Analyzing Software Build Architectures

Bo Zhang Vasil Tenev Martin Becker
Fraunhofer Institute for Experimental Software Engineering (IESE)
Kaiserslautern, Germany
Email: {bo.zhang, vasil.tenev, martin.becker}@iese.fraunhofer.de

Abstract: *In order to derive executable software artefacts in an efficient and effective manner, a sound build system needs to be maintained properly along with the evolution of source code. However, in large-scale software projects the building process often becomes effort consuming and sometimes error prone, which is often caused by an eroded architecture of the build system. While sound method and tool support to analyze the evolution and the erosion of software architecture exists, the situation for the architecture of the build system is different. This renders the evolution of the build system a non-trivial task. In consequence, especially change-intensive software projects, which either evolve fast or have to deliver many software variants at a point in time, are often facing serious challenges in the long-term run. To cope with these challenges, we first discuss typical challenges and their root causes in the context of build architectures. Then we present our analysis approach and tool chain, which consists of a make file parser, build dependency model and a respective visualization of the build system architecture.*

1 Introduction

While normal source code (also known as production code) implements the behavior of a software product, its build system (including build tools and build code, such as makefiles) derives the executable software from its production source code. In large industrial software systems, the complexity of the build system is often high (in terms of build jobs and build dependencies), and the building process is time-consuming (over one hour in large systems) even in a distributed environment using high-performance and multi-core computers. This is not acceptable in real continuous integration settings with frequent code revisions and builds per day.

While sound method and tool support to analyze the evolution and the erosion of software architecture exists, the situation for the architecture of the build system is different. There are a few tools for build system analysis and optimization (e.g., [2][4][6]), but they have limitations either in system scalability or in supported build techniques and tools. In consequence, especially change-intensive software projects, which either evolve fast or have to deliver many software variants at a point in time, are often facing serious challenges in the long-term run.

In this paper, we first discuss the existing build system environment and challenges (in Section 2) and then introduce MArZ (in Section 3), which is our build architecture analysis approach and tool support.

2 Build Systems in the Wild

From our industrial project experiences in the recent years, we have seen build systems in complex large-scale systems typically with following settings:

a) A build system often uses multiple build tools (e.g., GNU Make, CMake, EMake [2]) and programming languages (e.g., Makefiles, Bash scripts, Python scripts).

b) There are inclusion and (recursive) invocation relationships between build scripts, either within the same programming language or across different programming languages.

c) Build scripts could be generated at the building time.

d) Build tools could be also built and configured at the building time depending on the building environment (e.g., in Android).

e) A build system uses additional build optimization tools (e.g., EMake [2]). Although such tools provide certain optimization (e.g., build acceleration) support, they also change build behavior internally, which is hard to analyze and verify from outside.

f) A build system is executed in a single run to build multiple software system variants (typically for a product family) instead of building a single software system. This makes the build process more complex.

g) In large-scale systems, the build system and process is often deployed in a number of distributed machines. This makes the build analysis more difficult.

Given these settings regarding the software and hardware environment of build systems, industrial practitioners are have challenges in developing and maintaining a complex large-scale build system against build errors and low build efficiency. The root cause is lack of knowledge with respect to the build jobs, build artefacts, and their dependencies. This also causes erosion of the build architecture over time. In practice, we have seen different erosion symptoms such as build redundancy, build obsoleteness, and suboptimal build parallelism.

3 MArZ Build Analysis

In order to understand and maintain build systems in industry, we have developed a build management approach and tool chains called MArZ (Makefile Architecture Analyzer). For an existing build system, it can extract the build architecture, generate the corresponding build model, and conduct further build measurement and optimization.

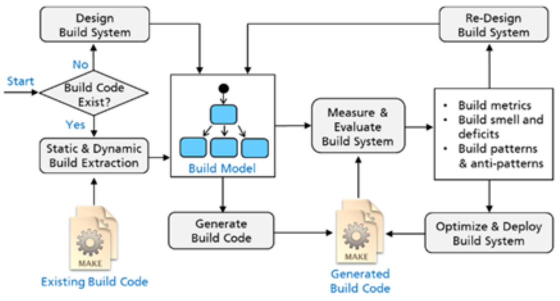

Fig. 1. MArZ Build Analysis Approach.

3.1 Build Extraction

In order to extract the build architecture from an existing build system as well as its run-time information, we have developed techniques and tools for both static and dynamic build analyses.

Currently, the static build extraction essentially focuses on Makefiles (they are typically the most frequently used build language) as well as its executing shell environment (e.g., Bash or Python). By implementing a Makefile parser (based on the open source tool Kati [3]), we extract Makefile targets as build jobs (including the build command as well as input and output artefacts), Makefile inclusion, recursive make calls in a Makefile, as well as variables and conditional logics defined in a Makefile. Moreover, we can also extract the Makefile and target that is executed in a shell script. It is important especially when the same Makefile target is executed multiple times in different shell contexts.

Besides static extraction, we have also managed to monitor the build process in a dynamic analysis. This helps in extracting the duration time of each build job (based on the open source tool remake [5] and a callgrind parser). Moreover, the dynamic analysis can also trace make calls executed in a Bash script.

3.2 Build Modeling

Based on the extracted build information, a build model can be created as a graphical representation. To this end, we have defined the build model syntax based on UML 2.0 and implemented a tool for generated such a model in Enterprise Architect automatically. In practice, a developer can create a build model either from scratch or based on the extraction of an existing build system. Such a graphical model is easy to understand comparing with build code. For build modules that are complex or change-intensive, one can only maintain the build model and have the build code generated automatically.

3.3 Build Measurement and Optimization

As a follow-up step, we conduct automated measurement on the extracted build model to identify build smells and deficits (e.g., redundant and obsolete build jobs) as well as build patterns or anti-patterns. This helps improve the build system.

4 Conclusion

This paper discusses challenges in developing and maintaining build systems in practice, and introduces the MArZ approach and tool chains for analyzing the build architecture. We have applied this analysis in recent industrial projects, and will enhance it in the future work.

References

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

[2] ElectricInsight. http://electric-cloud.com/.

[3] Kati tool. https://github.com/google/kati.

[4] Makefile::Parser. https://github.com/agentzh/makefile-parser-pm.

[5] Remake tool. http://bashdb.sourceforge.net/remake/

[6] Tamrawi, A., Nguyen, H. A., Nguyen, H. V., and Nguyen, T. N. 2012. SYMake: a build code analysis and refactoring tool for makefiles. In the 27th IEEE/ACM International Conference, 366–369.

[7] B. Zhang, V. Tenev, and M. Becker, "Android build dependency analysis," in 2016 IEEE 24th International Conference on Program Comprehension (ICPC). IEEE, May 2016, pp. 1-4.