Build Code Analysis with Symbolic Evaluation

Ahmed Tamrawi, Hoan Anh Nguyen, Hung Viet Nguyen, Tien N. Nguyen
Electrical and Computer Engineering Department
Iowa State University
{*atamrawi,hoan,hungnv,tien*}@*iastate.edu*

***Abstract*—Build process is crucial in software development. However, the analysis support for build code is still limited. In this paper, we present SYMake, an infrastructure and tool for the analysis of build code in make. Due to the dynamic nature of make language, it is challenging to understand and maintain complex Makefiles. SYMake provides a symbolic evaluation algorithm that processes Makefiles and produces a *symbolic dependency graph* (SDG), which represents the build dependencies (i.e. rules) among files via commands. During the symbolic evaluation, for each resulting string value in an SDG that represents a part of a file name or a command in a rule, SYMake provides also an acyclic graph (called T-model) to represent its *symbolic evaluation trace*. We have used SYMake to develop algorithms and a tool 1) to detect several types of code smells and errors in Makefiles, and 2) to support build code refactoring, e.g. renaming a variable/target even if its name is fragmented and built from multiple substrings. Our empirical evaluation for SYMake's renaming on several real-world systems showed its high accuracy in entity renaming. Our controlled experiment showed that with SYMake, developers were able to understand Makefiles better and to detect more code smells as well as to perform refactoring more accurately.**

***Keywords*-build code maintenance; build code analysis**

I. Introduction

Software building is the process that converts and integrates source code, libraries, and other data in a software project into stand-alone deliverables and executable files. The build process is managed by a *build tool*, i.e. a program that coordinates and controls others [1]. A build tool needs to execute the *build commands* according to the *rules* specified in *build files*, which are written in a build language supported by the tool. Popular build tools are make, ant, and maven.

Prior research found that build maintenance could impose from 12%-36% overhead on software development [20]. In a large-scale system, build files grow quickly and become very complex because they must support the building of the same software in multiple platforms with various configuration and environment parameters [4]. McIntosh *et al.* [5] found that from 4-27% of tasks involving source code changes require an accompanied change in the related build code. They concluded that build code continually evolves and is likely to have defects due to high churn rate [5]. Importantly, those studies call for better tool support for build code.

Toward providing automatic tool support for developers to deal with complex build code, we have developed SYMake, an infrastructure and tool for the analysis of build code in GNU make. make is a scripting language in which a build file (called *Makefile*) is used to specify the build dependencies among the configuration files in a project via make's program entities. With a specific input/environment, make first evaluates a Makefile into a dependency graph among concrete file names and commands. Then, it executes the commands with those files. With such dynamic nature in make's evaluation, it is challenging for developers to understand and maintain over time multiple large, complex, and dependent Makefiles. Importantly, errors are hard to detect at static time and even at run time as the evaluation result depends on the input, the operating environment, and the files in the file system.

To address those challenges in the maintenance of build code in Makefiles, SYMake provides a symbolic evaluation algorithm that processes Makefiles and produces a single *symbolic dependency graph* (SDG) to represent the build rules and dependencies among files via build commands. It differs from a concrete dependency graph of make in that file names and commands in an SDG might not be completely resolved into strings. Instead, the SDG's node for a file refers to a data structure, called *V-model*, i.e. a graph-based representation for *symbolic string values* for the file's name. A V-model often contains symbols to represent the inputs or data retrieved from user environment. SDG enables static analysis on Makefiles and supports program understanding.

During the symbolic evaluation, for each resulting string value that represents a part of a file name or a command of a rule in an SDG, SYMake provides also an acyclic graph (called *T-model*) to represent its *symbolic evaluation trace*. That is, the T-model shows how that string value is initialized and manipulated via various Makefile's program entities.

We used SYMake to develop algorithms and a tool to detect several types of code smells and errors in Makefiles, e.g. cyclic dependencies, rule inclusion, duplicate prerequisites, recursive variable loops, etc. The tool supports also build code refactoring e.g. rule extraction/removal, target creation, target/variable renaming, prerequisite extraction, etc.

Our empirical evaluation for SYMake's renaming on several real-world systems has shown that it can achieve high accuracy in entity renaming. We also conducted a controlled experiment whose result showed that with SYMake, human subjects were able to understand the Makefiles better and to detect more code smells as well as to perform refactoring more accurately in shorter time. Our contributions include:

1. An AST building algorithm and a symbolic evaluation algorithm on Makefiles to create SDGs (Sections III and IV),
2. A symbolic evaluation tracing algorithm (Section V),
3. Makefiles' code smell detection algorithm (Section VI),
4. Automatic refactoring algorithm for make elements that is able to handle fragmented identifiers (Section VII),
5. An empirical evaluation and a controlled experiment to show SYMake's accuracy and usefulness (Section VIII).

II. Motivating Example

This section explains how make works via an example. Figure 1 shows myMakefile, a Makefile inspired from GNU make's documentation [6]. The goal of this file is to tell make how to build the *demo*, *server*, and *client* programs in both Linux and Windows from its respective source and data files.

GNU make processes a Makefile in two distinct phases:

- **Evaluation phase**: For an input and environment, make first resolves all variables and expressions into concrete values to produce a *concrete set of rules*, and *internally* constructs a *dependency graph*. Each rule in that graph has a dependency between *prerequisites* and *targets*, and a *recipe* (i.e. shell commands) to generate the targets from their prerequisites. Figure 2 shows the rules after the evaluation as make runs on myMakefile on Linux.
- **Execution phase**: make then uses the constructed dependency graph and executes the required rules based on their prerequisites and recipes. For example, assume that user's command is 'make all', the rule for target all is executed, which leads to the rules for server.o and client.o (Figure 2). If the modifying time of a prerequisite file is later than that of its target file, the command is executed to produce the updated version of the target file.

Variable OS models the current operating system (line 1). Its value is obtained via a call to the shell built-in function to execute the shell command uname. Depending on the value of OS ('Linux' or not), the variables ext (i.e. file extension) and cmd (i.e. build command) are set with different values (lines 3-9). The server's and client's file names are stored in serverNM and clientNM. The variable programs (line 13) is evaluated into a list of both names: 'server.o client.o'.

Four variables at lines 15-18 are used to store the names of libraries and object files. At line 15, the name of the variable server.o_libs is fragmented and is constructed from the value of expression $(serverNM). It is initialized with the concatenation of 'priv protocol' and the file names in the current directory that match with the pattern '*.conf' via wildcard.

Line 20 is a rule without a recipe for the target all. After the evaluation phase, make transforms that rule into line 1 of Figure 2, where the variable programs is replaced with its actual value 'server.o client.o'. To update or build for the target all, make needs to create or update each of its prerequisites, which in turn can be either a file or a different target.

