Studying the Evolution of Build Systems

by

Shane McIntosh

A thesis submitted to the
School of Computing
in conformity with the requirements for
the degree of Master of Science

Queen's University
Kingston, Ontario, Canada
January 2011

Abstract

As a software project ages, its source code is improved by refining existing features, adding new ones, and fixing bugs. Software developers can attest that such changes often require accompanying changes to the infrastructure that converts source code into executable software packages, i.e., the build system. Intuition suggests that these build system changes slow down development progress by diverting developer focus away from making improvements to the source code.

While source code evolution and maintenance is studied extensively, there is little work that focuses on the build system. In this thesis, we empirically study the static and dynamic evolution of build system complexity in proprietary and open source projects. To help counter potential bias of the study, 13 projects with different sizes, domains, build technologies, and release strategies were selected for examination, including Eclipse, Linux, Mozilla, and JBoss.

We find that: (1) similar to Lehman's first law of software evolution, Java build system specifications tend to grow unless explicit effort is invested into restructuring them, (2) the build system accounts for up to 31% of the code files in a project, and (3) up to 27% of source code related development tasks require build maintenance. Project managers should include build maintenance effort of this magnitude in their project planning and budgeting estimations.

Co-authorship

Earlier versions of the work in this thesis were published as listed below:

1) The Evolution of ANT Build Systems (Chapter 4)

Shane McIntosh, Bram Adams, and Ahmed E. Hassan. In Proceedings of the 7th IEEE Working Conference on Mining Software Repositories (MSR), pages 42–51, Cape Town, South Africa, 2010. IEEE Computer Society Press. (Acceptance ratio: 16/51 = 31%, *Invited for Special Issue*).

My contribution – Drafting the research plan, gathering and analyzing the data, and drafting manuscripts.

2) The Evolution of Build Systems for Java Projects (Chapter 4)

Shane McIntosh, Bram Adams, and Ahmed E. Hassan. Under review for the Journal of Empirical Software Engineering, Special Issue on Mining Software Repositories. Springer Press. (Invited extension of "The Evolution of ANT Build Systems", Impact factor: 1.612 [1]).

My contribution – Drafting the research plan, expanding upon our collection of gathered data, analyzing the data, and drafting manuscripts.

[1] Based on 2009 Journal Citation Report®, Thomson Reuters

3) An Empirical Study of Build Maintenance Effort (Chapter 5 and 6)

Shane McIntosh, Bram Adams, Thanh H. D. Nguyen, Yasutaka Kamei, and Ahmed E. Hassan. To appear in Proceedings of the 33rd International Conference on Software Engineering (ICSE), Honolulu, Hawaii, USA, 2011. ACM Press. (Acceptance ratio: 62/441 = 14%).

My contribution – Drafting the research plan, expanding upon an existing collection of gathered data, analyzing the data, and drafting manuscripts.

Acknowledgments

With the utmost respect, I would like to thank my co-supervisors, Dr. Ahmed E. Hassan and Dr. Bram Adams. You have each left an indelible mark on my life, and for that I am humbled and eternally grateful. Ahmed, you have motivated not only to set big goals, but to put into motion a plan of action to achieve them. Bram, your enthusiasm, talent, and dedication are truly awe-inspiring.

I would also like to thank my colleagues, at the Software Analysis and Intelligence Lab (SAIL). You have each become personal role models of mine, exemplifying the type of strong work ethic and commitment to quality that I can only hope to emulate.

My sincere thanks to my thesis examiners, Dr. G. Scott Knight of the Royal Military College of Canada and Dr. James R. Cordy of Queen's University, for their fruitful suggestions.

I would like to dedicate this work to my family and friends. Without your support, this thesis would not have been possible. Also, to Victoria, for your patience, understanding, and love, I am forever grateful.

Statement of Originality

I, Shane McIntosh, hereby declare that I am the sole author of this thesis. All ideas and inventions attributed to others have been properly referenced. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners. I understand that my thesis may be made electronically available to the public.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

> *[...] there are few things more important to a programmer's work flow and therefore productivity than how their system is built* [50].
>
> George V. Neville-Neal

As a software project ages, its source code is changed continuously to address rapidly changing environments and new user demands [33]. Each time the source code has been changed, new deliverable artifacts that reflect the latest changes must be produced in order to test the actual software.

The build system automates the process of producing deliverable artifacts rapidly, correctly, and across all supported platforms, with the aim of simplifying the lives of all software development stakeholders. For instance, software developers use the build system to produce installable packages and test their changes after completing a source code modification. Software testers rely on the build system to execute automated tests that report when the deliverables no longer produce expected output, i.e., regress. Further, project managers use the build system to generate releases of

the software system for delivery to customers.

Although the build system plays such a pivotal role in the lives of software development stakeholders, it is one of many artifacts that software engineering researchers tend to overlook [57]. This is unfortunate since the build system can have a dramatic impact on a software project, as the following evidence suggests:

1) U.S. Department of Energy (DoE)

By performing a developer survey, Kumfert *et al.* estimate that the build induces a 12% mean overhead on the development process [32]. That is, 12% of a developer's time is spent maintaining the build system rather than implementing features, fixing bugs, or restructuring source code.

2) The Linux Kernel

The Linux build engineers spent many years, and numerous releases, evolving the core build machinery of the Linux kernel, which contains over 15,000 lines of build code, to make integration of new code easy for contributors [2].

3) KDE

The KDE 3 project's build system was such a burden to maintain that it limited the productivity of KDE developers, and even warranted migration to newer build technologies. The SCons and CMake build technologies were briefly experimented with until CMake was established as the supported build technology. The migration efforts themselves required a substantial investment of developer time and effort, as build infrastructure was reimplemented entirely using the new technologies, and had to be introduced incrementally to avoid disruption to development progress [48].

The maintenance of the build system seems to be a real nuisance for developers, i.e., build maintenance diverts developer focus away from the main tasks of fixing bugs, improving existing features, and adding new features. A brief survey of the developer mailing lists of Tomcat shows that developers often need help understanding their build systems [54, 58]. Contributors frequently vent their frustrations about difficulties executing the build [30, 49]. To illustrate the frustrating role that the build system plays in the lives of developers, we manually analyzed the bug repositories of Mozilla and ArgoUML by examining defect records that relate to build system issues, and found the following examples of build-related frustrations:

1) Disruption of development progress

While working on a defect, one Mozilla developer removed an obsolete part of the build code and tested his changes locally without any issue [42]. However, when another developer merged the build changes with his working copy, he could no longer run the build because of subtle differences between his build environment and the first developer's. The second developer (and conceivably many others) could no longer build or test changes locally until the build defect was resolved.

2) Inappropriate feedback

On the ArgoUML continuous integration server [52, 62], which regularly updates a local copy of the ArgoUML source code to execute all automated tests, the build process failed to complete [49]. However, developers could not determine which change was the one that broke the build. Developer time and effort was then invested to determine which change is responsible and what corrective action was needed. This investigation diverts the developer's attention away from the core tasks of fixing bugs and adding new features.

3) Impact on product quality

In Firefox 3.0, there was a "show-stopper" defect [59] that prevented users in a networked environment from accessing a web page via the address and search bar, the core feature of any web browser. The fix for the issue was not delivered for four months. Users in this networked environment could not use the Firefox 3.0 product until the release of the first service pack, i.e., 3.0.1. It turned out that a change to the Firefox build system was including an incorrect version of the SQLite library, which caused inconsistencies in the delivered packages.

Despite the crucial role of build systems and their non-trivial maintenance effort, software engineering research rarely focuses on them [57]. Initial findings show that the size and complexity of `make`-based build systems grow over time [2, 66]. To the best of our knowledge, no study has focused on the evolution of the size and complexity of non-`make` build systems, nor on the impact that build maintenance has on software development. Without a strong understanding of how build systems evolve, resources cannot be properly allocated, and software releases may be delivered late and over-budget. Similarly, since feature-rich build technologies such as ANT [5], Maven [6], and CMake [36] are gaining momentum, projects are beginning to migrate away from `make` [17, 26, 35, 48]. During such a migration, the development team must throw away thousands of lines of build code and reimplement the build logic using the selected technology, requiring a large investment of development time and effort, and risking the disruption of development progress.

To better understand the impact that the build system has on software maintenance and evolution, this thesis studies the evolution of software build systems as standalone entities and with respect to the source code that the build system breathes

life into. With a better understanding of build system evolution, development stakeholders can better estimate software maintenance effort. For instance, (1) developers and testers could more accurately estimate the time and effort required for a development task with build maintenance effort included, and (2) project managers could include a more informed estimation of maintenance effort in their project planning and budget.

1.1 Research Statement

Prior research, experience maintaining large software projects, and interaction with industrial software developers lead to us to the following hypothesis:

Build system maintenance plays an important role in software development.

Motivation – Prior work reports that the effort required to maintain the build system is so great that a dedicated team of build experts is required, e.g., in the Perl interpreter [61], and the Linux kernel [2]. Furthermore, Kumfert *et al.* estimate that developers spend 12% (median) of their time maintaining the build system rather than performing core development tasks [32]. Tu *et al.* report that build systems for projects that support many platforms are difficult for people to interpret [61]. We are interested in studying the evolution of the build system to determine its impact on different stakeholders in the software development process.

1.2 Thesis Overview

This thesis studies the evolution of build systems across the lifetime of 13 open source projects and one commercial project. We perform three analyses to validate our research hypothesis:

1) High-level evolution analysis of Java build systems (Chapter 4)

To assess whether build maintenance is impacting Java developers, we study the evolution of Java build systems to complement earlier release-level studies that find that `make`-based build systems of C projects evolve.

2) Fine-grained evolution analysis (Chapter 5)

By analyzing each source code revision committed to Version Control System (VCS) repositories of ten software projects, we study the scale of the build system and its maintenance.

3) Maintenance overhead analysis (Chapter 6)

We study each committed developer revision and groups of related revisions recorded in the VCSs of ten software projects to measure the impact that build system maintenance has on the development process.

1.3 Major Thesis Contributions

This thesis presents support for the research hypothesis as listed below:

Results – As demonstrated by the Mozilla project, managers for C projects should anticipate that 27% of development tasks that involve source code changes will require accompanying build system changes. Managers for Java projects should anticipate

that build maintenance is required for 4–16% of the source code tasks. We observed two build maintenance ownership styles: (1) *Centralized build ownership*, where most build system changes are performed by a small team of build experts, or (2) *Distributed build ownership*, where build changes are rather evenly dispersed amongst the team of developers.

1.4 Organization of Thesis

The remainder of the thesis is organized as follows: Chapter 2 provides a brief background on the build system, its role in software development, and a comparison of common build technologies. Chapter 3 presents research related to our analysis of the evolution of build systems.

In Chapter 4, we present a coarse-grained study of Java build system evolution at the *release-level*, i.e., development releases are considered data points. In the chapter, we study six open source Java projects to verify that build system evolution also exists in non-`make` build systems, complementing prior work on `make`-based build systems [2, 66].

In Chapter 5, we present a fine-grained study of build system evolution in `make` and non-`make` build systems at the *revision-level*, i.e., each development change is considered a data point.

In Chapter 6, we study how successful projects cope with the overhead of maintaining the build system from task-centric and developer-centric perspectives.

Finally, Chapter 7 concludes the thesis, and discusses the limitations and potential directions for future work in this area.

Chapter 2

Background and Definitions

> *If someone can provide a link to something like "Make for Dummies" I'd appreciate it. Having constructed abstract Turing Machines to do arithmetic, read and written technical papers, I'd like to think the problem [of understanding the build system] is more than brain-death on my part.*
>
> Anonymous developer

Before we analyze the evolution of build systems, we first provide an overview of build systems themselves. The following questions are addressed in this chapter:

2.1) What is a build system?

We define the concept of a build system and discuss its role in the software development process.

2.2) What is the typical architecture of a build system?

Based on prior research and personal experiences with build systems, we propose

a reference architecture for build systems. The reference architecture breaks the task of building software down into four layers.

2.3) What are the typical build system languages?

We introduce the popular `make`, ANT, and Maven build system languages, which are used by the studied projects.

2.1 What is a build system?

For the purposes of this thesis, we define the build system as:

> *The infrastructure responsible for transforming development artifacts such as source code, into a deliverable format that is ready for testing and subsequent release.*

Build systems play an important role in software development, since they interact with many software development stakeholders [50]:

1) Developers

Developers use the build system on a daily basis to test a software system after adding a new feature or creating a (potential) bug fix.

2) Testers

Testers weave automated unit and integration tests into the build process in order to quickly detect regression, i.e., incorrect behaviour that was correct in prior versions of the software.

3) Managers

Managers use the build system to generate releases for distribution to users.

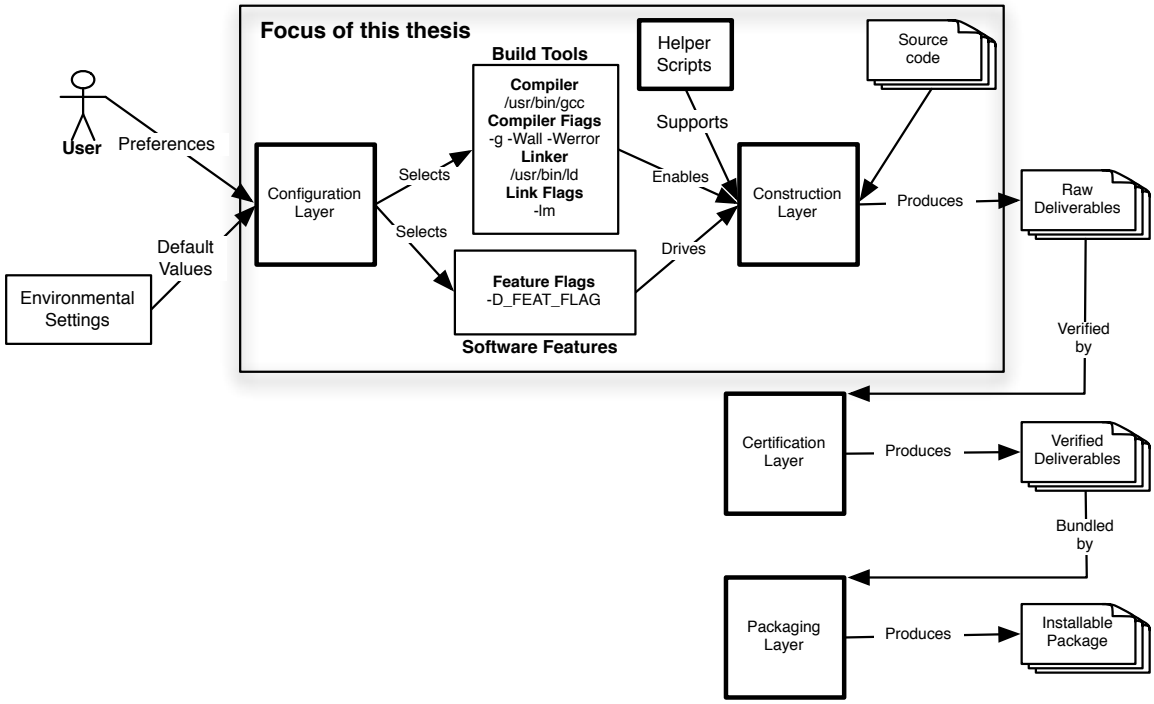

Figure 2.1: Conceptual architecture of a typical build system.

2.2 What is the typical architecture of a build system?

Figure 2.1 shows our conceptual architecture of a build system consisting of four layers. Each layer is derived from prior research. We describe each layer below.

2.2.1 Configuration Layer

The *configuration layer* allows build system users to select the code features that should be included in the final product, and the build tools to use during the build process [1, 9, 60]. A configuration tool identifies which build tools are needed during the build and checks whether the configuration of software features selected by the

user is valid. These requirements and constraints of build tools and software features are derived from specifications written in a configuration language, such as KConfig or CDL [9].

2.2.2 Construction Layer

After a configuration of build tools and software features has been ratified by the configuration layer, the *construction layer* executes the commands necessary to produce the deliverables [1]. A construction tool parses the build specification files to determine the necessary commands and the order in which they must be executed in order to produce the final product correctly. Construction layer (or build) specifications are typically expressed in terms of a build system language. Among build system languages, popular choices include `make` (i.e., GNU make, BSD make, etc.) and ANT. However, new languages such as Maven and CMake are gaining popularity [17, 48].

Build specification files conceptually specify build targets. A build target represents an abstract build goal (or collection of goals) T such as "complete all compilation commands". A target T typically has two key characteristics, (1) a build rule that defines the sequence of commands that must be executed when T is triggered, and (2) a list of dependent targets that determine whether or not T should be triggered. Heuristics are used to speed up a build such that a target is only triggered if its output files do not exist yet, its output files are older than its input files, or at least one dependent target has been rebuilt.

The construction layer can have complex architectures [1, 61]. Our research reveals that in addition to code implemented directly in build specification files, many build systems also depend upon a layer of build-related helper scripts such as Perl and Bash

scripts. These scripts implement common build rule logic and reduce repetition in build specification files.

