Four Languages and Lots of Macros

Analyzing Autotools Build Systems

Jafar M. Al-Kofahi
Electrical and Computer Engineering Department
Iowa State University
USA

Suresh Kothari
Electrical and Computer Engineering Department
Iowa State University
USA

Christian Kästner
School of Computer Science
Carnegie Mellon University
USA

Abstract

Build systems are crucial for software system development. However, there is a lack of tool support to help with their high maintenance overhead. GNU Autotools are widely used in the open-source community, but users face various challenges from its hard to comprehend nature and staging of multiple code -generation steps, often leading to low quality and error-prone build code. In this paper, we present a platform, AutoHaven, to provide a foundation for developers to create analysis tools to help them understand, maintain, and migrate their GNU Autotools build systems. Internally it uses approximate parsing and symbolic analysis of the build logic. We illustrate the use of the platform with two tools: ACSense helps developers to better understand their build systems and ACSniff detects build smells to improve build code quality. Our evaluation shows that AutoHaven can support most GNU Autotools build systems and can detect build smells in the wild.

CCS Concepts • **Software and its engineering → Software maintenance tools**;

Keywords build-system, GNU Autotool, Autoconf, build maintenance

ACM Reference Format:
Jafar M. Al-Kofahi, Suresh Kothari, and Christian Kästner. 2017. Four Languages and Lots of Macros: Analyzing Autotools Build Systems. In *Proceedings of 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE'17).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3136040.3136051

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
GPCE'17, October 23–24, 2017, Vancouver, Canada
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-5524-7/17/10…$15.00
https://doi.org/10.1145/3136040.3136051

1 Introduction

Build systems are a crucial part of software development. Prior work has shown that build systems are complex [18, 28, 39], come with high maintenance overhead [29, 30], and are defect prone as they keep evolving with the software system [34]. Those studies emphasize the importance of build systems and call for better *tool support* to help developers understand, maintain, and migrate their build systems.

For the developers to properly maintain or migrate their build systems, they first need to have a solid understanding of their build system mechanics and how it builds the software. A lack of understanding would lead to increased maintenance burden [22, 39], making build systems more defect prone, and would make migration more challenging [40]. Comprehension is a challenge as build tools tend to use domain specific, powerful, and complex languages.

Generator-based build tools can be particularly challenging to understand as they have multiple stages involved in the build process, where each stage generates build artifacts to be executed in a later stage during the build. For example, in GNU Autotools the M4 preprocessor creates a shell script that subsequently generates a Makefile from a template, which is eventually executed by *make*. Such staged build process adds to the system's complexity. Such systems are not easy to analyze, neither for humans nor machines.

In this work, we target GNU Autotools [7] as an instance of particularly difficult generator-based build tools that requires an analysis of multiple languages and their interactions. In contrast to prior tools for maintaining and understanding build systems that focused mostly on specific tasks [19–21, 31, 41], we aim to build a generic analysis infrastructure that can be used for various maintenance tasks. We demonstrate our infrastructure by providing two tools for *build comprehension* and *build-smell detection*.

Both build comprehension and build-smell detection address real issues for users of Autotools. We noticed that Autotools users frequently complain about maintaining and making changes to their Autoconf scripts. Developers face challenges in understanding their build system, specifically their configuration script and how it fits into the big picture. A trial-and-error approach to build system maintenance leads to overly complicated and hard to maintain configuration scripts. In line with prior work [39], we identified

that developers neglect to properly maintain their build system, causing various issues within their build system adding to their maintenance overhead, such as inconsistencies between user documentation and how is the build system implemented. We also identified the existence of *build smells*, which are issues related to the quality of the build system that adds to its technical debt and complexity but does not break the build, such as the existence of unused variables, dependencies. These build smells exists within specific parts of the build script but can also span multiple languages, including configuration scripts, build logic, and the target code.

We approach the analysis of generator-based build systems written with Autotools by attempting to parse and recognize common patterns in un-preprocessed build code (i.e., without running the generators). We subsequently symbolically execute the configuration logic to identify options, their interactions, and their build actions on the remainder of the build process. A build action is any action executed (i.e., check for dependency, generate artifacts) to accomplish the build process. For build-smell detection, we further extract the targets of build actions in templates for the build logic and in the target C code itself and define analyses for build smells that compare the various extracted information. We package our analysis infrastructure providing a structural representation of the build system created by parsing and symbolic analysis as a tool platform called AutoHaven. We further explain how the extracted information can be used to support developers in build comprehension tasks and build a tool for build-smell detection. We evaluate infrastructure on ten real-world build system written with GNU Autotools and identify that our approximation is practical and useful for build analysis. Our evaluation shows that we can correctly capture the configuration actions and detect build smells in real build systems.

Overall, we contribute:

- An analysis infrastructure for generator-based build systems written in GNU Autotools that extracts information from configuration scripts with approximate parsing and symbolic execution of the configuration logic.
- A comprehension tool support for developers to better understand their GNU Autotools build systems.
- A definition of build smells, and a detection tool that identifies build smells, including issues that span multiple languages.
- An evaluation of our infrastructure on ten real-world build systems, demonstrating accuracy despite approximations and the ability to identify real build-smells in the wild.

This paper extends a prior workshop paper on the AutoHaven infrastructure [22]. In that workshop paper, we argued for the need of such an infrastructure and sketched a possible solution. We also surveyed criticism of Autotools users in practice and presented the first version of our parsing approach for configuration logic. In this paper, we extend our prior work by adding a symbolic analysis of the configuration logic, by building two tools (comprehension and build-smell detection) on top of our infrastructure, and by evaluating our approach with ten real-world build systems.

2 Background and Motivation

This section provides background on GNU Autotools to motivate the need for the proposed AutoHaven platform.

