The evolution of Java build systems

Article *in* Empirical Software Engineering · August 2011

DOI: 10.1007/s10664-011-9169-5

CITATIONS
50

READS
1,152

3 authors, including:

Bram Adams
Polytechnique Montréal
199 PUBLICATIONS **5,998** CITATIONS

SEE PROFILE

Ahmed E. Hassan
Queen's University
427 PUBLICATIONS **14,900** CITATIONS

SEE PROFILE

Noname manuscript No.
(will be inserted by the editor)

The Evolution of Java Build Systems

Shane McIntosh · Bram Adams · Ahmed E. Hassan

the date of receipt and acceptance should be inserted later

Abstract Build systems are responsible for transforming static source code artifacts into executable software. While build systems play such a crucial role in software development and maintenance, they have been largely ignored by software evolution researchers. However, a firm understanding of build system aging processes is needed in order to allow project managers to allocate personnel and resources to build system maintenance tasks effectively, and reduce the build maintenance overhead on regular development activities. In this paper, we study the evolution of build systems based on two popular Java build languages (i.e., ANT and Maven) from two perspectives: (1) a static perspective, where we examine the complexity of build system specifications using software metrics adopted from the source code domain; and (2) a dynamic perspective, where the complexity and coverage of representative build runs are measured. Case studies of the build systems of six open source build projects with a combined history of 172 releases show that build system and source code size are highly correlated, with source code restructurings often requiring build system restructurings. Furthermore, we find that Java build systems evolve dynamically in terms of duration and recursive depth of the directory hierarchy.

Keywords Build Systems · Software Evolution · ANT · Maven · Software Complexity

1 Introduction

Software build systems are responsible for automatically transforming the source code of a software project into a collection of deliverables, such as executables and development libraries. A build process may involve hundreds of command invocations that must be executed in a specific order to correctly produce a set of deliverables. First, 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. The

Shane McIntosh · Bram Adams · Ahmed E. Hassan
Software Analysis and Intelligence Lab (SAIL)
Queen's University
E-mail: {mcintosh, bram, ahmed}@cs.queensu.ca

requirements and constraints of build tools and software features are derived from specifications written in a configuration language [10]. Second, a construction tool like `make` or ANT constructs the configured build deliverables in the correct order by observing the dependencies in the build specification files (e.g., `makefile` or ANT build.xml files).

Build systems play a key role in the software development process. They simplify the lives of developers, who constantly need to re-build testable artifacts after completing a code modification. Build systems also play a key role in team coordination. For example, the continuous integration development methodology requires automatic execution of project builds and publication of results via email or web sites to provide direct feedback to developers about software quality [1]. Maintaining a fast and correct build system is pivotal to the success of modern software projects.

Unfortunately, build systems require substantial maintenance effort. Kumfert *et al.* find that on average, build systems induce a 12% overhead on development effort [20], i.e., 12% of development effort is spent maintaining the build system. Build maintenance can involve the addition of build rules to accommodate new source code modules, or adjustments to the configuration of compiler/linker flags. Build maintenance drives the evolution of build specification files in many projects. For example, the Linux build engineers went out of their way to make integration of new code into the build process trivial to encourage contributions. The core build machinery, which is hidden behind an intricate facade, has evolved into a highly complex build system that requires considerable effort to maintain [3]. As a second example, the maintenance of the build system in the KDE 3 project was such a burden that it drastically impacted the productivity of KDE developers, and even warranted migration to a new build technology, requiring a substantial investment of effort [30]. To most developers, it would appear that as a build system ages, only its perceived complexity increases, i.e., builds consume more computational resources and take longer to complete.

Despite the crucial role of build systems and their non-trivial maintenance effort, software engineering research rarely focuses on them. Initial findings have shown that the size and complexity of build systems grow over time [3, 36]. However, this evolution primarily has been studied in `make`-based build systems for C projects. Little is known about build specifications for Java projects.

Similar to C and C++ code, Java code must be compiled and bundled before it can be delivered to end users. Hence, Java projects also require build systems to translate source code files into a deliverable format. Due to improvements in the compiler behaviour, the Java compiler can resolve dependencies at compile time [11], while most C and C++ compilers cannot. Hence, we suspect that Java build systems evolve differently than C and C++ build systems studied in prior work [3, 36].

In this paper, we present an empirical study of traditional source code evolution phenomena in six Java build systems. We address the following two research questions:

RQ1) Do the size and complexity of source code and build system evolve together?
Our static code analysis of build system specifications shows not only 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 perceived build-time complexity evolve?
Our build-time analysis of Java build systems did not reveal a common pattern in the build-time length of the studied projects, although for some projects we

observe linear growth or other trends in build-time length, recursive depth, and build coverage.

This paper is an extended version of our earlier work [24]. The original work provides:

- An empirical study of the evolution of ANT build systems in four small-to-large open source systems;
- A definition of the Halstead suite of complexity metrics for the domain of build systems;
- Evidence of high correlation between the evolution of ANT build systems and the source code.

We extend this work to provide an empirical study of the evolution of Maven build systems in two medium-sized open source projects. Maven is a different Java build technology that is gaining momentum recently. In addition, we further expand the Halstead suite of complexity metrics to the domain of Maven build systems. Furthermore, we draw comparisons between the ANT and Maven build system evolution, and extrapolate higher-level comparisons between Java and C build systems.

The remainder of the paper is organized as follows. Section 2 introduces the ANT and Maven build languages and associated terminology. Section 3 elaborates on the research questions that we address. Section 4 discusses the methodology for the case studies that we conducted on six open source systems, while Sections 5 and 6 present the results. Section 7 draws comparisons between the evolution of ANT versus Maven build systems, and build systems for C versus Java projects. Section 8 discusses the threats to validity. Section 9 surveys related work. Finally, Section 10 draws conclusions.

2 Background

We first provide an overview of build system concepts in general, then introduce the ANT and Maven build languages for Java projects, which are the focus of our study.

2.1 Build System Concepts

A typical build system consists of two major layers [2]. The *configuration layer* allows a user or developer to select code features, compilers, and third-party libraries to use during the build process, and enforces any constraints or conflicts between these configuration options. The configuration layer may automatically detect a default set of configuration options by scanning the build environment, but these default values can be overridden by the user. In this paper, we ignore the configuration layer and assume that the default set of configuration options has been selected (similar to Adams *et al.* [2]).