2.2.3 Certification Layer

After constructing the deliverables, the build system drives the deliverables through a sequence of automated tests [28]. We refer to the scripts and build logic necessary for this phase of the build process as the *certification layer*. The build system drives the execution of these tests using the certification layer. When the regression tests fail, the certification layer reports issues to the developers. The certification layer ensures that when software regresses, reports are produced promptly so that developers may address the issues early in the development cycle.

2.2.4 Packaging Layer

The final step in the build process is to produce an installable package [16]. The *packaging layer* gathers the constructed deliverables, third-party redeliverable libraries, and associated product documentation and bundles them into an installation bundle format, such as Microsoft Installer (.msi) for Windows users or RedHat Package Management (.rpm) format for RedHat-like "flavours" of GNU/Linux.

2.3 What are the typical build system languages?

This section presents a brief history and the major features of the popular `make`, ANT, and Maven build system languages. Table 2.1 shows which reference architecture layers are typically implemented by which build system languages that we will

Table 2.1: Build technologies and their appropriate build layers.

	Configuration	Construction	Certification	Packaging
`make`		✓		
ANT	✓	✓		
Maven	✓	✓	✓	✓

Figure 2.2: Example Makefile target expression

```
1 main.o : main.c message.h
2         $(CC) -c main.c
```

consider in this thesis. The languages may be used to implement layers that are not indicated in Table 2.1, however, they are not designed for them.

2.3.1 Make

The `make` build language first appeared in literature in 1979. Feldman declares that `make` is "a program for maintaining computer programs" [18]. More accurately, `make` can be used to automatically synchronize program source code with its deliverables. If a properly constructed `make` build system is deployed, the `make` command will execute only the necessary commands to create an up-to-date deliverable. In this sense, `make` was revolutionary. Before `make`, there existed no specialized language for implementing build systems.

Figure 2.2 shows an example of a `make` target expression. Line 1 describes the dependency relationship between the target file "main.o" and its dependencies "main.c" and "message.h". Line 2 defines the rule that will (re)generate the target, i.e., compile "main.c" using the compiler assigned to the $(CC) `make` variable.

Target expressions are listed in a `Makefile`. When the `make` command is executed,

it automatically attempts to bring the first target listed in the `Makefile` up-to-date. A target is considered up-to-date when its output has a newer modification time than its dependencies. If this is not true, the `make` command will execute the specified rule. If the target is up-to-date when the `make` command is executed, `make` will skip the execution of that rule, since it is not necessary. In this sense, `make` builds are incremental and will only execute the minimal set of rules necessary to bring the output in sync with its dependencies.

`Makefile`s list dependencies among files. However, it is sometimes useful to collect conceptual dependencies between targets that do not correspond to files. For such cases, `make` provides "phony" targets, i.e., virtual targets that do not correspond to files.

The dependencies among targets in a `Makefile` together form a build graph. This is a Directed Acyclic Graph (DAG) that represents the dependency model of the build system. The DAG can differ between build executions depending on the targets that are out-of-date, or differences in the selected tools and configurable features.

2.3.2 ANT

ANT, an acronym for Another Neat Tool, was created by James Duncan Davidson in 1999. He was fed up with some of the inconsistencies in the `make` build language [47], which was and still is the de facto standard among build system languages for C and C++ projects. Although `make` pioneered many build system concepts, there are serious flaws in its design, such as the inherent platform dependence of commands in build rules and the common recursive architecture found in many `make`-based build systems [43]. To resolve these flaws, ANT was designed to be small, extensible, and

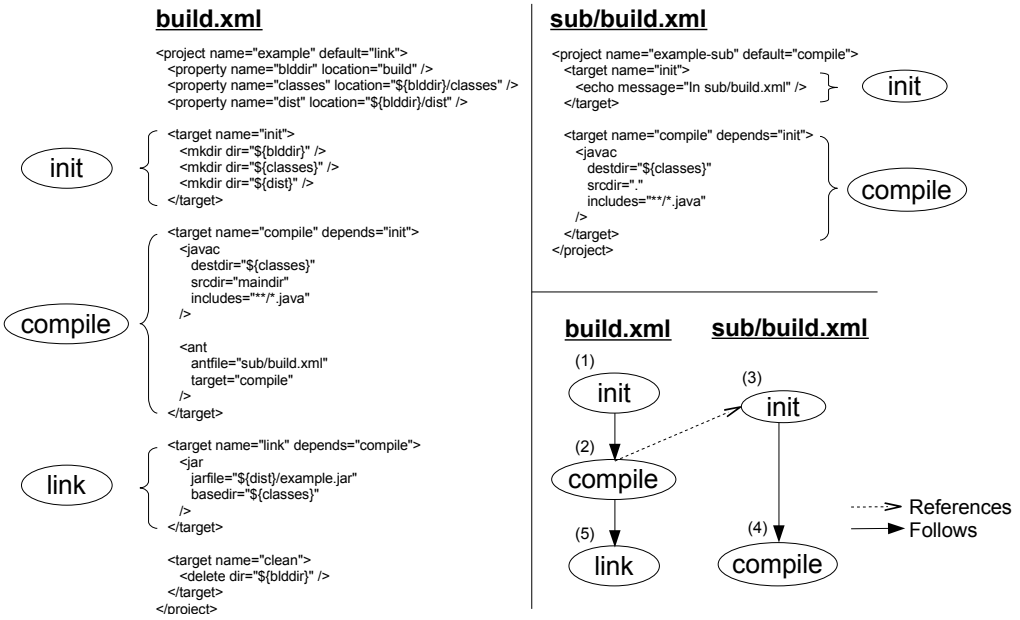

Figure 2.3: Example ANT build.xml files (left, top-right) and the resulting build graph (bottom-right). The build graph has a depth of 2 (i.e., "compile" in build.xml references "init" in sub/build.xml) and a length of 5 (i.e., execute (1), (2), (3), (4), then (5)).

operating system independent. Still, many of the concepts introduced by `make` survive in ANT. An example ANT specification file and the resulting build graph is shown in Figure 2.3.

An ANT build system is specified by a collection of XML files. <**project**> tags contain all of the build code related to a software project. In ANT, a target does not correspond to a file, but to a sequence of related conceptual tasks (c.f., phony targets in `make`), such as "compile all source files" ("compile" <target> in Figure 2.3) or "collect all class files in a jar archive" ("link" <target> in Figure 2.3). <**task**> tags represent atomic commands inside a build <target>'s build rule. A task may "create a directory" ("mkdir" tasks in the "init" target of build.xml) or "run the compiler on the given set of source files" ("javac" task in the "compile" target of either XML

file). The <task> is the most fine-grained element in an ANT build specification file.

The ANT build language comes stocked with a library of common build <task>s. If a <task> implementation does not exist, ANT provides an Application Programmer Interface (API) for developing expansion tasks. The Task API, like the ANT parser itself, is implemented for the Java SE platform.

Targets may "depend" on one another. Using these dependencies, a graph may be constructed. We only provide an example of such a graph using an ANT script, however this same concept may be applied to targets in many build languages.

Consider the build graph shown in the bottom-right section of Figure 2.3. In this example, ANT has been instructed to execute the "link" target, yet its dependencies must be satisfied first. The "link" target depends on the "compile" target, which in turn depends on the "init" target.

Targets may also depend upon targets in other build files. Build developers often leverage this feature to preserve build system modularity. As an example of a depth dependency, the "compile" target (via its <ant> task) depends on another "compile" target in a different specification file (i.e., sub/build.xml). The "compile" target may depend upon "compile" targets in subdirectories of the "sub" directory producing a chain of depth dependencies.

The build graph shown in Figure 2.3 is said to have a length of five since five targets were triggered, and a depth of two since two was the maximum depth encountered in the graph.

2.3.3 Maven

Maven was created with build process standardization in mind, since many Java projects of the Apache foundation shared similar ANT scripts [6]. The Maven build process is called the *Build Lifecycle*. A lifecycle is composed of one or more sequential phases. For example, a simple lifecycle may contain (1) a "compile" phase where source code is compiled into bytecode, followed by (2) a "package" phase where bytecode is bundled into a deliverable format.

Each build phase may have zero or more sequential goals bound to it. A phase without any goals bound to it is skipped during the build, however those with one or more goals bound execute these goals during the build. For example, the "compile" phase may (1) enforce a coding style standard by binding an "enforce" goal to it, then (2) compile the source code by binding a "compile" goal to it. Each goal is implemented in a Maven plugin.

The build lifecycle is composed of 23 phases. A subset of the 23 phases is bound to default goals depending on the deliverable package type. The studied Maven projects use JAR packaging, and as such, use the JAR package phase-to-goal bindings outlined in Table 2.2.

Additional goals may be bound to lifecycle phases by configuring additional Maven plugins in build specification files. For example, integration testing may be executed during the build process by loading an appropriate plugin and binding an integration testing goal to the `integration-test` lifecycle phase.

Maven prescribes to the design principle of "convention over configuration". A project that is built with Maven can minimize the amount of build code necessary by conforming to the Maven convention. Deviation from the Maven convention requires

Table 2.2: The Maven default lifecycle for JAR packages.

Phase	Description
`process-resources`	Pre-process the resource files.
`compile`	Compile the source code.
`process-test-resources`	Pre-process the test resource files.
`test-compile`	Compile the test code.
`test`	Execute the *unit* tests.
`package`	Package the compiled code into the deliverable format.
`install`	Install the deliverables in the local Maven repository.
`deploy`	Upload the installed deliverables to a remote repository.

additional build code.

In addition to build process standardization through the build lifecycle, Maven also features automatic management of third party libraries. Java projects often struggle with managing these external dependencies, often opting to either (1) commit the exact versions of the libraries into the project's VCS, or (2) download them automatically using hard coded ANT targets. Maven provides support for specifying required versions and sharing them in a local cache repository for use in all Maven-built projects.

2.4 Chapter Summary

This chapter lays the foundation necessary for the remainder of the thesis. We first defined the build system and discussed its impact on various stakeholders in the software development process. Next, based on prior research, we established a reference architecture for build systems. Finally, we introduced the three build technologies that are used to implement build systems in the studied projects.

In the next chapter, we discuss the prior research that has been performed involving the build system.

Chapter 3

Related Research

The study of history is a powerful antidote to contemporary arrogance. It is humbling to discover how many of our glib assumptions, which seem to us novel and plausible, have been tested before, not once but many times and in innumerable guises; and discovered to be, at great human cost, wholly false.

Paul Johnson

In this chapter, we present a survey of prior build system research. In performing the survey, we identified two common research topics:

1) Build system design

2) Build system evolution

The remainder of the chapter discusses the prior research with respect to these two topics.

3.1 Build System Design

Build system design is critical. Just as design and architecture bugs are more expensive than implementation bugs in source code [38], the same is true for build systems. Neglecting to carefully design the build system may result in a highly complex build system implementation that is difficult to maintain and requires a considerable investment of effort to restructure.

3.1.1 Analysis of Design

De Jonge showed that the lack of modularity in build systems limits the reusability of software components, since components cannot be built independently [13, 14]. To remedy this, de Jonge proposes the use of Component-Based Software Engineering (CBSE) principles in the design of build systems, i.e., build systems should consist of build components that communicate through a standard public interface. Such a design allows the build component implementation to vary independently of the build interface. A large-scale case study on Mozilla [44] shows that CBSE-based build systems can improve the reusability of software components.

Adams *et al.* present MAKAO (Makefile Architecture Kernel featuring Aspect Orientation), a design recovery tool for `make`-based build systems. MAKAO visualizes build targets executed during an execution of a `make` build. MAKAO is useful for restructuring, since it models concrete executions of the build process. The produced model can be queried and filtered, allowing users to navigate and customize the build visualization. Case studies of open and proprietary systems, such as Linux and the Kava system, show that `make`-based build systems can be very complex.

De Jonge and Adams' findings show that build systems for large projects have complex and non-trivial designs and implementations. Intuition suggests that these complex build system designs require careful maintenance, which would induce an overhead on the development process (c.f., our research hypothesis).

3.1.2 Patterns and Pitfalls

Software designers use design patterns [21] and avoid anti-patterns [19] to improve the design of software. The following studies investigate similar patterns in build system design.

Miller presents a study of `make` build systems implemented using the common "recursive `make`" paradigm and argues that it should be considered an anti-pattern [43]. Traditionally, UNIX software projects use the recursive `make` technique to transform source code into deliverables. As these projects age, their recursive `make` systems begin to exhibit symptoms such as slow performance, incomplete or highly redundant builds, and build sensitivity to irrelevant changes in the source code. Miller attributes these symptoms to the recursive `make` technique itself when used in an unbounded fashion.

Conversely, Tu *et al.* [61] discuss the *code robot*, a build system design pattern with a positive effect on a build system structure. A code robot is a program that is built during an initial phase in the build process and used in later phases to generate platform-specific code from code templates. Use of the code robot pattern allows development teams to avoid having to ship and maintain dozens of platform-specific files in the project VCS. The GCC [23] and Perl [53] projects provide examples of the code robot. In GCC, the code robot comprises a smaller version of GCC used

to build the remainder of the compiler collection. Similarly, in Perl, the code robot is used to build a small Perl interpreter that interprets Perl scripts containing build tasks.

Adherence to patterns and avoidance of anti-patterns further suggests that the build system design process is non-trivial and imposes an implicit overhead on the development process (c.f., our research hypothesis).

3.1.3 Build Performance

The build system is an important part of software development and maintenance. After completing a code fix, developers must run the build process in order to test the changes. While the build process is executing, the developer must wait. If the build process is slow, this idle period is extended, frustrating developers and slowing development progress. The time consumed by the build process is a perceived form of build system overhead. Hence, we survey related work in the field of build process acceleration.

There is much work on build acceleration strategies. Build tools such as `make`, ANT, and Maven operate incrementally by examining the last modification time of target output and input, executing only the smallest list of commands necessary to bring the deliverable artifacts up-to-date. The SCons [31] build tool uses checksums of each file to avoid performing link commands when only code comments have been modified. Adams *et al.* empirically evaluate strategies for accelerating the build process [3]. Yu *et al.* improve both incremental and fresh build speed by automatically removing unnecessary dependencies between files [65] and redundant code from C header files [64].

Improving build performance is a means of mitigating the considerable impact that the build system imposes on development stakeholders (c.f., our research hypothesis).

3.2 Build System Evolution

Lehman *et al.* present observations of program evolution in their body of work, commonly referred to as "Lehman's Laws" [8, 33, 34]. In this thesis, we focus on the first two laws:

1) A program evolves due to changes in its environment, e.g., platform upgrades, new customer requirements.

2) Due to the modifications induced by law 1, the program grows in complexity.

Lehman *et al.* recorded these observations by analyzing large, proprietary software systems as they aged during the 1970's and 1980's. Godfrey *et al.* verified these results for open source systems [24].

Similar environmental factors that cause software systems to evolve (law 1) are also present for build systems. Since transforming source code into a usable artifact is the main goal of the build system, the source code is a part of the environment of a build system. Hence, source code evolution may act as a catalyst to the evolution of build systems.

Furthermore, build systems have two dimensions in which they may evolve: (1) A *static* dimension that evaluates the static characteristics of the build system specification files, and (2) A *dynamic* dimension that evaluates the build system run-time execution properties. The below subsections discuss studies of each dimension.

3.2.1 Build Specification Evolution

Build specification evolution is evaluated by analyzing changes in source code metrics adapted for use with build systems. The use of source code metrics is justified since build specification files share many similarities with source code implemented in an interpreted programming language. Case in point, the SCons build language is entirely based on the Python programming language.

Zadok studied the effect of changing build technology. He examined the static size and complexity of the Berkeley Automounter build system [66]. He found that the build system was growing in size and complexity until a build technology migration from `make` to GNU Autotools. The migration reduced the size and complexity metrics, which encourages other project maintainers to migrate to Autotools. While the paper does not consider the development investment that was required to migrate build technologies, the paper does hint that build systems have a tendency to grow unless disturbed by major project restructuring.

Adams *et al.* studied the static evolution of the Linux kernel build system, which is implemented using `make` [2]. They found that the Linux kernel build system is growing exponentially in terms of the Build Lines of Code (BLOC). Godfrey *et al.* [24] found that the Linux source code also grows exponentially in terms of LOC. Furthermore, the build and source code appear to grow together or shrink together, suggesting that source code and build system *co-evolve*.

In summary, not only do `make`-based build systems appear to evolve [66], but they appear to co-evolve with project source code [2]. That is, changes in the source code often induce changes in the build system, and vice versa. We conjecture that this co-evolution imposes an overhead on the development process (c.f., our research

hypothesis).

3.2.2 Build-time Evolution

Build-time evolution is observed by analyzing the results of build execution. Such an analysis is not representative of the entire build system, but rather a single path of possible build execution. The studies below focus on a single build configuration with a consistent build environment, i.e., the same operating system, versions of third-party libraries, etc. The evolution is observed using metrics intuitive to the data, e.g., the number of executed targets (build length).

Using the MAKAO tool [1], Adams *et al.* study the number of targets and dependencies in executed builds of different versions of the Linux kernel [2]. They find that there was a large structural change from the 2.4 version to the 2.6 version of the Linux kernel. Using the querying capabilities of MAKAO, they identify the root cause of the change in build system structure.

