Co-Evolution of Source Code and the Build System

Bram Adams
Software Analysis and Intelligence Lab
School of Computing
Queen's University (Canada)
bram@cs.queensu.ca

Abstract

A build system breathes life into source code, as it configures and directs the construction of a software system from textual source code modules. Surprisingly, build languages and tools have not received considerable attention by academics and practitioners, making current build systems a mysterious and frustrating resource to work with. Our dissertation presents a conceptual framework with tool support to recover, analyze and refactor a build system. We demonstrate the applicability of our framework by analyzing the evolution of the Linux kernel build system and the introduction of AOSD technology in five legacy build systems. In all cases, we found that the build system is a complex software system of its own, trying to co-evolve in a synchronized way with the source code while working around shortcomings of the underlying build technology. Based on our findings, we hypothesize four conceptual reasons of co-evolution to guide future research in the area of build systems.

1. Introduction

A build system turns a set of source code and data files into a suite of executable programs. As such, it (1) provides a uniform interface for selecting the desired features and environment (*configuration layer*), and (2) efficiently invokes construction tools like compilers in the correct order with the appropriate flags and input (*build layer*). The configuration layer is modeled by configuration scripts, which specify configuration parameters and their constraints. It configures any parametrized source code file or build script before the build layer starts its work. The build layer is modeled by build scripts (e.g., makefiles [7]), which specify build dependencies and instructions.

Most stakeholders in the software development process interact directly or indirectly with the build system on a daily basis, as the build system forms the backbone of modern software development (e.g., continuous integration or release management). Surprisingly, the evolution and maintenance of build systems has been largely ignored in research, except for work on modularising a build system [5], recovering its high-level architecture [14] and speeding up the build process. As a consequence, little (tool) support exists to improve the understanding and re-engineering of build systems. In particular, little is known about the intricate relation between the build system and the source code. This can have dramatic consequences for stakeholders, as the following two observations show:

1. **Development slowdown**. KDE is a highly configurable, large, open source desktop environment. By 2005, its build system had become so complex that even simple tasks like moving a file from one folder to another were tedious, often with unforeseen consequences [12]. Switching to a new build system seemed to be the only way out, but this required developers to recover and reconstruct the knowledge hidden inside the old build system. Only their second attempt succeeded.

2. **Hindering reuse**. De Jonge [5] warns of the hidden impact of the build system on source code reuse. A component without a separate build system cannot be extracted from its originating system and plugged into other systems, as its configuration options and constraints are spread throughout the global build system. Build system limitations discourage developers from reusing components.

In both observations, software engineers are either not aware of the co-evolution of source code and the build system, or not able to propagate changes between source code and the build system in a synchronized way [6]. For example, if new source code features are not properly specified or constrained in the configuration scripts, users cannot benefit from them. Likewise, linking an application with a wrong library version causes unpredictable run-time behavior. On the other hand, if the build system defines new conditional compilation flags or header file locations, the source code needs to be updated to reflect these changes.

The goal of our dissertation is to improve the understand-

ing of build systems and to investigate the **C**o-evolution of **S**ource code and the **B**uild system (**CSB**), by studying real-world systems and exploring a worst-case scenario from the CSB's point of view. This paper first presents our research hypothesis (Section 2), followed by an overview of the main dissertation parts (Section 3). Finally, we summarize our major contributions in Section 4.

2. Research Hypothesis

The build system is crucial in software development, but is not well understood by practitioners. Tools and techniques are needed to understand a build system and its co-evolution with the source code.

Co-evolution is a software engineering phenomenon in which different software models are causally connected [6]: if one model changes, this may have an impact on the other one and vice versa. Co-evolution can occur between any two or more artifacts (or phases) of software development: architecture/implementation, design regularities/-source code, source code/tests, etc. Our dissertation focuses on the co-evolution of source code and the build system.

The work is broken down into four parts. First, we design and implement a framework for understanding and re-engineering build systems. Second, we validate this framework by analyzing the evolution of the Linux kernel build system. Third, we apply the framework to examine the impact of CSB on the introduction of AOSD technology in legacy systems. This co-evolution scenario confronts a significant change on the source code level with rigid build system technology. Fourth, we distill four conceptual reasons from the case study findings for the existence of CSB. These reasons represent important areas of future work in build system research and practice.