2.1 GNU Autotools

To understand our solution, we first need to introduce how GNU Autotools and their build systems work. Figure 1 shows the structure of such build systems. For a given system, the developers provide *Makefile.am* Automake files, to describe how to build the system at a high level; they also provide a *configure.ac* Autoconf file to describes the system's external dependencies and the build-system configurations. These declared configurations can alter the build process to include or exclude system features, at the file level within the build script, or at a more granular level in the source code.

GNU Automake processes the *Makefile.am* files to identify the source files to be built, and then it generates *Makefiles.in* templates that hold the build logic for the system. Special placeholders annotate these templates, which are later used to adjust the build logic according to the selected build configurations; these placeholders are called substitution variables. They are used by the configuration script to pass configuration related settings (i.e. which file to include) and environment configurations (i.e., which compiler to use, compiler flags).

The Autoconf *configure.ac* script is written using GNU M4 macros [12], shell scripting, and can also have snippets of C, C++, or Perl programming languages to check for certain libraries or features in the environment. The M4 language is macro based, where each macro expands to a predefined snippet of text; in the Autoconf case, these M4 macros are expanded to snippets of shell scripts. GNU Autoconf processes the *configure.ac* file, expand the M4 macros and generate the *configure* shell script file. The *configure* script consumes the *Makefile.in* templates and generates concrete Makefiles. For example, a template would have the following line: *CC = @CC@*. This variable holds the C compiler command for the build process. When the *configure* script is executed, it identifies the default C compiler, then substitute the *@CC@* placeholder in the templates with the C compiler command (e.g., *CC = gcc*).

In GNU-Autotools-based build systems, developers can control what to compile from the source code at file level with the Makefiles, or on a more granular level using conditional compilation. In conditional compilation, a block of code would have an associated condition, and if the condition

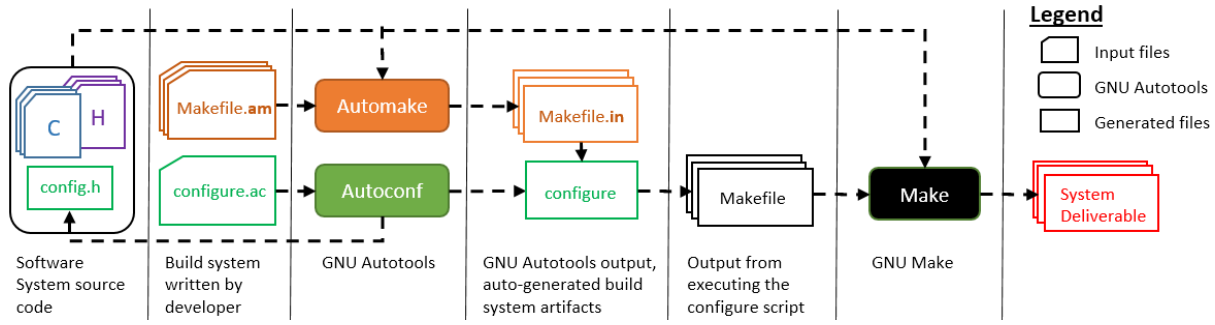

Figure 1. Autotools workflow

is met then the block of code is included in the source code compilation; otherwise, it gets excluded. For conditional compilation, developers use C-Preprocessor macros to surround blocks of code with CPP control constructs (i.e., #ifdef's) and declare the condition using CPP macros.

The configuration script uses the build script's substitution variables and conditional compilation to alter the build according to the selected build configurations. The script will decide the values for the substitution variables and decide which CPP macros to declare.

Once the configuration script has executed and it generates the concrete Makefiles, the user runs GNU Make to build the system and produce the deliverables.

2.2 AutoHell, a Closer Look

"Some developers, not only in KDE, like to nickname the autotools as 'auto-hell' because of its difficult to comprehend architecture" [36]. GNU Autotools are widely used in the open source community but often criticized. As systems evolve and become more sophisticated, many have migrated away from GNU Autotools to other build tools, such as CMake [3]. We investigated some recent migrations and looked into their code repository commits, email archives, and developer blogs to identify the reasons for migrating [22].

For example, in version 4, the KDE [11] team decided to migrate away from Autotools. They had two attempts in this migration [40], and the second attempt succeeded in migrating to CMake [36, 40, 42]. Another example is Map Server [13] that migrated away from Autotools to CMake [14]. Often the following challenges were mentioned as reasons:

- **Steep learning curve:** Understanding the different tools that come into play and their role in the workflow, is not straightforward. Also one needs to be familiar with multiple languages such as M4, shell scripting, make, and any other language used in *configure.ac*.
- **Staged build process:** The workflow in Figure 1 involves multiple dependent stages. When debugging the build system, the developer needs to generate the configure script and makefile templates; then run them. Any issues identified would need to be fixed on the input *.ac* and *.am* files, which has an associated performance and maintenance costs.
- **Large Autoconf files:** Maintaining *configure.ac* files can be intimidating due to their complexity and large size. Checking some popular open source systems [1], their *configure.ac* files averaged at 3200 SLOC.
- **Lack of tool support:** Developers have limited visibility into the Autotools-based build systems and how they work, and they tend to rely on domain expert for support.

2.3 Build Smells in Autoconf Configuration Script

In line with prior research [39], our prior study of open-source build systems [22] revealed multiple examples of negligence by developers to keep their Autoconf up to date. We present here one such example of developer negligence that resulted in an unused build code. In revision 189 of the D2X-XL [4] system, a developer made a source code change with a message "*Removed a ton of unused code*", that took out all usage of the *OGL_ZBUF* CPP macro from the source code. But they neglected to modify the Autoconf script, to remove the declaration of *OGL_ZBUF* CPP macro. The Autoconf script still declares the *OGL_ZBUF* macro after nine years in revision 14729.

It is not clear why this unused macro still exists after all of this time. But this shows that over time the quality of the Autoconf script degrades. And it becomes overly complicated and error-prone.

