Extracting Configuration Knowledge from Build Files with Symbolic Analysis

Shurui Zhou,* Jafar Al-Kofahi,† Tien N. Nguyen,† Christian Kästner,* Sarah Nadi‡
*School of Computer Science, Carnegie Mellon University
†Electrical and Computer Engineering Department, Iowa State University
‡Technische Universität Darmstadt

***Abstract*—Build systems contain a lot of configuration knowledge about a software system, such as under which conditions specific files are compiled. Extracting such configuration knowledge is important for many tools analyzing highly-configurable systems, but very challenging due to the complex nature of build systems. We design an approach, based on SYMake, that symbolically evaluates Makefiles and extracts configuration knowledge in terms of file presence conditions and conditional parameters. We implement an initial prototype and demonstrate feasibility on small examples.**

I. Introduction

Build systems are an essential part of software systems. Among others, they control variability and manage configurations, deciding which files and features to include in the compiled product. With the increasing number of configuration options and increasing complexity of build systems, build scripts become increasingly complex [2, 10, 11]. For example, in the Linux kernel, the build system (consisting of Makefiles) decides which files are compiled and with which parameters, based on over 10,000 configuration options. It is hard to understand, analyze, and maintain such a system without understanding its Makefiles. Unfortunately, there are few tools that support analyzing, refactoring, and maintaining makefiles. In this paper, we provide tool support with a specific focus on extracting such variability information.

Our goal is to identify the presence condition of each file or code fragment that is controlled by the build system, as well as any additional (potentially optional) parameters needed for its compilation. A presence condition describes under which configurations a block of code or an entire file is compiled, expressed as a boolean formula over configuration options [5, 12]. That is, a file is included in the configuration process if and only if the presence condition evaluates to true for a given configuration. Configuration decisions are pervasive in build scripts, as exemplified in our running example in Figure 1: the value of the *CCFLAGS* parameter depends on the current operating system (Lines 5-9); also, whether the target *distrib* has a prerequisite of *createDBDir* (Lines 15-17) depends on whether *DB* is defined or not. Hence, which files are compiled and how depends on configuration options.

Extracting configuration knowledge from build systems is important for many forms of variability analysis for highly-configurable systems. For example, tools as TypeChef [9] and Undertaker [15] analyze the entire configuration space of a program to identify type errors or dead code. They extract presence conditions from conditional compilation directives in the code (*#ifdef*) and reason about properties in the code, such as whether function calls always match a function declaration in all configurations. For example, in Figure 4, function *foo* is called in all configurations with option *B* selected, but only defined in configurations with option *A* deselected (or executed with parameter '-UA'), leading to a compile-time error in all configurations with $A \wedge B$. However, analyses that do not consider variability knowledge from the build system can be misleading, causing both false positives and false negatives: For example, if we knew that the file in Figure 4 is only compiled if option *A* is deselected, we would not issue an error. While extracting presence conditions from #ifdef directives is relatively straightforward, extracting variability knowledge from the build system is difficult—it is one of the main challenges of using TypeChef and similar tools for variability analyses.

Determining configuration knowledge from a build system is challenging, because most build systems are written in sophisticated Turing-complete scripting languages. Extracting presence conditions exactly is often undecidable, because many build systems may perform arbitrary computations by calling shell scripts, such as the $(eval), $(shell), or $(wildcard) functions. Current approaches use sampling

```
1  OS=$(shell uname)
2  POI4R:=$(shell pwd)
3  DISTRIB=Poi-0.1.0
4
5  ifeq ($(OS),Linux)
6  CCFLAGS=-O0
7  else
8  CCFLAGS=-O2
9  endif
10
11 distrib:
12   install Poi4R.rb $(DISTRIB)/python
13   g++ -o $@ $(CCFLAGS) -I$(POI4R) poi.o
14
15 ifdef DB
16 distrib: createDBDir
17 endif
18
19 createDBDir:
20   mkdir -p Poi/dbl
21 ifneq($(OS),Linux)
22   install libgcj.5.dylib $(DISTRIB)/gcj
23 endif
```

