Escaping AutoHell: A Vision for Automated Analysis and Migration of Autotools Build Systems

Jafar Al-Kofahi [†], Tien N. Nguyen [‡], and Christian Kästner [*]
[†]Electrical and Computer Engineering Department, Iowa State University, USA
[‡]School of Engineering and Computer Science, University of Texas-Dallas, USA
[*]School of Computer Science, Carnegie Mellon University, USA

ABSTRACT

GNU Autotools is a widely used build tool in the open source community. As open source projects grow more complex, maintaining their build systems becomes more challenging, due to the lack of tool support. In this paper, we propose a platform to build support tools for GNU Autotools build systems. The platform provides an abstraction of the build system to be used in different analysis techniques.

CCS Concepts

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

Keywords

Build system; Autotools; Autoconf; Build analysis

1. INTRODUCTION

Build systems are a crucial part of software systems, as they describe how to process the system's artifacts to produce the deliverables (i.e., executables). Build systems also have configurations that control which features to be included in the produced deliverables. Build systems are described and executed using build tools; some popular build tools are GNU Autotools, Ant, and Maven. Prior research found that build maintenance could impose 12-36% overhead on software development [12]. A study on Linux found that developers have invested significant effort to make their build system as simple as possible [6]. The authors reported that the complexity of build code in Linux co-evolves with fast-growing source code. For large-scale systems, build files grow quickly and complex because they must support the building of the same software in multiple platforms with various configuration and environment parameters [11]. McIntosh *et al.* found that 4-27% of tasks involving source code changes require accompanying change in corresponding build code. They concluded that build code continually evolves and is likely to have defects due to high churn rate [13]. Importantly, those studies call for better tool supports for build code.

To better understand the tool support needed, we took a closer look into open source systems and what kinds of challenges they face with their build systems. A widely used build tool in that community is GNU Autotools [2]. GNU Autotools build systems have a configuration stage. A configuration consists of a declaration, and a set of actions to be taken if it was enabled/disabled, these actions usually adjust the build logic behavior. The second stage is the build logic stage, where developers describe how the configurations actions can adjust the build process, and how to produce the deliverables from the system's resources. Then to build the system, a builder would select the build configurations, run the configuration stage to adjust the build logic accordingly, and then kicks-off the build by running GNU Make. More on this in Section 2.

Autotools has its own challenges and drawbacks. Several open source projects [3, 4] that have used Autotools, recently migrated away from Autotools to CMake, or other build tools. Looking into their commits, email archives, and developer blogs, we identified the following common reasons for migration: **(1)** steep learning curve for Autotools, **(2)** multi stage build system(s), **(3)** lengthy and complicated configuration stage, **(4)** and lack of tool support.

In this work, we aim to create visibility into Autotools build systems, by providing a platform, AutoHaven, to serve as the foundation for build analysis and migration tools, to help developers overcome the challenges associated with build systems. AutoHaven would provide an abstraction of the build system using parsing and symbolic analysis. This abstraction can be leveraged by support tools to perform their analyses on build systems. For example, to help build system migration, a tool can use our abstraction to extract the build system semantics (e.g. configurations), and use that to reduce the effort needed in the migration process. We already started in our endeavored to build this platform; at this time, we can parse the configuration stage, and are building an Abstract Syntax Tree (AST) and planning symbolic execution for it. The AST is not sound or complete, due to the nature of staged applications, but, from studying open source systems, we noticed that developers follow common patterns for declaring configurations and their actions, this helps us define the structure for our AST. The AST will provide sufficient information about the configurations and their actions. Eventually we are planning to also build an AST for build logic, more on this and example applications later in Section 3.

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.

RELENG'16, November 18, 2016, Seattle, WA, USA
© 2016 ACM. 978-1-4503-4399-2/16/11...$15.00
http://dx.doi.org/10.1145/2993274.2993279

2. GNU AUTOTOOLS

2.1 Overview

Figure 1 shows a typical GNU Autotools build system. For a given system, the developers provides *Makefile.am* Automake files, to describe how to build the system on a high level; they also provide *configure.ac* Autoconf file, to describes the system external dependencies, and the build system configurations. Autoconf is written using GNU M4 macros, shell scripting, and other programming languages. GNU Automake processes the *Makefile.am* files to identify the source files to be built, and then it generates *Makefiles.in* templates that holds the build logic for the system. These templates are annotated with special placeholders, that are later used to adjust the build logic according to the selected build configurations. GNU Autoconf processes the *configure.ac* file, expand the M4 macros and generate the *configure* shell script file. The *configure* 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 a similar fashion the configure script can adjust the build logic based on the selected build configurations and environment settings. After that the user can run GNU Make on the concrete Makefiles to build the system and produce the deliverables.