3 Approach

In this work, we extract information from GNU Autotools build systems and used this information to build tools for understanding the configuration scripts and detecting build smells originating from the configuration script. We proceed in four steps: (1) We parse the configuration and build scripts, using unsound approximations to avoid having to execute the involved generators. (2) We symbolically execute the

[1] OpenVPN, OpenSSH, Emacs, GCC, and MapServer

parsed configuration scripts to identify how configuration options depend on each other and how they affect actions in the build. (3) We extract information from the two previous steps and present them to users for build comprehension tasks. (4) We define common issues and inconsistencies as *build smells* and build a detector to automatically identify them, also considering information from other parts of the system's build and source code.

3.1 Parsing Build Scripts

To reason about builds, we need to understand both the build logic and the configuration logic. To parse the build logic in make templates (`Makefile.am`, see Fig. 1), we extend *SyMake* [20, 41, 44] to support AutoMake constructs—a relatively straightforward extension to identify the targets of substitutions by the configuration script. The key challenge though is in parsing the configuration logic (`configure.ac`, see Fig. 1), as it involves multiple languages and generation steps.

While it is not generally possible to parse the configuration logic without executing at least the M4 preprocessor, we suspected that in practice developers follow a more restricted set of development practices that do not arbitrarily intermix M4 and shell instructions. We also expect that most developers follow a few frequently used rather than developing their own custom macros. Before developing a parser, we confirmed this belief with a preliminary study.

Preliminary study on configuration scripts. In a preliminary study, we aimed to answer a set of research questions to inform the feasibility of parsing configuration scripts: *How do developers write their configuration scripts? Are there any common characteristics in how they write them? Can these characteristics be exploited to address the needed cross-language analysis?*

To answer these questions, we carefully studied the Autoconf mechanisms in the Autoconf manual [6] and manually studied the configuration scripts of four open-source systems: Emacs [9], OpenVPN [16], OpenSSH [15], and GCC [8].

From this study we identified the following: First, Autoconf comes with a vast library of M4 macros (e.g.. declare a configuration, check for C header file, execute a snippet of C code, and much more). However, in our study, we noticed that developers tend to use only a small common subset of them. As a consequence, it is likely that we can handle a large number of configuration scripts by understanding the mechanisms behind only a small number of M4 macros.

Second, when writing the configuration scripts, developers indeed tend to follow common patterns to achieve commonly needed functionality (i.e., declare configuration and how it modifies the build script, check for dependency, and others). They tend to use a small set of M4 macros in common and repeated patterns in shell scripts, mostly simple

Listing 1. Snippet from Configure.ac

```
1  #Declare long-message feature
2  AC_ARG_ENABLE(localization,
3   [--enable-localization=ar/en.
4     ar for Arabic, and en for english.],
5   LANG=$enableval)
6
7  #ensure proper macros are defined
8  #in source header
9  if test "$LANG" = "ar"; then
10      AC_SUBST(LANG)
11 fi
```

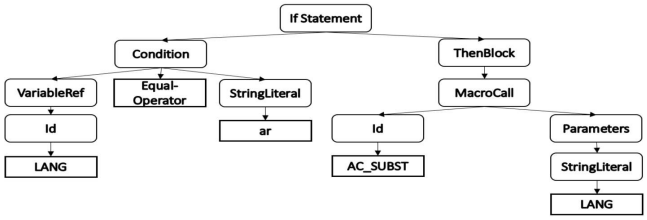

Figure 2. Example Autoconf AST

assignments and substitutions rather than arbitrary computations. This is encouraging and suggests that configuration scripts tend to be fairly regular such that we can recognize and analyze a small number of common patterns.

Third, the script is long due to the repetition of these patterns (i.e., same pattern to check for dependency is used to check for the various dependencies needed). This again suggests that parsing might be possible since the large size of many configuration scripts does not come from complicated interactions but merely from repetitions. This is also encouraging for the subsequent symbolic analysis.

Fourth, developers tend to use simple control-flow constructs, typically simple and not particularly deeply nested branching statements. Loops are uncommon; the few loops are mostly used as for-each loops to iterate over a collection. This is encouraging for the symbolic analysis.

In Listing 1, we show an excerpt from a configuration script exemplifying the common patterns used: The M4 macro, *AC_ARG_ENABLE*, is commonly used to declare configurations. It assigns the user provided value for configuration localization to the variable *LANG*. Subsequently, this variable is used to decide whether a particular action should be taken in the build or not, in this case, it controls an *AC_SUBST* to declare a substitution variable for the `Makefile.in` templates. These kinds of simple actions with a small number of macros and simple control-flow decisions are representative of most of the configuration logic.

Approximate parsing. We exploit the insights from our preliminary study to build a parser that works well for configuration scripts that follow the commonly used patterns we have identified. Specifically, we extend a grammar for shell scripts with the most commonly occurring M4 macros.

For those macros, we can also decide how to parse the corresponding macro parameters, for example, to parse some as text and others as shell code. In our resulting abstract syntax tree, we preserve the macros rather than replacing them by shell code that M4 would generate because we are interested in the sources (options) and targets (actions) of the configuration logic. This preservation would simplify our parsing compared to attempting to parse and understand what the shell script is doing. In Figure 2, we illustrate the resulting abstract syntax tree for our prior configuration snippet from Listing 1. Although shell scripts can be large, the use of repeated patterns helps us in parsing them efficiently.

Parsing is technically unsound, and we may stumble over unsupported or user-defined M4 macros, but those seem to be relatively rare. If we encounter unknown structures, we attempt to parse them as unknown macros (upper case names followed by arguments in parentheses) and represent them as unknown entities in our abstract syntax tree. As part of our evaluation, we will investigate to what degree we can successfully parse real-world configuration scripts with our approximate technique.

The abstract syntax tree is the central representation for all subsequent analysis.

