be fragmented. Let us describe our techniques.

A. Symbolic Dependency Graph

SYMake first processes a Makefile and performs a symbolic evaluation to produce a data structure called a *Symbolic Dependency Graph* (SDG), which represents all possible build rules 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 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, a node in SDG, representing a component of a rule, refers to a data structure, called *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,

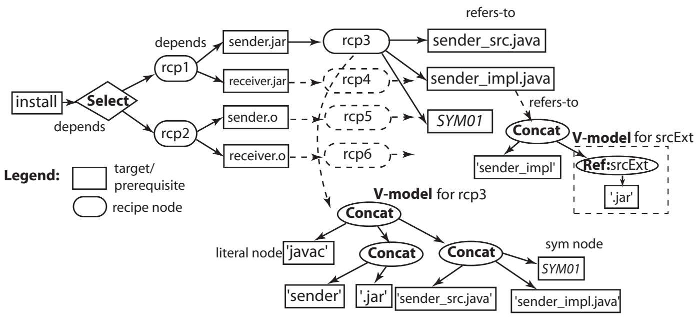


Figure 3. Symbolic Dependency Graph and V-models

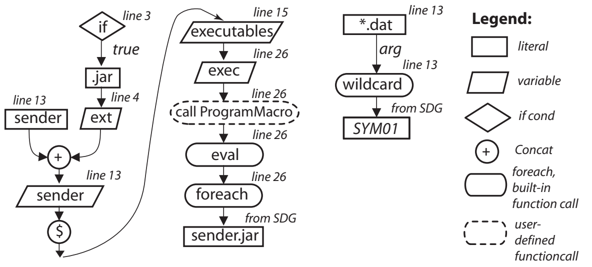


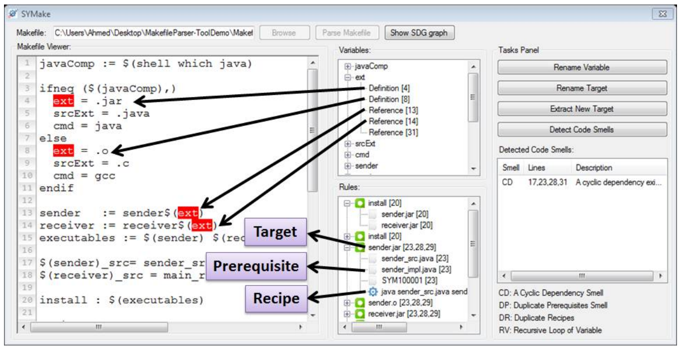
Figure 4. T-models for sender.jar (left), SYM01 (right)

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. V-model contains also a **Reference** node represents a reference to a variable and its child 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 Java compiler is installed or not (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 sender_src.java, sender_impl.java, and a symbolic node SYM01 representing the result returned from a call to wildcard to get data files from the current directory (line 17).

B. Evaluation Trace Model

During symbolically evaluating a Makefile, for each resulting string value that represents a part of a rule or a recipe in an SDG, SYMake provides a labeled acyclic graph (called *T-model*) to represent how that string value is computed and manipulated via program entities in the Makefile. The T-model contains 3 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. For representing an operation/action, an T-model contains 1) a **Concat** node, 2) **Evaluation** node to represent variable evaluation, and 3) **Function Call** node. Figure 4 shows the



```
1  javaComp := $(shell which java)
2
3  ifneq ($(javaComp),)
4    ext = .jar
5    srcExt = .java
6    cmd = java
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) $(rec
16
17 $(sender)_src= sender_sr
18 $(receiver)_src = main_r
19
20 install : $(executables)
21
```

Variables:
- javaComp
- ext
  - Definition [4]
  - Definition [8]
  - Reference [13]
  - Reference [14]
  - Reference [31]
- srcExt
- cmd
- sender

Rules:
- install [20]
  - sender.jar [20]
  - receiver.jar [20]
- install [20]
- sender.jar [23,28,29]
  - sender_src.java [23]
  - sender_impl.java [23]
  - SYM100001 [23]
  - java sender_src.java send
- sender.o [23,28,29]
- receiver.jar [23,28,29]

Detected Code Smells:

Smell	Lines	Description
CD	17,23,28,31	A cyclic dependency exi...