The changes to build system structure required a large investment of developer effort over three years of development of the Linux kernel (c.f., our research hypothesis).

3.3 Chapter Summary

In this chapter, we survey prior work focused on two topics:

3.1) Build system design

Findings – Build system design is critical to the smooth evolution of source

code. A poorly designed build system can limit the reusability of software components [13, 14] and produce incorrect artifacts [43].

3.2) Build system evolution

Findings – `make`-based build systems for C projects evolve [66], and furthermore, appear to co-evolve with the source code at the release-level [2].

This thesis focuses on further expanding upon the impact that the evolution of build systems has on source code development. In the next chapter, we evaluate build systems for Java projects to find out whether prior evolution findings for `make`-based build systems at the release-level [2, 66] apply to Java build systems as well.

Chapter 4

Java Build System Evolution at the Release-level

I suspect one of the reasons that [build systems] mushroom so much is that most people don't understand [them well] enough, so they just take an existing example and hack some more into it (sic).

Anonymous developer

In the prior chapters, we have introduced the build system and related terminology (Chapter 2), as well as surveyed the prior research (Chapter 3). We found that little is known about the evolution of the Java build systems as a project ages. Adams *et al.* made initial findings in the Linux kernel, which suggest that the build system evolves across major releases [2]. Zadok also found evolution-like patterns in the Berkeley automounter build system across major releases [66]. However, this research is limited to `make`-based build systems for C projects and to the release-level (more on this in the next chapter).

Little is known about build specifications for Java projects. Java is a popular programming language that is used both in industry and academia. Similar to C source code, Java source code must be compiled and linked before it can be delivered to end users. Hence, Java projects also require build systems to translate source code files into deliverable bytecode. However, the Java compiler differs from the C compiler in two ways: (1) a single invocation of the Java compiler will automatically resolve dependencies between all of the input source files, while the C compiler must rely on external dependency management through build tools like `make`, and (2) Java compiler invocations are expensive, since the Java Virtual Machine (JVM) must be started before and shut down after each invocation [15]. For these reasons, we conjecture that Java build systems require less maintenance than C build systems, since the compiler performs tasks formerly required of the construction layer.

In order to validate our hypothesis, i.e., build system maintenance plays an important role in software development, we expand upon prior work on `make`-based build systems for C projects [2, 66] by studying the evolution of Java build systems across software releases. An earlier version of this study was published at the 7th IEEE Working Conference on Mining Software Repositories [39]. A more recent extension is under review for a special issue of the journal of Empirical Software Engineering [40]. To focus our analysis, the release-level study addresses two research questions:

RQ1) Do the static size and complexity of source code and build systems evolve similarly?

Motivation – Traditionally, evolution studies measure evolutionary trends in source code metrics to analyze how projects evolve. In our case, we need to study the evolution of specialized complexity measures for build systems. In

addition, we need to analyze how the evolution of build systems is related to that of the source code in order to: (1) validate earlier build system findings for `make`-based build systems, and (2) contrast the evolution of build systems against the evolution of source code.

Findings – The static analysis of build system specifications in this chapter not only shows that Java build systems follow linear or exponential evolution patterns in terms of size and complexity, but also that such patterns are highly correlated with the evolution of the Java source code.

RQ2) Does the build-time complexity evolve?

Motivation – Measuring dynamic properties of a build system provides a complementary perspective of the complexity of a build system. We define dynamic complexity as the amount of build code that a typical build exercises and the time elapsed during a typical build. Intuition suggests that as a build executes more build code and takes longer to complete, it grows in complexity. We investigate whether this complexity exhibits evolutionary trends.

Findings – The dynamic analysis of Java build systems in this chapter does not reveal a common pattern in the studied projects, although we observe linear growth and other interesting trends in build-time length, recursive depth, and build coverage dimensions.

The chapter is organized as follows. Section 4.1 discusses the setup of our case studies on six open source Java projects. Section 4.2 presents the results for our case studies of ANT build systems, and Section 4.3 presents the results of our case studies of Maven build systems. Section 4.4 compares the case studies of ANT and Maven

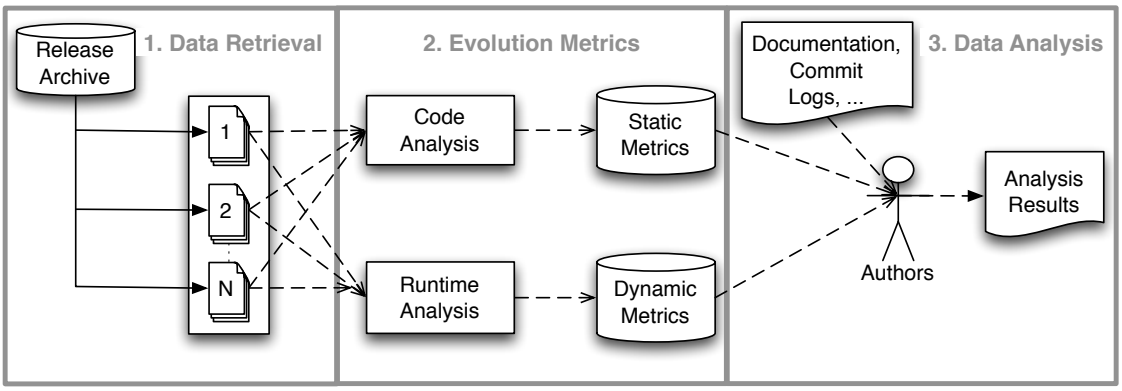

Figure 4.1: Overview of our approach for studying the release-level evolution of Java build systems.

build systems, and furthermore compares our Java case studies to prior work on C build systems. Section 4.5 concludes the chapter.

4.1 Case Study Setup

We track the evolution of software build systems across release snapshots of six open source Java projects. There are some existing metrics from the source code domain that we can study across time, but we also need to define metrics that are customized to the domain of build systems. The focus of these metrics is on the identification of trends related to RQ1 and RQ2. An overview of our approach is shown in Figure 4.1. The remainder of this section discusses each step in our approach:

4.1.1) Data Retrieval

Release snapshots are gathered from project release archives and version control systems (VCS).

Table 4.1: Metrics used in release-level build system analysis

Group	Metric	Description
Static	Build Lines of Code (BLOC)	The number of lines of code in build specification files.
	Target Count	The number of build targets in the build specification files.
	Task Count	The number of tasks in the build specification files.
	File Count	The number of specification files in the build system.
	Halstead Complexity	The quantity of information contained in the build system (Volume), the mental difficulty associated with understanding the build system specification files (Difficulty), and the weighted Difficulty with respect to Volume (Effort).
Dynamic	Build Graph Length	The length of a build graph, either in terms of the total number of executed tasks *or* of the total number of executed targets.
	Build Graph Depth	The depth of a build in terms of the maximum level of depth references made.
	Target Coverage	The percentage of targets in the build system that are exercised by the default or clean targets.
	Dynamic Build Lines of Code (DBLOC)	The percentage of code in the build system that is exercised by the default or clean targets.

4.1.2) Evolution Metrics

A set of metrics are calculated for each release snapshot.

4.1.3) Data Analysis

We analyze the set of releases using the calculated metrics, investigating trends and anomalies.

4.1.1 Data Retrieval

In order to validate our hypothesis, this chapter starts with a course-grained analysis of build system evolution. We consider official software releases of a project to achieve this the level of granularity.

We consider each type of release equal, including major releases that increment the first digit in the version numbering of a project, minor releases that increment the second digit, and service pack releases that increment the third digit. For example, Eclipse release number 3.2.1 is major release number 3, minor release number 2, and service pack 1.

For each project, a collection of source code snapshots were retrieved corresponding to official project releases. These releases were downloaded from the official release archives, except for the ArgoUML and Hibernate data, which was retrieved from the project VCS. The released versions of ArgoUML and Hibernate are easily retrieved, since they are marked with annotated tags in the respective repositories.

4.1.2 Evolution Metrics

In our study, we use various static and dynamic metrics to quantify a wide variety of build system characteristics across the release snapshots. The metrics are summarized in Table 4.1. BLOC, build target/task/file count, and Halstead complexity are gathered statically. Dynamically, build system content is measured with the length and depth dimensions of the build graph. Metrics such as BLOC, file count, DBLOC and the Halstead suite of complexity metrics are inspired by corresponding source code metrics, whereas others, such as target count and task count were used in earlier studies [2]. Build graph depth and target coverage are new metrics proposed by this study.

Most of the metrics are self-explanatory, except for the Halstead complexity metrics, as we had to adapt their definition from source code to build systems. To our knowledge, the notion of such an explicit metric for static build system complexity is new. We use a source code metric to measure the complexity of build files because build specification files share many similarities with source code implemented in an interpreted programming language. With this in mind, we conjecture that build system complexity can be measured using source code complexity metrics on build system description files.

Since establishing a definitive measure of static complexity for build systems is not the focus of this thesis, we only focus on the Halstead suite of complexity metrics [27]. In future work, we plan to examine the McCabe cyclomatic complexity [37] and how it applies to build systems, although results of our case study indicate that (similar to source code [25, 56]), size metrics already provide a good approximation of build complexity metrics.

Halstead Metrics

We now define the Halstead suite of complexity metrics for build system languages. The Halstead complexity metrics measure [27]:

Eq 4.1) Volume

How much information a reader has to absorb in order to understand a program's meaning.

Eq 4.2) Difficulty

How much mental effort a reader must expend to create a program or understand its meaning.

Eq 4.3) Effort

How much mental effort would be required to recreate a program.

Each Halstead metric depends on four tally metrics that are based on programming language characteristics. First, we must tally the number of *operators*, i.e., functions that take input parameters to produce some output. Within the scope of build systems, we consider an operator as any target or task in ANT or any XML tag in Maven.

Next, we must tally the number of *operands* used in the source code. Within the scope of build systems, we consider operands as the parameters passed to a target or task tag in ANT or any child tag in Maven.

Tallies of both the operators/operands that occur at least once (n1 or n2) and the total number of operators/operands (N1 or N2) are collected. The four tallies are described below:

- n1 – The number of distinct operators.
- n2 – The number of distinct operands.
- N1 – The total number of operators.
- N2 – The total number of operands.

These tallies are then used to calculate the Halstead volume, difficulty, and effort as follows:

$$\text{Volume} = (N1 + N2) \times \log_2(n1 + n2) \tag{4.1}$$

$$\text{Difficulty} = \frac{n1}{2} \times \frac{N2}{n2} \tag{4.2}$$

$$\text{Effort} = \text{Difficulty} \times \text{Volume} \tag{4.3}$$

4.1.3 Data Analysis

As suggested by our choice of metrics in Table 4.1, we analyze each release snapshot from two perspectives.

RQ1) Static analysis

Build system files and program source files (including unit tests) of each release are examined statically. We measure the size of the source code (SLOC) so we can compare it against the size of the build system (BLOC). SLOC was measured using David A. Wheeler's `sloccount` utility [63]. We developed a SAX-based Java tool to measure static build metrics such as target count, task count, and the Halstead complexity of build system specification files. Since comment and whitespace lines are discarded by the `sloccount` tool, our BLOC count also discards them using a `sed` script. The surviving lines are tallied using `wc`.

RQ2) Dynamic analysis

The build system of each release was exercised using the default build configuration [2] and the results were logged. The ANT output was exported to an XML log using the built-in ANT XML logger (-logger XmlLogger). The Maven output was exported in text, since Maven does not support XML output. The log of a build embodies the dynamic build graph. To analyze the graph, we extended our Java tool to calculate dynamic metrics such as target coverage, build graph length and depth in terms of both targets and tasks, and the time elapsed during the build.

Historical project documentation such as mailing list archives, release notes, and source code revision comments were consulted in order to investigate our findings for RQ1 and RQ2.

Table 4.2: Java projects studied at the release-level

	ArgoUML	Tomcat	JBoss	Eclipse	Hibernate	Geronimo
Build Tech.	ANT	ANT	ANT	ANT	Maven	Maven
Domain	UML Editor	Web Container	App Server	IDE	ORM	App Server
Max. Source Size (KSLOC)	176	277	731	2,900	328	219
Max. Build System Size (KBLOC)	6	11	29	200	3	30
Timespan	2002-09	1999-09	2002-09	2001-09	2008-10	2006-10
Number of Releases	12	90	25	25	9	11
Shortest Rel. Cycle	53 days	2 days	13 days	32 days	15 days	8 days
Longest Rel. Cycle	593 days	714 days	398 days	176 days	286 days	328 days
Average Rel. Cycle	228 days	95 days	130 days	110 days	91 days	100 days
Release Style	Single	Parallel	Parallel	Single	Parallel	Single

4.2 ANT Case Study

In this section, we present the results of our ANT case study with respect to our two research questions.

4.2.1 Studied Projects

We selected four open source projects built using ANT with different sizes, domains, and release styles. Table 4.2 summarizes the characteristics of the projects.

ArgoUML is a Computer Aided Software Engineering (CASE) tool for producing

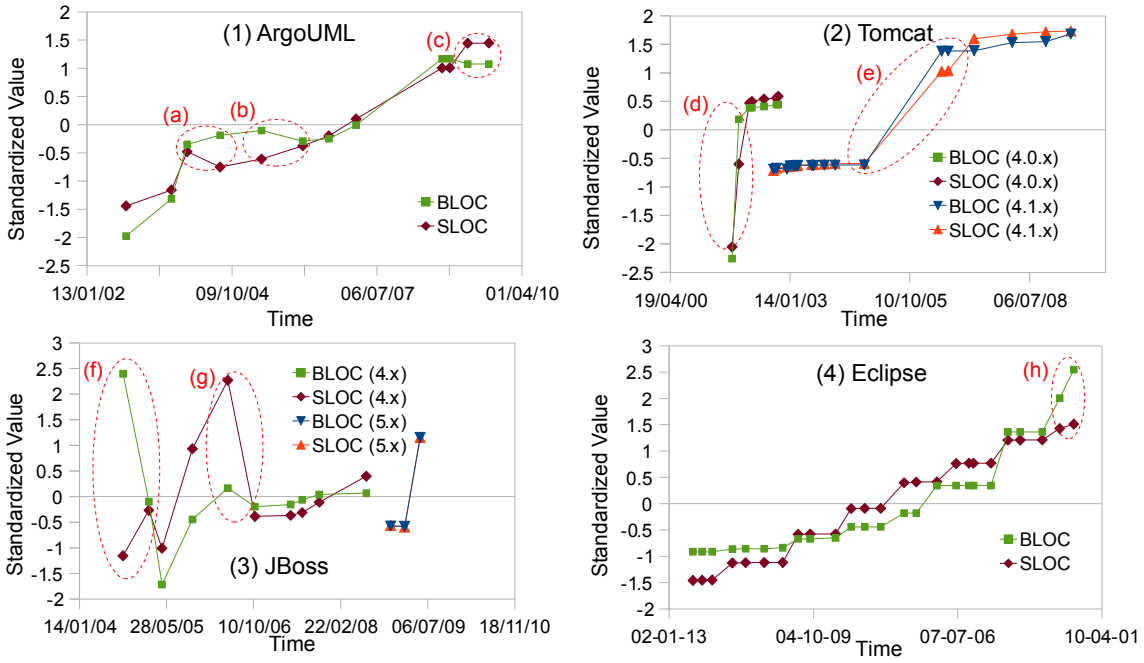

Figure 4.2: Standardized BLOC and SLOC values. In most projects, the source code and build system evolution trends are very similar. Anomalies are discussed in the text.

Unified Modelling Language (UML) diagrams. Tomcat is popular implementation of the Java Servlet and Java Server Pages (JSP) technologies. JBoss is a well-known Java Application Server. Eclipse is a general-purpose Integrated Development Environment (IDE) developed by IBM.

4.2.2 Do the static size and complexity of source code and build systems evolve similarly?

We explored the evolution of ANT build system specification files from three angles. First, we use Figure 4.2 to show a general trend of increasing size in the four projects, then we use Table 4.3 and 4.4 to show that there is a strong correlation between the

Table 4.3: Correlation of static size metrics (ArgoUML, Tomcat, JBoss, and Eclipse). Most size metrics have a high correlation (≥ 0.8). Those that do not are printed in bold.

	Task Count				File Count			
	ArgoUML	Tomcat	JBoss	Eclipse	ArgoUML	Tomcat	JBoss	Eclipse
Target Count	0.99	0.99	0.88	0.97	0.98	0.99	**0.75**	0.98
Task Count					0.95	0.98	**0.64**	0.99

	BLOC				SLOC			
	ArgoUML	Tomcat	JBoss	Eclipse	ArgoUML	Tomcat	JBoss	Eclipse
Target Count	0.98	0.99	**0.40**	0.98	**0.78**	0.97	0.89	0.95
Task Count	1.00	1.00	**0.15**	1.00	0.90	0.97	**0.78**	0.99
File Count	0.94	0.99	**0.59**	0.99	0.88	0.98	0.88	0.98
BLOC					0.89	0.98	**0.40**	0.99

growth in the static size and in the complexity of a build system, and finally we use Figure 4.2 and Table 4.3 again to show that the build system and source code show similar evolutionary trends in terms of size.