3.2 Symbolic Analysis of Configuration Logic

For many analysis and comprehension tasks, it is important to understand the build and configuration logic, for example, to identify which configuration options affect which actions in the build. To that end, we symbolically analyze the configuration logic parsed in the previous step (a similar analysis of the build logic within the Makefile templates can be reused from SyMake [41]).

In a nutshell, we create a symbolic execution engine that executes the statements in the abstract syntax tree for the configuration logic. Symbolic execution is fairly straightforward: We introduce symbolic values for unknown values (e.g., for inline shell commands) and configuration options (recognized by specific M4 macros). We implement symbolic versions of many standard shell instructions, such as assignments and string substitutions; we evaluate expressions in control-flow decisions (concretely if possible, symbolically otherwise) track path conditions and explore all feasible paths. For increased accuracy, borrowing from variational execution [23, 35, 37] and similar to MultiSE [38], we aggressively track alternative concrete values in choices rather than merging them into fresh symbolic values. Our symbolic execution supports many, but not all M4 macros and shell statements; it executes loops at most once and does not support recursion (both of which are fortunately not common, as discussed above).

In our configuration script excerpt in Listing 1, the variable LANG would be represented symbolically after the declaration of the option as

```
Choice(Feature(localization),
       Symbolic(userInput), "")
                \vspace{1em}
```

indicating that it may have a symbolic user value when *localization* is selected and is empty otherwise. Subsequently, we can determine that the action AC_SUBST can only be reached if *localization* is enabled and the user provides the value "ar".

Results from symbolic execution, such as information about which build actions are reachable under which path conditions, or where build options are used for decisions, can be used subsequently by other tools.

3.3 Comprehension Support for Developers

As discussed earlier (Sec. 2.2), developers face challenges understanding how the configuration scripts fit into the build system. This is due to its lengthy and multi-language nature, with impact across domains to alter the build script and/or source code.

To provide comprehension support for developers, we provide support for three common comprehension tasks: **(T1)** to understand which configuration options the system manages and how they interact. **(T2)** To understand which options affect the build process and when (e.g., substitution variables for build scripts and CPP macros for the source code). And **(T3)** to understand which dependencies (e.g., external libraries) are needed for the system to be built, and under what configurations are they needed.

We extract the information for all three questions from the results of our prior analysis steps. For listing options (T1), we simply identify all relevant M4 macros in our abstract syntax tree; for identifying interactions, we investigate whether two options ever co-occur in values or path conditions during symbolic execution. To identify the effect of options (T2), in the symbolic trace, we observe which option affect build actions, either in path conditions of the build actions or as parameters to those actions. To track external dependencies (T3), we collect M4 macros that test for libraries or other dependencies in the configuration scripts and collect dependencies from the build logic in terms of optional build targets in the Makefile templates.

As output, we produce a summary of the configuration script that lists the options and corresponding build actions with their path conditions and symbolic parameters. For example, in Listing 1, we describe that the substitution action only occurs when *localization* is selected, and the parameter *"ar"* is provided:

```
MAKEFILE_SUBST(name = LANG, value = "ar",
       if = Feature(localization) AND (Symbolic(
            userInput) = "ar"))
```

3.4 Detecting Build Smells

In addition to supporting comprehension tasks, we also provide a tool to detect *build smells*. Build smells can exist within

a single configuration script or they can span multiple languages and artifacts including the configuration script, the build script, and the source code. Due to such complexity, it is difficult to detect the build smells without tool support.

We have built a build-smell detector on top of our analysis infrastructure. In addition to the structure and symbolic traces of the configuration and build logic, we also collect information about used preprocessor usage in C code with a simple lightweight analysis tracking tokens across all C files.

We demonstrate the capability to detect different types of build smells, both within a single artifact and those that span across multiple artifacts:

- **Unused variables:** We report any variable in the configuration logic that is never subsequently used (dead store). Without any use, the variable complicates the build logic without any effect. A simple analysis of the symbolic execution trace is sufficient to detect unused variables, solely within the configuration logic.
- **Unused build substitution variables:** We report substitution variables in Makefile templates if they are never actually substituted by the configuration logic. Such substitution variables complicate the Makefile logic without having any effect and can lead to unnecessary maintenance and dead build code. To detect this smell, we identify all substitution variables in the Makefile template and match them against build actions in the configuration logic, reporting those for which we do not find a match. This analysis is simple in that it only matches structures in abstract syntax trees, but it analyses both the build logic and the configuration logic.
- **Unused C-Preprocessor (CPP) macros:** We report C-Preprocessor macros (i.e., #define) declared from a build action and are never used anywhere within the source code. Similar to build substitutions, we match the declared macro against all tokens in the source code and report build actions without any match; this analysis compares configuration logic against the source code.

4 Implementation

This section will briefly describe the implementation details behind the AutoHaven platform and the proposed tools. For parsing the configuration script, we use ANTLR4 [1] and write a grammar to parse the shell script. And then update it per our identified common patterns to parse the other Autoconf script constructs. Anything that does not conform to our grammar is kept as a lump of text as a special AST node. This kind of special nodes includes any programming languages snippets (i.e., C) written in the configuration script, and any Autoconf script constructs not covered by our common patterns. ANTLR4 generates a parser that parses the script to generate a parse tree; we use that to construct our AST.

For static analysis on the generated AST, we utilize a visitor pattern to traverse the AST and extract any needed information. For doing symbolic analysis, we implement it per Section 3.2. For analyzing the build script to extract all substitution variables, we take advantage of their unique pattern (e.g.,*@variable name@*) and utilize a string pattern matching algorithm to identify all of the variables. And for analyzing the system's source code to extract all of the used CPP macros, we use JavaCPP [10] to preprocess the source code base and identify all used CPP macros.

With these, we have all needed analysis to implement the ACSense tool and provide comprehension support, and the ACSniff tool to detect build smells as discussed in the approach section.