The *construction layer* considers the configuration options that were selected by the user and parses the build specification files to determine the necessary build tasks and the order in which they must be executed to produce the desired build output. Construction layer (or build) specifications are typically expressed in a build system language. Among build languages, popular choices include `make` [13], ANT [5], and Maven [6]. Table 1 compares the features of these build languages. We discuss ANT and

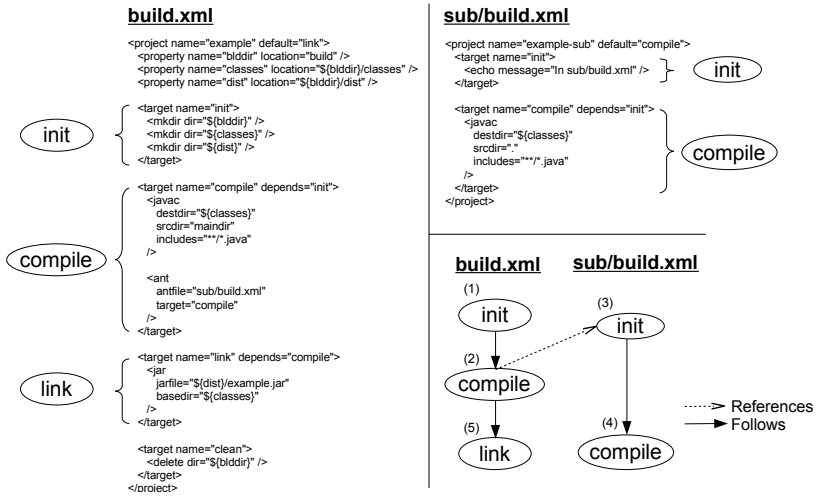

Fig. 1 Example ANT build.xml files (left, top-right) and the resulting build graph (bottom-right). The build graph has a depth of 2 (i.e., "compile" in build.xml references "init" in sub/build.xml) and a length of 5 (i.e., execute (1), (2), (3), (4), then (5)).

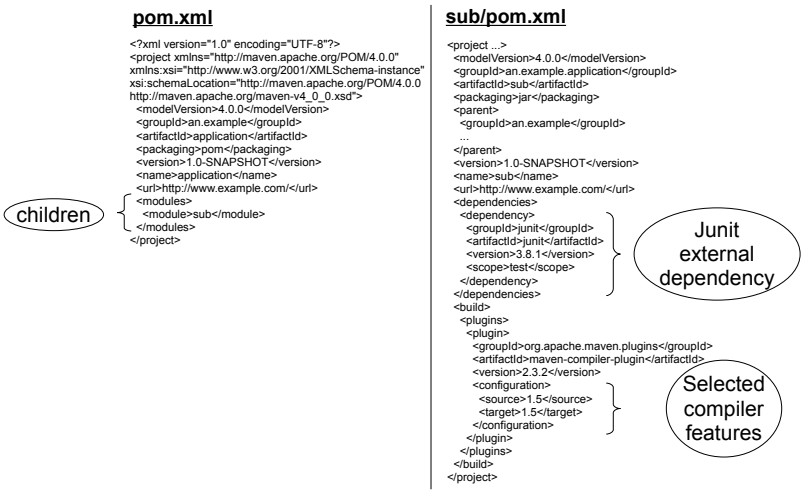

Fig. 2 Example Maven pom.xml files.

Maven in the next subsections, but first we briefly explain the general build language concepts.

Every build language depends at the lowest level on a unit of build activity. For instance, in `make`, these units are invocations of concrete build commands, such as shell scripts, compilers, and other tools. These units of build activity are typically encapsulated in an abstraction to describe a high-level build task, e.g., the construction of one library in the system, or a specific conceptual task like "preprocessing all files". In `make`, this abstraction corresponds to build rules that specify how to construct build targets like executables, dynamic libraries or perform more abstract tasks. Finally the build process is defined in terms of dependencies between the abstractions. For example,

Table 1 Build concepts.

	`make`	ANT	Maven
Build specifications	`Makefiles`, *.mk	build.xml	pom.xml
Unit of build activity	Build commands inside build rule.	ANT tasks inside ANT targets.	Goals bound to a particular phase.
Unit of abstraction	Build rules specifying how to build particular build target (file or abstraction).	Custom ANT targets specifying how to perform a conceptual build activity.	Standardized Maven phases specifying how to perform a conceptual build activity.
Dependency management	A target is only rebuilt if one or more of its dependent targets have been rebuilt.	A target is only rebuilt if one or more of its dependent targets have been rebuilt.	A fixed sequence of phases.

a `make` target is only built if at least one target on which it depends has changed. The abstractions and their dependencies in the build system form a directed acyclic graph ("build graph"), which is the basis for most build languages [2].

2.2 ANT

This paper studies the evolution of open source build systems implemented in the ANT build language. 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, which was and still is the de facto standard among build system languages for C and C++ projects [29]. Although `make` pioneered many build system concepts, there are serious flaws in its design, such as the inherent platform dependence of commands inside the `make` build rules and the common recursive architecture found in many `make` build systems [27]. To resolve these flaws, ANT was designed to be small, extensible, and operating system independent. Still, many of the concepts introduced by `make` survive in ANT. An example ANT specification file and the resulting build graph are shown in Figure 1.

An ANT build system is specified by a collection of XML files. <**project**> tags contain all of the code related to a software project. <**target**> tags correspond to build targets (unit of abstraction) we explained above. Such a <target> is responsible for conceptual build activities like "compile all source files" ("compile" target in Figure 1) or "collect all class files in a jar archive" ("link" target in Figure 1). <targets>s are essentially sequences of <**task**> tags (unit of build activity) that specify for example how to "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 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. This enables the <task>s to be imple-

mented in a platform-independent way, in contrast to the shell scripts and tools used by `make`.

Similar to targets in `make` build systems, ANT targets "depend" on one another. These dependencies can be modeled as a build graph consisting of length and depth dependencies. For instance, consider the build graph shown in the bottom-right section of Figure 1. 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. 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 build graph shown in Figure 1 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 Maven

This paper also studies the evolution of open source build systems implemented with Apache Maven [6]. Maven was created with build process standardization in mind, since many Java projects of the Apache foundation had to re-implement the same ANT targets and tasks over and over again. These common build activities were consolidated into the dedicated concept of a Maven *Build Lifecycle*. A lifecycle is composed of one or more sequential phases in a fixed order (imposed by the Maven designers). 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 phase may contain zero or more sequential goals. A phase without goals is skipped during the build, whereas those with one or more goals bound are executed in the order dictated by the build lifecycle. 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 as a Maven plugin.

The default build lifecycle is composed of 23 phases [8]. Typically, not all 23 phases are needed for a particular project. Instead, the build engineers select a subset of the 23 phases, or for certain types of deliverable (e.g., JAR or WAR), Maven already provides a default set of phases to accelerate build system development. For example, Figure 2 shows an example Maven build system with a module "sub" that is built using JAR packaging (sub/pom.xml line 5). Hence, when building this module, maven will execute the default build lifecycle for JAR packaging projects as shown in Table 2. All of the studied Maven projects use JAR packaging, so we do not discuss the lifecycles of other packaging types in the paper.

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 (not configured by default).

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, typically opting to either (1) commit the exact versions of the libraries into the project's Version Control System (VCS), or (2) download them automatically using hard-coded ANT targets. Maven provides support

Table 2 The Maven default lifecycle for JAR packaging.

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.

for specifying required versions and maintaining them in a local cache repository for use in all Maven-built projects.