CD: A Cyclic Dependency Smell
DP: Duplicate Prerequisites Smell
DR: Duplicate Recipes
RV: Recursive Loop of Variable

Figure 5. A variable is selected and a rule node is expanded

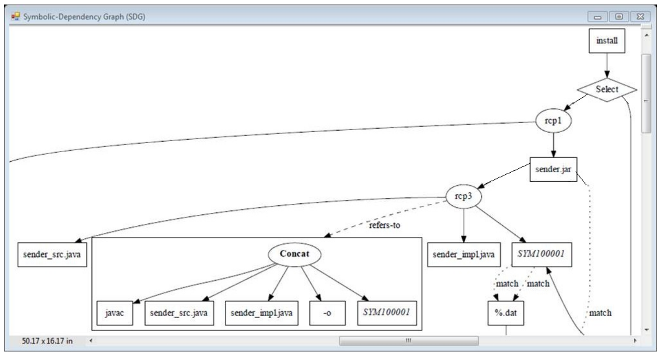


Figure 6. A screen shot of SYMake with SDG

T-model of sender.jar (left), and that of SYM01 (right).

From the infrastructure of SDG and associated V-models, and T-models, we develop algorithms in SYMake to detect code smells and errors such as cyclic dependencies, loops of recursive variables, duplicate prerequisites, rule inclusions, etc. With T-models, SYMake is also able to support automatic renaming for variables and targets/prerequisites, and extracting new targets. More details can be found in [2].

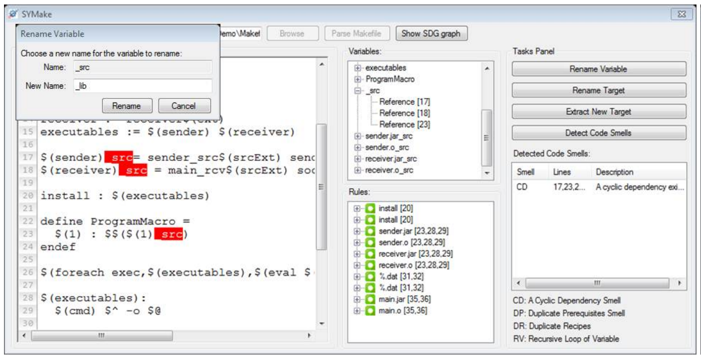
III. SYMake's Functionality

Generally, SYMake has the following functions: Makefile's symbolic evaluation and SDG displaying, renaming and refactoring support, and Makefile's code smell detection.

A. SYMake Symbolic Evaluation

SYMake's interface contains four main views: the Makefile view, rules and variables view, and a task view for code smell detection and refactoring tasks. SYMake allows a user to load a Makefile for analysis and it will symbolically evaluate the loaded Makefile. Figure 5 shows myMakefile in SYMake. The resulting SDG graph can be viewed as in Figure 6. For example, install node corresponds to the install target (line 20, Figure 1). sender.jar depends on rcp3 which refers to its V-model. In turn rcp3 depends on the prerequisite nodes sender_src.java, sender_impl.java, and SYM100001.

SYMake displays also the views for variables and rules in the loaded Makefile. When a variable is selected from



Rename Variable

Choose a new name for the variable to rename:

Name: _src

New Name: _lib

Rename | Cancel

```
15 executables := $(sender) $(receiver)
16
17 $(sender) _src= sender_src$(srcExt) senc
18 $(receiver)_src = main_rcv$(srcExt) soc
19
20 install : $(executables)
21
22 define ProgramMacro =
23   $(1) : $$($(1)_src)
24 endef
25
26 $(foreach exec,$(executables),$(eval $
27
28 $(executables):
29   $(cmd) $^ -o $@
30
```

Variables:
- executables
- ProgramMacro
- _src
  - Reference [17]
  - Reference [18]
  - Reference [23]
- sender.jar_src
- sender.o_src
- receiver.jar_src
- receiver.o_src

Rules:
- install [20]
- install [20]
- sender.jar [23,28,29]
- sender.o [23,28,29]
- receiver.jar [23,28,29]
- receiver.o [23,28,29]
- %.dat [31,32]
- %.dat [31,32]
- main.jar [35,36]
- main.o [35,36]

Tasks Panel

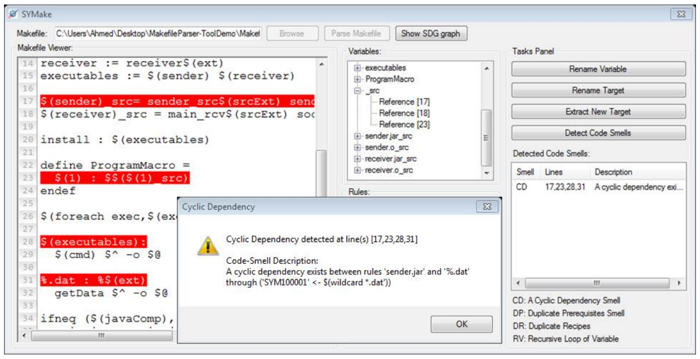
Rename Variable | Rename Target | Extract New Target | Detect Code Smells

Detected Code Smells:

Smell	Lines	Description
CD	17,23,2...	A cyclic dependency exi...