ANT Build systems grow in size: In Figure 4.2, we plot the standardized BLOC and SLOC metrics so that we may compare these two metrics in one graph, as SLOC values have a much higher scale than BLOC values (see Table 4.2). This standardization is calculated by weighting each data point in terms of its distance from the average BLOC or SLOC across all releases of a system. The standardization is measured in units of standard deviation from the mean (i.e., $Y = \frac{n-\mu}{\sigma}$).

In the JBoss and Tomcat projects, multiple release branches are supported in parallel. For example, JBoss developers produce service packs for the JBoss 4 and JBoss 5 releases simultaneously. For these projects, we standardize values with respect to each branch rather than across all releases. A logarithmic transformation was explored, but we found that it compressed many of the subtle characteristics of the

trends.

The BLOC of ArgoUML in Figure 4.2 shows a clearly increasing trend with the exception of one period in between releases 0.18.1 and 0.20 (Figure 4.2(b)). During this period, ArgoUML underwent a restructuring where modules for C# code generation and internationalization were migrated from the main ArgoUML repository into separate repositories. In doing so, the ArgoUML team seized an opportunity to revise the associated build specifications for these modules. As a result, the overall build system size was reduced. The ArgoUML team confirmed our findings.

Tomcat shows two unique trends of growth in BLOC. In the 4.0.x releases, the build system was initially subject to a rapid growth (Figure 4.2(d)). This was due to extensive work in the Catalina subproject. 568 lines of BLOC were added to implement configuration detection and release packaging logic in the Catalina build specification file. This period of rapid growth was followed by a rather calm period where only critical bug fixes were committed to the branch as it neared the end of its maintenance life. The 4.1.x branch begins its life with a calm period, followed by an 18-month hiatus between revisions 4.1.31 and 4.1.32 (Figure 4.2(e)) as Tomcat moved out of the Jakarta project and was rebranded as a standalone Apache project. This period shows an explosive increase of both BLOC and SLOC as a result of the 18 month project structure overhaul. After the restructuring was complete, the branch returns to a relatively calm progression as it approaches its end of maintenance life.

Restructuring efforts shown in Figure 4.2(f) and Figure 4.2(g) skew the first half of the results in JBoss, which otherwise has an increasing trend in BLOC. During Figure 4.2(f), an entire rewrite of the enormous test suite build specification file resulted in the removal of approximately 5,000 BLOC. During Figure 4.2(g), code for

supporting JAX-RPC was moved out of the main JBoss project and into a separate plugin project called JBoss WS (Web Services). In addition, the 'common' module was removed and its source code was integrated into other areas of the project hierarchy. As a result, the main JBoss project lost two build specification files and 568 BLOC.

Figure 4.2 shows that the Eclipse build system is growing in terms of BLOC. Further inspection of the trend in Figure 4.3 shows that the Eclipse build system is actually growing exponentially. This exponential trend is accounted for by the plugin nature of Eclipse. The Eclipse project maintains a modular and self-contained build system for each plugin. The top level of the build system simply chains together the builds for each plugin. It then follows that with each new plugin added, a large amount of build code is also introduced. As popularity rises and more plugins make their way into the Eclipse project mainline, these new plugins introduce with themselves more build code. This suggests that the exponentially rising trend in build system size strongly correlates with the trend in the number of plugins per release, which is also growing exponentially.

Similar to Lehman's first law of software evolution, build system specifications tend to grow over time unless explicit effort is invested to restructure them.

All dimensions of ANT build systems grow: Table 4.3 shows the Pearson correlation between the static size metrics for each studied system. With the exception of the JBoss project, which will be discussed later, the high correlation values indicate that BLOC, target and task count evolve similarly. Since the general trends of BLOC are growing, we can say that all dimensions of the ANT build systems grow.

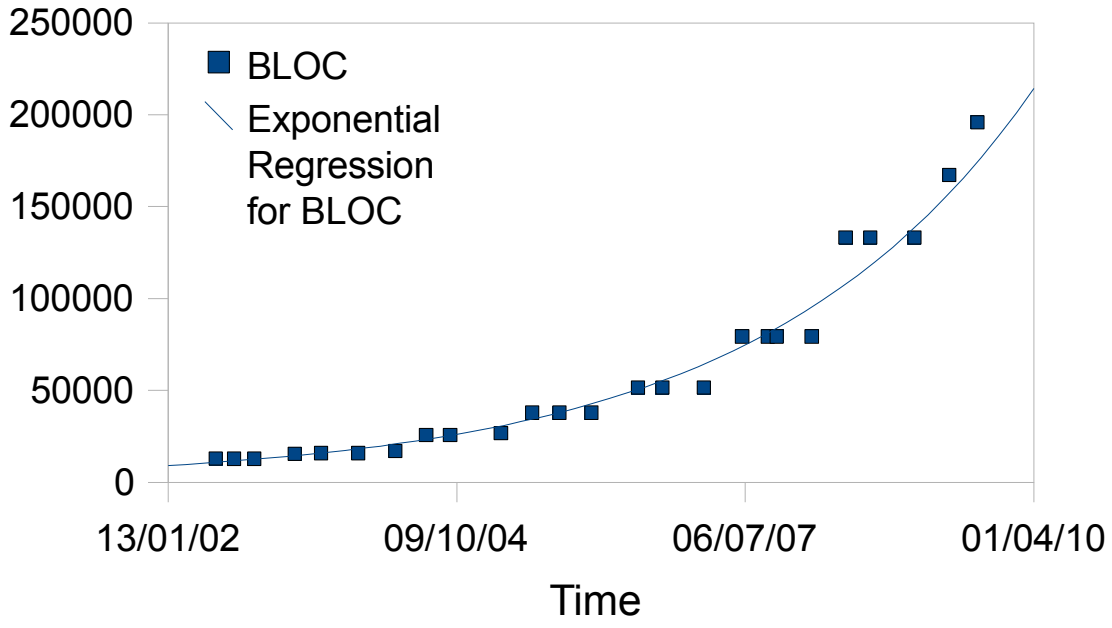

Figure 4.3: The exponential trend in Eclipse BLOC. The trend line has an R^2 value of 0.98.

We find that the Halstead complexity metrics follow trends similar to BLOC. Table 4.4 shows, for each studied system, the Pearson correlation between each Halstead complexity metric and the BLOC. With the exception of the JBoss project, our results indicate that build specification complexity metrics are highly correlated with build specification size metrics (BLOC). This result seems to agree with similar findings from research in the source code domain [25, 56].

In the JBoss build system, the Halstead complexity metrics and build system size are not highly correlated, as the JBoss build system is implemented using a different style called JBoss buildmagic. It leverages the underlying XML roots of ANT specification files to introduce a system of abstraction. The <!ENTITY> macro substitution tag is used extensively to import build specification code from external

Table 4.4: Pearson correlation between Halstead Complexity Metrics (Rows) and BLOC size (Columns).

	ArgoUML	**Tomcat**	**JBoss**	**Eclipse**
Volume	0.99	1.00	**0.17**	1.00
Difficulty	0.98	0.99	**0.20**	1.00
Effort	0.93	0.98	**0.11**	0.96

files, similar to header file inclusion in C. The expansion is performed at run-time. This causes skew in our results since we study BLOC in the unexpanded build files, whereas for the three other systems there is no difference between expanded and unexpanded form.

The Halstead complexity of a build system is highly correlated with the build system's size (BLOC), indicating that BLOC is a good approximation of build system complexity.

Source code and ANT build system size are highly correlated: Based on our observations of size and complexity trends, we are now able to verify whether trends in the size of the build system coincide with trends in the size of the source code. For each project, we: (1) calculate the Pearson correlation between BLOC and SLOC, and (2) visually compare the trends of BLOC and SLOC in Figure 4.2.

Table 4.3 shows that BLOC and SLOC are highly correlated, suggesting that the build system and source code tend to evolve together. Once again, the JBoss results are skewed because of their <!ENTITY> code inclusion method.

Figure 4.2 illustrates the correlation between the growth in BLOC and SLOC for the four subject systems. In most cases, the characteristics of the source code and build specification curves are very similar, which suggests that build system and source code are co-dependent. Deviations from the trend are analyzed by investigating

individual commits in the respective source code repositories.

In ArgoUML, anomalies occur at Figure 4.2(a), (b), and (c). During Figure 4.2(a), a restructuring was performed where source code that was previously hard-coded in six java source files, became automatically generated from an ANTLR grammar file. The build specifications were updated to perform the Java code generation task. Hence, we see an increase in BLOC and a sharp decrease in SLOC. During Figure 4.2(b), C# code generation and internationalization modules were moved out of the main ArgoUML repository and into individual repositories (as mentioned above) and the test source code of the unit tests module was distributed across different areas of the project hierarchy. The build specifications for the original unit tests module were deleted. Since no source was removed in the restructuring process and development work in other areas was continuing, we see an increase in normalized project source code. During Figure 4.2(c), another restructuring effort was undertaken where the documentation module was removed and placed into its own repository. In ArgoUML, the majority of build system restructuring seems to be instigated by source code evolution.

In the Tomcat project, the trends suggest that the source code and build system are growing in sync with each other. The increases at Figure 4.2(d) (i.e., Catalina subproject build growth) and (e) (i.e., Jakarta to Apache rebranding) are explained above.

For the first of the parallel release branches of the JBoss project, it would appear that there is little correlation between the BLOC and SLOC trends. During the rewrite of the build specification file in the testsuite module in the Figure 4.2(f) interval, the system source code was unaffected and hence was subject to the standard

growth. During Figure 4.2(g), JAX-RPC support was moved out of the main JBoss project and as a result, the SLOC reduced by 72 KSLOC. These two events produce considerable noise in otherwise highly correlated BLOC and SLOC trends.

In Eclipse, the trends in BLOC and SLOC are very similar. However, in between releases 3.5 and 3.5.1 (Figure 4.2(h)), we observe a sharp increase in BLOC and a moderate increase in SLOC. The BLOC increase is due to the introduction of a special plugin with the express purpose of driving the build system. The org.eclipse.releng.eclipsebuilder plugin contains ANT code that invokes script generators to build all of the shipped Eclipse plugins. The plugin contains nine new ANT files and 1,127 BLOC.

In most projects, BLOC and SLOC are highly correlated. Manual inspection suggests that many large restructurings in the build system are caused by major restructurings in the source code.

4.2.3 Does the build-time complexity evolve?

We study the evolution of build-time complexity from three angles. First, we use Figure 4.4 to show growth of build graph length and depth in the four studied build systems, then we use Table 4.5 to examine the build recursion complexity, and finally we analyze changes in target coverage.

ANT Build Graph Behaviour Analysis: We study the dynamic behaviour of a build system, by examining changes to the standardized length and depth of its build graph.

During Figure 4.4(a), ArgoUML shows a large change in both dimensions of the build graph. This was caused by the introduction of new internationalization and

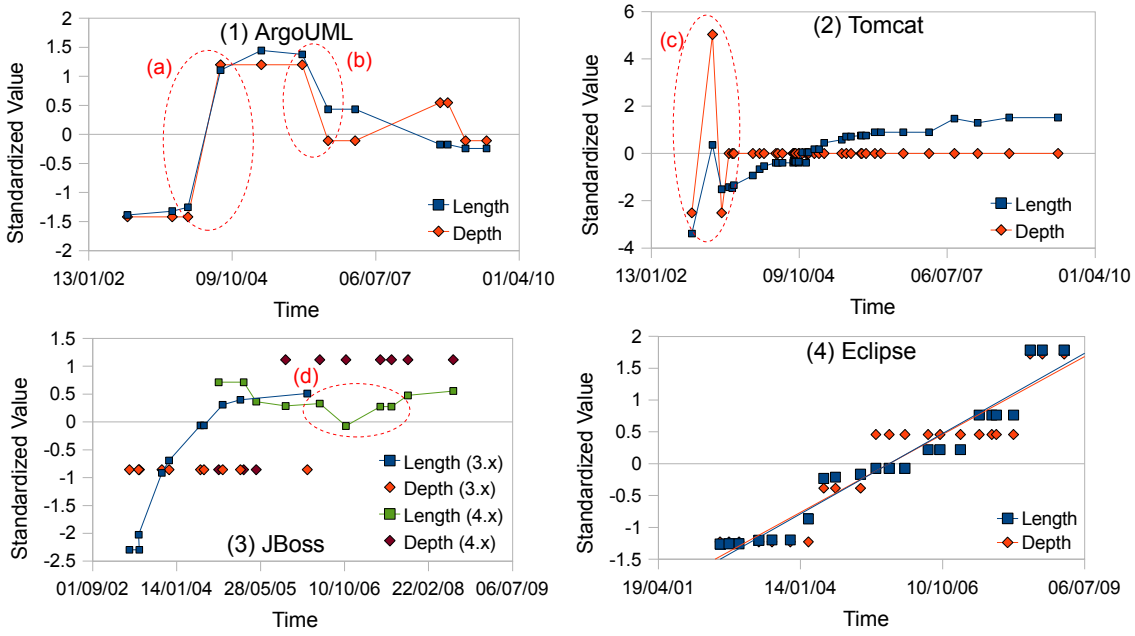

Figure 4.4: Standardized build graph dimensions (Dynamic analysis). Build graph length (in targets) and depth. Linear regressions are plotted for the dimensions in Eclipse, which have R^2 values of 0.94 (length) and 0.88 (depth).

unit test compilation targets that became part of the default build. The Figure 4.4(b) interval corresponds to the Figure 4.2(b) interval. The restructuring of modules into independent projects results in a considerable decrease in the build graph dimensions, and hence build time of the main project.

Figure 4.4(2) does not show data for Tomcat 3.x, 4.x and 6.x because of an interesting evolution. The Tomcat build system automatically downloads required third party Java archives (.jar files) based on hard coded URLs of the archived releases. The hard coded URLs for Tomcat 3.x and 4.x have become stale by now, preventing us from building these releases. The Tomcat 5.x URLs were still valid, allowing us to build these releases. During Figure 4.4(c), Tomcat shows an increase in build graph length and depth where a collection of third party library dependencies were, for a brief period, built from source instead of downloaded prebuilt. The inability to build Tomcat 3.x and 4.x shows that managing third-party dependencies is an important driver for build system evolution. This is why the Maven build technology integrates third-party library dependency management into the build system as discussed in Section 2.3.3.

In JBoss 3.x, the trend in build graph length sees rapid change initially, followed by a lull in later releases. However, JBoss 4.x shows a decrease in build length at (d) due to the removal of the JAX-RPC support and its build files from the main project at release 4.0.5 (mentioned above). JBoss 5.x is not plotted since only three releases in this branch are analyzed and this is not enough data to derive a solid trend.

In the Eclipse project, we see a steady linear increase for both the length and depth dimensions.

We found no general laws for build graph behaviour. Studied systems show either increasing trends in build graph length, or periods of growth and reduction. Trends are due to build restructurings or functionality being added to the default build.

Constant Depth vs. Varying Depth: Figure 4.4 shows two distinct trends in the build graph depth: (1) a near-constant depth (Tomcat and JBoss), and (2) a varying depth (ArgoUML and Eclipse). Table 4.5 shows the Pearson correlation between build graph depth and length metrics. The table indicates that the ArgoUML and Eclipse builds grow similarly in both length and depth dimensions, while Tomcat and JBoss do not. Manual investigation of the build systems of the projects reveals that the ArgoUML and Eclipse builds are recursive, while the Tomcat and JBoss ones are not. A recursive build process is one that divides the build process into smaller builds of each component, and each component build is further divided into builds of subcomponents, and so on. Conversely, non-recursive builds are performed in one build process. We observe that the recursive builds vary in depth, while the non-recursive builds have a constant depth. Through manual inspection of the release snapshots, we find that only the Tomcat project briefly switched between recursive and non-recursive build system designs early in the life of the project (Figure 4.4(c)).

Interestingly, Table 4.5 shows that the build length in targets and the elapsed time in the ArgoUML build process are not correlated. Since ArgoUML is a relatively small project, there is very little variation between the fastest build of 5 seconds and the slowest build of 23 seconds. Hence, ArgoUML's build is more susceptible to noise due to process scheduling and other environmental factors.

As the Eclipse project ages, the maximum depth of recursion reached during

Table 4.5: Pearson correlation between dynamic metrics (Rows) and build graph depth in each project (Columns). ArgoUML and Eclipse grow similarly in length and depth, while Tomcat and JBoss do not. Anomalies for a particular project are printed in bold and are discussed in the text.

	ArgoUML	**Tomcat**	**JBoss**	**Eclipse**
Elapsed Time	**0.37**	0.14	0.40	0.92
Build Graph Length (Targets)	0.92	0.37	0.48	0.96
Build Graph Length (Tasks)	0.94	0.12	0.26	0.96

its build process increases. This implies that as the project ages, the build process actually grows in both length and depth dimensions. The build system had grown to such a state that the Eclipse team has introduced in version 3.5.1 the org.eclipse.releng.eclipsebuilder plugin mentioned earlier.

The studied projects either select a recursive design or a non-recursive one. Once a design has been selected, the studied projects only switch early in the life of the project (e.g., Tomcat), since design changes during the implementation phase are not trivial.