Lines 22-24 show another way for variable initialization. Variable ProgramTmp is initialized with the string at line 23.

```
1  OS := $(shell uname)
2
3  ifeq ($(OS),Linux)
4     ext = o
5     cmd = build.sh
6  else
7     ext = exe
8     cmd = build.bat
9  endif
10
11 serverNM := server.$(ext)
12 clientNM := client.$(ext)
13 programs := $(serverNM) $(clientNM)
14
15 $(serverNM)_libs = priv protocol $(wildcard *.conf)
16 $(serverNM)_objs = server_impl.$(ext) server_access.$(ext)
17 $(clientNM)_objs = client_impl.$(ext) client_api.$(ext)
18 $(clientNM)_libs = protocol
19
20 all: $(programs)
21
22 define ProgramTmp =
23    $(1): $$($(1)_objs) $$($(1)_libs)
24 endef
25
26 $(foreach prog,$(programs),$(eval $(call ProgramTmp,$(prog))))
27
28 $(programs):
29       $(cmd) $@ $^
30
31 %.conf : %.$(ext)
32       genConf $^ −o $@
33
34 ifeq ($(OS),Linux)
35 demo.o : demo.c linux.conf
36       install $^ −o $@
37 else
38 demo.exe : demo.c win.conf
39       install.bat $^ −o $@
40 endef
```

Figure 1. myMakefile: An Example of make Build Code

```
1  all: server.o client.o
2
3  server.o : server_impl.o server_access.o priv protocol sample.conf
4       build.sh server.o server_impl.o server_access.o priv protocol sample.
            conf
5
6  client.o : client_impl.o client_api.o protocol
7       build.sh client.o client_impl.o client_api.o protocol
8
9  linux.conf : linux.o
10      genConf linux.o −o linux.conf
11
12 sample.conf : sample.o
13      genConf sample.o −o sample.conf
14
15 demo.o : demo.c linux.conf
16      install demo.c linux.conf −o demo.o
```

Figure 2. Internal Representation after Evaluation Phase on myMakefile

As it is evaluated, the string at line 23 will be evaluated. In fact, in this example, ProgramTmp is used as a *user-defined function* with the built-in function call at line 26.

Line 26 shows a foreach loop with the variable programs as the iteration list and prog as its iterator. The body of foreach is a call to eval built-in function, which parses its argument into Makefile's rules/statements and considers them as part of the current Makefile. Thus, a function call is made to evaluate ProgramTmp and its parameters are assigned with the values of the temporary variables $(0), $(1), etc. For example, $(1) (line 23) contains the variable prog's value. Thus, at the first iteration, a reference to $(1) is resolved to 'server.o' as prog's value is 'server.o'. The returned value from call is 'server.o : $$(server.o_objs) $$(server.o_libs)', which is a parameter passed to eval. It is parsed into a new rule server.o whose prerequisites are the values of the variables server.o_objs and server.o_libs. The same applies in the second iteration as prog's value is 'client.o'. Thus, lines 3 and 6 of Figure 2 show the rules created from evaluating line 26 of Figure 1.

Lines 28-29 define a rule for two targets named 'server.o' and 'client.o' via $(programs). That is, both targets share the same recipe at line 29. Since the target 'server.o' occurs in the previous rule, those rules are combined to form a complete rule at lines 3-4 (Figure 2). The same applies to client.o rule (lines 6-7). Note that $@ and $^ are *automatic variables* whose values are equal to the rule's target and prerequisites.

An implicit rule serves as a template/pattern for any prerequisite file that does not have an explicit rule for creating/updating. Lines 31-32 specify the rule to produce any configuration file %.conf from the corresponding %.o or %.exe file. For example, linux.conf at line 35 is a prerequisite file for demo.o and there is no explicit rule to make that file. make finds that linux.conf matches with %.conf. Thus, it creates a new rule for linux.conf (lines 9-10, Figure 2). The same process applies when make sees sample.conf at line 3 of Figure 2. It creates a new rule sample.conf as in lines 12-13.

Scenario 1. Assume that a user enters 'make -f myMakefile' on a Linux machine. make builds its dependency graph (Figure 2), and runs the first rule (i.e. rule all). It examines the prerequisites, and then executes the rules at lines 3 and 6. It continues the same process and successfully builds for the rule all.

Interestingly, there is a subtle error in myMakefile if 'make all' is requested. The server.o rule depends on the configuration files fetched from the current directory via $(wildcard *.conf) (line 15). The error will occur if a configuration file in the current directory has the name server.conf. In that case, make will consider server.conf as a prerequisite of server.o. It will match server.conf with the implicit rule at line 31, create the server.conf rule, and add it to the dependency graph:

```
server.conf : server.o
        genConf server.o −o server.conf
```

A cyclic dependency now occurs because server.o lists server.conf as one of its prerequisites, and server.conf also has server.o as a prerequisite in the new rule. That loop 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 users' current environment/directory, and the input.

Scenario 2. 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. At line 15, the prefix of an identifier is defined from the value of the variable $(serverNM). A regular text search tool cannot distinguish between the identifiers and the string values (e.g. line 11). Moreover, the identifier of variable $(serverNM)_objs (line 16) is fragmented and composed of multiple substrings. If a tool renames the suffix objs, it must also rename at line 23 and line 17 since line 23 affects both of these lines.

Scenario 3. Over time, a different developer works on a different component of the project, and myMakefile includes other Makefiles (e.g. mk1). Assume that (s)he adds into his/her own file mk1 a rule comp1.conf to handle the building of the configuration file for his/her component as follows:

```
comp1.conf : comp1.o
      genConf comp1.o −o comp1.conf
```

Dealing with multiple, dependent, and complex Makefiles, (s)he might not be aware of the implicit rule at line 31 in myMakefile that was designed to handle any .conf files. Such redundancy increases the complexity and decreases the maintainability of the Makefiles. If one wants to change the implicit rule at line 31, (s)he must change comp1.conf rule in mk1. Thus, it is helpful to detect rule inclusion/redundancy.

Summary. The analysis of Makefiles is challenging because:

1) The analysis for entities' *names* and build *dependencies* is not trivial due to the dynamic nature of make. It is also hard to detect code smells and errors at static time.

2) As a project evolves, Makefiles become more complex. Bad smells such as rule redundancy, duplications, circular dependencies, etc create several maintenance problems.

3) Understanding large, complicated, and dependent Makefiles requires much effort and time from developers.

III. Make's Abstract Syntax Tree

Let us describe make's syntactical rules that we use to build an Abstract Syntax Tree (AST) for a Makefile. We have read GNU make's documentation and source code [6], and specified its grammar production rules as in Figure 3.

1. *Makefile* node (rule 1) is the root of a Makefile's AST. A Makefile consists of a list of *statements/rules*. The order of statements/rules in a Makefile is important in its execution.

2. *Statement* node (rule 2): A Makefile's statement represents a source code line or block that occurs as a stand-alone evaluation unit. Typical statements are in line 2 of Figure 3.

3. *Assignment* node (rule 3): An assignment to a variable can be either *simple* or *recursive*. In a simple variable assignment (:=), the right hand side (RHS) expression is *evaluated* and assigned to the variable. A recursive one (=) is similar to a pointer assignment with no update from LHS to RHS variables. A variable can be referred by its name or an expression whose value is its name. Line 15 of Figure 1 defines a variable server.o_libs since $(serverNM)_libs is evaluated first.

4. *Definition* node (rule 6): A variable defined within a define can be used in three ways. First, a defined variable

```
1)  Makefile → {Statement|Rule}
2)  Statement → Assignment|Definition|FunctionCall|Foreach|If|Directive
3)  Assignment → [private|export|override] (Id|Expr) (+=|:=|=) Expr
4)  Id → IdPart ((WS)* IdPart)*
5)  IdPart → [^WS = : ; \n]+
6)  Definition → [private|export|override] define Id [+=|:=|=] \n ∼\n
    endef
7)  FunctionCall → $(FunctionName [Expr[{,Expr}]])
8)  FunctionName → subst|patsubst|strip|findstring|filter|...
9)  Expr → Term{[WS]Term}
10) Term → FunctionCall|ELiteral|Evaluation|Foreach|If
11) ELiteral → WLiteral ((\WS)* WLiteral)*
12) WLiteral → [^WS \n]+
13) Evaluation → $(Id|Expr)
14) Rule → Expr (: | ::) (Assignment| [|]Expr)[Recipe]
15) Recipe → (;|\n\t)RecipeExpr{\n\t RecipeExpr}
16) RecipeExpr → RecipeTerm{[WS]RecipeTerm}
17) RecipeTerm → FunctionCall|Evaluation|RecipeLiteral|AutoEval|Foreach|If
18) RecipeLiteral → [^\n]+
19) AutoEval → $@|$<|$?|$^|$+|...
20) Foreach → $(foreach Id,Expr,Expr|RecipeExpr)
21) If → (((ifeq|ifneq) (Expr,Expr) | ((ifdef|ifndef) Expr))
    {Rule|Statement}|RecipePart [else {Rule|Statement}|RecipePart] endif) |
    $(if Expr,Expr|RecipeExpr[,Expr|RecipeExpr]))
22) RecipePart → \n\t RecipeExpr{\n\t RecipeExpr}
23) Directive → Include|Vpath|Export|Undefine
24) Include → (include|sinclude|-include) Expr
25) Vpath → vpath [Expr]
26) Export → (unexport|export) [Expr]
27) Undefine → [override] undefine Id
```