3. Overview of the Dissertation

3.1. Conceptual Framework for Understanding Build Systems

Build systems are very hard to understand and modify. At the level of individual configuration and build scripts, users face arcane, complex configuration and build tools, under- or over-specified configuration and build dependencies, platform-specific build instructions and problems with enforcing versioned dependencies. There are hardly any debugging tools for build systems either. At the system level, users need to identify performance bottlenecks, grasp the different phases in the build process [14], and understand the build architecture and its implications. For example, "recursive make" is a popular "make" architecture in which each directory has its own build script that is invoked in a separate "make" process by the parent directory's build script. Although conceptually simple, it requires workarounds like adding multiple build loops or redundant build dependencies to deal with inconsistent build dependencies, race conditions during parallel build and pluggability of build scripts of new subsystems.

As changes to the build system occur very frequently and need to propagate through many configuration/build scripts at once [13], tools are required to support developers and the many other users of the build system. Our dissertation proposes a re(verse)-engineering framework for build systems, named "MAKAO" [2], which provides five important features to improve build system understanding and modification. First, MAKAO visualizes the build dependency graph of a concrete build, with flexible navigation. In Figure 1, colors identify the different file types, edges have the same color as their destination node and all files described by the same build script are enclosed by the same convex hull. Second, users can interactively query for static information in the build scripts and for the dynamic values of build script variables. Third and fourth, MAKAO's Prolog-based pattern matching facility allows to filter out patterns in the build dependency graph, either to make the dependency graph easier to interpret or to detect bad smells. Fifth, MAKAO can re-engineer a make-based [7] build layer.

3.2. Evolution of the Linux Kernel Build System

We used MAKAO to recover the design of the Linux kernel build system in all major releases from 1991 until 2007 [3]. For each release, we generated the build dependency graph of the default build configuration, measured graph metrics like the number of nodes/edges in the build dependency graph and consulted external resources like the dedicated mailing list and documentation. We found that the kernel build system evolved from a couple of build scripts to a complex system of nearly one thousand build scripts, half as many configuration scripts and almost one hundred additional shell scripts and programs. Between the 2.4.x and 2.6.x kernel series the build system even saw a major overhaul, as the highly dissimilar build dependency graphs in Figure 1 show. MAKAO enabled us to understand the kernel build system and to recover the re-engineering steps performed between each successive version.

3.3. Introduction of AOSD Technology in Legacy Build Systems

In order to analyze a concrete scenario in which CSB plays an important role, we chose to study the introduction of AOSD technology [10] in legacy systems [11]. As-