3 Research Questions

Software evolution is concerned with the aging process of source code. For example, Lehman *et al.* established the laws of software evolution, which suggest that as software ages, it increases in size and complexity [9, 21, 22]. Godfrey *et al.* found that the Linux source code grows super-linearly in size and complexity [14].

We conjecture that build systems also evolve in terms of size and complexity. Understanding this evolution is important, since the build system plays a critical role in the software development process. Indeed, most software development stakeholders interact with the build system [31]. Developers use a build system to run a software system after adding a new feature or fixing a defect. Software testers use the build system to automate the validation of deliverables by executing unit and integration tests during the build process. Development methodologies based on continuous integration depend on a fast and correct build system in order to swiftly deliver reports on the current status of the project source code. For all of these stakeholders, maintaining a correctly functioning build system is of the utmost importance.

This paper improves the understanding of build system evolution by studying the evolution of Java build systems. Prior work focused exclusively on C and C++ build systems. For example, Adams *et al.* found initial evidence of increasing complexity in the Linux kernel build dependency graphs [3]. Zadok also found evolving complexity in the Berkeley Automounter build system [36], measured in terms of lines of build code and the number of conditionally compiled code branches.

Similar to C and C++ code, Java code must be compiled and bundled 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 dependencies between C files traditionally can only be managed by external dependency management tools like `make`, requiring separate compiler invocations per source code file; and (2) Java compiler invocations are expensive, since the Java Virtual Machine (JVM) must be

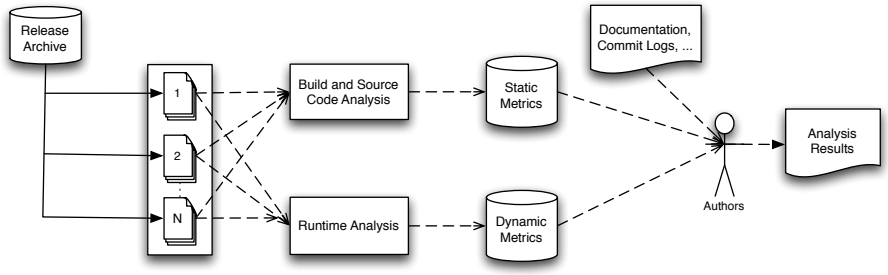

Fig. 3 Overview of our approach to study the evolution of build systems.

started before and shut down after each invocation [11]. Both compiler differences result in a reduction of the number of Java compiler invocations needed to build a Java system. Based on this, we conjecture that Java build systems are not only specified in a different manner, and hence, evolve differently than C build systems, but also that Java build systems should require less maintenance than C build systems. To validate these claims, this paper focuses on the following two research questions:

RQ1) Do the size and complexity of source code and build system evolve together?
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, and (2) contrast the evolution of build systems against the evolution of source code. Since traditionally, evolution studies typically only considered complexity metrics for source code, we first need to define specialized complexity measures for build systems.

RQ2) Does the perceived build-time complexity evolve?
Developers and users often complain about how long their build takes and how the build seems to become slower over time. We are interested in investigating whether this perceived build-time complexity indeed exhibits evolutionary trends. That is, how much build code is routinely exercised and how long does a typical build take across different releases of a software project.

4 Methodology

To address the two research questions, we track the evolution of software build systems for different releases of six open source projects. There are some existing metrics from the source code domain that we can use, 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 3. We now explain each step of our approach.

4.1 Data Retrieval

We consider official software releases of a project as the level of granularity for our analysis. While no software team can guarantee their product to be buildable at all

Table 3 Metrics used in our build system analysis. The examples in Figures 1 and 2 are used to provide example calculations for ANT and Maven.

Group	Metric	Description	Example (ANT)	Example (Maven)
Static	Build Lines of Code (BLOC)	The number of non-empty lines of code in build specification files.	$30 + 12 = 42$	$16 + 34 = 50$
	Target Count	The number of build targets in the build specification files.	$4 + 2 = 6$	N/A
	Task Count	The number of task invocations in the build specification files.	$7 + 2 = 9$	N/A
	File Count	The number of specification files in the build system.	2	2
	Halstead Complexity	The amount of information contained in the build system (Volume).	$(15 + 24) \times log_2(10 + 10) \approx 168.6$	$(35 + 33) \times log_2(20 + 18) \approx 356.9$
		The mental difficulty associated with understanding the build system specification files (Difficulty).	$\frac{10}{2} \times \frac{24}{10} = 12$	$\frac{19}{2} \times \frac{33}{18} \approx 17.4$
		The weighted Difficulty with respect to Volume (Effort).	$168.6 \times 12 \approx 2,023.2$	$356.9 \times 17.4 \approx 6,210.1$
Dynamic	Build Graph Length	The length of a build graph, either in terms of the total number of executed tasks (fine-grained) *or* of the total number of executed targets (coarse-grained).	Targets: $3 + 2 = 5$; Tasks: $6 + 2 = 8$	7
	Build Graph Depth	The maximum level of depth references made in chains of references from one build target to another in different build specification files.	2	2
	Target Coverage	The percentage of targets in the build system that are exercised by a given build target.	Default: $\frac{5}{6} \approx 83.3\%$	N/A
	Dynamic Build Lines of Code (DBLOC)	The percentage of code in the build system that is exercised by a given build.	Default: $\frac{38}{42} \approx 90.5\%$	N/A

times in the development cycle, an official software release is by nature a buildable and runnable version of a project. This decision is critical for our dynamic build analysis in RQ2.

For each project, a collection of official source code releases were retrieved. These releases were downloaded from the official release archives, except for the ArgoUML and Hibernate data, which were retrieved from the project Version Control System

Table 4 Studied Projects

	ArgoUML	Tomcat	JBoss	Eclipse	Hibernate	Geronimo
Domain	UML Editor	Web Container	App Server	IDE	ORM	App Server
Source Size (KSLOC)	≤ 176	≤ 277	≤ 731	$\leq 2,900$	≤ 328	≤ 219
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

(VCS). The released versions of ArgoUML and Hibernate were marked in the VCS with annotated tags.

4.2 Evolution Metrics

In our study, we use various static and dynamic metrics to quantify a wide variety of build system characteristics across the releases. The metrics are summarized in Table 3. BLOC, build target/task/file count, and Halstead complexity are gathered statically. Dynamically, build system content is measured using 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 [3]. Build graph depth and target coverage are new metrics proposed by this study. Some metrics only apply to ANT build systems.

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. To measure the complexity of build files, we adapt a source code metric, because build specification files share many similarities with source code implemented in an interpreted programming language. Case in point, the SCons [19] and Rake [17] build languages are entirely based on the Python and Ruby programming language, respectively. With this in mind, we conjecture that build system complexity can be measured by applying 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 paper, we only focus on the Halstead suite of complexity metrics [16]. In future work, we plan to examine the McCabe cyclomatic complexity [23] and how it applies to build systems, although results of our case study indicate that (similar

to source code [15, 32]), size metrics already provide a good approximation of build system complexity.