ANT Build Coverage Behaviour Analysis: To study the dynamic coverage of a typical build, we calculate the proportion of targets exercised in a typical build relative to the total number of targets. We do not show a graph for coverage because the values remain relatively constant unless a major event occurs.

In ArgoUML, the coverage varies between 14-29%, with two notable increases of 7% and 8% corresponding to the project restructuring periods discussed earlier (Figure 4.4(b) and (c)). The BLOC shrank during the restructuring, which implies that the ArgoUML build system was bloated with unused code prior to the project

restructuring.

The coverage metrics in both Tomcat and JBoss are rather consistent at around 30% and 40% respectively. Minor fluctuations of ±3% occur between release branches (e.g., Tomcat 5.0.x to 5.5.x), however the major restructurings that were mentioned above do not seem to have an effect on the build system coverage.

In Eclipse, there is one notable change in the otherwise constant coverage showing an increase of 36% from 2.x to 3.x. This was caused by a decrease in the total number of existing targets and an increase in the number of targets hit by the default build. The decrease in total targets was caused by the removal of redundant build logic. This indicates that while major changes were made to system functionality (enough to warrant an increase in major release number), a similar amount of work was invested in the build system.

Target coverage remains more or less constant for each project. Major fluctuations of ±10% *correspond to major project events like restructurings and major releases, suggesting that build maintenance can impact the performance of a build system as perceived by developers and users.*

4.3 Maven Case Study

In this section, we present the results of our Maven case study with respect to our two research questions, which are the same as for the ANT study.

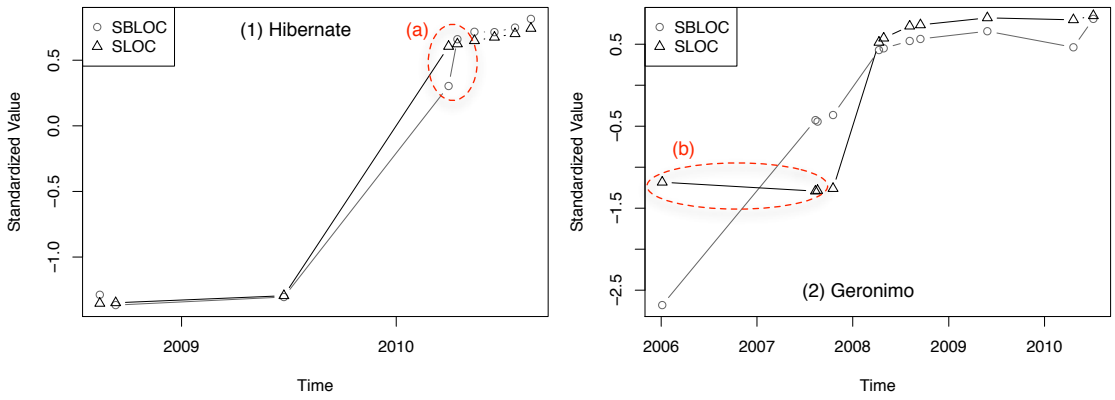

Figure 4.5: Standardized BLOC and SLOC values for the Maven projects. Source code and build system evolution trends are very similar.

4.3.1 Studied Projects

We selected two open source projects built using Maven with different sizes, domains, and release styles. Table 4.2 summarizes the characteristics of the projects. Hibernate is an Object-to-Relational mapping framework for Java programs, of which we studied the "core" subsystem. Geronimo is a web application server.

Four of the studied projects use ANT as the build technology (ArgoUML, Tomcat, JBoss, Eclipse), while only two studied projects use Maven (Hibernate and Geronimo). Since Maven is a newer build technology that is starting to gain momentum, there is less project data available for analysis.

4.3.2 Do the static size and complexity of source code and build systems evolve similarly?

We explored the evolution of Maven build specification files using the same three angles as ANT. First, we use Figure 4.5 to show a general trend of increasing size in the two projects, then we use Table 4.6 to show that there is a strong correlation between the growth in the static size and complexity of a build system, and finally we use Figure 4.5 and Table 4.6 again to show that the build system and source code evolve similarly.

Maven builds also grow: Figure 4.5 shows that, similar to ANT and `make`, the size of Maven-based build systems also grows over time. Below, we discuss the anomalies in each project.

In May 2007, the Hibernate project migrated their existing ANT build system to Maven build technology [17]. The exact motivation for the migration is unclear. Hibernate version 3.3.0, which was released in August of 2008, is the first Hibernate release that used the Maven build system to produce the official deliverables. In Figure 4.5, we show only the Hibernate releases built with Maven, i.e., from version 3.3.0 onward. In this section, we analyze the Maven-built portion of the Hibernate project evolution.

The Hibernate Maven build system shows consistent growth throughout it's life-time. The large spike in Figure 4.5(a) is due to major changes from the 3.3.2 to the 3.5.0 releases, while the smaller increases are due to small changes between service pack releases (e.g., 3.3.x). Large changes to the build are delayed until a new minor release (e.g., 3.5.0) in order to avoid breaking the existing build infrastructure of a

Table 4.6: Pearson correlation between BLOC (Columns) and the build system's Halstead complexity and SLOC (Rows). Anomalies in bold.

	Hibernate	**Geronimo**
Volume	1.00	1.00
Difficulty	0.99	**0.33**
Effort	1.00	0.84
SLOC	0.99	0.76

released branch.

The Geronimo project used Maven for their build system from project birth. The Geronimo build is consistently growing, with the exception of the encircled 1.0 to 2.0 transition, when the build shrank (Figure 4.5(b). In Geronimo, the 1.0 build system was implemented using Maven 1.x technology. In version 2.0, the Geronimo build system was migrated to Maven 2.x technology, which required major build specification changes [7]. Specifically, the project.properties and build.properties files are merged into a settings.xml file, and the maven.xml and project.xml files are replaced with the pom.xml file.

Maven builds grow unless explicit effort is invested to restructure them.

Maven specification static complexity evolves: Table 4.6 shows that, similar to ANT build systems, the Halstead complexity of Maven specification files is highly correlated with BLOC. Again, this is similar to prior work in the source code domain that suggests that size is a good approximation for source code complexity [25, 56].

In Geronimo, we observe little correlation between BLOC and Halstead Difficulty (0.33). The p-value for this metric was 0.30, much larger than the standard cutoff of 0.05, indicating that this correlation is not statistically significant. The Pearson correlation of the Volume and Effort Halstead metrics had p-values that were less

than 0.01, indicating that those correlations are statistically significant.

Since the studied build systems grow in size, and the build system complexity metrics are highly correlated with the size, we can say that the studied build systems also grow in complexity, as projects age.

Similar to findings in the source code domain [25, 56], build system complexity can be reasonably approximated using size in BLOC.

Maven build growth is highly correlated with source growth: The positive correlations in Table 4.6 also show that trends of growth or reduction in the source code are often accompanied by similar trends in the build system. We encircle periods in Figure 4.5 when the build and source code do not agree. Below, we elaborate on each anomaly with respect to each project.

In Hibernate, most periods of growth in the source code have similar growth in the build system. However, in the encircled period between releases 3.5.0 and 3.5.1, the build grew quicker than the source code. The build files were modified to add Groovy source code generation to the build process, which introduced a family of new library dependencies to the build.

In Geronimo, the encircled discrepancy between build and source code was due to the migration of Maven versions 1 and 2 mentioned above. Otherwise, the source and build size trends are similar.

Source code and Maven build systems tend to grow and shrink together.

4.3.3 Does the build-time complexity evolve?

We study the evolution of perceived build system complexity in Maven build systems using a similar approach as used to study ANT systems. We use Figure 4.6 to show

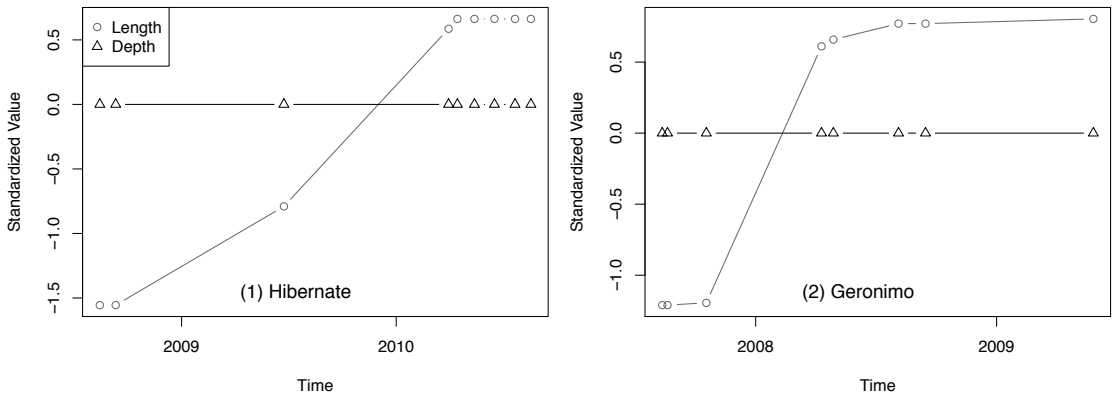

Figure 4.6: Standardized build graph dimensions (Dynamic analysis).

the growth of build graph dimensions in the two studied build systems.

Similar to our ANT study, we measure two dimensions of Maven build graphs. We measure the length of the build by counting the goals executed during the default build, and we measure the depth of a build by counting the number of directories from the deepest module containing a Maven specification file to the top of the source tree. Figure 4.6 shows the standardized versions of these two build graph dimensions for each release.

Since versions prior to 3.3.x of Hibernate were not built using Maven, we refrain from presenting them in Figure 4.6. In Geronimo, builds prior to 2.x require libraries that are no longer served in the Geronimo Maven repositories, hence we only present Geronimo builds of the 2.x releases.

Maven build length slowly increases as a project ages: Figure 4.6 shows growth in the length of a build as projects age. The steep increases in length happen during minor release changes, i.e., 3.3.2 and 3.5.0 in Hibernate and 2.0.2 and 2.1.0 in Geronimo. There is much less growth between service pack releases, e.g., 2.0.1 and

2.0.2 in Geronimo. There are fewer large changes in service pack releases, since they are more likely to introduce defects [45]. Thus, there is little growth in the build length since there is little new code to build. In minor and major releases, there are larger amounts of source code change, and hence longer build lengths since the new code must be compiled and linked.

Maven builds consistently grow longer as a project ages. There is much more growth between minor releases than in between service pack releases.

Maven build depth remains constant: In Maven, multi-module builds may be achieved using "Reactor" builds. In a Maven Reactor build, the top-level specification file is parsed first. The specification lists any modules that must be built in order to complete the build. The Maven process will then parse all of the module build files, each of which may contain their own specification lists that are processed recursively until there are no longer any module specifications to parse. The Maven process then proceeds to execute the necessary goals in each module until the build request is satisfied.

Figure 4.6 shows that depth is constant for both Hibernate and Geronimo projects. This suggests that Hibernate and Geronimo do not need to grow deeper. This may be due to the evolutionary activity before the period that we examine. For instance, we study Hibernate builds 3.3.x-3.5.x. This means that Hibernate has had 2 major releases, i.e., 1.x.x and 2.x.x, to solidify a source tree structure before we begin examining the build. Similarly, in Geronimo, we study the 2.x builds, leaving out the 1.x builds where much of the depth growth may have occurred.

The depth of the studied Maven projects does not change. More data is necessary to support a hypothesis about the depth dimension in Maven build systems.

4.4 Discussion

We divide our post-experiment discussion into (1) a comparison of our findings for ANT and Maven build technologies, and (2) a comparison of our findings for Java build systems to earlier findings for C and C++ build systems.

4.4.1 ANT and Maven comparison

We study the evolution of both ANT and Maven build systems in open source projects. We find that both build system types: (1) grow as a project ages unless explicit effort is invested to restructure them, (2) the build system size in BLOC is a good approximation for build system complexity, and (3) build system and source code grow together, and in cases when they disagree, they were often reacting to the same development event.

Although the Hibernate project migrated its existing ANT build infrastructure to Maven between versions 3.2.7 and 3.3.0, we are unable to directly compare the ANT and Maven build evolution. Such a comparison would not be fair for three reasons:

1. While the ANT build was much smaller, only ever reaching 1,152 BLOC, it provided much less functionality. Maven builds provide built-in mechanisms for library dependency management, automated test execution, report publishing,

and website generation. While these three tasks are achievable in ANT, they require a large investment of development effort.

2. The Hibernate migration to the Maven build was accompanied by a project restructuring. The Hibernate ANT build only needed to produce client and back end libraries, whereas the Maven build must produce several smaller libraries. This decomposition of the larger libraries was done to allow Hibernate users to only link their applications with those classes that they require. However, the smaller source code components that produce the smaller libraries must also have a build component to allow for seamless decomposition [14]. Thus, the number of build files increased. Furthermore, we find that the last Hibernate ANT build (version 3.2.7) had 288 BLOC/file on average in four files, and the first Maven build (version 3.3.0) had 78 BLOC/file in 27 files. This drop in average size was likely due to the restructuring effort and is likely not a generalizable trend across all Maven migrations, however more case studies are required to clarify this.

3. Due to Maven's convention over configuration design principle, Maven build systems inherently perform more functionality with less code than ANT build systems. Hence, comparing the size of an ANT build system to the size of a Maven build system is not productive.

4.4.2 C and Java build system comparison

We expected to find that Java build systems would require less effort to remain in sync with the source code than C build systems for two reasons:

1. A single invocation of the Java compiler automatically resolves dependencies between the input source files, while the C compiler must rely on external dependency management through build tools like `make`.

2. Since the Java compiler invocations are expensive (the Java Virtual Machine (JVM) must be started before and shut down after each invocation), build developers capitalize on the Java compiler's ability to compile many Java source files in one invocation.

In this section, we compare our findings for Java to prior work on `make`-based C build systems.

Adams *et al.* made three observations about the evolution of the `make`-based Linux build system: (1) the Linux build system evolves, (2) the complexity of the build increases over time, and (3) maintenance drives the evolution of the build system. These findings are mirrored by our findings with both ANT and Maven build systems for Java projects, i.e., both ANT and Maven build systems grow in size and complexity unless explicit effort is invested to restructure them.

In the studied projects, we found that the build and source code grow at similar rates when standardized (RQ1). This indicates that, similar to C build systems, effort is still invested in keeping the build in sync with the source code. However, there are differences in the driving motivations of the evolution. For instance, in `make`, there are serious flaws in the common recursive `make` paradigm used to implement modular `make`-based build systems [43]. The Linux build engineers invested much effort in maintaining a modular build system that is not susceptible the flaws associated with recursive `make` [2]. Build system modularity support is built into ANT via the <ant> task, and into Maven via Reactor builds. Modularity support provided by ANT

and Maven relieves ANT and Maven build engineers from concerns about potential modularity flaws, unlike `make`. Instead, ANT and Maven build system evolution is driven mainly by development events such as restructurings.

The Linux build engineers were also greatly concerned with maintaining a simple interface for driver developers to integrate code into the Linux build process. This is a major concern since driver source code contributions make up the majority of the Linux source code [24]. We found that similar concerns drive the evolution of the Eclipse and JBoss build systems. Eclipse build engineers maintain a separate plugin that simplifies the Eclipse build process for plugin developers. The JBoss buildmagic code increases build code reuse and simplifies the process of adding a JBoss component to the JBoss project.

Finally, Linux build engineers must maintain explicit dependency listings among targets in the build specifications, i.e., `makefile`s. ANT and Maven build specification are not concerned with such details, since the Java compiler handles dependency management among source files.

4.5 Chapter Summary

Software build systems are complex entities in and of themselves. They evolve both statically and dynamically in terms of size and complexity. We find that Lehman's first two laws apply in the context of build systems. That is, our case study indicates that:

1. Build systems change continuously, especially due to changes in their environment (i.e., source code and development libraries).

2. Build systems grow in complexity as a side effect of the changes induced by Lehman's first law.

Through a case study of six open source Java projects, we made the following important observations across ANT and Maven build systems:

- Both the static and dynamic size and complexity of build systems show differing patterns of growth over time that correlate with the size of the source code.
- The exponential growth of Eclipse's build system is highly correlated with the project plugin count.
- Once a build system has established either a recursive or non-recursive design, it rarely switches between designs.
- The Halstead complexity of a build system is highly correlated with the build system's size (BLOC).
- As observed in Tomcat, management of third-party libraries is a crucial factor in build system evolution.
- Major fluctuations in build-time target coverage of $\pm 10\%$ correspond to major project events like restructurings and major releases, suggesting that build maintenance can impact the performance of a build system as perceived by developers and users.
- The findings above are consistent with earlier findings for `make`-based systems with slightly different drivers of build system evolution.

We conclude that C and Java build systems evolve similarly at the release-level. Java build systems in general appear to co-evolve with the project source code, which agrees with prior work on `make`-based build systems [2, 66]. Armed with this understanding, project managers can predict that periods of substantial change in the source code will be accompanied by similar change in the build system.

Reason suggests that such a co-evolution of build and source code imposes some degree of overhead on software development, yet we cannot measure it with such a coarse level of analysis. In the next chapter, we study the co-evolution of build and source code at a finer granularity, i.e., individual source code revisions instead of releases, to further validate our research hypothesis, i.e., build system maintenance plays an important role in software development.