Fig. 1. Modified excerpt of the Apache POI project

```
Configuration: [$(OS)=Linux]
  install Poi4R.rb Poi-0.1.0/python
  g++ -o distrib -O0 -I/Users/repos/makefiles poi.o

Configuration: [$(OS)!=Linux]
  install Poi4R.rb Poi-0.1.0/python
  g++ -o distrib -O2 -I/Users/repos/makefiles poi.o

Configuration: [$(OS)=Linux, DB]
  mkdir -p Poi/dbl
  install Poi4R.rb Poi-0.1.0/python
  g++ -o distrib -O0 -I/Users/repos/makefiles poi.o

Configuration: [$(OS)!=Linux, DB]
  mkdir -p Poi/dbl
  install libgcj.5.dylib Poi-0.1.0/gcj
  install Poi4R.rb Poi-0.1.0/python
  g++ -o distrib -O2 -I/Users/repos/makefiles poi.o
```

Fig. 2. Actual executed scripts in different configurations for the Makefile in Figure 1.

```
install Poi4R.rb Poi-0.1.0/python ---> [TRUE]
g++ -o $@ -O0 -I$(POI4R) poi.o ---> [$(OS) = Linux]
g++ -c $@ -O2 -I$(POI4R) poi.o ---> [$(OS) != Linux]
mkdir -p Poi/dbl ---> [DB]
install libgcj.5.dylib Poi-0.1.0/gcj ---> [($(OS) != Linux) ∧ DB]
```

Fig. 3. Conditional build script with presence conditions

strategies or inaccurate heuristics only [4, 6, 13]. Theoretically, we could execute the build system in every configuration to see which files get compiled and how (see scripts in Figure 2), but such brute-force strategy does not scale and sampling would yield imprecise results (e.g., the Linux Kernel's 10,000 options can be combined into more configurations than there are atoms in the universe).

To extract configuration knowledge, we symbolically execute the build script. Symbolic execution allows analyzing all configurations at once, abstracting over unknown values. We record options and environment interactions with symbolic values and approximate all possible executions with corresponding path conditions. From the symbolic execution result, we extract a conditional dependency graph that describes the dependencies among targets with presence conditions. From the conditional dependency graph, we finally derive a conditional script that describes all possible executions with presence conditions for every recipe entry, as illustrated in Figure 3, and the file presence conditions.

In summary, we contribute: (1) an approach to analyze Makefiles to extract presence condition for files and code blocks; (2) an extension of the symbolic execute tool SYMake [14]; (3) a demonstration of feasibility on small examples with a prototype; (4) a test strategy to ensure correctness.

II. Related Work

Analyzing build files has been recognized as increasingly important. Adams et al. [1, 2, 11] have shown how build systems continue to grow in size and complexity, emphasizing the importance of analysis and tool support. Researchers have investigated build system analysis from different perspectives and some even have started extracting presence condition.

```
#ifndef A
void foo() { }
#endif
#ifdef B
void bar() { foo(); }
#endif
```

Fig. 4. Code example with a type error in configurations with $A \wedge B$ detectable with variability analysis tools as TypeChef.

Most analysis approaches are dynamic and actually execute the build to extract information. For example, van der Burg et al. [16] dynamically detect which files are included in a build to check license compatibility, Metamorphosis [8] dynamically analyzes build system to migrate them, and MkFault [3] combines runtime information with some structural analysis to localize build faults. However, such dynamic approaches can only analyze one configuration at a time.

Alternatively, some researchers have investigated static analyses for build files. We build on SYMake [14], which uses symbolic execution to conservatively analyze all possible executions of a Makefile. It produces a symbolic dependency graph, which represents all possible build rules and dependencies among targets and prerequisites, as well as recipe commands. It was originally designed to detect several types of errors in Makefiles and help building refactoring tools.

We are specifically interested in extracting variability information in terms of file presence conditions and conditional parameters, a challenge which has been addressed by three research groups so far. A simple dynamic analysis of executing all possible configurations would yield accurate variability information, but obviously does not scale. Instead, Dietrich et al. [6] sample a subset of configurations, trying to activate each configuration option once. Their approach is simple due to its sampling nature, but incomplete; it cannot recover complex conditions with several disjunctions and negations. Using a different strategy, both Berger et al. [4] and Nadi and Holt [13] have tried to statically approximate file presence conditions by parsing specific patterns in build scripts. Their approaches are designed for common patterns used in Linux's Kbuild infrastructure and achieves relatively high precision for the Linux kernel, but are unable to cope with build files (or parts thereof) that do not follow these patterns. In our approach, we use symbolic execution which does not rely on sampling or specific patterns.