5 Empirical Evaluation

Our evaluation focuses on the technical side and the accuracy of our proposed platform and tools, by answering the following research questions:

(RQ1) How many of our simplified assumptions (i.e., targeting common patterns for parsing, and evaluating for loops once) hold when analyzing actual systems?
(RQ2) What is the accuracy of our configuration script summarization? Are we able to accurately capture all build actions and correctly calculate their path conditions?
(RQ3) How pervasive are build smells in the build systems?
(RQ4) What is the accuracy and performance of our approach to detect build smells?

5.1 Subject Systems

To answer our research questions, we need to use actual GNU Autotools based build systems. To find such systems, we use Boa [25], Boa is a domain-specific language and infrastructure that eases mining software repositories. The Boa platform is available online [2] and comes preloaded with a snapshot of SourceForge [17] and GitHub [5] software repositories. We use Boa to collect ten open source systems from SourceForge. Our query looks for systems that have a GNU-Autotools-based build system and counts the number of revisions that made changes to Autotools configuration files. We sorted them in descending order and picked the top ten systems for evaluation. The systems are shown in Table 1.

5.2 (RQ1) Evaluating Assumptions

We evaluate how well our simplifying assumptions hold in practice. The assumptions are: **(a)** configuration scripts follow identified patterns, **(b)** loops are rarely used, and when used, their bounds are easy to compute. For evaluating the impact, we count the number of constructs and macros being

Table 1. Subject systems

System	Rev	LOC	#for	#if	#Configs
BogoFilter	7053	916	6	64	11
BZFlag	22835	1379	6	98	16
D2X XL	14729	483	1	30	15
Gmerlin	5129	1399	0	116	21
Hercules-390	a9bea8a	2075	0	88	15
Illumination	4138	463	0	44	17
OpenVRML	4316	684	0	65	13
Qore	7401	1872	8	239	21
QTractor	c9ca26b	1480	2	143	60
TuxBox2	15664	652	0	45	35
Average		**1141**	**3**	**94**	**21**

Column *Rev* shows software revision used for evaluation (SVN or Git), *LOC* is total lines of code in configuration script file, *#for* is the number of loop constructs in the script, *#if* is the number of branching constructs in the script, and *#Configs* is total number of build configurations.

Table 2. Evaluating parsing assumption

System	#Constructs	#In AST	Recall
BogoFilter	326	310	95%
BZFlag	455	444	98%
D2X XL	150	145	97%
Gmerlin	353	325	92%
Hercules-390	444	440	99%
Illumination	249	247	99%
OpenVRML	267	262	98%
Qore	600	584	97%
QTractor	497	493	99%
TuxBox2	308	0	0%

Column *Constructs* shows the total number of control structures and M4 macros in the configuration script. Column *In AST* shows the total number of these constructs that is parsed and represented in the AST. Column *Recall* shows the percentage of constructs recalled in our AST.

used in the configuration script manually and compare them to what the AST has.

Table 2 shows the results. Overall we are able to parse nine out of the ten subject systems. The last system, TuxBox2, that we could not parse, did not conform to our common patterns; instead, it had declared its own macros and used them to declare configuration options and build actions. To evaluate our first assumption, we compare the number of structures in the original script to what we capture in our AST. We manually count the number of structures (e.g., if statement, M4 macros...etc.) and compare them to what we have in our AST. For the nine subject systems we were able to parse, we have a 97% recall for the script structures. This indicates that our identified characteristics are common, and even though our approach is not generic, it is still applicable and useful.

To evaluate our second assumption, we check how many subject systems have loop constructs, and how frequent are they. Then we check whether they are bound or not. Table 1 shows that only five out of the ten subject systems uses loop constructs. Compared to the other constructs they are not used often. Then we manually checked all of the loop constructs and they were bound to a constant number of iterations. This confirms our assumption.

5.3 (RQ2) Accuracy of Summarization

To provide comprehension support, we summarize the configuration script by listing the configuration options and the build actions along with their path conditions. To evaluate our solution, we study the subject systems and manually identify their configuration options and build actions, by identifying the relevant M4 macros (i.e., AC_ARG_ENABLE, AC_SUBST) within the script, and for every build action, we manually calculate their path conditions. We accumulate this information as our oracle of truth and use it to answer this research question. Refer to section 6 for a discussion on threats to validity.

To answer this research question, we run the ACSense tool to evaluate how many of the configuration options and build actions the tool reports compared to the oracle. For the build actions, we also evaluate the accuracy of the reported path conditions; if the path condition reported by the tool does not match the one we manually calculated for the oracle, we presume it is incorrect.

Table 3 shows the results. Overall ACSense reports build actions with a high recall of 86% and is able to identify their path conditions with a high precision of 88% on average. The build actions we missed are either missing in our AST representation (e.g., part of an unrecognized pattern) or are part of Autoconf macros that are currently not supported in our symbolic evaluation. For instance, *OpenVRML* relies on Autoconf macros to control the path conditions for most build actions, and our symbolic evaluation does not support these macros at the moment, as they are not commonly used. This is the reason why ACSense did not perform well for calculating path conditions for *OpenVRML*.

5.4 (RQ3) Presence of Build Smells

To answer our third research question on how pervasive build smells are, we investigate our subject systems to see how many of them do have build smells and whether it is a common problem across Autotools-based build systems.

As discussed earlier (Section 3.4), manually detecting build smells is complex and not trivial, and it would be infeasible to attempt to answer this research question by manually detecting the build smells and how widespread they are across our subject systems. Instead, we use our ACSniff tool

Table 3. Evaluating ACSense

System	#Actions	Recall	#PC	Precision
BogoFilter	44/55	80%	32/33	97%
BZFlag	63/87	72%	37/37	100%
D2X XL	32/35	91%	27/27	100%
Gmerlin	49/49	100%	15/19	79%
Hercules-390	35/36	97%	28/30	93%
Illumination	42/66	64%	23/33	70%
OpenVRML	31/31	100%	1/16	6%
Qore	63/74	85%	38/39	97%
QTractor	77/77	100%	57/57	100%
Average	**49/57**	**86%**	**29/33**	**88%**

Column *#Actions* shows the total number of actions detected by ACSense over those declared in the script. Column *Recall* shows the percentage of how many actions ACSense was able to recall. *#PC* shows the number path conditions correctly computed by ACSense to the ones we manually computed for the oracle. Column *Precision* is how many path conditions did ACSense identify correctly.

and run it on all subject systems that we can parse. Then we manually verify whether the reported build smells are actual ones to answer the third research question.