Chapter 5

Build System Evolution at the Revision-level

[Build system migration is] surely a lot of work, that also includes the risk of scaring people away. Answer this: what is more scary: the current build system or the idea of throwing anything you know about the current build system away?

Anonymous developer

In the prior chapter, we presented our study of release level build system evolution in Java projects. We find that Java build systems evolve both statically in the build specification files, and dynamically at build execution time. These findings agree with those of prior work on `make`-based build system for C projects [2, 66].

The prior chapter focused on a coarse analysis from release to release. However, these releases are composed of numerous developer changes, i.e., revisions. In our release-level analysis, these revisions are blurred together, and as a result, we may be

overlooking phenomena that occur in between the releases.

To further validate our research hypothesis, i.e., build system maintenance plays an important role in software development, this chapter presents our study of the day-to-day maintenance of the build system throughout revisions. An earlier version of this study will be published at the 33rd International Conference on Software Engineering (ICSE) [41]. The study addresses three research questions:

RQ1) How many files does a typical build system consist of?

Motivation – We want to study the size of a typical build system and how it evolves at the revision-level to better understand the magnitude of the build system.

Findings – The build system accounts for a relatively small proportion of the files in a project (9% median).

RQ2) How much does a typical build system churn?

Motivation – Churn measures the rate of change in source code. Prior studies have found that frequently changing source code, i.e., code with high churn, has a higher defect density [45] and causes more post-release defects [46]. We want to measure churn in the build system to gain insight into how susceptible the build system is to defects.

Findings – The build system has a churn rate comparable to the source code. This suggests that build systems are constantly evolving and are likely to have defects [45].

RQ3) How large are typical build system changes?

Motivation – There is no prior work that quantifies the size of typical build

Table 5.1: Projects studied at the revision-level

	ArgoUML	Hibernate-core	Eclipse-core	Jazz	GCC	Git	Linux	Mozilla	PLplot	PostgreSQL
Timespan	'98-'09	'01-'07	'01-'10	'07-'08	'88-'05	'05-'09	'05-'10	'98-'10	'92-'09	'96-'09
Program lang.	Java	Java	Java	Java	C	C	C	C	C	C
Build techs.	Make* ANT	ANT* Maven	PDE	PDE	Autotools	Make Autoconf	Make KConfig	Make Autoconf	Make* Autotools* CMake	Autotools
# Config. Files	289	54	437	5,707	942	10	1,708	2,394	314	50
# Const. Files	325	157	46	260	777	33	2,018	8,315	338	721
Total (TB)	614	211	483	5,967	1,719	43	3,726	10,709	652	771
# Prod Files	7,116	9,272	2,391	45,275	14,181	743	42,912	43,952	659	2,683
# Test Files	891	7,426	1,211	14,738	21,109	824	340	30,835	791	1,377
Total (TS)	8,007	16,698	3,602	60,013	35,290	1,567	43,252	74,787	1,450	4,060
$\frac{\text{TB}}{\text{TB+TS}}$	7%	1%	12%	10%	5%	3%	8%	12%	31%	16%

* Build technologies used before migration

system changes. Large changes imply that considerable effort is put into build system maintenance.

Findings – A typical build change adds or removes 3 to 4 build lines of code (BLOC), while a typical source change adds or removes 4 to 17 source lines of code (SLOC).

The chapter is organized as follows. Section 5.1 presents the studied projects. Section 5.2 discusses the design of our case studies used to address the research questions, while Sections 5.3, 5.4, and 5.5 present the results.

5.1 Studied Projects

We conduct a large scale study of ten different open and closed-source software projects. Table 5.1 summarizes the characteristics of the studied projects.

Table 5.2: File type classification examples

Build	Production	Test
Config. and Const. layer (Makefile*, configure*)	Production code (*.c*, *.h*, *.java)	Unit tests (*.c*, *.h*, *.java)

Jazz [TM1] is a commercial next-generation IDE developed by IBM. The GNU Compiler Collection (GCC) is a popular source code compiler with front-ends for many programming languages. Git is a distributed version control system. Linux is an operating system kernel. Mozilla is a suite of internet tools, such as the Firefox web browser. PLPlot is a plotting library with bindings for many popular programming languages. PostgreSQL is an object-relational database system. ArgoUML, Hibernate, and Eclipse are introduced in the prior chapter.

We selected open source projects of different domains, build technologies, and programming languages to reduce bias. Jazz is one of the few available data sets with high-quality work item linkage [10, 51].

5.2 Case Study Setup

We first classify each file that existed in the given timespan as either build, production, or test code. Those files that do not fit in any category are marked as "other". Table 5.2 provides some example file type classifications that we used.

The classification process was semi-automated. Most files could be classified using file type naming conventions. However, patterns such as ".java" and ".xml" were ambiguous, i.e., some .java files are production code while others are test code. After initial filtering of unambiguous file types, the remaining files had to be classified

[1] `http://www.jazz.net`. IBM and Jazz are trademarks of IBM Corporation in the US, other countries, or both.

manually. For example, of the 49,364 files in Linux, approximately 40,000 could be classified automatically. The remaining 9,000 or so files had to be tagged manually based on our prior experience with build systems.

5.2.1 Build Abstraction

The GCC, Git, Mozilla, PLplot, and PostgreSQL projects make use of the GNU Autotools and CMake build abstraction languages. These languages allow build engineers to implement build logic for many different platforms using an abstract representation of the build process. A build code generator produces the necessary platform-specific code at build time. For these case studies, we focus on our analysis on revisions to the Autotools and CMake abstraction files.

5.3 How many files does a typical build system consist of?

The build system accounts for 9% of the maintained files (median). Table 5.1 shows that the build accounts for 1-31% of the maintained files that existed in the given timespan, with a median of 9%. These low values indicate that in most cases (PLplot being the exception), the build system is dwarfed by the other development artifacts.

Hibernate-core (1%) and PLplot (31%) are anomalies. Being entirely composed of a single library, the Hibernate-core project has little build code (211 files), which explains the low 1%. On the other hand, the PLplot project has the most inflated build file percentage of 31%. While the PLplot project is rather small, it provides bindings

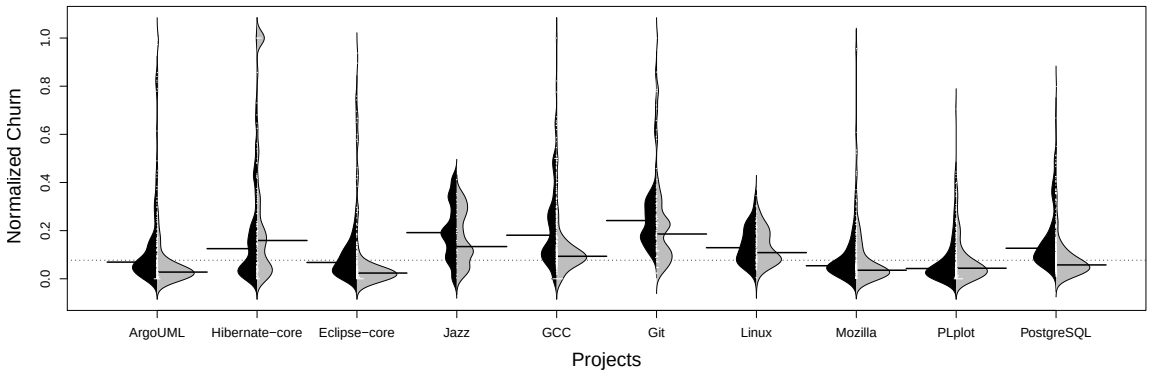

Figure 5.1: Distribution of monthly churn in source (black) and build (grey) files.

to many programming languages. Each binding has its own construction layer component and extensive configuration code, increasing the build system size. The problem is compounded by two build technology migrations that PLPlot has undergone. The migrations reimplemented build code from `make` [18] into GNU Autotools [22], and later from Autotools into CMake [36], as mentioned in 5.1.

5.4 How much does a typical build system churn?

The normalized churn of build files is similar to source code. To study the churn rate in the build system, we compare its churn rate against that of the source code. We measure the churn rate in the source code and build system using *normalized churn* to take system size into account. We count the number of source files and the number of build files that were changed in each month-long development period. We then divide each count by the total number of source files or the total number of build files that existed in the period. We repeat this process for each development month. We chose a development month period length rather than shorter periods, such as a

day or week, because we feel that a month is enough time for a significant amount of development to occur.

Figure 5.1 plots the distribution of the monthly normalized change using a *beanplot*. Beanplots are boxplots in which the vertical curves summarize and compare the distributions of different data sets [29], which in our case correspond to the normalized churn of build and source code in month-long development periods. The horizontal black lines indicate the median of the normalized change for each project's source (black) and build files (grey).

In most of the studied projects, the median of the monthly normalized change for the source and build files are relatively close to each other, only differing by at most 7% (GCC).

The Hibernate-core is the only project with a median value of the normalized change for the build files greater than that of the source files. The Hibernate-core project had only 1-7 build files during the first 12 months of development, and easily reached 100% normalized churn, skewing the median.

The comparable rate of change in the source and build files is concerning, since rapidly changing source code modules often contain more defects than slowly changing ones [45]. Build maintainers must take great care to ensure that the build system does not become defect-prone, since a defect-ridden build system may: (1) slow development progress due to suboptimal build routines [43], or (2) fail to produce correct deliverables, which grinds development progress to a halt [30, 49].

Table 5.3: Number of lines changed per revision

	Quantiles	ArgoUML		Hibernate		Eclipse		GCC		Git		Linux		Mozilla		PLplot		PostgreSQL	
		+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-
Bld	**Lower**	1	1	1	1	1	1	2	2	1	1	1	1	1	1	2	1	1	1
	Median	2	2	4	2	3	2	6	5	2	1	2	2	4	3	5	4	4	3
	Upper	9	5	13	6	9	4	36	32	6	3	8	6	11	11	16	12	14	12
Prod	**Lower**	4	3	4	2	3	2	3	2	3	2	3	2	3	2	5	2	4	2
	Median	15	9	17	9	10	5	8	5	8	4	9	6	11	6	14	7	15	9
	Upper	49	36	61	33	32	19	30	21	24	13	29	21	44	28	44	26	65	43

5.5 How large are typical build system changes?

A typical build change adds and removes 3–4 lines of code (median). Table 5.3 shows the median, the lower and the upper quartiles of the number of line of code added and deleted per revision in the nine studied projects. Jazz data was unavailable for analysis because we do not have access to the actual code.

Most source changes add between 8–17 lines and remove between 4–9 lines (median). The corresponding numbers for build changes are 2–6 lines and 1–5 lines (median). Thus, when the build changes, the size of the change is about $\frac{1}{4}$–$\frac{1}{2}$ of the size of a typical source change.

5.6 Chapter Summary

Our findings indicate that while the build system is small, many developer revisions include build maintenance. Build system maintenance generates a churn rate comparable to that of the source code when normalized by their respective sizes. As such,

the build system may be similarly susceptible to defects [45].

In analyzing the evolution of build systems at the revision level, we make the following important observations that support our hypothesis, i.e., build system maintenance plays an important role in software development:

- The build system accounts for up to 31% of the code files in a project, with a median value of 9%.
- While most build systems are small in comparison to the source code, the normalized churn is comparable to that of the source code.
- A typical build change involves adding and deleting 3–4 lines of build code.

Chapter 6

An Empirical Study of Build Maintenance Overhead

The results of Chapter 4 and prior work [2, 66] suggest that build system and source code co-evolve with each other between releases. In Chapter 5, we find that the build system is relatively small in size, being composed of 9% (median) of the code files in the studied projects. However, it churns frequently, at rates similar to the source code when normalized by their respective sizes, suggesting a high overhead on developers. To analyze whether these revision-level build changes are burdensome on developers, we now investigate how tightly coupled developer changes to production and test code are to build system changes. To do so, we address two research questions:

RQ1) How often are build changes required to complete development tasks?

Motivation – Kumfert *et al.* estimate that developers spend 12% of their time keeping the build system in sync with the source code, rather than fixing bugs

and adding new features [32]. These results are based on a survey asking developers about their overall build maintenance effort. We are interested in rigorously validating these findings with the actual changes developers make.

Findings – Managers of C projects should explicitly account for up to 27% of production code work items to require build maintenance. Java projects can expect 4–16% of production code work items to require build changes.

RQ2) How do projects distribute build maintenance work?

Motivation – Since build systems have high churn, some projects designate members of the development team as build experts. To study the different ways in which projects are allocating personnel to build maintenance, we want to see how many developers have to modify the build system.

Findings – We find that the teams in the analyzed projects adopt one of two build ownership styles: (1) most build changes are performed by a small team of build experts (Linux and Git), or (2) build changes are dispersed amongst the development team (Jazz).

The chapter is organized as follows. Sections 6.1 and 6.2 present the results of our case studies used to address our research questions.

6.1 How often are build changes required to complete development tasks?

In this section, we study how successful projects maintain consistency between the build system and the production or test code. We measure the coupling between

these entities by evaluating various association rules using "interest" metrics.

6.1.1 Approach

We study consistency management at two levels of granularity:

1. *Revision Level*

   We study coupling between production/test code and build code in revisions, as is typically done in empirical studies. However, revisions offer too fine of an analysis, since typical developer tasks such as adding a new feature or fixing a serious bug involve multiple revisions.

2. *Work Item Level*

   The set of changes resolving a developer task is called a "work item". Since developers typically reason in terms of work items rather than revisions, we also studied coupling between production/test code and build code at the work item level. However, most projects do not record this data in a recoverable fashion [11, 51]. Hence, our work item analysis is limited to three of the studied projects.

We adopt association rule interest metrics to measure the relationships between production, test, and build files. An association rule is a statistical description of co-occurring elements in a dataset [4]. For example, Amazon.com recommends additional purchases by mining association rules from their database of prior customer purchases. We do not mine association rules from the data, but rather we evaluate the following associations:

Table 6.1: Association rule interest metrics

Metric	Calculation
Support(A)	$\frac{\#\ class\ A\ transactions}{\#\ total\ transactions}$
Conf(A $\Rightarrow$ B)	$\frac{Support(A,B)}{Support(A)}$
Conv(A $\Rightarrow$ B)	$\frac{Support(A) \times Support(\neg B)}{Support(A,\neg B)}$

Prod $\Rightarrow$ **Bld** measures the coupling between production code and the build system layers, i.e., the implication that a production code change will be accompanied by a build code change in the same revision. Similarly, **Test** $\Rightarrow$ **Bld** measures the implication that test code change will be accompanied by a build code change in the same revision.

Bld $\Rightarrow$ **Prod** measures the implication that a change to the build system layers will be accompanied by a production code change in the same revision. Intuitively, we expect this implication to be strong since Chapter 4 showed that the majority of build system maintenance is the result of production code change, and hence should be grouped together. Similarly, **Bld** $\Rightarrow$ **Test** measures the implication that a build layer change requires an accompanying test change.

We evaluate the association rules above using the "interest" metrics in Table 6.1. Support(X) is defined as the proportion of revisions that contain X [4].

Confidence(X $\Rightarrow$ Y) (Henceforth abbreviated to Conf(X $\Rightarrow$ Y)) measures the strength of the implication that X implies Y [4]. In our context, Conf(X $\Rightarrow$ Y) is the probability that a revision or work item changes Y, given that it changes X. Gall *et al.* use an identical metric to measure logical dependencies between modules, i.e., logical coupling [20]. Thus, we use confidence to measure the strength of the logical

coupling from X to Y. Note that confidence measures are directional, i.e., Conf(X $\Rightarrow$ Y) $\neq$ Conf(Y $\Rightarrow$ X).

Conviction(X $\Rightarrow$ Y) (Henceforth abbreviated to Conv(X $\Rightarrow$ Y)) is a measure of the departure of Conf(X $\Rightarrow$ Y) from independence [12]. A Conv(X $\Rightarrow$ Y) of 1 indicates that the Conf(X $\Rightarrow$ Y) is no different than would be expected if X and Y were independent of each other. Conviction values less than one indicates that the confidence observed is less than expected for independent variables. Conviction values greater than one indicate that the confidence observed is more than for independent variables. Throughout this study, we use conviction to evaluate whether the logical coupling induced by build maintenance is exceptionally lower or higher than expected, i.e., conviction values much less than or greater than one, or as expected in the case of independence, i.e., approximately one.

Finally, we use a χ^2 goodness-of-fit test [55] to validate the statistical significance of the coupling between the production and test code components and the build system. If the χ^2 statistic is greater than 3.84 ($\alpha \leq 0.05$), the relationship is statistically significant. We report the p-value of the χ^2 test, rather than the χ^2 statistic.

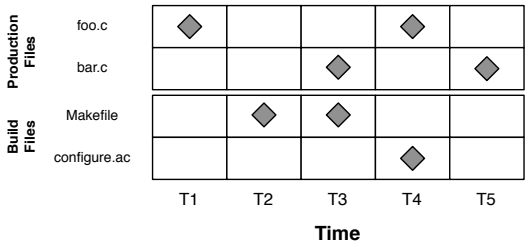

Figure 6.1: An association rule example scenario.