Figure 3. Makefile Abstract-Syntax Production Rules

can be used as an assigned variable, except that its value can contain multiple lines. Second, it can serve as a user-defined function as it is used with Make's call (ProgramTmp, line 26). Finally, it can be used later with eval function and evaluated into a list of statements and/or rules. The resulting statements and rules are treated as part of the current Makefile and make continues the evaluation to build the dependency graph.

5. *FunctionCall* node (rule 7) represents a call to a built-in function. A few built-in functions are listed in rule 8.

6. *Expr* node (rule 9): An expression represents a part of a line that will be evaluated. It can be a concatenation of terms (i.e. Terms) with or without whitespaces in-between. A term can be a function call, variable evaluation, a foreach or if statement (rule 10). It can also be a literal (ELiteral).

7. *Evaluation* node (rule 13) is for a variable's evaluation. E.g., $(programs) is evaluated into 'server.o client.o' (line 20).

8. *Rule* node (rule 14) represents a build action. It has:

- A target, which can be an expression that is evaluated into one or multiple targets. E.g., $(programs) is evaluated to create 2 targets server.o and client.o (line 28).
- An expression whose value after evaluation will result in a set of files forming the prerequisite list for the rule.
- A recipe representing the shell commands of the rule.

A rule can be 1) non-terminal rule(:), i.e. it is executed as requested in a command-line or if its target is a prerequisite of another to-be-executed rule; or 2) terminal rule (::), i.e., it is executed once its prerequisites exist. The same target name can occur in multiple rules, and those must be of the same type. If they are non-terminal, they will be combined into a single one. A rule can contain a local variable assignment.

9. *Recipe* node (rule 15) models a concatenation of recipe expression(s) that is evaluated to a string forming the rule's commands. It will be evaluated into a list of shell commands. The expression within a recipe (RecipeExpr) is the same as a regular Expr, except that if it has a string literal, the literal must not include a new line. We call it RecipeLiteral (rule 18).

10. *AutoEval* (rule 19) is used to evaluate automatic variables that can occur only in a recipe (line 36, Figure 1).

11. *Foreach* node (rule 20): A foreach statement consists of three parts: 1) an identifier node as the loop iterator (e.g. prog at line 26); 2) an expression whose value represents the iteration list (e.g. $(programs)); and 3) an Expr or RecipeExpr representing the loop body that will be evaluated (e.g. $(eval $(call ProgramTmp,$(prog)))). The return value after executing a foreach is a concatenated string of all values resulted from the evaluation of the body expression at each iteration.

12. *If* node (rule 21) represents a condition. It can be either a conditional block (e.g. as in lines 3-9) or an inline condition (e.g. $(if $(var1),$(var2),$(var3))). In a conditional block, a branch can contain any combination of Makefile rules/statements. A branch can also be evaluated into a part of a rule's recipe (represented by RecipePart). The value of an if statement will be viewed as part of the current Makefile. An inline condition is used to check an empty string.

13. A *directive* is aimed to tell Make 1) to include another Makefile (include), 2) to undefine a variable (undefine), 3) to add/remove the variables from the environment ((un)export), and 4) to setup Make's searching path (Vpath).

IV. Makefile Symbolic Dependency Graph (SDG)

An SDG captures the dependencies among prerequisites and targets, and respective recipes. An SDG's node refers to a data structure, called *V-model*, representing a *symbolic string value* for a name. A V-model can contain symbols to represent the inputs or data retrieved from user environment.

Definition 1: A **Symbolic Dependency Graph** (SDG) is a directed graph representing a Makefile's rules and dependencies among prerequisites and targets through the recipes. An SDG can contain the following types of nodes:

- A **target/prerequisite** node represents a rule's target or prerequisite. A target can be a prerequisite for another rule and vice versa. It refers to a V-model representing the symbolic string value of the target/prerequisite.
- A **recipe** node represents the recipe of a rule. There is an *edge* from a target node to the recipe node representing the dependency of that target to this recipe. Similarly, there is an *edge* from the recipe node to a prerequisite node. A recipe node also refers to a V-model.
- A **Select** node represents alternative dependencies from a target to one of multiple recipes and prerequisites.
- A **rule** block contains all nodes/edges related to a rule.

Definition 2: A **V-model** is a labeled, ordered, and directed acyclic graph representing the symbolic string values for the parts of a rule in an SDG. In a V-model, leaf nodes represent string values. Inner nodes model either *operations* for combining those values or *references* to other V-models.

1. There are two kinds of leaf nodes:

- a *literal* node represents a concrete string value, and
- a *symbolic* node represents an undetermined/unresolved string value (e.g. an environment value).

2. There are three types of inner nodes:

- A **Concat** node represents a value concatenated from the values corresponding to the sub-trees of that node. The order of the sub-trees is that of the concatenation.
- A **Select** node represents a value that could be selected from either of the values corresponding to its sub-trees.
- A **Reference** node represents a reference to a variable and its child is a V-model representing the value of that variable. For a simple variable, its V-model does not contain any reference nodes. For a recursive variable, its V-model may contain multiple reference nodes since it can be computed from multiple recursive variables.

3. The nodes on V-models have their attributes describing additional information, such as the Makefile expressions and code positions associated with literal and symbolic nodes.

Figure 4a shows part of the SDG and its V-models for Figure 1. With its **Select** node, the target node all (line 20) depends on either of the two recipes and the corresponding prerequisites: one for Linux and one for Windows (line 3). The recipes rcp1 and rcp2 are empty and do not refer to any V-model. Thus, to build for target all, make must build either {server,client}.o or {server,client}.exe. The file server.o depends on the execution of the recipe rcp3. Each part of a rule refers to the respective V-model (not all are shown). For example, the string content of rcp3 is represented by a V-model, which is the concatenation of several literals and **Concat** nodes. Note that the last **Concat** represents the concatenation of two strings priv and protocol, and a symbol SYM01 representing the result returned from a call to wildcard to get the files from the current directory. The V-model for 'server_impl.o' starts with a **Concat** between a literal node 'server_impl' and a V-model for the variable ext, which in turn starts with a reference node Ref:ext pointing to the string literal 'o'. That reference node allows ext to be used as either a value or a reference.

Figure 4b represents the implicit rule %.conf (line 31). The string content of the recipe is computed via the respective V-model in each case, which contains no symbolic node.