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
```

Listing 1 shows an Autoconf file snippet, of a Hello World system, that prints a hello world message in different languages. Lines 3-6, call the *AC_ARG_ENABLE* macro to declare a feature called *localization*, this would be interpreted as an argument to the configure script passed as *–enable-localization*. If the user does use this argument, the value will be stored in variable *LANG* at line 6. The if statement in line 10 would check the variable *LANG* and if it was assigned *"ar"* value, it then calls macro *AC_SUBST* that would declare a substitution variable called *LANG*, when consuming Makefile.in templates, this substitution variable would replace any instances of *@LANG@* with the value *"ar"*. Autoconf uses M4 macros (e.g.*AC_SUBST*) to express certain functionality, that get expanded to pre-defined shell script blocks.

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" [14]. GNU Autotools is widely used in the open source community, but often criticized. As systems evolve and become more complex, many have migrated away from GNU Autotools to other build tools, such as CMake [1]. We investigated some recent migrations, and looked into their code repository commits, email archives, and developer blogs to identify the reasons for migrating.

For example, in version 4, the KDE [3] team decided to migrate away from Autotools. They had two attempts in this migration [15], and the second attempt succeeded in migrating KDE to CMake [15, 14, 17]. Another example is Map Server [4], that migrated away from Autotools to CMake [5]. The following challenges were often mentioned as reasons.

- **Steep learning curve:** Understanding the different tools that come into play and their role in the workflow, is not as straight forward as it looks. Also one need 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 need to generate the configure script and makefile templates and actually run the two stages of the build system, but any fixes must be on his input, the *.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.

3. PROPOSAL: AutoHaven PLATFORM

To address the challenges associated with GNU Autotools, we propose to build a platform to be used as a foundation for creating tools to support the developers understanding, maintaining, or migrating their GNU Autotools build systems. To accomplish this, AutoHaven will provide an abstract approximation of the build system, this abstraction would provide syntactical structure for the build system (AST). Then analysis techniques for build systems can be created using this abstraction. Such platform can have different kinds of applications depending on the developer's needs. Here we discuss some potential applications:

- **Migration:** When developers migrate to a new build tool, they tend to consider new requirements for their build system [15]. But one important requirement is to ensure the configuration knowledge and build logic are migrated in a complete and accurate fashion. This is where AutoHaven would help, the developers can process the generated ASTs to derive feature declarations and their expected effect on build and source code. For example, if the build code is expected to define certain macros for the source code it is important to capture those and their conditions correctly. Similarly in the build logic, using our AST the developer can identify the needed dependencies and build script to build any target, thus it would avoid unnecessary errors due to migration.

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

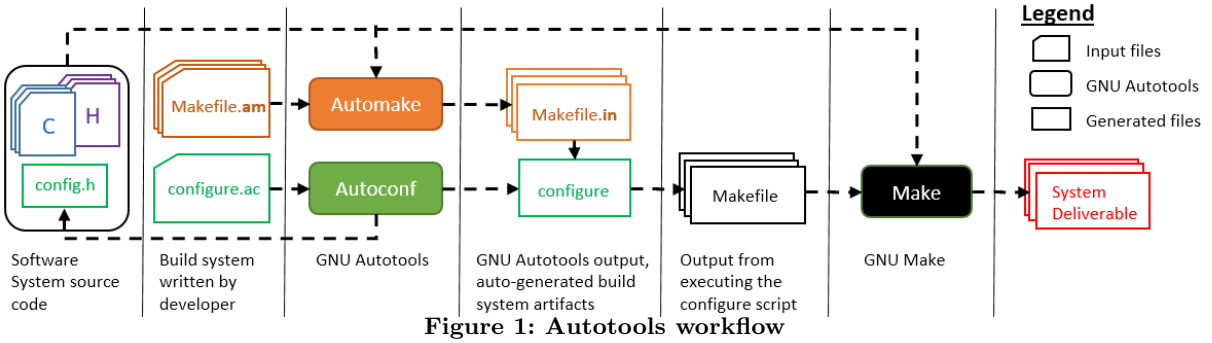


Figure 1: Autotools workflow

- **Testing:** To the best of our knowledge, there isn't a way for developers to test their builds, especially the configuration space, other than manually running the build process as much as needed. Using AutoHaven AST, the developers can define a coverage metric for their build system, and automate their testing entirely or partially. Thus achieving more meaningful coverage efficiently, compared to the developers manually identifying and running test builds.

In building this platform, we address the following challenges:

- **Nature of Autoconf configure.ac script:** Autoconf files are written using shell script, M4 macros, and any other language the developer uses in their configure.ac files. Due to this the abstract representation is not sound nor complete, and we will be limited with what we can represent.

- **Staged Program Analysis:** Autotools build systems are split into two domains: configuration and build logic, and they involve various generation steps to build the system. An analysis platform need to include both domains to allow for meaningful analysis to be performed.

3.1 Autotool Build System ASTs

From studying existing Autoconf scripts, we noticed that developers don't have any restrictions on how to write their scripts, and they can use any tool or language to accomplish configuring their systems. But we also noticed that they follow common patterns for declaring configurations and how those configurations affect build and source codes. One of those patterns already shown in the example script in Listing 1. Usually the developers would declare a feature using Autoconf M4 macro *AC_ARG_ENABLE*, then this feature is expressed via a variable (i.e.*LANG*), that is used later on to adjust the build process (e.g.via calls to the *AC_SUBST* macro), having these common patterns, and the use of M4 macros, gives a structure to the Autoconf files, and this allows us to actually approximate the Autoconf files. It is important to also note that the logic within the Autoconf usually involves simple *if-statements* and no loops to propagate values used during executions and within constraints, which can be executed/analyzed symbolically. For the remainder of the Autoconf scripts that does not conform to shell grammar, M4 macros, nor to those common patterns, we will hold onto them as they were presented in the Autoconf script and represent them as plain-text nodes in our representation, and leave it to the analysis tool to decide whether and how to interpret them. Figure 2 shows an example Autoconf-AST for the *if* statement in *configure.ac* script in Listing 1.

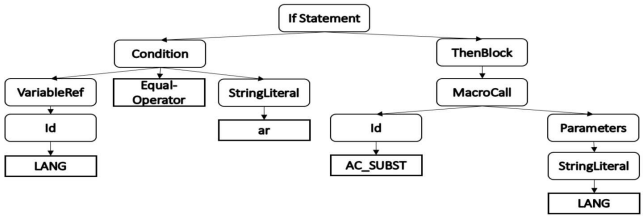


Figure 2: Example Autoconf AST

The Automake makefiles are high-level description of the build logic, and they do not describe the build for all the system artifacts. Due to that, using the Makefile.in templates, the output from Automake in workflow 1, would provide more insight into the build system for our purposes. For their AST's we will expand on the existing work of Symake [16]. Symake symbolically executes GNU Make makefiles, in this work we will be expanding on that to handle the Automake makefile templates.

3.2 AutoHaven Analysis

The platform needs to represent configuration and build logic domains to enable meaningful analysis of the build system. To do that it will mimic the Autotools workflow and provide the ability to symbolically execute the build process from configuration to build logic. For a given input to the configuration step, it evaluates the Autoconf AST and identifies the expected outputs, then generates the proper configuration header file (i.e. config.h) and provides the expected substitution variables to the Makefile.in ASTs. In testing, this can help automate selecting different combinations of build configurations, instead of the tedious manual process done by the developers.

3.3 Implementation

For parsing Autoconf files, we started with an existing grammar for Shell script, we expanded upon it to handle the Autoconf M4 macros. Anything not recognized by those two is considered as an unfamiliar structure and stored as plain-text in a special node type. Using this grammar we successfully can parse Autoconf *configure.ac* files of various projects (e.g., Emacs, OpenVPN, and OpenSSH). Currently we are building the AST from the parser output for Autoconf files. The next step is to expand upon Symake AST to handle variable substitutions from Makefile.in templates. Then to build the rest of the AutoHaven platform to symbolically execute per Autotools workflow.

4. RELATED WORK