III. Extracting Variability Knowledge with Symbolic Execution

The goal of our approach is to extract configuration knowledge from Makefiles. Our idea is to get symbolic scripts of all executions and extract presence conditions and parameters from them. We proceed in two steps. First, we use SYMake to symbolically evaluate the variables and collect rules in Makefiles. Then we extract the presence conditions of files and code blocks from the output of SYMake.

A. Step 1: Symbolically evaluate variables and collect rules

SYMake mirrors the first part of the build process in which targets and recipes are collected into a graph without executing the build steps' recipes. However, when faced with unknown values from options or environment interactions (e.g., from executing shell commands), instead of concrete values, we use symbolic values and explore all possible paths. Figure 5 shows the output graph of our approach that analyzes the Makefiles of Figure 1. For example, in Line 3, *DISTRIB* is assigned a concrete value 'Poi-0.1.0', whereas, in Lines 1-2, variables *OS* and *POI4R* are stored as symbolic values, because we cannot statically know the operating system and directory of the Makefile's environment.

When *if* statements are encountered in which the condition cannot be evaluated to a concrete value, SYMake executes both branches with corresponding path conditions. If we assign a variable with different values under different path conditions, we preserve all possible values and their corresponding conditions using *Select* nodes. For example, in Figure 5, *CCFLAG* is assigned with two alternative concrete values, depending on the value of *$(OS)*. When such a variable is used, for example, in recipes as '*g++ -o $@ $(CCFLAGS) -I$(POI4R) poi.o*', SYMake substitutes all possible values and their corresponding conditions.

While executing Makefiles, SYMake collects all the targets and their dependencies and recipes in a graph. Targets, dependencies, or recipes found under a path condition are added to the graph conditionally, storing the condition as an annotation on the edge. In our example, target *distrib* depends on *createDBDir* only if option DB is selected and *createDBDir*'s recipe depends on the current operating system. This leads to a conditional dependency graph as illustrated in Figure 5.[1]

B. Step 2: Extracting Conditional Build Script

After symbolic execution, we extract a conditional build script and presence conditions from the output graph of Step 1. First, we explode recipes that contain internal *Select* nodes until we have a number of recipes with conditions that no longer contain further *Select* nodes (but possibly still symbolic values). This corresponds to simple factoring operations in the choice calculus [7]. Mirroring *make*'s behavior, given a main target, we then traverse the graph in depth first order to get all directly and indirectly dependent targets of the build graph, as well as their recipes. During the traversal, we collect all conditions on edges and assign a presence condition to each node (a conjunction of all edge conditions). For example, in Figure 5, when processing the second recipe for *createDBDir* (rcp4), we know that it is executed only under condition $(\$(OS)!=Linux) \wedge DB$.

By listing all identified recipes and their corresponding conditions in order, we create a conditional build script as illustrated in Figure 3. The conditional build script is a single compact description of all possible build scripts that could be created with a brute-force strategy (cf. Figure 2), except that it may include symbolic values where SYMake cannot resolve environment interactions. It can be considered as a single file with #ifdef directives that a preprocessor could process to create a concrete build script for a specific configuration.

Finally, we extract presence conditions of files and parameters from the conditional scripts we obtained. We identify all build steps that use specific commands, as *gcc* or *g++*, and parse the instruction to identify files and additional parameters. If a file is compiled under different conditions, we track that file's presence condition as a disjunction of all individual conditions. We proceed similarly with parameters per file. For example, in Figure 1, the file *poi.o* is compiled if $(\$(OS) = Linux) \vee (\$(OS) != Linux) \equiv True$, but it is compiled with different parameters under different conditions. This is precisely the information needed for TypeChef and similar tools.

IV. PROTOTYPE/TESTING

We have built a prototype implementation for our approach that covers most language constructs of *make* and performs the described postprocessing to extract configuration knowledge. The implementation is not complete yet. For example, submake calls are not yet supported (i.e., tracking values when descending to nested make calls). Although this prevents us from running large-scale experiments on Linux or similar systems, we can already evaluate simpler makefiles.