Building SDG and V-models via Symbolic Evaluation

Our symbolic evaluation algorithm recursively evaluates a Makefile for all statements/rules in all branches, updating/creating small V-models, and combines them into larger ones. Moreover, it extracts all the resulting rules in all branches, and combines the scattered ones into single rules if needed. SYMake processes statements/rules as in Table I:

1. var := E: SYMake maintains for each variable var a V-model corresponding to its most recent value during symbolic evaluation. As meeting a *simple* assignment, it expands

Table I
SYMBOLIC EVALUATION RULES TO BUILD SDG FROM A MAKEFILE

Makefile Syntax	**Evaluation Rule**
1. var := E	var.simple = true, var.V = new RefNode(Expand(E.V))
2. var = E	var.simple = false, var.V = new RefNode(E.V)
3. var += E	if (var is defined) if (var.simple = true) var.V = new RefNode(new Concat(Expand(var.V), Expand(E.V))) else var.V = new RefNode(new Concat(var.V.child,E.V)) else var.V = new RefNode(E.V), var.simple = false
4. define var := L endef	var.simple = true, var.V = new RefNode(Expand(Build-V(L)))
5. define var = L endef	var.simple = false, var.V = new RefNode(Build-V(L))
6. define var += L endef	if (var is defined) if (var.simple = true) var.V = new RefNode(new Concat(Expand(var.V, Expand(Build-V(L)))) else var.V = new RefNode(new Concat(var.V.child, Build-V(L))) else var.V = new RefNode(Build-V(L)), var.simple=false
7. E $\rightarrow$ **\$(call** E*, $\{E_i\}$)	$\forall i$ i.V = new RefNode(Expand(E_i.V)), E.V = Expand(E*.V)
8. E $\rightarrow$ **\$(eval** E*)	E.V = $\emptyset$, Eval(Parse(Flatten(E*.V)))
9. E $\rightarrow$ \$(func $\{$E$_i\}$)	func.retValue = $\emptyset$, $\forall i$ func.param$_i$.V = Expand(E$_i$.V), eval func, E.V = Expand(func.retValue.V)
10. E $\rightarrow$ \$(var)	if (var is defined) E.V= var.V else E.V= new Symbol()
11. E_1 $\rightarrow$ \$(E_2)	E_1.V=new Select($\{$getvar(Id_i).V:$Id_i$$\in$Flatten($E_2$.V)$\}$)
12. E $\rightarrow$ **\$(foreach** Id, E_1, E_2)	E.V = new Select($\{$Build(e_i) : e_i $\in$ Flatten(E_1.V)$\}$), Build(e_i)= new Concat($\{$Expand(E_2.V): $Id_j$$\in$Tokenize($e_i$), Id.V = new RefNode(new LiteralNode(Id_j))$\}$)
13. E $\rightarrow$ WLiteral	E.V = new LiteralNode(WLiteral)
14. E $\rightarrow$ ELiteral	E.V = new LiteralNode(ELiteral)
15. E $\rightarrow$ RecipeLiteral	E.V = new Concat(Tokenize(RecipeLiteral))
16. **undefine** var	var.V = new Symbol()
17. include E	$\{$Eval(Parse(f_j)): f_j $\in$ Tokenize($e_i$$\in$Flatten(E.V))$\}$
18. E_1:E_2(Recipe)	$\{$BuildRule(t_{ij}):$t_{ij}$$\in$Tokenize($T_j$),$T_j$$\in$Flatten($E_1$.V)$\}$ BuildRule(t_{ij}): R = getRule(t_{ij}) if (R is defined) UpdateRule(R, E_2.V) else new Rule(t_{ij}, E_2.V, Recipe)
19. E_1::E_2(Recipe)	$\{\{$new Rule(t_{ij}, E_2.V, Recipe)$\}$:t_{ij} $\in$ Tokenize(T_j), T_j $\in$ Flatten(E_1.V)$\}$
20. Recipe $\rightarrow$ $(;\mid\backslash n\backslash t)E_1\{\backslash n\backslash t\ E_i\}$	Add a new RecipeNode(new Select($\{$new Concat (Tokenize(r_j)):r_j $\in$ Flatten((new Concat(E_i.V)))$\}$))
21. RecipePart $\rightarrow$ $\backslash n\backslash t\ E_1\{\backslash n\backslash t\ E_i\}$	RecipePart.V = new Select($\{$new Concat(Tokenize(r_i)):r_i $\in$ Flatten((new Concat(E_i.V)))$\}$)
22. E $\rightarrow$ RecipePart	E.V = RecipePart.V
23. E_1 $\rightarrow$ if (E_2) E_3 else E_4	eval E_2, $\forall$var$\in$VARS*, var.V=new RefNode(new Select(var$_{E_3}$.V, var$_{E_4}$.V)), if E_3,E_4: RecipePart $\Rightarrow$ E_1.V=new Select(E_3.V,E_4.V) RULES$_{E_3}$$\in$FindRules($E_3$), RULES$_{E_4}$$\in$FindRules($E_4$) $\forall$ $rule$ $\in$ (RULES$_{E_3}$ $\cup$ RULES$_{E_4}$): R = getRule($rule.t$) if ((R is defined) AND ($rule$ is single-colon)) Update_If_Rule(R, rule$_{E_3}$, rule$_{E_4}$) else new If_Rule(rule$_{E_3}$, rule$_{E_4}$)
24. E_1 $\rightarrow$ \$(if E_2, E_3,E_4)	eval E_2, E.V = new Select(Expand(E_3.V), Expand(E_4.V))

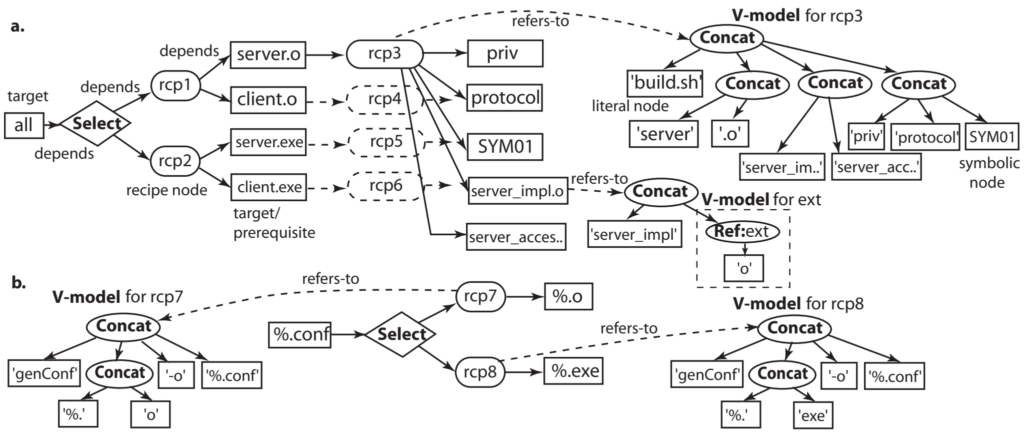

Figure 4. Symbolic Dependency Graph and V-models

(*de-references*) the V-model of E to replace all reference nodes with their children in a breadth-first traversal. It then creates a reference node Ref:var and attaches the expanded V-model as the child of Ref:var. The V-model rooted at Ref:var models var's recent value. var is marked as a simple variable.

2. var = E: similarly to the simple assignment in case 1. However, no expansion is needed since var is recursive.

3. var += E (concatenating E with the current value): If var is undefined, make considers this as recursive variable assignment, and SYMake uses rule 2. If var is simple and defined, it assigns var a new V-model rooted at a new reference node Ref:var. The child of the reference node is a newly created **Concat** of the expanded versions of the current V-models of var and E. If var is recursive, it handles similarly except that the reference node in the old V-model of var is skipped.

4. define var := L endef: SYMake first parses the string L and uses Table I to build the V-model from L. Then, it similarly expands that V-model and creates the reference node as in 1.

5. define var = L endef: Similar to the rule 4 except that the expansion of V-model is not needed since var is recursive.

6. define var += L endef: If var is undefined (no value yet), it treats this as in rule 5. Otherwise, it uses rule 3 except that it parses L and builds the V-model with the rules in Table I.