Table 4 shows our results. We were able to find 184 build smells in all parseable subject systems; this indicates that the problem exists and is frequent across Autotools build systems.

5.5 (RQ4) Accuracy and Performance of Build Smell Detection

Given that build smells exist and are widespread in Autotools build systems, we evaluate the accuracy of our provided solution in detecting build smells. As it involves multiple analysis techniques, we also evaluate the performance of our solution.

Again, it is infeasible to build an oracle of truth that has all build smells in our subject systems and use that to measure the accuracy of ACSniff reported build smells. In this work, we focus on providing initial results that later work can improve on, the manual work involved to measure soundness is out of scope.

For our evaluation, we run the ACSniff tool and manually check if the reported build smells are actual ones or not. Refer to section 6 for a discussion on threats to validity. Table 4 shows the results, as can be seen, ACSniff accurately detects build smells with high precision of 96% on average. To evaluate performance, we measured how long it took to analyze each system. On average it took 15 seconds per system.

The number of detected unused variable build smells is much higher than the other build smells, after investigating this kind of build smells, we learned that the developers do not use variables implicitly declared by the Autoconf macros to hold user input (i.e., configuration options) or hold the results from checking the build environment. Instead, they explicitly declare their own variables to track that same information, leaving the implicit ones unused. This is the main reason behind the high numbers of unused variables, and we did not see a clear indication on why the developers would not use the implicitly declared variables. We speculate that this could be due to developers not fully understanding what Autoconf macros actually do for them, thus adding their own variables, which aligns with our findings discussed in the motivation section. The false positive reported for unused variables build smells, were due to our approach for symbolic evaluation. As we explicitly declare how to evaluate M4 macros and we do not evaluate all of them, we miss on evaluating snippets of scripts were the reported unused variables are actually being used, thus ACSniff reported them as build smells.

For the remaining build smells, many were due to neglected maintenance as the example discussed in the motivation section.

6 Threats to Validity

For evaluating comprehension support for developers, we built our own oracle of truth, which is prone to human error and might be incomplete or inaccurate. Our mitigation to this is our own experience from studying and analyzing Autotools build systems, as that helped us build this oracle as accurately as we could.

Furthermore, we only evaluated the accuracy of capturing the build actions and their path conditions. But we did not evaluate how much it would actually help the developers better understand their configuration scripts. This is something we leave for future work, where we plan to engage the actual developers in a more thorough study, then report their feedback on the proposed approach.

Lastly, despite a diverse selection of subject systems, our results may not generalize to other systems.

7 Discussion

In this section we discuss our insight for future research direction and recommendations for build tools authors.

7.1 Build Systems Maintenance

Build systems are a crucial part of software systems. Prior work has shed light on the high maintenance overhead associated with build systems, stating the need for tool support to reduce this overhead [18, 30]. And that this is true across the various build tools currently available [28, 34]. More interestingly, they point out that developers tend to be hesitant when it comes to maintaining their build systems, in fear of breaking the build [22, 27, 29]. And as the software system

Table 4. Evaluating build smells detection using ACSniff

	Unused variables		Unused build substitution variable		Unused CPP Macros		
System	Detected	Correct	Detected	Correct	Detected	Correct	**Accuracy**
BogoFilter	57	55	0	0	11	11	97%
BZFlag	40	39	0	0	3	3	98%
D2X XL	10	10	0	0	12	9	86%
Gmerlin	37	37	0	0	0	-	100%
Hercules-390	200	180	3	3	1	1	90%
Illumination	28	27	0	0	1	1	97%
OpenVRML	6	6	0	0	0	-	100%
Qore	113	111	0	0	3	3	98%
Qtractor	94	93	2	2	7	7	99%

The first row shows the targeted build smells, underneath that, the *Detected* column shows the total number of detected build smells by the ACSniff tool, and the *Correct* column shows how many of them are actual build smells. The *Accuracy* column shows ACSniff precision in detected build smell, how many of the ones it reports are actual build smells, and its calculated by dividing the total number of actual build smells reported, over the total number of build smells reported.

evolve, they might miss on properly updating their build system to keep it up to date [29, 39].

We argue that regardless of the build tool used to implement the build systems, the maintenance challenges will be similar in nature and that the solutions to address them might also be similar. For instance, the developers might be hesitant to maintain their build systems due to lack of understanding of how it works. In this work, we introduce a solution to this problem for Autotools build systems, and we introduce the concept of build actions, as a set of common tasks the configuration script executes. One can argue that regardless of the build tool used, build systems can be translated into a series of common build actions: check for dependencies, compile a file, setup deployment packages, and so on. And arguably, by summarizing the build system into a list of build actions, it would provide an easier way for the developers to understand their build system and what it is doing, addressing their hesitation at its root.

More research is needed to study the different natures of build tools. To identify how common the challenges are and whether existing solutions are applicable with minor modifications. This would help guide the future research for build-systems maintenance.

7.2 Consideration for Future Build Tools

Prior work had discussed the importance of build systems, and shed light onto the maintenance overhead associated with them. There is a shortage of analysis tools to help developers cope with this overhead. Future build tools need to learn from the past and design their build tools with analysis in mind, instead of retrofitting after the fact as we had to do. They could avoid having many stages in the build process and isolate the complexity of each stage within itself to avoid cross-stage analysis. They could aim to make the build tool more declarative, as that would make the build system easier to comprehend and analyze. These considerations would simplify the build systems and would enable developers to easily create the analysis tools they need to maintain their build systems.