Analyzing build files has been recognized as increasingly important. Adams et al.[7, 6, 13] 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. Most analysis approaches are dynamic and actually execute the build to extract information. For example, van der Burg et al.[18] dynamically detect which files are included in a build to check license compatibility, Metamorphosis [10] dynamically analyzes build system to migrate them, and MkFault [8] combines runtime information with some structural analysis to localize build faults. However, such dynamic approaches can only analyze one configuration at a time.

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 [7], provides visualization and code smell detection support for Makefiles. Our own prior work, SYMake [16], uses symbolic execution to conservatively analyze all possible executions of a GNU Make 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. MkDiff [9] expands on top of SyMake to detect semantic differences, in the build logic, by comparing changes on the symbolic dependency graphs between two versions for a given Makefile. Zhou et al.[19] also 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.

5. CONCLUSION

We investigate the pain points associated with GNU Autotools build systems, and why developers migrate away from them to other build tools. As a solution to help mitigate the challenges associated with GNU Autotools build systems, we propose AutoHaven, a platform to build AST representation for such build systems, to provide a needed foundation for building support tools for developers using GNU Autotools. We propose how to build those ASTs, and provide potential applications of such platform and its advantages.

6. ACKNOWLEDGEMENTS

This work was supported in part by US NSF grants CCF-1320578, CCF-1318808, and CCF-1552944.

7. REFERENCES

[1] CMake Official Site. cmake.org.

[2] GNU Autotools. gnu.org/software.

[3] K Development Environment. kde.org.

[4] Map Server Official Site. mapserver.org.

[5] Map Server: Request to migrate to CMake. mapserver.org/development/rfc/ms-rfc-92.html.

[6] B. Adams, K. de Schutter, H. Tromp, and W. de Meuter. The evolution of the Linux build system. In *Electronic Communications of the ECEASST*, Aug 2008.

[7] B. Adams, H. Tromp, K. de Schutter, and W. de Meuter. Design recovery and maintenance of build systems. In *2007 IEEE International Conference on Software Maintenance*, pages 114–123, Oct 2007.

[8] J. Al-Kofahi, H. V. Nguyen, and T. N. Nguyen. Fault localization for Make-Based build crashes. In *2014 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 526–530, Sept 2014.

[9] J. M. Al-Kofahi, H. V. Nguyen, A. T. Nguyen, T. T. Nguyen, and T. N. Nguyen. Detecting semantic changes in Makefile build code. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 150–159, Sept 2012.

[10] M. Gligoric, W. Schulte, C. Prasad, D. van Velzen, I. Narasamdya, and B. Livshits. Automated migration of build scripts using dynamic analysis and search-based refactoring. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 599–616. ACM, 2014.

[11] L. Hochstein and Y. Jiao. The cost of the build tax in scientific software. In *2011 International Symposium on Empirical Software Engineering and Measurement*, pages 384–387, Sept 2011.

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

[13] S. McIntosh, B. Adams, T. H. D. Nguyen, Y. Kamei, and A. E. Hassan. An empirical study of build maintenance effort. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 141–150, May 2011.

[14] A. Neundorf. Why the KDE project switched to CMake. lwn.net/Articles/188693/.

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

[16] A. Tamrawi, H. A. Nguyen, H. V. Nguyen, and T. N. Nguyen. Build code analysis with symbolic evaluation. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 650–660, June 2012.

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

[18] S. van der Burg, E. Dolstra, S. McIntosh, J. Davies, D. M. German, and A. Hemel. Tracing software build processes to uncover license compliance inconsistencies. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 731–742. ACM, 2014.

[19] S. Zhou, J. Al-Kofahi, T. N. Nguyen, C. KÃd'stner, and S. Nadi. Extracting configuration knowledge from build files with symbolic analysis. In *2015 IEEE/ACM 3rd International Workshop on Release Engineering (RELENG)*, pages 20–23, May 2015.