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

- **Volume**: How much information a reader has to absorb in order to understand a program's meaning.
- **Difficulty**: How much mental effort a reader must expend to create a program or understand its meaning.
- **Effort**: How much mental effort would be required to recreate a program.

Each Halstead metric depends on four tally metrics that are based on source code 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 (excluding the target "name" parameter; e.g, 'b' in <a b=c>) or to any child tag in Maven.

We tally both the number of distinct operators and operands in the build code (n1 and n2), as well as the total number of operators and operands in the build code (N1 and N2). The tallies with the '1' suffix represent the number of operators, and the tallies with the '2' suffix represent the number of operands. These values are then used to calculate the Halstead volume, difficulty, and effort as follows [16]:

$$\text{Volume} = (N1 + N2) \times \log_2(n1 + n2) \tag{1}$$

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

$$\text{Effort} = \text{Difficulty} \times \text{Volume} \tag{3}$$

Table 3 shows calculations of the Halstead metrics for the ANT and Maven examples in Figures 1 and 2. Here we briefly discuss how we arrived at the N1, N2, n1, and n2 values.

For the ANT example in Figure 1, there are 5 targets and 10 tasks in the two build.xml files ($N1 = 5 + 10 = 15$). Together, these targets and tasks have 24 operands ($N2 = 24$). There are 4 distinct targets and 6 tasks ($n1 = 4 + 6 = 10$). Finally, there are 10 distinct operands ($n2 = 10$).

For the Maven example in Figure 2, there are 10 XML tags in pom.xml and 25 XML tags in sub/pom.xml ($N1 = 10 + 25 = 35$). 33 of the 35 XML tags are child tags ($N2 = 33$). 20 of the XML tags are distinct ($n1 = 20$), while 18 child XML tags are distinct ($n2 = 18$).

4.3 Analysis Methodology

As suggested by our choice of metrics in Table 3, we analyze each release using two perspectives. For RQ1, build system files and program source files of each release are examined statically. SLOC was measured using David A. Wheeler's `sloccount` utility [35]. To measure static build metrics such as target count, task count, and the Halstead complexity of build system specification files, we developed a SAX-based

Java tool. 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`.

For RQ2, the build system of each release was exercised using the default build configuration and the results were logged, similar to Adams *et al.* [2]. 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, our Java tool was extended 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. To facilitate future work, we have made the raw ANT and Maven build logs available on the web [25].

Developers may implement customized ANT tasks using the Java-based ANT API. While such .java files are technically maintained as part of the build system, they specify the internals of a custom task, comparable to the implementation of a shell script invoked by makefiles. Hence, they should not simply be aggregated with the ANT build specifications, which are task-agnostic. We elect to leave custom task implementations out of this study, but do plan to revisit the problem in future work.

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.

4.4 Studied Projects

We selected six open source projects of different size, domain, build technology and release style. Table 4 summarizes the characteristics of the projects, ranked from small to large.

ArgoUML is a Computer Aided Software Engineering (CASE) tool for producing Unified Modelling Language (UML) diagrams. Tomcat is a 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. Hibernate is an Object-to-Relational mapping framework for Java programs, of which we studied the "core" subsystem. Geronimo is a Java 2 Enterprise Edition (J2EE) application server runtime environment.

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). Maven is a newer build technology that is starting to gain momentum. Thus, there is less data available for analysis.

5 ANT Case Study

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

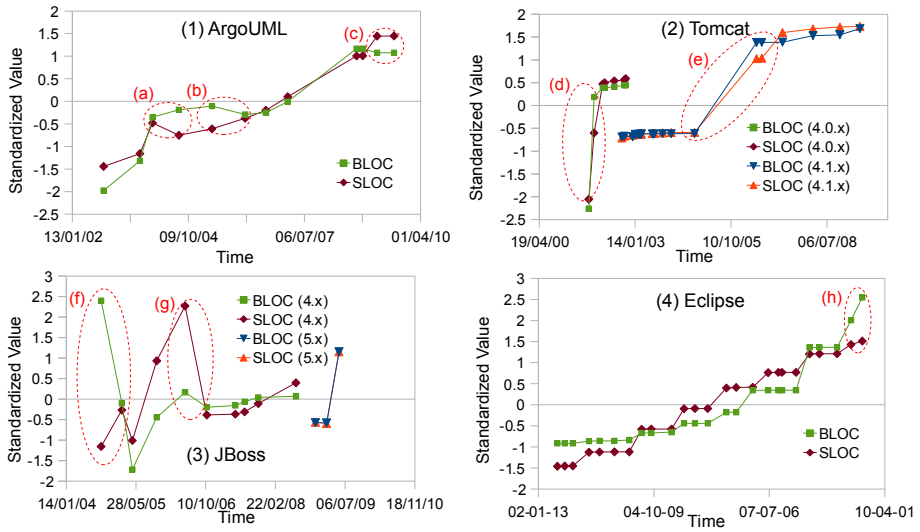

Fig. 4 Standardized BLOC and SLOC values. In most projects, the source code and build system evolution trends are very similar. Anomalies are discussed in the text.

Table 5 Correlation of static size metrics (**A**rgoUML, **T**omcat, **J**Boss, and **E**clipse). P-values are mostly <0.01, except for some of the low correlations shown in bold, whose p-values are shown in parentheses.

		Task Count				**File Count**			
		A	T	J	E	A	T	J	E
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**			
		A	T	J	E	A	T	J	E
Target Count	0.98	0.99	**0.40** (0.05)	0.98	0.98	0.97	0.89	0.95	
Task Count	1.00	1.00	**0.15** (0.47)	1.00	0.95	0.97	**0.78**	0.99	
File Count	0.94	0.99	**0.59**	0.99	0.96	0.98	0.88	0.98	
BLOC					0.94	0.98	**0.40** (0.05)	0.99	

RQ1) *Do the size and complexity of source code and build system evolve together?*

We explored the evolution of ANT build system specification files from three angles. First, we use Figure 4 to show a general trend of increasing size in the four projects, then we use Table 5 and 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 and Table 5 again to show that the build system and source code evolve similarly in terms of size.

ANT Build systems grow in size: In Figure 4, 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). 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, measured in units of standard deviation (i.e., $Y = \frac{n-\mu}{\sigma}$, where n is the size in BLOC or SLOC, μ is the mean size in BLOC

or SLOC, and σ is the standard deviation). In projects with parallel releases, i.e., two or more release branches supported simultaneously, we standardized 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 shows a clearly increasing trend with the exception of one period in between releases 0.18.1 and 0.20 (Figure 4(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 these 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 increase in BLOC (Figure 4(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 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(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.