8 Related Work

Analyzing build files has been recognized as increasingly important. Adams et al.[18, 19] and S.McIntosh et al. [34] have shown how build systems continue to grow in size and complexity. Martin et al. [32, 33] work focused specifically on Makefiles and shed light on the complexity associated with maintaining Makefiles due to the various features and constructs utilized within them. Seo et al. [39] studied builds at Google and stated that up to 37% of their build failed, mostly due to neglected build maintenance. And that build maintenance is associated with high overhead on the developers, Kerzazi et al. [29] found that up to 18% of build fails, and in their study that has an estimated associated cost of more than 336 man-hours, as once the build is broken, the development team is blocked until the build is fixed. This emphasizes on the importance of analysis and tool support for build systems.

Researchers have investigated build system analysis from different perspectives. Most analysis approaches are dynamic and actually execute the build to extract information. For example, van der Burg et al. [43] dynamically detect which files are included in a build to check license compatibility, Metamorphosis [26] dynamically analyzes build system to migrate them, Dietrich [24] analyzes Kbuild based systems

dynamically to derive presence conditions for source files, and our prior work, MkFault [21], combines runtime information with some structural analysis to localize build faults. However, dynamic approaches can only analyze one configuration at a time.

On the other hand, Macho et al., [31] statically analyze Maven based build systems to identify build changes to help developers cope with the evolution of their build systems. Hardt and Munson [27] developed a tool to monitor the source code for structural refactoring to identify the need for build maintenance, and it updates the ANT based build script associated with the system. But, to the best of our knowledge, there are no analysis tools support for GNU Autotools build systems. The KDE developers built, am2cmake, specifically for their needs, to help migrating their Automake Makefile.am to CMake, but it does not provide any means of analyzing the logic within them, nor does it handle the Autoconf configuration scripts. On the other hand, there is some tool support for GNU Make: MAKAO [19] provides visualization and code smell detection support for Makefiles, but it does not support Autotools. SYMake [41] uses symbolic execution to conservatively analyze all possible executions of a GNU Make Makefile. It produces a symbolic dependency graph, which approximates 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. Zhou et al.[44] expands on top of SyMake to identify presence conditions for source files from build code. But all of these tools are built for GNU Make makefiles, and none can analyze GNU Autotools build systems.

9 Conclusion

Build systems are crucial to software systems, and they come with high maintenance overhead. In this work, we provide support for developers in maintaining their GNU Autotools based build systems. Toward that, we provide the AutoHaven platform, which provides an abstract syntax tree representation for the configuration script. This would enable developers to create analysis techniques to aid them in maintaining their GNU Autotools based build systems. Our evaluation shows that AutoHaven provides a practical and useful structural representation of the configuration script.

During our study of open-source systems, we identified that developers face challenges in understanding their configuration scripts and that overtime they neglect to properly maintain their configuration script, leading to the existence of build smells. To aid developers in better understanding their configuration scripts, we introduce ACSense, a solution to summarize the configuration script by capturing the build actions associated with the script, and their path conditions. Toward addressing build smells, we introduce ACSniff, a solution to detect build smells within the configuration script and across multiple build artifacts (i.e.build script and source code). Our evaluation shows that we can accurately capture much of the configuration script's build actions and it's path conditions and that we accurately detect build smells.

For future work, we plan to evaluate how practical our provided solutions are in real system setup. Also, our work opens the door for more accurate and in depth analysis of build systems. We plan to expand on prior work [44] that depends on extracting configuration knowledge from the build system and see how AutoHaven can improve the accuracy of the existing solutions.

Acknowledgments

Kaestner's work has been supported by the National Science Foundation awards National Science Foundation under Grant No. 1318808 and Grant No. 1552944 and the Science of Security Lablet (H9823014C0140). We would like to acknowledge Tien N. Nguyen for his contributions to earlier versions of this work.

References

[1] 2017. ANTLR 4.0. (2017). https://github.com/antlr/antlr4/blob/master/doc/index.md
[2] 2017. BOA website. (2017). http://boa.cs.iastate.edu/boa/
[3] 2017. CMake Official Site. (2017). cmake.org
[4] 2017. D2X-XL. (2017). https://sourceforge.net/projects/d2x-xl/
[5] 2017. GitHub software code repository. (2017). https://github.com/
[6] 2017. GNU Autoconf Manual. (2017). gnu.org/software/automake/manual/html_node/GNU-Build-System.html#GNU-Build-System
[7] 2017. GNU Autotools. (2017). gnu.org/software
[8] 2017. GNU Compiler Collection (GCC). (2017). https://gcc.gnu.org/
[9] 2017. GNU Emacs. (2017). https://www.gnu.org/software/emacs/
[10] 2017. JavaCPP. (2017). https://github.com/bytedeco/javacpp
[11] 2017. K Development Environment. (2017). www.kde.org
[12] 2017. M4 macro language. (2017). https://www.gnu.org/software/m4/m4.html
[13] 2017. Map Server Official Site. (2017). mapserver.org
[14] 2017. Map Server: Request to migrate to CMake. (2017). mapserver.org/development/rfc/ms-rfc-92.html
[15] 2017. Open SSH. (2017). https://www.openssh.com/
[16] 2017. Open VPN. (2017). https://openvpn.net/
[17] 2017. SourceForge software code repository. (2017). https://sourceforge.net/
[18] B. Adams, K. de Schutter, H. Tromp, and W. de Meuter. 2008. The evolution of the Linux build system. In *Electronic Communications of the ECEASST*.
[19] B. Adams, H. Tromp, K. de Schutter, and W. de Meuter. 2007. Design recovery and maintenance of build systems. In *Proceedings of the 23rd International Conference on Software Maintenance (ICSM '07)*. 114–123.
[20] J. M. Al-Kofahi, H. V. Nguyen, A. T. Nguyen, T. T. Nguyen, and T. N. Nguyen. 2012. Detecting semantic changes in Makefile build code. In *Proceedings of the 28th International Conference on Software Maintenance (ICSM '12)*. 150–159.
[21] J. M. Al-Kofahi, H. V. Nguyen, and T. N. Nguyen. 2014. Fault localization for Make-Based build crashes. In *Proceedings of the 30th International Conference on Software Maintenance and Evolution (ICSME '14)*. 526–530.
[22] J. M. Al-Kofahi, T. Nguyen, and C. Kästner. 2016. Escaping AutoHell: A Vision for Automated Analysis and Migration of Autotools Build