We use the example in Figure 6.1 to illustrate the confidence and conviction metrics. A series of five revisions appear on the Time axis and a series of four files (two

Table 6.2: Association rule metric values for production, test, and build code

		ArgoUML	Hibernate-core	Eclipse-core	Jazz	GCC	Git	Linux	Mozilla	PLplot	PostgreSQL
Support	Prod	0.62	0.62	0.68	0.69	0.56	0.61	0.87	0.70	0.39	0.55
	Test	0.06	0.32	0.23	0.18	0.13	0.11	0.01	0.08	0.19	0.10
	Bld	0.07	0.08	0.08	0.09	0.15	0.07	0.10	0.16	0.36	0.16
	Prod, Bld	0.01	0.03	0.02	0.03	0.04	0.03	0.06	0.06	0.03	0.05
	Test, Bld	<0.01	0.02	0.01	0.01	0.01	<0.01	<0.01	0.01	0.03	0.02
Conf	Prod ⇒ Bld	0.02	0.05	0.03	0.04	0.07	0.04	0.06	0.08	0.08	0.10
	Bld ⇒ Prod	0.16	0.36	0.28	0.28	0.27	0.41	0.56	0.35	0.09	0.34
	Test ⇒ Bld	0.05	0.05	0.03	0.07	0.07	0.04	0.13	0.16	0.17	0.19
	Bld ⇒ Test	0.04	0.20	0.09	0.13	0.06	0.07	0.01	0.08	0.09	0.11
Conv	Prod ⇒ Bld	0.95	0.96	0.95	0.94	0.92	0.98	0.96	0.91	0.69	0.93
	Bld ⇒ Prod	0.45	0.59	0.44	0.43	0.60	0.66	0.30	0.46	0.67	0.68
	Test ⇒ Bld	0.98	0.97	0.95	0.97	0.91	0.97	1.04	1.00	0.77	1.03
	Bld ⇒ Test	1.02	0.91	1.34	0.94	0.96	0.96	1.01	1.02	0.97	1.05
χ^2 (p-value)	Prod, Bld	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01
	Test, Bld	0.06	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	0.93	<0.01	<0.01

production and two build) appear on the Y-axis. Of the four production file revisions, two have build file changes as well (T3 and T4), thus the Conf(Prod ⇒ Bld) is 0.5. The Conf(Bld ⇒ Prod) is 0.67 since two of the three revisions have a production file change. The Conv(Prod ⇒ Bld) is 0.8 due to a more complex calculation, but can be interpreted as the Conf(Prod ⇒ Bld) of 0.5 is 20% less than expected if production and build system were independent entities (conviction of 1). We do not calculate the χ^2 value for this example, since the test was intended for a larger sample size. However, suppose the statistic for the production-build relationship is larger than 3.84 (and hence, the p-value smaller than 0.05), then the low coupling and conviction of Prod ⇒ Bld would be statistically significant, i.e., not just an artifact of noise.

6.1.2 Revision-Level Results

In this section, we use Table 6.2 to show that there is low revision-level coupling between the source and build files.

Low revision-level coupling from the production or test code to the build system: The confidence values in Table 6.2 show that the coupling values from production or test files to the build files are lower than Kumfert *et al.*'s 12% survey-based estimate [32]. We examine the coupling from production and test code to the build system below.

A revision rarely includes both production and build file changes as reflected by the low Support(Prod, Bld) values. The Conv(Prod ⇒ Bld) values reveal that in most cases (except Jazz) the observed coupling is consistently less than coupling expected of independent production and build files. The χ^2 statistic shows that the production and build files are significantly independent of each other in all of the studied projects.

In PLplot, the Conv(Prod ⇒ Bld) and Conv(Test ⇒ Bld) values show that the coupling between test and build code is low. The changes in PLplot are especially disconnected due to the two migration efforts that generated many build-only changes.

> *There is low revision-level coupling from production and test code to the build system.*

Low revision-level coupling from the build system to the production or test code: The confidence values in Table 6.2 indicate that the build system is coupled more to the source code than vice versa. The Conv(Bld ⇒ Prod) values are all much less than 1 (0.43–0.68), indicating that there is much less coupling from the build and production files than expected. This finding is counter-intuitive, since we would expect that most build changes would be accompanied by production code

Table 6.3: Overview of work item data.

	Eclipse-core	Jazz	Mozilla
Txs	6,391	36,557	210,400
Txs w/ Work Items	4,092	22,485	79,242
% Tx w/ Work Items	64%	62%	38%
Work Items	2,452	11,611	55,199

changes. The Conv(Bld ⇒ Test) are all close to 1 (except Eclipse-core), indicating that the coupling values are not out of the ordinary.

In PLplot, the confidence values are low due to the slow migration period. Much of the build migration effort was committed in revisions that were not related to any source code. Hence, the confidence values are reduced.

While the coupling from the build system to the production and test code is higher than in the other direction, the conviction values indicate that the observed coupling is lower than statistically expected.

6.1.3 Work Item Results

The confidence values we have observed suggest that there is low coupling between the production or test files and the build system. These values are lower than we had anticipated based on the Kumfert *et al.*'s survey-based estimation of 12% [32]. However, they confirm an earlier study of KDE that found that build revisions are often dominated by the build and do not co-change with other entities [57].

Table 6.4: Work item interest metrics

		Eclipse-core	Jazz	Mozilla
Support	Prod	0.87	0.85	0.83
	Test	0.31	0.24	0.17
	Bld	0.17	0.05	0.26
	Prod, Bld	0.14	0.04	0.22
	Test, Bld	0.06	0.02	0.08
Conf	Prod ⇒ Bld	0.16	0.04	0.27
	Bld ⇒ Prod	0.82	0.72	0.86
	Test ⇒ Bld	0.20	0.08	0.44
	Bld ⇒ Test	0.36	0.36	0.29
Conv	Prod ⇒ Bld	0.99	0.99	1.01
	Bld ⇒ Prod	0.74	0.52	1.15
	Test ⇒ Bld	1.03	1.03	1.31
	Bld ⇒ Test	1.07	1.19	1.16
χ^2 (p-value)	Prod, Bld	0.02	<0.01	<0.01
	Test, Bld	0.16	<0.01	<0.01

We conjecture that the low observed coupling is due to developer commit behaviour. For example, while some developers commit related build and source changes under one revision, others may commit build changes in separate revisions from source code changes, which introduces noise in the revision-level data. To address this, we find that all related revisions should be linked to a single ITS work item, i.e., groups of related revisions can be linked together according to the ITS work item that they collectively resolve. By investigating the relationship between source and build files at the work item level, we aim to reduce the noise caused by different developer commit behaviour.

Our work item analysis is limited to the three projects in Table 6.3. Prior work notes the lack of availability of high quality work item linkage in VCS data [11, 51]. With this in mind, a three project study is actually quite unique.

Table 6.3 shows that a large portion of the VCS revisions of Eclipse-core (64%), Jazz (62%), and Mozilla (38%) could be linked to ITS work item IDs. The other projects did not adopt a pattern for linking revisions to work item identifiers.

Production code work items are more tightly coupled to the build system in C projects than Eclipse-based Java ones: The Conf(Prod $\Rightarrow$ Bld) and Conf(Test $\Rightarrow$ Bld) values in Table 6.4 shows that there is considerable coupling from the production and test code to the build system in Mozilla (27% and 44%). However, the Eclipse-core and Jazz projects have less coupling. We investigate this phenomenon below.

27% of Mozilla work items that contain a related source code change also contain changes to the build system. These numbers indicate that production code and build system consistency requires considerable developer participation. However, in Eclipse-core, the coupling is reduced to 16% and in Jazz, the observed coupling is a mere 4%.

Eclipse-core and Jazz achieve lower build coupling by leveraging the automated Eclipse Plugin Development Environment (PDE) build technology called PDE build. Each Eclipse subsystem contains a build specification file "build.properties", that lists the high-level build system configuration. The PDE build parses these property files to either: (1) generate ANT scripts to perform the build appropriately, or (2) use an appropriate Eclipse plugin to perform the compilation and packaging. Since the

developer must only maintain the build.properties file, which does not contain low-level details that change frequently, the daily build maintenance overhead is reduced.

The Mozilla Conv(Test ⇒ Bld) value in Table 6.4 indicates that the logical coupling between test and build code of 44% is considerably higher than expected, while the Conv(Test ⇒ Bld) values for Eclipse-core and Jazz indicate no significant increase. The χ^2 results further reflect the significance of the Mozilla relationship with a significant p-value. The p-value for Eclipse-core indicates that the observed relationship between test and build code is not statistically significant.

By studying Mozilla at the work item level, we find that there is a substantial coupling between production and test code with the build system. We observe a 19% increase in Conf(Prod ⇒ Bld) and a 28% increase in Conf(Test ⇒ Bld) over the revision-level analysis. We observe similar increases in Eclipse-core of 13% and 17% respectively. There was little change in the observed coupling for Jazz.

The maintenance of the build system impacts both production and test development in Mozilla. The Eclipse and Jazz build code is automatically generated, resulting in reduced build system maintenance and coupling to the source code.

6.2 How do projects distribute build maintenance work?

Our study of RQ1 reveals that Mozilla developers will have to perform build changes for roughly one in every four work items they are tasked with. However, we did not consider build ownership, i.e., which developers actually make changes to the build

Table 6.5: Developer-based interest metrics.

		Jazz	Git	Linux
	All	156	795	6,502
Support	Prod	0.81	0.85	0.97
	Test	0.36	0.22	0.02
	Bld	0.73	0.22	0.26
	Prod, Bld	0.63	0.19	0.24
	Test, Bld	0.32	0.05	0.01
Conf	Prod ⇒ Bld	0.79	0.22	0.25
	Bld ⇒ Prod	0.87	0.85	0.93
	Test ⇒ Bld	0.89	0.24	0.58
	Bld ⇒ Test	0.44	0.23	0.06
Conv	Prod ⇒ Bld	1.26	1.00	0.99
	Bld ⇒ Prod	1.46	0.98	0.48
	Test ⇒ Bld	2.51	1.02	1.76
	Bld ⇒ Test	1.14	1.02	1.03
χ^2 (p-value)	Prod, Bld	0.02	1.00	<0.01
	Test, Bld	0.01	0.95	<0.01

system. Prior work reports that projects may elect to dedicate a team of experts to build maintenance tasks, e.g., the Perl interpreter [61], and the Linux kernel [2]. In those cases, although build coupling seems high, the work is delegated to build experts.

We study the relationship between production, test, and build developers by evaluating the association between them with the Support, Confidence, and Conviction "interest" metrics introduced above.

6.2.1 Approach

We label authors as a build, test, or source code developers. An author may hold one or more labels. We assume that developers who produce source code revisions are source code developers, since source code development is the main focus of a development team. Hence, we label authors as source developers if they produce at least one source code modifying revision. However, we only label authors as build developers if their personal source-build coupling is greater than or equal to the project wide source-build coupling. Similarly, we only label authors as test developers if their personal source-test coupling is greater than or equal to the project wide source-test coupling. We choose such a definition to identify those developers responsible for a significant portion of build system (and test) development.

Our study is limited to projects that retain correct author names. A common practice in open source development is to restrict VCS write access to a set of core developers [11]. Many authors send patches, i.e., files containing their changes, to the core developers for review. After engaging in a review process, the core developer will send the changes to the VCS. Only the Git, Linux, and Jazz project's VCS retain the name of the original author of the patch (instead of the core developer), so our analysis is limited to these three projects.

6.2.2 Results

We use Table 6.5 to illustrate two build ownership styles that projects adopt for maintaining the build system.

Table 6.6: Number and percentage of developers responsible for 80% of the file changes to production, test, and build files.

	Jazz	Git	Linux
Prod	41 (26%)	57 (7%)	523 (8%)
Test	58 (37%)	95 (12%)	484 (7%)
Build	53 (34%)	44 (5%)	365 (5%)

Concentrated and dispersed build ownership: We observe two patterns of build ownership:

1. Concentrated ownership (Linux and Git)

   Most build maintenance comes from a small team of build engineers

2. Dispersed ownership (Jazz)

   Most developers contribute code to the build system.

The Conf(Prod ⇒ Bld) values of Git (22%) and Linux (25%) in Table 6.5 show that the majority of source code contributors do not have to change the build system frequently. However, the χ^2 p-value shows that the coupling in Git is not statistically significant.

While the build system in the Jazz project rarely changes, the changes are made by most production and test developers. Jazz's Support(Bld) value in Table 6.4 shows that only 5% of work items require build changes, however, the Conf(Prod ⇒ Bld) values in Table 6.5 show that 79% of production code developers make a considerable number of changes to the build. Keeping the coupling between source code and build system changes at a low 5% for production ensures that although the distribution of build maintenance affects most developers, it does not affect them greatly.

Table 6.6 shows that to make 80% of all build changes, a, smaller proportion of developers are needed in Git (7%) and Linux (8%) than in Jazz (34%). This indicates that build expertise is concentrated in the Git and Linux projects whereas it is dispersed among developers in the Jazz project. Comparing the numbers to those of the production and test code, we see that the build consistently has the lowest proportion of developers that contribute 80% of the changes in the two open source projects (5%). In Jazz, we see that the build has a higher proportion of developers involved (34%) than the two open source projects.

Since most of the build changes in Linux and Git are made by a core team of build experts, contributors are saved the hassle of build maintenance. In 2001, the Linux project in particular invested time and effort into reducing the build system impact that the build system has on contributors [2]. Our findings suggest that they were successful in concentrating the maintenance of the build system onto a core team of build experts.

While we do not have the data to speculate about which style performs best universally, we conjecture that build ownership style (1) is more suitable for open source teams. Open source development depends on casual developer contributions. Casual developers will have a hard enough time learning the intricacies of the foreign source code without having to struggle with the build system. Thus, offloading the build maintenance on a core engineer seems advisable.

The studied projects adopt either a concentrated (Linux and Git), or dispersed (Jazz) build ownership style to limit the overhead of build maintenance on individual developers.

Most test developers have to make build changes: The Conf(Test ⇒ Bld) values in Table 6.5 reveal that 89% of Jazz test developers, 58% of Linux test developers, and 24% of Git test developers also make changes to the build code. This indicates that the build system maintenance is impacting most test developers in Jazz and Linux. The corresponding conviction and χ^2 values for Jazz and Linux show that these percentages are higher than expected and statistically significant. Project managers should keep this in mind when performing test development planning and budget estimations.

6.3 Chapter Summary

In analyzing the relationship between the source code and the build system in ten software projects, we make the following important observations:

- There is low revision-level coupling between the production or test code and the build system. However, there is considerable work item coupling. Developers may not commit all related code under one revision, but the related work will be filed under one work item ID. We suggest that future co-evolution studies consider analysis at the work item level, as we feel it more accurately represents the development workflow.

- A larger proportion of developers are responsible for maintaining the build system in the analyzed commercial project than in the open source ones. In the three studied projects, the actual overhead of build system maintenance is limited for individual developers.

- Many test developers must also maintain the build, meaning that the test to build code consistency management affects many people.

These findings support our research hypothesis, i.e., build system maintenance plays an important role in software development. However, in practice, projects try to mitigate the overhead by using an automated build generation framework, such as the Eclipse PDE (RQ1), or by dedicating a team of build experts to the majority of build maintenance tasks (RQ2).

In Chapter 5, we found that typical build changes add and remove 3 to 4 build lines of code. In this chapter, we find that 4–16% of source code tasks in Java projects and up to 27% of source code tasks in C projects require an accompanying build change. Project managers should explicitly account for this when performing project planning and budgeting exercises.

In Chapter 5, we also found that the build system has a high churn rate when normalized by the build system size. In this chapter, we observe that the analyzed projects have two build ownership styles for coping with this high churn rate in the build system: (1) a small team of build experts handle most of the maintenance, and (2) maintenance is dispersed amongst most developers. In future work, we plan to investigate the advantages and disadvantages of these two build ownership styles.

Chapter 7

Summary and Conclusions

What we call the beginning is often the end. And to make an end is to make a beginning. The end is where we start from.

T. S. Eliot

This chapter concludes the thesis. The concepts presented throughout this thesis are summarized and our hypothesis is resolved. The limitations and possible directions for future work are also presented.

7.1 Summary

Developers fix defects and add new features in source code to adapt software projects to changing environments and address user demands [33]. Build systems are critical to support these developer changes, since build systems automate the translation of the project source code into testable and deliverable artifacts.

However, similar to source code, build systems require maintenance in order to

keep producing project deliverables correctly and rapidly. Neglecting to keep the build system up-to-date can cause latent defects that are difficult to diagnose and can have a dramatic effect on product quality (e.g., Firefox 3.0 [59]).

To better understand the overhead induced by the build system, we study build system maintenance over time. We perform empirical studies of 13 large-scale open source and proprietary software projects to validate our research hypothesis:

Build system maintenance plays an important role in software development.

In Chapters 4 and 5, we measured the build maintenance activity of Java build systems across releases (Chapter 4), and C, C++, and Java build systems across revisions (Chapter 5). Our findings and prior research [2, 66] suggest that build systems evolve at the release and revision levels. When normalized by their respective sizes, we find that source and build components of the software project churn at similar rates. Since the build system accounts for up to 31% of the files in a software project, 9% on average, this means that the build system is susceptible to defects [45].