Refactoring efforts at Figure 4(f) and Figure 4(g) skew the first half of the results in JBoss, which otherwise has an increasing trend in BLOC. During Figure 4(f), an entire rewrite of the enormous "testsuite" build specification file resulted in the removal of approximately 5,000 BLOC. During Figure 4(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 shows that the Eclipse build system is growing in terms of BLOC. Further inspection of the trend using an exponential regression (Figure 5) suggests that the Eclipse build system is growing exponentially (with an R^2 value of 0.98). 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 each introduce more build code. We calculated the Pearson correlation between the number of plugins in each release and BLOC to be 0.99. This suggests that the exponentially rising trend in build system size strongly correlates with the trend in the number of plugins per release. ArgoUML and Tomcat 4.1.x appear to have a linear growth trend with R^2 values of 0.86 and 0.91 respectively. JBoss and Tomcat 4.0.x do not have clear linear or exponential regression trends, since the R^2 values are less than 0.5.

Table 6 Pearson correlation between Halstead Metrics (Rows) and BLOC (Columns). P-values are mostly <0.01, except for some of the low correlations shown in bold, whose p-values are shown in parentheses.

	ArgoUML	**Tomcat**	**JBoss**	**Eclipse**
Volume	0.99	1.00	**0.17** (0.43)	1.00
Difficulty	0.98	0.99	**0.20** (0.34)	1.00
Effort	0.93	0.98	**0.11** (0.61)	0.96

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

All dimensions of ANT build systems grow: Table 5 shows the Pearson correlation (and p-value statistic) between the static size metrics for each studied system. With the exception of the JBoss project, which will be explained later, the high correlation and statistically significant p-values with $p < 0.05$ 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.

The static complexity of ANT build systems increases: We find that the Halstead complexity metrics follow trends similar to BLOC. Table 6 shows, for each studied system, the Pearson correlation between each Halstead complexity metric and the BLOC. With the exception of the JBoss project, the results indicate that build specification complexity is highly correlated with build specification size (BLOC) and the values are statistically significant with $p < 0.05$. This finding seems to agree with similar findings from research in the source code domain [15, 32].

In the JBoss build system, the Halstead complexity metrics and build system size are not highly correlated, as the JBoss build system is implemented in a different style. 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 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 growth are highly correlated: Based on our observations of size and complexity trends, we are now able to verify whether growth periods of the build system coincide with growth periods of the source code. For each project, we: (1) calculated the Pearson correlation between BLOC and SLOC; and (2) visually compare the trends of BLOC and SLOC in Figure 4.

Table 5 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.

The correlation between the growth in BLOC and SLOC for the four subject systems is illustrated in Figure 4. In most cases, the characteristics of the source code and build specification curves are very similar, which suggests that BLOC and SLOC

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(a), (b), and (c). During Figure 4(a), a refactoring 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 the period encircled in Figure 4(b), C# code generation and internationalization modules were migrated from the main ArgoUML repository to 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 project source code. During Figure 4(c), another refactoring 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(d) and (e) 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(f) interval, the system source code was unaffected and hence was subject to the standard growth. The build system size apparently reached such a critical point that explicit steps were taken to restructure the build system. During Figure 4(g), JAX-RPC support was moved out of the main JBoss project and as a result, the SLOC reduced by 72 KSLOC. These 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(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.

RQ2) *Does the perceived build-time complexity evolve?*

We study the evolution of perceived build system complexity from three angles. First, we use Figure 6 to show growth of build graph length and depth in the four studied build systems, then we use Table 7 to examine the build recursion complexity, and finally we analyze changes in target coverage, which is a measure of perceived build system complexity.

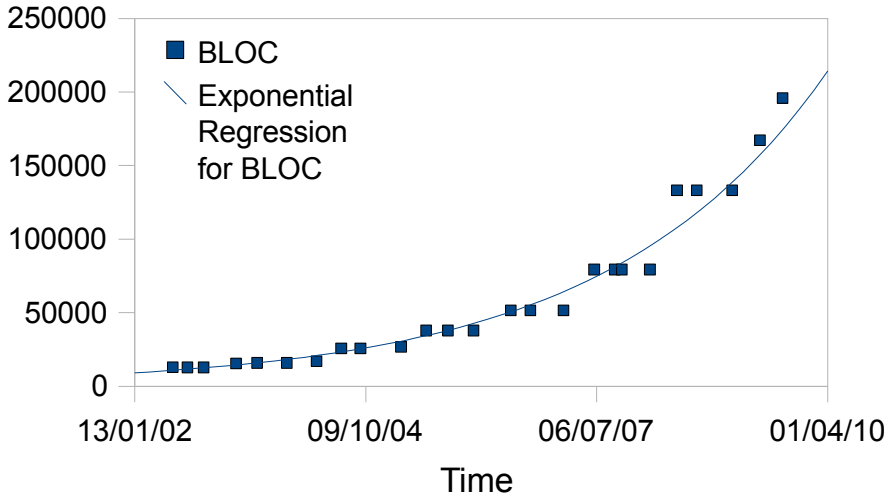

Fig. 5 The exponential trend in Eclipse BLOC. The trend line has an R^2 value of 0.98.

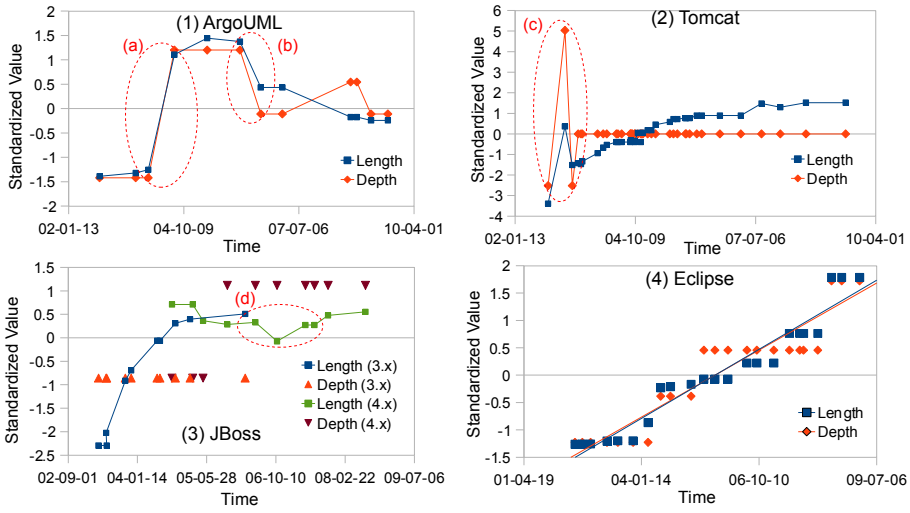

Fig. 6 Standardized build graph dimensions. Build graph length (in targets) and depth have an R^2 value of 0.94 and 0.88 in Eclipse.

We elected to exercise the build target that generates all deliverable jar files in each project, conventionally called the "all" target. We selected the "all" target, since it is available in each of the studied projects, and has the same meaning in each project.

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 6(a), ArgoUML shows a large change in both dimensions of the build graph. This was caused by the introduction of new internationalization and unit test compilation targets that became part of the default build. The Figure 6(b) interval corresponds to the Figure 4(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 6(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 6(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 code instead of downloaded pre-built. 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.

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 (with R^2 values of 0.94 and 0.88 respectively). The correlation between these two dimensions is discussed below.