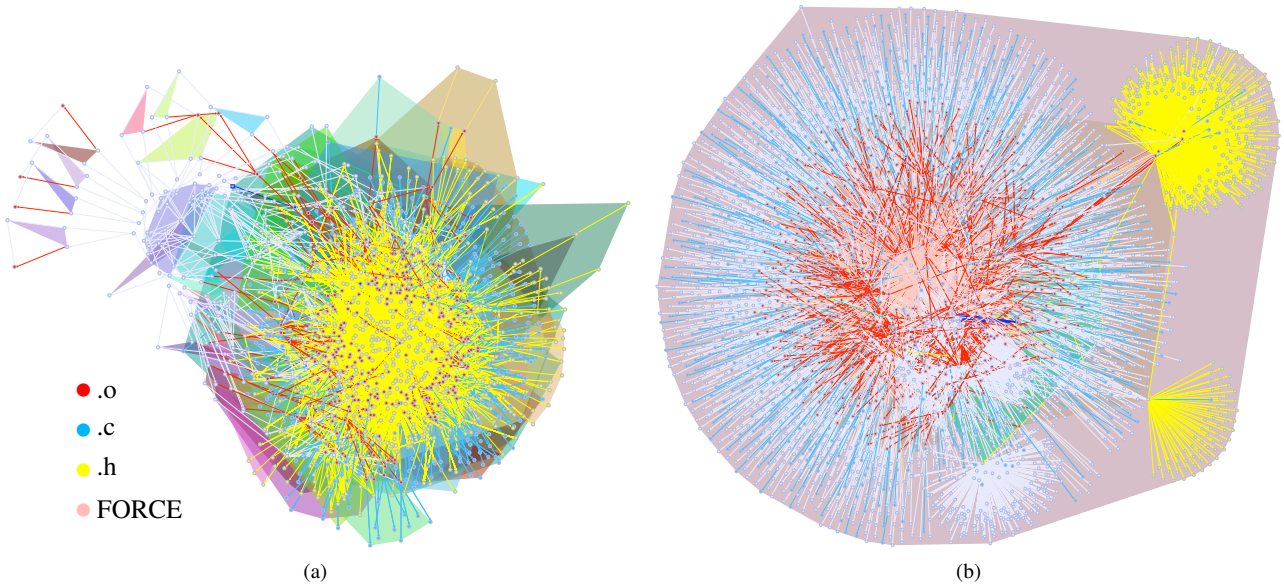

Figure 1. Build dependency graphs of Linux 2.4 (a) and 2.6 (b).

pects are modules that encapsulate "crosscutting concerns" (CCCs) like logging or caching. Without AOSD, the implementation of a CCC is typically "scattered" across multiple modules, where it is "tangled" with the implementation of other concerns. With AOSD, the CCC can be implemented as "advice" inside aspect modules. Contrary to a procedure, advice is triggered ("woven") automatically whenever a specified run-time condition ("pointcut") is satisfied. Although, conceptually, weaving is a run-time activity, most weavers transform the source code or bytecode at build-time. This is exactly where the CSB kicks in, as researchers and practitioners experience CSB problems with the implicit scope of weaving (1), consistent weaving across components (2), reliable speed-up of the weaving process (3) and fine-grained weaver configuration (4) [1, 9, 15].

To investigate CSB in the presence of AOSD technology, we designed a build-aware aspect language for C systems named "Aspicere" and implemented a source code and a link-time bytecode weaver for it. As Aspicere pointcuts are Prolog rules, pointcuts can be expressed in terms of the build system configuration and structure by representing build system concepts like "library", "component" and "selected feature" as logic facts. Build-aware pointcuts are intended to narrow the gap between composition in the source code (weaving) and in the build system (file manipulation).

We performed five case studies in which Aspicere aspects were woven into a C system. In two of them, tracing aspects were woven to understand the architecture of an industrial C system (269 makefiles) [15] and of Quake 3 (6 makefiles). A third case study proposed an aspect-based exception handling scheme [4], while a fourth one re-engineered the architecture of a small VM (1 makefile) [8]. A fifth case study refactored conditional compilation in the Parrot VM (60 makefiles) into advice [1]. MAKAO proved to be invaluable for understanding the case studies' build systems and for integrating Aspicere's weavers. Aspicere's build-aware advice makes aspects more robust to changes in the build system, and hence to CSB.

3.4. Conceptual Reasons of Co-evolution

Based on the experimental findings in the Linux and AOSD case studies, we hypothesize four important reasons why source code and the build system co-evolve:

1) *Modular source code needs a modular build system.*

The kernel's error-prone "recursive make" build architecture continually has been re-engineered to improve the seamless integration of new drivers. Aspect developers assume that the whole system is the scope of weaving ("whole-program reasoning"), but because of the complex interaction of binaries and libraries the build system cannot guarantee this. This mismatch between aspect and build modules causes important problems.

2) *The build system is an executable specification of the system architecture.*

A build system is a meta-program that manipulates and composes files according to the source code architecture. The Linux kernel build engineers have experimented with dozens of algorithms and tools to synchronize the build dependencies with the actual source

code architecture. Things are even worse for aspect languages, as the build system does not understand the advice construct's inversion of dependencies.

3) *Correctness trumps efficiency.*

The kernel build engineers are torn between deliberately omitting source code dependencies to improve build speed and adding more dependencies to ensure a correct build. For aspect languages, the issues with whole-program reasoning and aspect interaction complicate speeding up the weaving process in a reliable way.

4) *A build system's configuration layer controls the static variability of source code.*

The kernel's extreme source code variability has required multiple configuration languages and powerful graphical configuration utilities. The higher potential for source code variability offered by aspects, aspect interaction and the impact of the weaving order of aspects put even higher demands on the configuration layer.

The findings of our case studies provide initial proof for these four conceptual reasons of CSB, and we believe future research on build systems should focus on them.