7. E $\rightarrow$ **$(call** E*,{$E_i$}**)**: SYMake first creates a temporary variable i for each parameter E_i, and assigns its value to i by making the expanded V-model of E_i become a child of a new reference node for each i. Then, it creates a new V-model by expanding the V-model of the expression E*. That new V-model corresponds to the return value of function call.

8. E $\rightarrow$ **$(eval** E***)**: SYMake first flattens the V-model of the expression E* (Figure 5). The result is a set of symbolic or literal nodes each of which represents a possible value of E* via a specific path in that V-model. It then parses the content of each node into rules/statements and evaluates them as part of the current Makefile. Symbols are treated as strings.

9. E $\rightarrow$ $(func {E$_i$}). For a built-in function, SYMake assigns the V-models of its actual arguments to formal parameters, and evaluates the body. A V-model is created for the returned value. The V-model for E is the expanded version of that new V-model since the returned value must not contain any references. Symbolic values are used if needed.

10. E $\rightarrow$ $(var): When a variable var is retrieved for a computation, its latest V-model is used. However, if var does not have any V-model, SYMake returns a symbolic node.

11. $E_1 \rightarrow \$(E_2)$: In this case, the value of E_2 is resolved to variable identifier. However, E_2 may have different values with different paths. Thus, SYMake first flattens the V-model of E_2 to a set of string literal and symbolic nodes. Each node represents a possible variable identifier Id_i through a distinct path. Then, it creates a new V-model rooted at a new **Select** node whose children are the V-models of all variables Id_is.

12. E $\rightarrow$ **$(foreach** Id, E_1, E_2**)**: First SYMake flattens E_1. The returned set of symbolic/literal nodes represents all the possible iteration lists. For each list e_i, SYMake tokenizes the contents. Then, for each token Id_j, it creates a new V-model rooted at a new reference node whose child is a new literal node for that token. For each possible value e_i of E_1, SYMake creates a new V-model rooted at a **Concat** node and its children are the expanded V-models of the expression E_2 at each token Id_j. Finally, SYMake appends each of the new V-models with those **Concat**s to a new V-model rooted at a new **Select** node. This new V-model at this **Select** represents all possible values for the foreach statement.

13, 14. E $\rightarrow$ WLiteral/ELiteral: It creates a new V-model with one literal node whose string content is the literal string.

15. E $\rightarrow$ RecipeLiteral: It tokenizes the literal and creates a new literal node for each token. Then, it creates a V-model rooted at a **Concat** whose children are the new literal nodes.

16. undefine var: a symbol is used after var was undefined.

17. include E: SYMake first flattens the V-model of E to find all possible values e_is. It tokenizes each string literal to get the tokens f_j as file names. If the file is available, it parses the contents into rules/statements and symbolically evaluates them as part of the current Makefile.

18. E_1:E_2(Recipe): For a single-colon rule, SYMake flattens the V-model of E_1 to all possible values T_is. Each string T_i is tokenized to find the list of targets. Then, for each resulting target t_{ij}, it checks if there exists a rule R with the same target name via getRule(t_{ij}). If R exists, it updates R using UpdateRule. UpdateRule flattens the V-model of E_2 to all possible values representing all possible prerequisites of the rule and combines them with R's prerequisites. Then, for each set of possible combinations of prerequisites, it builds a rule graph as follows: the Recipe node is connected to all combined prerequisites. The target t_{ij} is connected to Recipe through a new **Select** node (see Figure 4). Each path of the Select node represents a set of recipes and prerequisites of the rule t_{ij}. If such a rule does not exist, SYMake creates a new Makefile rule, but without a **Select** node.

19. E_1::E_2(Recipe): Similar to a sub-case of case 18 where the rule is not defined earlier and no combination is needed.

20. Recipe $\rightarrow$ (;|\n\t)E_1{\n\t E_i}: For a recipe, SYMake creates a new RecipeNode, which refers to its V-model rooted at a **Select** node. Each child node represents a possible recipe string r_j resulted from flattening the V-model that starts at a **Concat** connecting all the V-models of the expressions E_is. Each r_j is then represented by a V-model rooted at a new **Concat** node connecting the different tokens of r_j.

21. RecipePart $\rightarrow$ \n\t E_1{\n\t E_i}: Similar to rule 20, however, SYMake does not create a RecipeNode.

22. E $\rightarrow$ RecipePart: E gets the V-model of RecipePart.

23. $E_1 \rightarrow$ if (E_2) E_3 else E_4: it processes as follows:

First, it collects into a set VARS* all variables modified or initialized in either branch. Let us use varE_3.V and varE_4.V to denote the V-models of var after evaluating each branch, respectively. For each var in VARS*, SYMake updates its value with a new V-model rooted at a new reference node Ref:var and its child is a new **Select** node whose children are varE_3.V and varE_4.V. If the else branch is empty, the latest V-model for var before if is used in place of varE_4.V.

Second, it collects into a set ($\text{RULES}_{E_3} \cup \text{RULES}_{E_4}$) all rules defined in either E_3 or E_4. For each *rule* in that set, if there exists a single-colon rule with the same target name getRule($rule.t$), it updates the existing rule R with the rules rule_{E_3} and rule_{E_4} (using Update_If_Rule). That function adds a **Select** node after the target node. Each path of the **Select** node represents a possible recipe and prerequisites for the target node corresponding to the rules rule_{E_3} and rule_{E_4}. If there exists no rule defined before that has the same target name as *rule*, SYMake creates a new rule with a **Select** node after the target node, and each of its branches represents the recipe and prerequisites of either rule_{E_3} or rule_{E_4}.

Third, if E_3 and E_4 are of type RecipePart, SYMake creates a new V-model rooted at a **Select** node whose children are the V-models of the expressions E_3 and E_4.

24. $E_1 \rightarrow$ $(if E_2,E_3,E_4): It creates a new V-model rooted at a **Select** node whose children are the expanded V-models of E_3 and E_4. If the condition is known, rule 11 is used.

```
1  function Flatten(VModel vmodel)
2    switch (vmodel.Type)
3      Literal: return new Set(vmodel.getString())
4      Symbolic: return new Set()
5      Reference: return Flatten(vmodel.Child)
6      Select: return Flatten(vmodel.Left) ∪ Flatten(vmodel.Right)
7      Concat: set ← new Set(); set.add("")
8              foreach childNode in concat.ChildNodes
9                childSet = Flatten(childNode); tmpSet = new Set()
10               foreach str1 in set
11                 foreach str2 in childSet
12                   tmpSet.add(str1 + str2)
13               set ← tmpSet
14             return set
```

Figure 5. Flatten Algorithm for a V-model

Figure 5 shows Flatten function that takes a V-model and returns the set of all possible string values represented by the V-model. Flatten computes the flattened values for a node by combining those of its children. A **Literal** leaf node has one possible value (line 3). A symbolic node has an empty set of flattened values (line 4). A **Reference** node has only one child, thus its set of flattened values is the same as that of its child (line 5). For a **Select** node, the set is the union of those for two branches (line 6). For a **Concat**, Flatten computes all possible concatenations of strings each of which is a flattened value from a child node in the children order to form a set of all possible full-length values of the **Concat** (lines 9-14). It does this by visiting each child from left to right, and concatenating each possible concatenated value of the previous children with each flattened value of this child.

V. Evaluation Trace Model

The values of any component (prerequisites, recipe, targets) of a rule are often composed and manipulated in different code locations in a Makefile. For example, server.o comes from $(1) (line 23, Figure 1), one iteration of foreach loop (line 26), an eval call with ProgramTmp (line 26), the variables prog, programs (line 13), and serverNM (line 11), etc. To support program understanding and automatic operations (e.g. renaming), we develop **T-model**, a model for tracing the evaluation steps. Figure 6 shows the T-model for server.o (left), and the T-model for SYM01 (right). A T-model is similar in spirit to an execution trace. Let us define T-model.