In Chapter 6, we studied how build maintenance is distributed across production code changes and developers. We find that up to 27% of source code work items require accompanying build changes in C projects, while 4–16% of source code work items require build changes in Java projects. Similarly, we find that up to 44% of test code work items require accompanying build changes in C projects, while 8–20% of test code work items require build changes in Java projects. Furthermore, we observe two build ownership styles adopted by projects to reduce the build maintenance overhead on individual developers: (1) dispersed, i.e., build maintenance is spread among most of the development team (observed in the Jazz project); (2) concentrated, i.e.,

build maintenance is performed by a small team of build experts.

The above evidence confirms our hypothesis, i.e., build system maintenance plays an important role in software development. Project managers, developers, and software testers should consider the overhead of build system maintenance when planning and budgeting for future releases, new features, test cases, and even bug fixes. Stakeholders should also consider adopting a build ownership style that suits their development environment.

7.2 Limitations and Future Work

The work presented in this thesis has several limitations. In this section, we outline those limitations and how future work may be designed to address them.

- The majority of the studied projects in this thesis are open source. Since open source and commercial project development styles differ, our results may be biased towards open source projects. For instance, in Jazz, our studied commercial project, build maintenance is dispersed amongst most developers, whereas in open source projects such as Linux and Git, build maintenance is managed by a core team of build engineers. More studies that analyze the build systems of commercial projects are needed.

- Our results may be biased by the studied projects. We attempted to prevent potential bias by selecting projects with different characteristics for analysis; replication of our studies using different projects may be necessary.

- We perform our dynamic analysis of build systems only at the release level because most software teams cannot guarantee their product to be buildable at

any arbitrary time in the development cycle. A release snapshot is by nature a buildable and runnable version of a project. By focusing our analysis at such a high level of granularity, we may miss development events that occur in between releases. Future work that studies the dynamic evolution of a build system at the revision level may prove useful.

- Similar to the work of Adams *et al.* [2], our dynamic analysis in chapter 4 is based on a single platform and configuration, i.e., GNU/Linux on an x86-based processor with the default configuration suggested for this platform. This decision was made to ensure that we used a consistent platform for comparison. By only exploring a single configuration, we have left areas of the build system unexplored.

- Our metric for static build system complexity is derived from the Halstead suite of complexity metrics. The Halstead complexity metrics are parametric, with different constants for different systems. Since the purpose of this thesis is not to establish a definitive measure for build system complexity, we assume that typical Halstead parametric constants apply to the domain of build systems. The striking similarities between build system languages and interpreted programming languages lead us to believe that this assumption is valid. However, our use of the Halstead metric has not been validated. Validating the use of the Halstead metrics or establishing a domain-specific static build system complexity metric may provide an interesting avenue for future work.

- Throughout the thesis, the BLOC metric measures lines of build specification code, but does not consider build task implementation code. As such, custom

ANT task implementations and custom Maven build goals do not factor into the build system size or complexity. Build task implementation code remains an unmeasured dimension of the build system size and complexity. Future work may find interesting patterns in the evolution of custom tasks in ANT build systems.

- Our file classification approach is subject to opinion and may not be 100% accurate. To classify files that may have fit in more than one category, the authors' best judgement was employed. Future work may address this by implementing a file discriminator that classified files by examining their content (first suggested by Robles *et al.* [57]).

- We make the assumption that the collected data is always correct, e.g., developers always commit necessary build changes in the same revision as the associated source code change, or all related revisions are linked to an appropriate work item. This assumption does not always hold, which introduces noise in our analysis. For instance, a developer neglects to commit all related changes under a single revision, or mistypes the work item ID in the revision message. We address noise in the revision level analysis by studying work items. However, noise in the work item level analysis is very difficult to detect and filter.

- Since the well-linked data necessary for our work item and build ownership analyses is hard to access [10, 51], our studies are limited to three projects. As such, our results may not generalize well. We selected three projects from different domains in order to combat this sort of bias in our results, yet future replication work may be necessary to further solidify our findings.

Bibliography

[1] Bram Adams, Kris De Schutter, Herman Tromp, and Wolfgang De Meuter. Design Recovery and Maintenance of Build Systems. In *Proc. of the 23rd Int'l Conf. on Software Maintenance (ICSM)*, pages 114–123, 2007.

[2] Bram Adams, Kris De Schutter, Herman Tromp, and Wolfgang De Meuter. The Evolution of the Linux Build System. *ECEASST*, 8, 2007.

[3] Rolf Adams, Walter Tichy, and Annette Weinert. The Cost of Selective Recompilation and Environment Processing. *Transactions On Software Engineering and Methodology (TOSEM)*, 3(1):3–28, January 1994.

[4] Rakesh Agrawal, Tomasz Imielinski, and Arun Swami. Mining Association Rules between Sets of Items in Large Databases. *ACM SIGMOD Records*, 22(2):207–216, 1993.

[5] Apache Software Foundation. Apache ANT Manual. `http://ant.apache.org/manual/`, 2010. Last viewed: 07-Jul-2010.

[6] Apache Software Foundation. Apache Maven. `http://maven.apache.org/`, 2010. Last viewed: 18-Mar-2010.

[7] Apache Software Foundation. Maven Migration Guide. `http://maven.apache.org/guides/mini/guide-m1-m2.html`, 2010. Last Viewed: 02-Sep-2010.

[8] Laszlo A. Belady and Meir M. Lehman. A Model of Large Program Development. *IBM Systems Journal*, 15(3):225–252, 1976.

[9] Thorsten Berger, Steven She, Rafael Lotufo, Andrzej Wasowski, and Krzysztof Czarnecki. Variability Modeling in the Real: A Perspective from the Operating Systems Domain. In *Proc. of the 25th Int'l Conf. on Automated Software Engineering (ASE)*. IEEE/ACM, 2010.

[10] Christian Bird, Adrian Bachmann, Eirik Aune, John Duffy, Abraham Bernstein, Vladimir Filkov, and Premkumar Devanbu. Fair and Balanced? Bias in Bug-Fix Datasets. In *Proc. of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Sym. on the Foundations of Software Engineering (ESEC/FSE)*, pages 121–130, 2009.

[11] Christian Bird, Peter C. Rigby, Earl T. Barr, David J. Hamilton, Daniel M. German, and Prem Devanbu. The Promises and Perils of Mining Git. In *Proc. of the 6th Working Conf. on Mining Software Repositories (MSR)*. IEEE Computer Society, 2009.

[12] Sergey Brin, Rajeev Motwani, Jeffrey D. Ullman, and Shalom Tsur. Dynamic Itemset Counting and Implication Rules for Market Basket Data. In *Proc. of the 1997 ACM SIGMOD Int'l Conf. on Management Of Data*, pages 255–264. ACM, 1997.

[13] Merijn de Jonge. Decoupling Source Trees into Build-Level Components. In J. Bosch and C. Krueger, editors, *Eighth International Conference on Software Reuse*, volume 3107 of *LNCS*, pages 215–231. Springer-Verlag, July 2004.

[14] Merijn de Jonge. Build-Level Components. *IEEE Transactions on Software Engineering*, 31(7):588–600, 2005.

[15] Mikhail Dmitriev. Language-Specific Make Technology for the Java Programming Language. In *Proc. of the 17th Annual Conf. on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)*. ACM, 2002.

[16] Eelco Dolstra. Integrating Software Construction and Software Deployment. *Lecture Notes in Computer Science*, 2649:102–117, 2003.

[17] Steve Ebersole. Maven migration. http://lists.jboss.org/pipermail/hibernate-dev/2007-May/002075.html, 2007. Last viewed: 18-Mar-2010.

[18] Stuart I. Feldman. Make-A Program for Maintaining Computer Programs. *Software - Practice and Experience*, 9(4):255–265, 1979.

[19] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, Reading, Mass, USA, 1999.

[20] Harald Gall, Karin Hajek, and Mehdi Jazayeri. Detection of Logical Coupling Based on Product Release History. In *Proc. of the Int'l Conf. on Software Maintenance (ICSM)*, pages 190–198, Washington, DC, USA, 1998. IEEE Computer Society.

[21] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-wesley Reading, MA, 1995.

[22] GNU Autotools Team. An Introduction to the Autotools. `http://www.gnu.org/software/hello/manual/automake/Autotools-Introduction.html`. Last viewed: 14-Aug-2010.

[23] GNU Development Team. GNU Compiler Collection. `http://gcc.gnu.org/`. Last viewed: 11-Apr-2010.

[24] Michael W. Godfrey and Qiang Tu. Evolution in Open Source Software: A Case Study. In *Proc. of the Int'l Conf. on Software Maintenance (ICSM)*, pages 131–140. IEEE Computer Society, 2000.

[25] Todd L. Graves, Alan F. Karr, J. S. Marron, and Harvey Siy. Predicting fault incidence using software change history. *IEEE Trans. Softw. Eng.*, 26(7):653–661, 2000.

[26] Lenz Grimmer. Building MySQL Server with CMake on Linux/Unix. `http://www.lenzg.net/archives/291-Building-MySQL-Server-with-CMake-on-LinuxUnix.html`. Last viewed: 20-Aug-2010.

[27] Maurice H. Halstead. *Elements of Software Science (Operating and Programming Systems Series)*. Elsevier Science Inc., New York, NY, USA, 1977.

[28] Ahmed E. Hassan and Ken Zhang. Using Decision Trees to Predict the Certification Result of a Build. In *Proc. of the 21st Int'l Conf. on Automated Software Engineering (ASE)*, Washington, DC, USA, 2006. IEEE Computer Society.

[29] Peter Kampstra. Beanplot: A boxplot alternative for visual comparison of distributions. *Journal of Statistical Software, Code Snippets*, 28(1):1–9, 2008.

[30] KDE developer: "mosfet". Autoconf/Automake errors in kdelibs. `http://lists.kde.org/?l=kde-core-devel&m=95953244511288&w=4`. Last viewed: 18-Aug-2010.

[31] Steven Knight. SCons Design and Implementation. In *Tenth Int'l Python Conf.*, 2002.

[32] Gary K. Kumfert and Tom G. W. Epperly. Software in the DOE: The Hidden Overhead of "The Build". Technical Report UCRL-ID-147343, Lawrence Livermore National Laboratory, CA, USA, February 2002.

[33] Meir M. Lehman. On Understanding Laws, Evolution and Conservation in the Large Program Life Cycle. *Journal of Systems and Software*, 1(3):213–221, 1980.

[34] Meir M. Lehman, Dewayne E. Perry, Juan F. Ramil, Wladyslaw M. Turski, and Paul D. Wernick. Metrics and Laws of Software Evolution – The Nineties View. In *Proc. of the 4th Int'l Software Metrics Symposium (METRICS)*, 1997.

[35] Linden Labs. CMake. `http://wiki.secondlife.com/wiki/CMake`, July 2010. Last viewed: 20-Aug-2010.

[36] Ken Martin and Bill Hoffman. *Mastering CMake, 5th Edition.* Kitware Inc., Clifton Park, NY, USA, 2009.

[37] Thomas J. McCabe. A Complexity Measure. In *Proc. of the 2nd int'l conf. on Software engineering (ICSE)*, page 407. IEEE Computer Society Press, 1976.

[38] Steve McConnell. *Code Complete, 2nd Edition*. Microsoft Press, 2004.

[39] Shane McIntosh, Bram Adams, and Ahmed E. Hassan. The evolution of ANT build systems. In *Proc. of the 7th working conf. on Mining Software Repositories (MSR)*, pages 42–51. IEEE Computer Society, 2010.

[40] Shane McIntosh, Bram Adams, and Ahmed E. Hassan. The Evolution of Build Systems for Java Projects (Under review). *Empirical Software Engineering*, 2011.

[41] Shane McIntosh, Bram Adams, Thanh H. D. Nguyen, Yasutaka Kamei, and Ahmed E. Hassan. An Empirical Study of Build Maintenance Effort. In *Proc. of the 33rd Int'l Conf. on Software Engineering (ICSE) (To appear)*. ACM Press, 2011.

[42] Andrew Miller. js/Makefile.in gone but still in allmakefiles.sh. `https://bugzilla.mozilla.org/show_bug.cgi?id=351377`. Last viewed: 18-Aug-2010.

[43] Peter A. Miller. Recursive Make Considered Harmful. In *Australian Unix User Group Newsletter*, volume 19, pages 14–25, 1998.

[44] Mozilla Foundation. Mozilla communications suite. `http://www.mozilla.com/`. Viewed on: 11-Apr-2010.

[45] Nachiappan Nagappan and Thomas Ball. Use of Relative Code Churn Measures to Predict System Defect Density. In *Proc' of the 27th int'l conf. on Software engineering (ICSE)*, pages 284–292, New York, NY, USA, 2005. ACM.

[46] Nachiappan Nagappan and Thomas Ball. Using Software Dependencies and Churn Metrics to Predict Field Failures: An Empirical Case Study. In *Proc. of the 1st Int'l Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 364–373, Washington, DC, USA, 2007. IEEE Computer Society.

[47] Adrian Neagu. What is Wrong with Make. `http://freshmeat.net/articles/what-is-wrong-with-make`, 2010. Last viewed: 26-Feb-2010.

[48] Alexander Neundorf. Why the KDE project switched to CMake – and how (continued). `http://lwn.net/Articles/188693/`, 2010. Last viewed: 06-Mar-2010.

[49] Thomas Neustupny. Build failed in Hudson, what to do? `http://argouml.tigris.org/ds/viewMessage.do?dsForumId=450&dsMessageId=2618367`. Last viewed: 18-Aug-2010.

[50] George V. Neville-Neal. Kode Vicious: System Changes and Side Effects. *Communications of the ACM*, 52(4):25–26, April 2009.

[51] Thanh H. D. Nguyen, Bram Adams, and Ahmed E. Hassan. A case study of bias in bug-fix datasets. In *International Conference in Software Maintenance*, page Accepted, Beverly, Massachusetts, 2010.

[52] Glenn Niemeyer and Jeremy Poteet. *Extreme Programming with Ant: Building and Deploying Java Applications with JSP, EJB, XSLT, XDoclet, and JUnit.* Sams, first edition edition, May 2003. ISBN-0672325624.

[53] Perl Development Team. Perl Scripting Language. `http://www.perl.org/`. Last viewed: 11-Apr-2010.

[54] Pier P. Fumagalli. Pain building... http://marc.info/?l=tomcat-dev&m=97537754000329&w=2. Last viewed: 2-Jan-2011.

[55] John A. Rice. *Mathematical Statistics and Data Analysis*. Duxbury press, 1995.

[56] Gregorio Robles, Juan J. Amor, Jesus M. Gonzalez-Barahona, and Israel Herraiz. Evolution and Growth in Large Libre Software Projects. In *Proc. of the Int'l Workshop on Principles of Software Evolution (IWPSE)*, pages 165–174, 2005.

[57] Gregorio Robles, Jesus M. Gonzalez-Barahona, and Juan J. Merelo. Beyond Source Code: The Importance of Other Artifacts in Software Development (A Case Study). *Journal of Systems and Software (JSS)*, 79(9):1233–1248, 2006.

[58] Roy Wilson. Make rant. http://marc.info/?l=tomcat-dev&m=97412077403986&w=2. Last viewed: 2-Jan-2011.

[59] Tim Steiner. mozStorage chokes on databases over AFP. https://bugzilla.mozilla.org/show_bug.cgi?id=417037. Last viewed: 18-Aug-2010.

[60] Andrew Sutton and Jonathan I. Maletic. How We Manage Portability and Configuration with the C Preprocessor. In *Proc. of the 23rd Int'l Conf. on Software Maintenance (ICSM)*. IEEE Computer Society Press, 2007.

[61] Qiang Tu and Michael W. Godfrey. The Build-Time Software Architecture View. In *Proc. of IEEE Int'l Conf. on Software Maintenance (ICSM)*, pages 398–407. IEEE Computer Society, 2002.

[62] Tijs van der Storm. The Sisyphus Continuous Integration System. In *Proceedings of the 11th Conference on Software Maintenance and Reengineering (CSMR)*, Amsterdam, The Netherlands, 2007. IEEE Computer Society Press.

[63] David A. Wheeler. Sloccount. `http://www.dwheeler.com/sloccount/`, 2010. Last viewed: 26-Feb-2010.

[64] Yijun Yu, Homayoun Dayani-Fard, John Mylopoulos, and Periklis Andritsos. Reducing Build Time Through Precompilations for Evolving Large Software. In *Proc. of the 21st Int'l Conf. on Software Maintenance (ICSM)*. IEEE Computer Society Press, 2005.

[65] Yijun Yu, Homy Dayani-Fard, and John Mylopoulos. Removing False Code Dependencies to Speedup Software Build Processes. In *Proc. of the 13th Int'l Conf of the IBM Center for Advanced Studies (CASCON)*. IBM, 2003.

[66] Erez Zadok. Overhauling Amd for the '00s: A Case Study of GNU Autotools. In *Proc. of the FREENIX Track on the USENIX Annual Technical Conf.*, pages 287–297, Berkeley (CA, USA), 2002. USENIX Association.