4. Conclusions and Contributions

Our dissertation investigates the build system and its peculiar co-evolution relation with source code (CSB). Build systems are very complex and hence very hard to understand and manipulate. However, if the build system does not evolve synchronously with the source code, the build system fails to generate a fully-functional software system, which affects most stakeholders in the software development process. These are our major contributions:

1. MAKAO framework to study build systems and CSB.
2. Experimental validation of MAKAO on the evolution of the Linux kernel build system, yielding proof of problems related to CSB.
3. Build-aware aspect language support to manage CSB.
4. Five case studies in which MAKAO supports the introduction of AOSD technology in legacy systems, yielding proof of problems related to CSB.
5. Four conceptual reasons for CSB.

An opaque build system forms a potential risk for software development, but simple tool and language support is able to foster understanding of a build system and its relation to the source code.

Acknowledgments. We would like to thank the anonymous reviewer and Nicolas Bettenburg for their comments.

References

[1] B. Adams, W. De Meuter, H. Tromp, and A. E. Hassan. Can we refactor conditional compilation into aspects? In *Proc. of the 8th ACM Int. conference on Aspect-Oriented Software Development (AOSD)*, pages 243–254, New York, NY, USA, 2009. ACM.

[2] B. Adams, K. De Schutter, H. Tromp, and W. De Meuter. Design recovery and maintenance of build systems. In L. Tahvildari and G. Canfora, editors, *Proc. of the 23rd Int. Conference on Software Maintenance (ICSM)*, pages 114–123, Paris, France, October 2007. IEEE Computer Society.

[3] B. Adams, K. De Schutter, H. Tromp, and W. De Meuter. The evolution of the Linux build system. *Electronic Communications of the ECEASST*, 8, February 2008.

[4] B. Adams and K. D. Schutter. An aspect for idiom-based exception handling. In *Proc. of the 5th Software-Engineering Properties of Languages and Aspect Technologies Workshop (SPLAT), AOSD*, Vancouver, Canada, March 2007.

[5] M. de Jonge. Build-level components. *IEEE Trans. Softw. Eng.*, 31(7):588–600, 2005.

[6] J.-M. Favre. Meta-model and model co-evolution within the 3D software space. In *Proc. of the Int. Workshop on Evolution of Large-scale Industrial Software Applications, ICSM*, Amsterdam, The Netherlands, September 2003.

[7] S. I. Feldman. Make - a program for maintaining computer programs. *Software - Practice and Experience*, 1979.

[8] M. Haupt, B. Adams, S. Timbermont, C. Gibbs, Y. Coady, and R. Hirschfeld. Disentangling virtual machine architecture. *IET Software: Special Issue on Domain-specific Aspect Languages*, 3(3):201–218, June 2009.

[9] A. Kellens, K. D. Schutter, T. D'Hondt, V. Jonckers, and H. Doggen. Experiences in modularizing business rules into aspects. In *Proc. of the 24th IEEE Int. Conference on Software Maintenance (ICSM)*, pages 448–451. IEEE, 2008.

[10] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proc. of the 11th European Conference on Object-Oriented Programming (ECOOP)*, volume 1241, pages 220–242, Jyväskylä, Finland, 1997. Springer-Verlag.

[11] K. Mens and T. Tourwé. Evolution issues in aspect-oriented programming. In T. Mens and S. Demeyer, editors, *Software evolution*, chapter 9, pages 203–232. Springer Verlag, 1st edition, February 2008.

[12] A. Neundorf. Why the KDE project switched to CMake – and how. https://lwn.net/Articles/188693/, June 2006.

[13] G. Robles. *Software Engineering Research on Libre Software: Data Sources, Methodologies and Results*. PhD thesis, Universidad Rey Juan Carlos, February 2006.

[14] Q. Tu and M. W. Godfrey. The build-time software architecture view. In *Proc. of the 17th Int. Conference on Software Maintenance (ICSM)*, pages 398–407, Florence, Italy, 2001.

[15] A. Zaidman, S. Demeyer, B. Adams, K. De Schutter, G. Hoffman, and B. De Ruyck. Regaining lost knowledge through dynamic analysis and aspect orientation. In *Proc. of the 10th Conference on Software Maintenance and Reengineering (CSMR)*, pages 91–102, Bari, Italy, March 2006. IEEE Computer Society.