Definition 3 (Evaluation Trace Model): **T-model** is a labeled, directed, and acyclic graph representing how an SDG node's value is computed and manipulated through different Makefile's program elements at an evaluation point. A node refers to its code locations. Edges represent the evaluation flows. An T-model has the following types of nodes:

1) **Data node**:

- A *Variable* node has its identifier as its label. An incoming edge from a variable node to another represents a recursive variable assignment, while an incoming edge from any other node represents a simple assignment.
- A **Literal** node is the same as in a V-model.

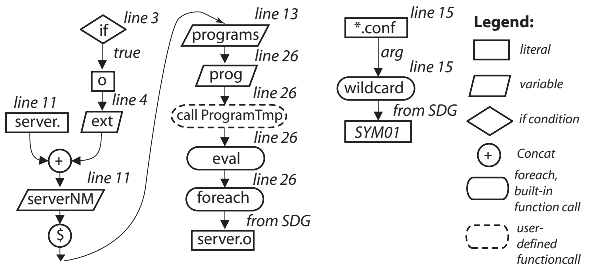

Figure 6. T-models for server.o (left), SYM01 (right)

2) **Operator/action node**:

- A **Concat** node is the same as in a V-model.
- An *Evaluation* node models the evaluation operation ($) on a variable. If two variable nodes are connected via an evaluation node, it is a simple variable assignment.
- A *Function call* node represents a call to a built-in or user-defined function F. T-models connected to this node represent the arguments passed to F. A node *eval* represents a call to *eval* (evaluating to rules/statements).

3) **Control node**: *if* and *foreach* in the evaluation process.

Building T-model. Building T-model for any node in a rule in an SDG (e.g. part of a prerequisite, a recipe, or a target) occurs during the symbolic evaluation. Generally, for each evaluation rule in Table I that constructs a new V-model, SYMake creates the new corresponding T-model:

1) For a simple assignment, it connects the T-model of the right-hand side expression to a variable node via an $ node (e.g. programs and serverNM in Figure 6). For a recursive assignment, the T-model and the variable node are connected.

2) As seeing a literal or symbolic value, it creates a new literal/symbolic node (e.g. 'server.' and SYM01 in Figure 6).

3) For a *Concat* operation, it creates a **Concat** node whose children are the T-models corresponding to two operands.

4) For a built-in function (e.g. $(wildcard *.conf)), it connects the T-models of the parameters to a newly created *function call* node (Figure 6). For an eval call, it connects the T-model of the *computed value* from eval to a new eval node.

5) For a call function, the T-model of the returned value is linked to the new call node (e.g. T-model for prog → call).

6) For a foreach statement, the T-model of its body is linked to a new foreach node (e.g. T-model of eval → foreach).

7) For an if, it creates an if node to represent the branch it took to build the traced value (e.g. true branch in Figure 6).

VI. Detection of Code Smells and Errors

Let us present an application of our infrastructure (SDG, V-, T-models) to detect code smells/errors in a Makefile.

Cyclic Dependency. An example of cyclic dependency among targets/prerequisites is in Scenario 1 (Section II). To detect this error, we use a graph algorithm to detect a directed cycle that goes through targets, prerequisites, recipes via dependency edges. Due to the presence of symbolic nodes (i.e. with the SYM prefix), a new type of edges is defined in an SDG, called *matching* edges. Two targets/prerequisites are connected via a matching edge if either 1) their labels are matched according to make's syntax (e.g. % in implicit rules), or 2) at least one of them is from a symbolic node.

In detection, SYMake considers those matching edges in addition to dependency edges. In Figure 4, the cycle includes SYM01 (Figure 4a) $\overset{match}{\rightarrow}$ %.conf (Figure 4b) → **Select** → rcp7 → %.o $\overset{match}{\rightarrow}$ server.o (Figure 4a) → rcp3 → SYM01.

Loop of Recursive Variables. That is, a recursive variable's value is indirectly dependent on itself. To detect this, when constructing the V-model for a variable var during symbolic evaluation, SYMake detects whether its reference node points to another V-model and from there indirectly refers back to var's reference node via one or multiple V-models.

Duplicate Prerequisites. make has a mechanism to combine multiple rules with the same target name into a complete rule. When multiple developers work on a Makefile, this mechanism allows them to focus on the dependencies of their own concerns. However, it could lead to the case that two or more prerequisites of a rule are exactly matched.

SYMake detects this smell by iterating over all rules in the SDG, and comparing the simple names of prerequisite nodes for an exact name match. Symbolic nodes are discarded.

Rule Inclusion. An example of rule inclusion is in Scenario 3 (Section II) in which one developer adds a specific rule for his/her file, while a general (implicit) rule already exists. For example, the following one is included in rule 31 (Figure 1).

```
comp1.conf : comp1.o
      genConf comp1.o −o comp1.conf
```

SYMake first runs the specific rule (without %) to generalize the recipe. E.g., the above one becomes genConf $^ -o $@. Then, it tries to match the target and each prerequisite to the counterpart in the implicit rule. If the pattern is matched for all (e.g. comp1.conf and %.conf), an inclusion is detected.

In those cases, GNU make might not be able to detect errors/smells since it could run on a different path that does not involve the duplication or cycle. In some cases, the errors are not revealed until the projects and Makefiles are deployed at user environments, directories, or configurations. SYMake performs symbolic evaluation to generalize possible evaluation results, thus, is able to detect them statically.

VII. Refactoring Support with SYMake

Renaming Variable. Another application is automatic renaming, where SYMake needs to find all code locations where a variable was initialized/referenced. The key challenges are listed in Scenario 2, Section II. E.g., the variable server.o_libs at line 15 has its name composed by the value of variable serverNM (i.e. 'server.o') and the substring '_libs'. The variable server.o_libs is then referenced when ProgramTmp is called at line 26, which leads to line 23 being evaluated,

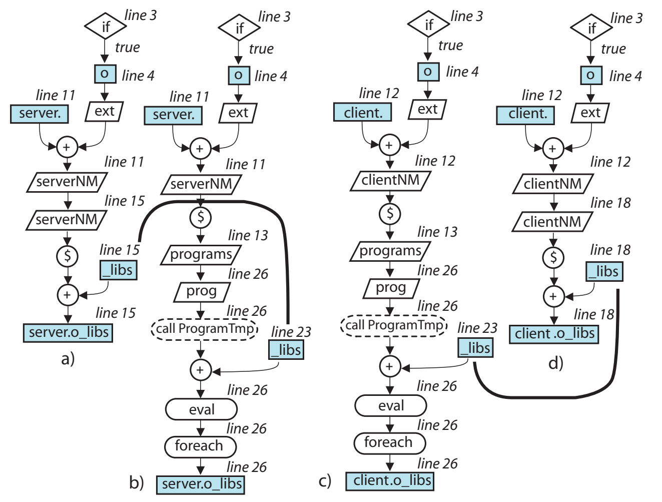

Figure 7. T-models for server.o_libs and client.o_libs

and $($(1)_libs) becomes $(server.o_libs) (i.e. a reference to server.o_libs). Thus, if one wants to change '_libs' as part of '$(serverNM)_libs' at line 15 into '_LIBS', then $($(1)_libs) at line 23 must be changed into $($(1)_LIBS). Also, '$(clientNM)_libs' at line 18 must be changed into '$(clientNM)_LIBS' since at line 23, $($(1)_libs) is evaluated into '$(clientNM)_libs' at foreach's second iteration (line 26). Thus, 3 locations of '_libs' at 15, 18, and 23 need consistent renaming. We call them a *locset*.