Systems. In *Proceedings of the 4th International Workshop on Release Engineering (RELENG '16)*. 12–15.

[23] T. H. Austin and C. Flanagan. 2012. Multiple Facets for Dynamic Information Flow. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '12)*. 165–178.

[24] C. Dietrich, R. Tartler, W. Schröder-Preikschat, and D. Lohmann. 2012. A Robust Approach for Variability Extraction from the Linux Build System. In *Proceedings of the 16th International Software Product Line Conference - Volume 1 (SPLC '12)*. 21–30.

[25] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen. 2013. Boa: A Language and Infrastructure for Analyzing Ultra-large-scale Software Repositories. In *Proceedings of the 35th International Conference on Software Engineering (ICSE '13)*. 422–431.

[26] M. Gligoric, W. Schulte, C. Prasad, D. van Velzen, I. Narasamdya, and B. Livshits. 2014. Automated Migration of Build Scripts Using Dynamic Analysis and Search-based Refactoring. In *Proceedings of the 2014 International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '14)*. 599–616.

[27] R. Hardt and E. V. Munson. 2013. Ant Build Maintenance with Formiga. In *Proceedings of the 1st International Workshop on Release Engineering (RELENG '13)*. 13–16.

[28] L. Hochstein and Y. Jiao. 2011. The Cost of the Build Tax in Scientific Software. In *Proceedings of the 5th International Symposium on Empirical Software Engineering and Measurement (ESEM '11)*. 384–387.

[29] N. Kerzazi, F. Khomh, and B. Adams. 2014. Why Do Automated Builds Break? An Empirical Study. In *Proceedings of the 30th International Conference on Software Maintenance and Evolution (ICSME '14)*. 41–50.

[30] G. Kumfert and T. Epperly. 2002. *Software in the doe: The hidden overhead of the build.* Lawrence Livermore National Laboratory.

[31] C. Macho, S. McIntosh, and M. Pinzger. 2017. Extracting Build Changes with BuildDiff. In *Proceedings of the 14th International Conference on Mining Software Repositories (MSR '17)*. 368–378.

[32] D. H. Martin and J. R. Cordy. 2016. On the Maintenance Complexity of Makefiles. In *Proceedings of the 7th International Workshop on Emerging Trends in Software Metrics (WETSoM '16)*. 50–56.

[33] D. H. Martin, J. R. Cordy, B. Adams, and G. Antoniol. 2015. Make It Simple: An Empirical Analysis of GNU Make Feature Use in Open Source Projects. In *Proceedings of the 23rd International Conference on Program Comprehension (ICPC '15)*. 207–217.

[34] S. McIntosh, B. Adams, T. H. D. Nguyen, Y. Kamei, and A. E. Hassan. 2011. An empirical study of build maintenance effort. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE '11)*. 141–150.

[35] J. Meinicke, C. Wong, C. Kästner, T. Thüm, and G. Saake. 2016. On Essential Configuration Complexity: Measuring Interactions in Highly-configurable Systems. In *Proceedings of the 31st International Conference on Automated Software Engineering (ASE 2016)*. 483–494.

[36] Alexander Neundorf. 2017. Why the KDE project switched to CMake. (2017). lwn.net/Articles/188693/

[37] H. V. Nguyen, C. Kästner, and T. N. Nguyen. 2014. Exploring Variability-aware Execution for Testing Plugin-based Web Applications. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. 907–918.

[38] K. Sen, G. Necula, L. Gong, and W. Choi. 2015. MultiSE: Multi-path Symbolic Execution Using Value Summaries. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE '15)*. 842–853.

[39] H. Seo, C. Sadowski, S. Elbaum, E. Aftandilian, and R. Bowdidge. 2014. Programmers' Build Errors: A Case Study (at Google). In *Proceedings of the 36th International Conference on Software Engineering (ICSE '14)*. 724–734.

[40] R. Suvorov, M. Nagappan, A. E. Hassan, Y. Zou, and B. Adams. 2012. An empirical study of build system migrations in practice: Case studies on KDE and the Linux kernel. In *Proceedings of the 28th International Conference on Software Maintenance (ICSM '12)*. 160–169.

[41] A. Tamrawi, H. A. Nguyen, H. V. Nguyen, and T. N. Nguyen. 2012. Build code analysis with symbolic evaluation. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. 650–660.

[42] Troy Unrau. 2017. The Road to KDE 4: CMake, a New Build System for KDE. (2017). dot.kde.org/2007/02/22/road-kde-4-cmake-new-build-system-kde

[43] S. van der Burg, E. Dolstra, S. McIntosh, J. Davies, D. M. German, and A. Hemel. 2014. Tracing Software Build Processes to Uncover License Compliance Inconsistencies. In *Proceedings of the 29th International Conference on Automated Software Engineering (ASE '14)*. 731–742.

[44] S. Zhou, J. M. Al-Kofahi, T. N. Nguyen, C. Kästner, and S. Nadi. 2015. Extracting Configuration Knowledge from Build Files with Symbolic Analysis. In *Proceedings of the 3rd International Workshop on Release Engineering (RELENG '15)*. 20–23.