To test our implementation, we have set up a differential testing framework, in which we compare the symbolic scripts extracted from SYMake (cf. Figure 3) with the scripts generated in a brute-force execution of all configurations (cf. Figure 2). Figure 6 outlines our testing approach: We first run symbolic execution on the input Makefiles to produce a conditional dependency graph and subsequently extract the conditional build script. To obtain the individual scripts to which to compare, we determine the different configurations the Makefile can execute under and execute make as *'make -n'* with *'-D'* or *'-U'* parameters to define or undefine values for each option. As shown in Figure 6, we expect that the conditional build script should be equivalent to all the individual scripts from the different configurations, except for symbolic values that are replaced by concrete values in the individual concrete executions. We automated this comparison by comparing the scripts' output and all values in the heap at the end of the execution.

While we expect our approach to scale for large build scripts with many configuration options, our testing approach relies on brute force and thus limits us to tests with few options. We use a selection of over 10 artificial Makefiles as well as parts of real Makefiles as tests, containing various configuration mechanisms, such as conditional assignments, conditional dependencies, conditional targets, and so forth. All tests have at most 8 options, yielding up tp 256 possible configurations, which are still quick to execute. All test cases pass in our current implementation, which gives us confidence in the correctness of our prototype. Our postprocessing and testing infrastructure is available online at https://github.com/shuiblue/MakefileScript.

[1] Technically, as discussed elsewhere [14], SYMake distinguishes between different internal models, shaded with different colors in the figure.

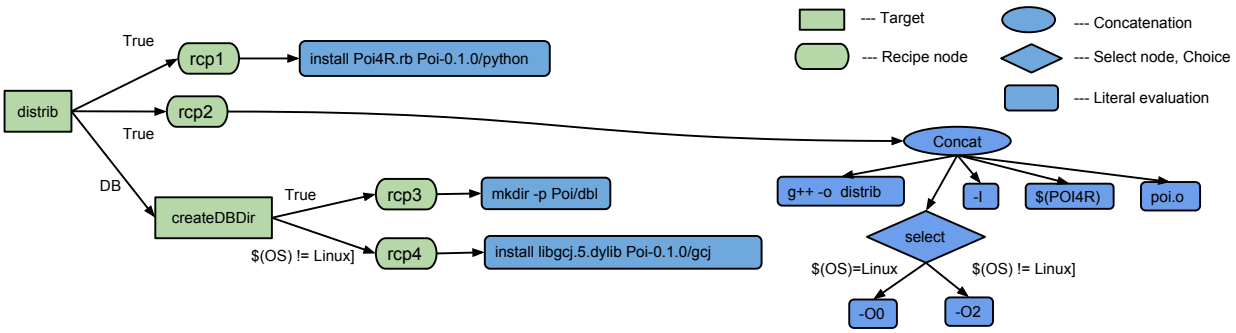

Fig. 5. Conditional dependency graph for the Makefile in Figure 1.

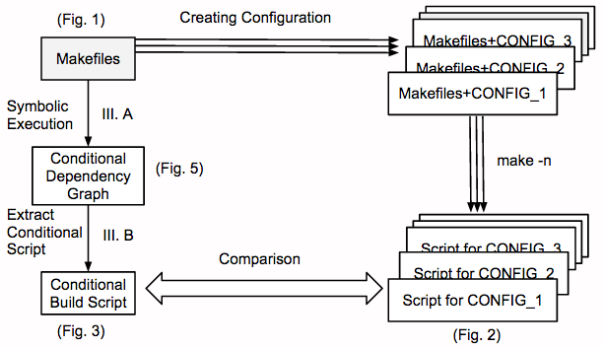

Fig. 6. Overview of our testing approach

V. Conclusion

We symbolically execute build scripts to extract configuration knowledge in the form of file presence conditions and conditional parameters. As intermediate steps, we build conditional call graphs and conditional build scripts. In contrast to prior work based on sampling, our approach can also detect more complex presence conditions. In contrast to prior parsing-based approaches, it does not rely on specific patterns in the build scripts, making it more generally applicable.

Our approach is conceptually limited though in that it replaces all environment interactions (e.g., escaping to the shell) by symbolic values. Such approximation is needed to deal with undecidability issues, but whether this is a significant obstacle when analyzing real-world systems is a question for future evaluations. In contrast to prior tools that would just miss certain conditions or provide imprecise constraints, we argue that symbolic values provide at least an exact indication of where our analysis is imprecise, which can be used to interpret the results in downstream tools. In the near future, we expect to extend our analysis to support real-world systems and integrate it with TypeChef.