CD: A Cyclic Dependency Smell
DP: Duplicate Prerequisites Smell
DR: Duplicate Recipes
RV: Recursive Loop of Variable

Figure 7. Variable renaming as _src is highlighted



Makefile: C:\Users\Ahmed\Desktop\MakefileParser-ToolDemo\Makef

Makefile Viewer:

```
14 receiver := receiver$(ext)
15 executables := $(sender) $(receiver)
16
17 $(sender)_src= sender_src$(srcExt) senc
18 $(receiver)_src = main_rcv$(srcExt) soc
19
20 install : $(executables)
21
22 define ProgramMacro =
23   $(1) : $$($(1)_src)
24 endef
25
26 $(foreach exec,$(ex
27
28 $(executables):
29   $(cmd) $^ -o $@
30
31 %.dat : %$(ext)
32   getData $^ -o $@
33
34 ifneq ($(javaComp),
```

Variables:
- executables
- ProgramMacro
- _src
  - Reference [17]
  - Reference [18]
  - Reference [23]
- sender.jar_src
- sender.o_src
- receiver.jar_src
- receiver.o_src

Rules:

Cyclic Dependency

Cyclic Dependency detected at line(s) [17,23,28,31]

Code-Smell Description:
A cyclic dependency exists between rules 'sender.jar' and '%.dat' through ('SYM100001' <- $(wildcard *.dat'))

OK

Tasks Panel

Rename Variable | Rename Target | Extract New Target | Detect Code Smells

Detected Code Smells:

Smell	Lines	Description
CD	17,23,28,31	A cyclic dependency exi...

CD: A Cyclic Dependency Smell
DP: Duplicate Prerequisites Smell
DR: Duplicate Recipes
RV: Recursive Loop of Variable

Figure 8. Cyclic dependency detection

the variable's view, SYMake highlights all corresponding locations where the variable is initialized/referenced. Figure 5 shows SYMake as a user selects 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 and recipe and the respective code locations.

B. *Renaming and Extracting*

To rename, the user selects a variable and clicks on *Rename Variable*. A pop-up window will ask the user for the new name. Figure 7 shows SYMake when the user requests to rename the suffix _src to _libs. 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.

C. *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 elements involved in that smell/error.

Figure 8 shows an example of a detected smell. A detected cyclic dependency in myMakefile is shown at the lines 17, 23, 28, and 31, and between the rules sender.jar and %.dat.

IV. Related Work

Prior work has shown that the maintenance of build files could cause a high percentage of overhead on general development efforts in a software process [3], [4], [5], [6]. 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 [7], [8]. It provides visualization and code smell detection supports for Makefiles. There are key departure points in SYMake in comparison with MAKAO. First, SYMake aims to provide program analysis on Make build code. MAKAO focuses more on visualization and reverse engineering for different views on build architecture. Moreover, MAKAO can only work on *concrete dependency graph* for a Makefile, thus it cannot support renaming/extracting, and code smell detections for Make code as in SYMake. As seen in Section I, due to dynamicism, program elements in a Makefile are not always fully exposed in build code (i.e. before evaluation phase).

Another work from Gunter [9] aims to formulate the execution phase on a concrete dependency graph (CDG), rather than the evaluation phase. CDG is modeled as a special Petri net and the execution phase is modeled as a marking process from source nodes to target ones. That work does not aim to support build code analysis as in SYMake.

V. Conclusions

SYMake [1] is a tool for build code analysis in Makefiles that is based on symbolic evaluation to statically detect code smells and errors and support renaming and extracting variables/targets. Tool demo can be viewed online [2].

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," Submitted to ICSE 2012.

[3] L. Hochstein and Y. Jiao, "The cost of the build tax in scientific software," in *ACM/IEEE 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, K. D. Schutter, H. Tromp, and W. D. euter, "The evolution of the linux build system," *Electronic Communications of the ECEASST*, vol. 8, 2008.

[6] S. McIntosh, B. Adams, and A. E. Hassan, "The evolution of ant build systems," in *MSR*, 2010, pp. 42–51.

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

[8] ——, "Makao," in *ICSM 2007*. pp. 517 –518, IEEE.

[9] C. A. Gunter, "Abstracting dependencies between software configuration items," *ACM TOSEM*, vol. 9, no. 1, pp. 94–131, 2000.

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