> *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 6 shows two distinct trends in the build graph depth: (1) a near-constant depth (Tomcat and JBoss), and (2) a depth that seems to vary in trends similar to build graph length (ArgoUML and Eclipse). Table 7 shows the Pearson correlation (and p-values) 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. We also find that once a recursive or non-recursive design has been selected, the project maintains the design and does not change.

As the Eclipse project ages, the maximum depth of recursion reached during its build process increases. This implies that as the project ages, the build process actually grows linearly 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. Both JBoss and Tomcat make

Table 7 Pearson Correlation between Dynamic Metrics (Rows) and Build Graph Depth (Columns).

	ArgoUML		**Tomcat**		**JBoss**		**Eclipse**	
	Cor.	p-val	Cor.	p-val	Cor.	p-val	Cor.	p-val
Elapsed Time	**0.37**	**0.24**	0.14	**0.40**	**0.40**	**0.70**	0.92	<0.01
Build Graph Length (Targets)	0.92	<0.01	**0.37**	0.02	**0.48**	0.02	0.96	<0.01
Build Graph Length (Tasks)	0.94	<0.01	**0.12**	**0.48**	0.80	<0.01	0.96	<0.01

limited use of recursion, only ever reaching a maximum depth of two. These projects only grow in length.

Our findings suggest that build systems require design before implementation, similar to source code. The studied projects either select a recursive design or a non-recursive one. Once a design has been selected, the studied projects do not switch.

ANT Build Coverage Behaviour Analysis: To study the dynamic coverage of a typical build, we calculate the proportion of code exercised in the default build target relative to the total amount of static specification code. 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 6(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 do not show any significant change in value hovering 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 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.

In general, the coverage values are low, ranging between 14% and 40%. There are numerous potential reasons for the low coverage, such as platform-specific behaviour, configuration-specific behaviour, or dead code accrual. Furthermore, release preparation and automated testing features of the build system were not exercised.

Target coverage remains more or less constant for each project. Major fluctuations of ±10% correspond with major project events such as restructuring efforts or major releases that change the default build functionality.

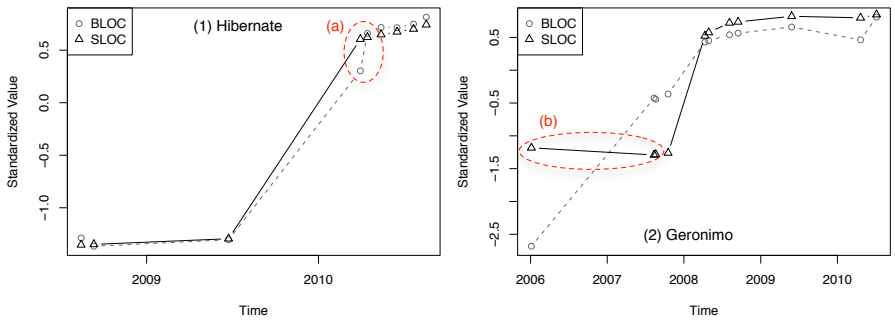

Fig. 7 Standardized BLOC and SLOC values for the Maven projects. Source code and build system evolution trends are very similar.

6 Maven Case Study

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

RQ1) *Do the size and complexity of source code and build system evolve together?*

We explored the evolution of Maven build specification files using the same three angles as ANT. First, we use Figure 7 to show a general trend of increasing size in the two projects, then we use Table 8 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 7 and Table 8 again to show that the build system and source code evolve similarly.

Maven builds also grow: Figure 7 shows that, similar to ANT and `make`, the size of Maven-based build systems also grows over time. Below, we discuss the anomalies with respect to each project.

In 2006, the Hibernate project migrated their existing ANT build system to Maven build technology [12]. The exact motivation for the migration is unclear. In this section, we analyze the Maven-built portion of the Hibernate project evolution. Version 3.3.0 is the first release that used the Maven build system to produce the official deliverables. In Figure 7, we show the Hibernate releases built with Maven, i.e., from version 3.3.0 onward.

The Hibernate Maven build system shows consistent growth throughout its lifetime. The large spike 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). Larger code changes are more likely to introduce defects [28]. Thus, to avoid breaking the existing build infrastructure of a release-producing branch of code, large changes to the build are delayed until a new minor release (e.g., 3.5.0).

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. 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

Table 8 Pearson correlation between BLOC (Columns) and the build system's Halstead complexity and SLOC (Rows). P-values are mostly <0.01, except for some of the low correlations shown in bold, whose p-values are shown in parentheses.

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

Maven 2.x technology, which required major build specification changes [7]. Specifically, the project.properties and build.properties files were merged into a settings.xml file, and the maven.xml and project.xml files were replaced with the pom.xml file.

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

The complexity of Maven specification files evolves: Table 8 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 [15, 32].

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, build system complexity can be reasonably approximated by its size in BLOC.

Maven build growth is highly correlated with source growth: The positive correlations in Table 8 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 7 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.

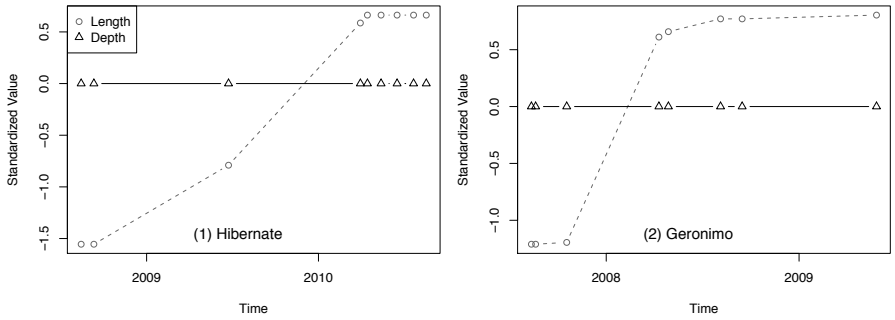

Fig. 8 Standardized build graph dimensions.

RQ2) *Does the perceived 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 8 to show growth of build graph dimensions in the two studied build systems.

Similar to our ANT study, we measure two dimensions of the Maven build graphs for the JAR-producing "package" phase. We measure the length of the build by counting the number of goals that are executed to reach the end of the "package" phase, 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 8 shows the standardized versions of these two build graph dimensions for each release.

Maven features automated downloading and maintenance of third-party component dependencies. In order to keep our comparison with ANT on an even plane, the third-party components were downloaded prior to our analysis.

Since versions previous to 3.3.x of Hibernate were not built using Maven, we refrain from presenting them in Figure 8. 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 8 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 [28]. 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: Before discussing our depth results, we briefly introduce the concept of modules and multi-module Maven build process. A Maven module is simply a component of a Maven-build product that is self-contained,

i.e., a subdirectory with its own Maven build specification file(s). Multi-module Maven builds are achieved by connecting modules together using "Reactor" builds. In a Maven Reactor build, the top-level specification file is parsed first. The specification lists all 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. We measure the depth of a Maven build by counting the number of directories between the deepest build module in the build process and the top directory in the source tree.