Acknowledgment

Zhou and Kästner's work is supported by NSF award CCF-1318808, Al-Kofahi and Tien's work is supported by NSF award CCF-1320578 and Nadi's work is supported by the DFG, project E1 within CRC 1119 CROSSING.

References

[1] Bram Adams, Herman Tromp, Kris De Schutter, and Wolfgang De Meuter. MAKAO. In *Proc. Int'l Conf. Software Maintenance (ICSM)*, pages 517–518. IEEE Computer Society, 2007.

[2] Bram Adams, Kris De Schutter, Herman Tromp, and Wolfgang De Meuter. The evolution of the linux build system. *Electronic Communications of the ECEASST*, 8, 2008.

[3] Jafar Al-Kofahi, Hung Viet Nguyen, and Tien N. Nguyen. Fault localization for build code errors in makefiles. In *Companion Proc. Conf. on Software Engineering*, pages 600–601. ACM, 2014.

[4] Thorsten Berger, Steven She, Rafael Lotufo, Krzysztof Czarnecki, and Andrzej Wąsowski. Feature-to-code mapping in two large product lines. In *Proc. Int'l Software Product Line Conf. (SPLC)*, pages 498–499. Springer-Verlag, 2010.

[5] Krzysztof Czarnecki and MichałAntkiewicz. Mapping features to models: A template approach based on superimposed variants. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*, pages 422–437. Springer-Verlag, 2005.

[6] Christian Dietrich, Reinhard Tartler, Wolfgang Schröder-Preikschat, and Daniel Lohmann. A robust approach for variability extraction from the linux build system. In *Proc. Int'l Software Product Line Conf. (SPLC)*, pages 21–30. ACM, 2012.

[7] Martin Erwig and Eric Walkingshaw. The choice calculus: A representation for software variation. *ACM Trans. Softw. Eng. Methodol. (TOSEM)*, 21(1):6:1–6:27, 2011.

[8] Milos Gligoric, Wolfram Schulte, Chandra Prasad, Danny van Velzen, Iman Narasamdya, and Benjamin Livshits. Automated migration of build scripts using dynamic analysis and search-based refactoring. In *Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 599–616. ACM, 2014.

[9] Christian Kästner, Paolo G. Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger. Variability-aware parsing in the presence of lexical macros and conditional compilation. In *Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 805–824. ACM, 2011.

[10] Rafael Lotufo, Steven She, Thorsten Berger, Krzysztof Czarnecki, and Andrzej Wąsowski. Evolution of the linux kernel variability model. In *Proc. Int'l Software Product Line Conf. (SPLC)*, pages 136–150. Springer-Verlag, 2010.

[11] Shane McIntosh, Bram Adams, Thanh H.D. Nguyen, Yasutaka Kamei, and Ahmed E. Hassan. An empirical study of build maintenance effort. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 141–150. ACM, 2011.

[12] Sarah Nadi and Ric Holt. Mining kbuild to detect variability anomalies in linux. In *Proc. Europ. Conf. Software Maintenance and Reengineering (CSMR)*, pages 107–116. IEEE Computer Society, 2012.

[13] Sarah Nadi and Richard C. Holt. The linux kernel: a case study of build system variability. *Journal of Software: Evolution and Process*, 26(8):730–746, 2014.

[14] Ahmed Tamrawi, Hoan Anh Nguyen, Hung Viet Nguyen, and Tien N. Nguyen. Build code analysis with symbolic evaluation. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 650–660. IEEE Press, 2012.

[15] Reinhard Tartler, Daniel Lohmann, Julio Sincero, and Wolfgang Schröder-Preikschat. Feature consistency in compile-time-configurable system software: Facing the linux 10,000 feature problem. In *Proc. Europ. Conf. Computer Systems (EuroSys)*, pages 47–60. ACM, 2011.

[16] Sander van der Burg, Eelco Dolstra, Shane McIntosh, Julius Davies, Daniel M. German, and Armijn Hemel. Tracing software build processes to uncover license compliance inconsistencies. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, pages 731–742. ACM, 2014.