To support renaming for variables with their names being fragmented (called *fragmented variables*), SYMake determines a locset with the following idea. During the evaluation, SYMake keeps track of all reference locations of the same variable and for each reference string s, it builds the corresponding T-model to keep track of where all substrings of s come from. Since the strings of all references of the same variable must match, SYMake is able to identify the matched substrings from the literals of the T-models of those references and group those substrings into a locset.

For example, during the evaluation at line 15, SYMake builds the T-model for the variable server.o_libs as in Figure 7a. At line 26, it sees a reference to server.o_libs and the corresponding T-model is in Figure 7b. Since two references at lines 15 and 26 are of the same variable, it can match the literal nodes in two T-models: Figures 7a and 7b. The string server.o_libs at line 15 comes from 3 literals: 1) 'server.' (line 11), 2) 'o' (line 4), and 3) '_libs' (line 15). The string server.o_libs at line 26 comes from 3 literals: 1) 'server.' (line 11), 2) 'o' (line 4), and 3) '_libs' (line 23). Thus, '_libs' (line 15) and '_libs' (line 23) belong to the same locset. Similar reason is applied to Figures 7c and 7d, and two strings '_libs' at lines 18 and 23 are of the same locset. Thus, all three strings '_libs' at lines 15, 18, and 23 belong to the same locset.

The T-models are created/updated over time. After the evaluation of a variable reference r, the entry for r in the entity table refers to its latest T-model. All locsets are maintained after symbolic evaluation. If renaming is requested for a variable (or part of it), SYMake searches in all locsets and renames all locations in the respective locset.

Table II
SUBJECT SYSTEMS AND BUILD CODE INFORMATION

System	MakeF	LOCs	Locsets	Loc.	Frag	NonFrag	Vars	Rules	Paths	Incl
SCST[34]	49	1786	870	2230	39	831	876	112	154	0
LINN[35]	67	4020	3417	7169	53	3364	3425	134	536	0
GCC[36]	68	5350	1972	16546	11	1961	1980	804	75	5
MIN[37]	95	2374	632	3324	0	632	632	121	95	95
LINS[38]	98	1255	973	1563	5	968	973	135	98	0
FIRE[39]	156	6374	1960	4668	12	1948	1991	2635	621	130
TS[40]	232	12950	2655	9711	50	2605	2655	2541	235	210

Rename Target. SYMake manages targets similarly as variables in which target definitions are viewed as variable declarations, and their usages as references. T-models in the entity table are updated as a rule or recipe is met (rules 18-20, Table I). SYMake supports renaming of a prerequisite only if it is a target of another rule but is not a file name.

Our tool supports also other refactorings such as rule extraction/removal, target/prerequisite/recipe extraction, etc.

VIII. EMPIRICAL EVALUATION

A. Accuracy and Efficiency Evaluation in Renaming

We conducted an evaluation on SYMake's renaming accuracy. We chose the Makefiles in 7 subject systems (Table II). We asked six Ph.D. students to independently identify the *locsets for consistent renaming*. We built a tool to assist them in that oracle building task. It skips the comments, keywords, built-in function names, and parses the Makefile's contents to retrieve string tokens. For each string s, it performs text-searching for all occurrences of s in a Makefile and included one(s). Human subjects marked the occurrences that are variables' names and require consistent renaming as belonging to the same locsets.

In Table II, columns Locsets and Loc. display the number of locsets and the total number of locations in locsets. A locset can have multiple variables. Columns Frag and NonFrag show the numbers of locsets with fragmented and non-fragmented variables. Complexity of Makefiles is shown via the numbers of program elements (last 4 columns). A detected locset LS_1 is considered as *correct* if there exists a locset LS_2 in the oracle such that all code locations in LS_1 match all locations in LS_2. If there is at least one matched location, but not all locations are matched, LS_1 is considered as *incorrectly* detected. If there does not exist any LS_2 in the oracle that has at least one matched location, LS_1 is viewed as *missing*.

In Table III, *Precision* (column Prec) is the ratio of the correctly detected locsets over the total detected ones. *Recall* (column Rec) is the ratio of the correctly detected ones over the total number of locsets. As comparing LS_1 with LS_2, we also evaluated accuracy at the detected locations. A detected location L in LS_1 is viewed as correct or incorrect if it is or is not in LS_2, respectively. To compare with the baseline

Table III
LOCSET DETECTION ACCURACY RESULT

System	Locsets			Locations		$\lvert\overline{SDG}\rvert$	$T(s)$	MB
	Prec	Rec	Total	Corr	Inc-Text			
SCST	100%	100%	870	2230	724	19184	12	51
LINN	100%	100%	3417	7169	323	19539	27	133
GCC	100%	100%	1972	16546	2188	10909	19	156
MIN	100%	100%	632	3324	611	9666	13	259
LINS	100%	100%	973	1563	167	10623	12	248
FIRE	100%	100%	1960	4668	893	55043	34	691
TS	100%	100%	2655	9711	1659	27302	22	552

Table IV
CONTROLLED EXPERIMENT'S RESULT

Tasks	Code Smell Detection						Refactoring		
	Cor	Incor	Miss	Prec-Loc	Rec-Loc	Time	Prec	Rec	Time
1	11-26	5-0	13-0	92-96%	67-100%	70-31	82-99%	77-99%	37-23
2	8-24	4-0	16-0	77-100%	65-92%	69-31	71-92%	68-91%	42-29
3	11-24	6-0	13-0	100-100%	64-99%	47-39	58-100%	39-100%	35-22
4	13-23	3-0	11-1	100-100%	96-100%	74-43	76-100%	71-97%	36-25
5	6 -24	6-0	18-0	81-100%	80-100%	72-46	77-100%	44-100%	33-23
6	16-25	3-0	8-0	98-94%	90-90%	76-47	75-92%	67-90%	42-23

method in text replacing, we counted the number of incorrect locations from the text-search tool (column Inc-Text).

The result shows that all detected locsets are correct and cover all the locsets in the oracle. SYMake also handles correctly all fragmented variables. In contrast, the text-search tool returned a large number of incorrect locations.

$\lvert\overline{SDG}\rvert$ shows the average number of nodes in an SDG with V-models for a Makefile. SYMake handles all SDGs for all Makefiles with efficient time and memory usage.

B. Usefulness in Smell Detection and Refactoring

1) Experiment Setting: We conducted a controlled experiment to evaluate how SYMake can help users in understanding, refactoring, and detecting the code smells in Makefiles. Six programming tasks on Makefiles were prepared. The Makefiles were selected from the systems in Table II. Each Makefile was then injected with 6 code smells of the types listed in Section VI. We invited 8 Ph.D. students with 4-8 years of programming experience and divided them into 2 groups. We also provided a short training session for all subjects. To further avoid the imbalance between two groups, we applied the crossover technique. That is, group 1 performed tasks 1, 3, and 5 using SYMake and other tasks without SYMake. The opposite is for group 2. The first set of tasks include 1) detecting at least 6 code smells, 2) reporting the locations/tokens for those code smells, and 3) a short explanation. The second set of tasks is build code refactoring including target creating/renaming, rule extraction/removal, variable renaming, and prerequisite extraction.

2) Evaluation Metrics: First metric is code quality. In the smell detection tasks, we compared the numbers of smells correctly/incorrectly detected and missed between without and with the tool (two numbers are side-by-side in Table IV). Precision and recall for the location detection are in Prec-Loc and Rec-Loc. In the refactoring tasks, we calculated the precision (Prec) and recall (Rec) of refactoring locations. Second metric is developers' effort measured via completion time. If the time limit had passed, (s)he was required to stop.

3) Result: As in Table IV, in both types of tasks, subjects with SYMake were able to detect smells more accurately and perform refactoring more correctly in less time.