Figure 8 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 appear to change.

7 Discussion

We divide our post-experiment discussion into (1) a comparison of our findings for ANT and Maven build technologies, (2) a comparison of our findings for Java build systems to earlier findings for C and C++ build systems, and (3) a discussion of our future work in the area of build systems.

7.1 ANT and Maven comparison

We study the evolution of both ANT and Maven build systems in open source projects. We find that both types of build systems: (1) grow in terms of static and dynamic complexity 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 two reasons. First, 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.

Second, 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 [18]. 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.

7.2 C and Java build system comparison

A single invocation of the Java compiler will automatically resolve dependencies between the input source files, while the C compiler must rely on external dependency management through build tools like `make`. For this reason, we expect to find that Java build systems should require less effort to create and keep in sync with the source code than C build systems. 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. The first two findings are mirrored by our findings for 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. 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 with the common recursive paradigm used to implement modular `make`-based build systems [27]. The Linux build engineers invested much effort in maintaining a modular build system that avoids the flaws associated with recursive `make` [3]. Build system modularity support is built into ANT via the <ant> task, and Maven via Reactor builds. Modularity support provided by ANT and Maven relieves ANT and Maven build engineers from concerns about potential modularity flaws. The engineers can focus on the actual restructuring of the build system.

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 [14]. 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. They need to implement large amounts of boilerplate code to prevent incorrect dependencies from producing inconsistent deliverables or even breaking the build. ANT and Maven build specification are not concerned with such details, since the Java compiler handles dependency management among source files.

To summarize, C and Java build systems evolve similarly from a high-level of analysis. Many of the drivers of the evolution of C build systems are present in Java build systems, yet there are important differences.

7.3 Future Work

Since the major finding of this paper is that the complexity of build systems increases unless explicit effort is invested to restructure them, we believe that the major task of future work is to identify concrete measures that practitioners can take to reduce the impact of build system maintenance. We provide two important areas for concrete measures.

The first area is at the developer level. Since changes to the source code often require changes to the build system, and vice versa, it is important for all stakeholders to know when build maintenance is necessary, and how. Since developers often struggle with code that they are not familiar with [4], similar issues are to be expected for build code, which uses lesser known languages and tools. Hence, a novice developer may easily introduce a source code change, unaware that build maintenance is required. If the build system is not changed when a change is required, the source code may not compile or may produce incorrect deliverables. Hence, we are currently working on a recommendation system to assist developers by identifying code changes that require build maintenance.

Apart from tool support, it is also important to identify concrete measures at the development process level. For example, projects such as Linux [3] and Perl [34] have dedicated teams of build experts. Centralizing build maintenance provides explicit support for developers to help them maintain the build system, yet can turn turn out to be a bottleneck. On the other hand, there are probably not enough build experts to distribute them across every development team. Hence, determining policies and strategies for build maintenance at the process level is a second important avenue for future work.

8 Threats to Validity

Construct Validity: Our analysis focuses on the release level. At this resolution, we miss build system events that happen during the development cycle. We avoid development revisions because there is no guarantee that the system is in a buildable and working state, a precondition to our dynamic analysis for RQ2.

Similar to earlier work [3], our builds are all based on a single platform and configuration. The platform we used is Linux on an x86-based processor and the configuration is 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 may have left areas of the build unexplored.

Internal Validity: The BLOC metric measures lines of build specification code and does not consider build task implementation code. As such, custom ANT task implementations and Maven plugin code did not factor into the build system size or complexity. Build task implementation code remains an unmeasured dimension of the build system size and complexity.

External Validity: Our case studies are based on open source projects and more specifically, open source projects built using ANT or Maven. Our results may not generalize to commercial systems or even open source systems in different domains. To combat this limitation, we considered projects of differing size, domain, and release style.

Our case studies are limited to ANT and Maven build technologies, and as such, our findings may not generalize to other build technologies. However, the similarities between our findings and those found for `make`-based build processes studied in prior work [3] suggest that this threat is limited.

9 Related Work

We present work related to our study in the areas of build system and software evolution.

Robles *et al.* argue that software artifacts other than source code also exhibit interesting maintenance and evolution patterns [33]. In our paper, we present a study of the evolution of build systems, a software artifact that co-evolves with project source code.

Adams *et al.* conjecture that not only do build systems evolve but also that they co-evolve with the program source code [3]. Their study of the Linux kernel build system, implemented using `make`, showed a super-linear (i.e., exponential) trend in the size of build specifications. We only found a super-linear growth in the Eclipse build system. We studied a variety of Java systems with different build technologies, while the prior work only studied the Linux build system.

Miller studies `make` build systems implemented using the common recursive paradigm [27]. He explains some of the rather gruesome pitfalls of the paradigm when used in an unbounded fashion. In our study, we use our build graph depth metrics to keep track of the maximum level that a build recursively encounters. We have no data about whether or not the practice of build recursion in ANT or Maven is a good design choice for Java build systems.

Kumfert and Epperly investigate the development overhead involved with maintaining the build system [20]. In a survey they conducted, developers claim that anywhere between 0% and 35.71% of their development time is spent maintaining the build system. For one specific case, Kumfert and Epperly validate their survey result of 20% by mining the project team's VCS history, categorizing each commit as relating to the build, the project source, and a few other categories that are out of this scope. We study software releases to try to uncover why developers find build systems complex, with the aim of eventually proposing better methods for managing build systems. In a follow-up study, we have also investigated the overhead of build maintenance on nine open source systems and one commercial system at the level of individual changes. We found that build maintenance can impose up to a 27% overhead on source code development [26], which confirms Kumfert *et al.*'s numbers.

Tu *et al.* identified a "Code Robot" build system design pattern [34]. In case studies of the Perl and GCC projects, they show that many build processes produce a preliminary, platform-specific version of a deliverable responsible for building the final version of the deliverable. For instance, in GCC, an initial phase of the build produces a restricted GCC compiler that is used to compile the remaining code and produce the final GCC deliverable. In this paper, we also study the build system, however we

focus on the evolution of build system specifications, both statically and dynamically in order to better understand the build system maintenance process.

Lehman *et al.* discuss their laws of program evolution [21, 22]. Based on the patterns observed in proprietary software, they find that source code tends to grow in size and entropy. Whereas Lehman *et al.* focus on the evolution of programs and changes in the *program* environment, we focus on the evolution of build systems and changes in the *build* environment.

Zadok studied the effect of the migration of the Berkeley Automounter build system to the GNU Autotools build infrastructure [36]. We did not consider migrations of build technology, since complexity metrics are hard to compare across technologies. For this reason, we refrained from comparing Hibernate's ANT and Maven build systems.

10 Conclusions

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, i.e., our case study indicates that build systems change continuously. Furthermore, build systems grow in complexity as a side effect of the changes induced by Lehman's first law. Changes to the build system often need to be accompanied by changes to the project's source code.

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 the complexity of build systems show patterns of growth over time that correlate with the size of the project 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 flat design, it does not switch to the other.
- 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.
- Large fluctuations in target coverage ($\pm 10\%$) correspond with major project events such as restructuring efforts and major releases.
- The findings above are consistent with earlier findings for `make`-based systems with slightly different drivers of build system evolution.

Together with our findings on build maintenance [26], the finding in this paper that large project restructurings are accompanied by similar restructuring in the build system suggests that software projects should dedicate more resources to build maintenance tasks, or at least consider these resources in their planning and budgeting. 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. This allows them to allocate resources to the maintenance and testing of the build system more effectively.

Acknowledgements

We would like to thank Linus Tolke, Tom Morris, and Bob Tarling of the ArgoUML development team for their assistance in validating our case study results, and the anonymous reviewers for their valuable feedback.

References

1. Abreu R, Premraj R (2009) How Developer Communication Frequency Relates to Bug Introducing Changes. In: Proc. of the joint int'l and annual ERCIM workshops on Principles of software evolution and software evolution workshops (IWPSE-Evol), ACM, pp 153–158
2. Adams B, De Schutter K, Tromp H, Meuter W (2007) Design Recovery and Maintenance of Build Systems. In: Proc. of the 23rd Int'l Conf. on Software Maintenance (ICSM), pp 114–123
3. Adams B, De Schutter K, Tromp H, De Meuter W (2008) The evolution of the linux build system. Electronic Communications of the ECEASST 8
4. Antoniol G, Guéhéneuc YG (2005) Feature Identification: A Novel Approach and a Case Study. In: Proc. of the 21st Int'l Conf. on Software Maintenance (ICSM), IEEE Computer Society, pp 357–366
5. Apache Foundation (2010) Apache ANT Manual. http://ant.apache.org/manual/, Last viewed: 07-Jul-2010
6. Apache Foundation (2010) Apache Maven. http://maven.apache.org/, last viewed: 18-Mar-2010
7. Apache Foundation (2010) Maven Migration Guide. http://maven.apache.org/guides/mini/guide-m1-m2.html, Last viewed: 02-Sep-2010
8. Apache Foundation (2011) Maven Build Lifecycle. http://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html, last viewed: 13-Apr-2011
9. Belady L, Lehman M (1976) A Model of Large Program Development. IBM Systems Journal 15(3):225–252
10. Berger T, She S, Lotufo R, Wasowski A, Czarnecki K (2010) 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
11. Dmitriev M (2002) Language-Specific Make Technology for the Java Programming Language. In: Proc. of the 17th Conf. on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA), ACM
12. Ebersole S (2007) Maven migration. http://lists.jboss.org/pipermail/hibernate-dev/2007-May/002075.html, last viewed: 18-Mar-2010
13. Feldman S (1979) Make-A Program for Maintaining Computer Programs. Software - Practice and Experience 9(4):255–265
14. Godfrey MW, Tu Q (2000) Evolution in Open Source Software: A Case Study. In: Proc. of the Int'l Conf. on Software Maintenance (ICSM), IEEE Computer Society, pp 131–140
15. Graves TL, Karr AF, Marron JS, Siy H (2000) Predicting fault incidence using software change history. IEEE Trans Softw Eng 26(7):653–661

16. Halstead MH (1977) Elements of Software Science (Operating and Programming Systems Series). Elsevier Science Inc., New York, NY, USA
17. Jim Weirich (2011) Rake – Ruby Make. http://rake.rubyforge.org/, last viewed: 30-Mar-2011
18. de Jonge M (2005) Build-level components. IEEE Transactions on Software Engineering 31(7):588–600
19. Knight S (2002) SCons Design and Implementation. In: Tenth Int'l Python Conf.
20. Kumfert G, Epperly T (2002) Software in the DOE: The Hidden Overhead of "The Build". Tech. Rep. UCRL-ID-147343, Lawrence Livermore National Laboratory, CA, USA
21. Lehman M (1980) On Understanding Laws, Evolution and Conservation in the Large Program Life Cycle. Journal of Systems and Software 1(3):213–221
22. Lehman M, Ramil J, Wernick P, Perry D, Turski W (1997) Metrics and Laws of Software Evolution – The Nineties View. In: Proc. of the 4th Int'l Software Metrics Symposium (METRICS)
23. McCabe TJ (1976) A Complexity Measure. In: Proc. of the 2nd int'l conf. on Software engineering (ICSE), IEEE Computer Society, p 407
24. McIntosh S, Adams B, Hassan AE (2010) The Evolution of ANT Build Systems. In: Proc. of the 7th working conf. on Mining Software Repositories (MSR), IEEE Computer Society, pp 42–51
25. McIntosh S, Adams B, Hassan AE (2011) ANT and Maven build logs. http://sailhome.cs.queensu.ca/~shane/
26. McIntosh S, Adams B, Nguyen THD, Kamei Y, Hassan AE (2011) An Empirical Study of Build Maintenance Effort. In: Proc. of the 33rd Int'l Conf. on Software Engineering (ICSE), ACM
27. Miller P (1998) Recursive Make Considered Harmful. In: Australian Unix User Group Newsletter, vol 19, pp 14–25
28. Nagappan N, Ball T (2005) Use of relative code churn measures to predict system defect density. In: Proc. of the 27th Int'l Conf. on Software Engineering (ICSE), ACM, New York, NY, USA, pp 284–292, DOI http://doi.acm.org/10.1145/1062455.1062514
29. Neagu A (2010) What is Wrong with Make. http://freshmeat.net/articles/what-is-wrong-with-make, last viewed: 26-Feb-2010
30. Neundorf A (2010) Why the KDE project switched to CMake – and how (continued). http://lwn.net/Articles/188693/, last viewed: 06-Mar-2010
31. Neville-Neal GV (2009) Kode Vicious: System Changes and Side Effects. Communications of the ACM 52(4):25–26, DOI 10.1145/1498765.1498777
32. Robles G, Amor J, Gonzalez-Barahona J, Herraiz I (2005) Evolution and Growth in Large Libre Software Projects. In: Proc. of the Int'l Workshop on Principles of Software Evolution (IWPSE), pp 165–174
33. Robles G, Gonzalez-Barahona JM, Merelo JJ (2006) Beyond Source Code: The Importance of Other Artifacts in Software Development (A Case Study). J Syst Softw 79(9):1233–1248, DOI http://dx.doi.org/10.1016/j.jss.2006.02.048
34. Tu Q, Godfrey M (2002) The build-time software architecture view. In: Proc. of Int'l Conf. on Software Maintenance (ICSM), IEEE Computer Society, pp 398–407
35. Wheeler DA (2011) Sloccount. http://www.dwheeler.com/sloccount/, last viewed: 26-Feb-2011
36. Zadok E (2002) Overhauling Amd for the '00s: A Case Study of GNU Autotools. In: Proc. of the FREENIX Track on the USENIX Technical Conf., USENIX Asso-

ciation, Berkeley (CA, USA), pp 287–297