C. Threats to validity: In the first experiment, human subjects were involved in oracle building and errors could occur. In the controlled experiment, injected code smells are not the real ones and might not be representative. The subject are students who are not professional programmers.

IX. RELATED WORK

MAKAO [18] is a visualization and smell detection tool for Makefiles. However, it works only on concrete dependency graphs [9], thus, cannot handle dynamic Makefiles. There are several empirical studies on build code maintenance. McIntosh *et al.* [5] showed that build changes induce more relative churn on build code than source code changes induce on itself. Hochstein and Jiao [4] found that 19-58% of the total commits are the ones involving only build code. Robles *et al.* [7] reported that many revisions contain only changes to build code. Adams *et al.* [3] showed that the complexity of build code in Linux co-evolves with source code. McIntosh *et al.* [8] reported the same result in ANT.

Recent advances in symbolic execution have been widely applied. Popular methods are compositional symbolic execution [10], pre-/post-conditions [11], a symbolic and concrete execution combination [12], a combination with model checking [13], [14], and differential symbolic execution [15]. PhpSync's symbolic execution [16] works on PHP to capture HTML/JS code, while SYMake aims to represent build rules.

No refactoring support has been yet available for make build code. However, research in refactoring has been extensive [17], [19], [2]. Recent advances include [21], [22], [23], [24], [25]. Several approaches aim to detect code smells [26]. Popular ones include clone analysis [27], meta-model [28], logic meta programming [29], [30], weak code structure detection [31], and visualization [32], [33].

X. CONCLUSIONS

We introduce SYMake, an infrastructure for make code analysis. SYMake includes AST building module, a symbolic evaluation algorithm, and an evaluation trace building algorithm. We used SYMake to develop a tool to detect code smells and to support refactoring in Makefiles. Our evaluation on real-world Makefiles showed that our renaming tool is accurate and efficient, and that with SYMake, users could detect code smells and refactor Makefiles more accurately.

ACKNOWLEDGMENT

This project is funded by US National Science Foundation (NSF) CCF-1018600 grant. It was also funded in part by Vietnam Education Foundation for the third author.

References

[1] "Software Building," en.wikipedia.org/wiki/Software_build.

[2] W. G. Griswold and D. Notkin, "Automated assistance for program restructuring," ACM TOSEM, vol. 2, issue 3, 1993.

[3] B. Adams, K. De Schutter, H. Tromp, W. De Meuter, "The evolution of the Linux build system," ECEASST, vol. 8, 2007.

[4] L. Hochstein and Y. Jiao, "The cost of the Build Tax in Scientific Software," in ESEM '11, pp. 384-387. IEEE, 2011.

[5] S. McIntosh, B. Adams, T. H.D. Nguyen, Y. Kamei, and A. E. Hassan, "An empirical study of build maintenance effort," in ICSE '11, pp. 141–150. ACM, 2011.

[6] "Make," http://www.gnu.org/software/make/manual/make.html.

[7] G. Robles, J. M. Gonzalez-Barahona, and J. J. Merelo, "Beyond source code: the importance of other artifacts in software development (a case study)," *J. Syst. Softw.*, vol. 79, issue 9, pp. 1233–1248, September 2006.

[8] S. McIntosh, B. Adams, and A. E. Hassan, "The evolution of ANT build systems," in *MSR '10*, pp. 42–51, IEEE, 2010.

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

[10] S. Anand, P. Godefroid, and N. Tillmann, "Demand-driven compositional symbolic execution," in TACAS'08/ETAPS'08, pp. 367–381. Springer-Verlag, 2008.

[11] P. Godefroid, "Compositional dynamic test generation," in POPL '07, pp. 47–54. ACM, 2007.

[12] C. S. Păsăreanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape, "Combining unit-level symbolic execution and system-level concrete execution for testing NASA software," in ISSTA '08, pp. 15–26. ACM.

[13] W. Visser, C. S. Păsăreanu, and S. Khurshid, "Test input generation with Java PathFinder," in ISSTA '04. ACM, 2004.

[14] W. Visser, C. S. Păsăreanu, and R. Pelánek, "Test input generation for Java containers using state matching," in ISSTA'06, pp. 37–48. ACM, 2006.

[15] S. Person, M. B. Dwyer, S. Elbaum, and C. S. Păsăreanu, "Differential symbolic execution," in FSE'08. ACM, 2008.

[16] H. V. Nguyen, H. A. Nguyen, T. T. Nguyen, T. N. Nguyen, "Auto-Locating and Fix-Propagating for HTML Validation Errors to PHP Server-side Code," in ASE '11, IEEE, 2011.

[17] T. Mens and T. Tourwé, "A survey of software refactoring," IEEE TSE, vol. 30, no. 2, pp. 126–139, 2004.

[18] B. Adams, H. Tromp, K. De Schutter, and W. De Meuter, "Design recovery and maintenance of build systems," in ICSM '07, pp. 114–123, IEEE CS, 2007.

[19] W. F. Opdyke, "Refactoring object-oriented frameworks," Ph.D. dissertation, Urbana-Champaign, IL, USA, 1992.

[20] G. Kumfert and T. Epperly, "Software in the DOE: The hidden overhead of "the build"" Lawrence Livermore National Laboratory, Tech. Rep. UCRL-ID-147343, 2002.

[21] D. Dig, J. Marrero, M. Ernst, "Refactoring sequential Java code for concurrency via concurrent libraries," in ICSE'09.

[22] I. Balaban, F. Tip, and R. Fuhrer, "Refactoring support for class library migration," in OOPSLA '05. pp. 265–279, ACM.

[23] H. Kegel and F. Steimann, "Systematically refactoring inheritance to delegation in Java," in ICSE '08, pp. 431–440. ACM.

[24] A. Kiezun, M. D. Ernst, F. Tip, and R. M. Fuhrer, "Refactoring for parameterizing Java classes," in ICSE'07. IEEE CS.

[25] J. Liu, D. Batory, and C. Lengauer, "Feature oriented refactoring of legacy applications," in ICSE '06, pp. 112–121. ACM.

[26] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*. Adison-Wesley, 2000.

[27] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis, "Advanced clone-analysis to support object-oriented system refactoring," in WCRE'00. IEEE CS, 2000.

[28] S. Ducasse, M. Rieger, and S. Demeyer, "A language independent approach for detecting duplicated code" in ICSM'99.

[29] S. Tichelaar, S. Ducasse, S. Demeyer, and O. Nierstrasz, "A meta-model for language-independent refactoring," in IW-PSE'00, pp. 157–169. IEEE CS, 2000.

[30] T. Tourwé and T. Mens, "Identifying refactoring opportunities using logic meta programming," in CSMR'03. IEEE, 2003.

[31] T. Dudziak and J. Wloka, "Tool-supported discovery and refactoring of structural weaknesses in code," Diploma thesis, Technical University of Berlin, 2002.

[32] E. Van Emden and L. Moonen, "Java quality assurance by detecting code smells," in WCRE'02, pp. 97–106. IEEE CS.

[33] M. Lanza and S. Ducasse, "Understanding software evolution using a combination of software visualization and software metrics," in *Proc. Langages et Modeles a Objets*, LMO '02, pp. 135–149. Hermes Publications, 2002.

[34] "SCST Project," github.com/bvanassche/scst-out-of-tree.

[35] "Linux2.6-net," github.com/mirrors/linux-2.6/tree/master/net.

[36] "Gcc compiler," github.com/mirrors/gcc.

[37] "Minix," http://www.minix3.org/download/

[38] "Linux2.6-sound," github.com/mirrors/linux-2.6/tree/master/sound.

[39] "Firefox development," http://hg.mozilla.org/mozilla-central/.

[40] "Current Thunderbird, SeaMonkey and calendar development," http://hg.mozilla.org/comm-central/.