The Evolution of ANT Build Systems

Shane McIntosh, Bram Adams and Ahmed E. Hassan
Software Analysis and Intelligence Lab (SAIL)
School of Computing, Queen's University, Canada
{*mcintosh, bram, ahmed*}*@cs.queensu.ca*

***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. With a firm understanding of build system aging processes, project managers could allocate personnel and resources to build system maintenance tasks more effectively, reducing the build maintenance overhead on regular development activities. In this paper, we study the evolution of ANT build systems from two perspectives: (1) a static perspective, where we examine the build system specifications using software metrics adopted from the source code domain; and (2) a dynamic perspective where representative sample build runs are conducted and their output logs are analyzed. Case studies of four open source ANT build systems with a combined history of 152 releases show that not only do ANT build systems evolve, but also that they need to react in an agile manner to changes in the source code.**

I. 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. Such 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, the build tools and configurable features are selected based on specifications written in a (mostly ad hoc) configuration language. Second, the deliverables are constructed in the correct order by observing dependencies that are typically specified in a build system language such as `make` or ANT.

Build systems play a key role in software development processes. Build systems 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 involves automatically executing project builds and publishing 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.

Build systems require substantial maintenance effort. A first example of this is given by Kumfert *et al.*, who find that on average, build systems induce a 12% overhead on development effort [2]. Second, the Linux build engineers made integration of new code 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]. Third, the maintenance of the KDE 3 project's build system 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 [4].

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, 5], yet this evolution has only been studied in two `make`-based build systems. In this paper, we present an empirical study of traditional source code evolution phenomena in four build systems. We address the following two research questions:

RQ1) Do the static size and complexity of source code and build system evolve similarly?
Static code analysis of build system specifications indicates not only that 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 source code.

RQ2) Does the perceived build-time complexity evolve?
Build-time complexity measures the perceived complexity observed by the build system user. Our dynamic analysis of build systems did not reveal a common pattern in the studied projects, although we observe linear growth and other interesting trends in build-time length and recursive depth dimensions.

This paper provides the following contributions:

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

The remainder of the paper is organized as follows. Section II introduces the ANT build language and associated terminology. Section III elaborates on the research questions that we address. Section IV discusses the methodology for the case studies we conducted on four open source systems, while Section V presents the results. Section VI surveys related work. Finally, Section VII draws conclusions.

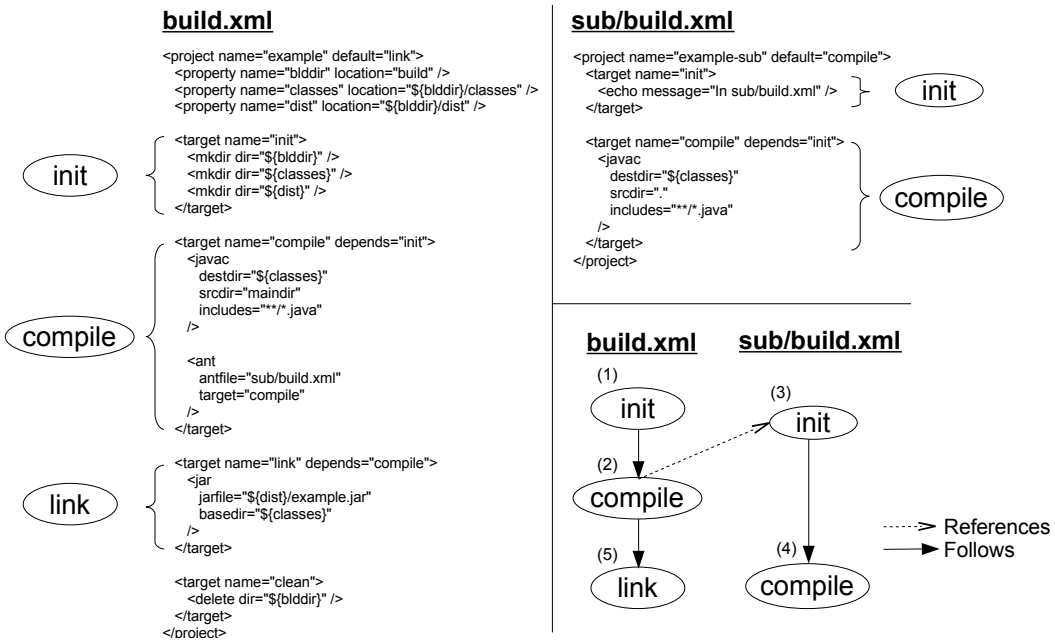

Figure 1. Example ANT build.xml files (left, top-right) and the resulting build graph (bottom-right). The build graph has a depth of 2 (i.e., "compile" in build.xml references "init" in sub/build.xml) and a length of 5 (i.e., execute (1), (2), (3), (4), then (5)).

II. Background

We now provide an overview of build system concepts and the ANT build language, which is the focus of our study.

A. Build Systems Concepts

A typical build system consists of two major layers [6]. The *configuration layer* allows a user or developer to select code features, compilers, and third-party libraries to use during the build process, while enforcing any constraints or conflicts between these configuration options. The configuration layer may automatically detect a default set of configuration options by examining 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 will suffice.

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 correct output. Construction layer (a.k.a. build) specifications are typically expressed in a build system language. Among build system languages, popular choices include `make` [7] and ANT [8].

Build specification files consist of various build target declarations. A build target represents an abstract build goal (or collection of goals) such as "completing all compilation commands". A target typically has two key characteristics, (1) a build rule that defines the set of commands that must be executed when the target is triggered, and (2) a list of dependent targets that determine whether the initial target 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, are older than its input files, or at least one dependent target has been rebuilt.

B. ANT

We study 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 [9]. Although `make` pioneered many build system concepts, there are some rather gruesome 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. 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 in a collection of XML files. <**project**> tags contain all of the code relating to a software project. <**target**> tags correspond to the build targets we explained above. Such a <target> is responsible for a sequence of related tasks such as "compile all source files" ("compile" target in Figure 1) or "collect all class files in a jar archive" ("link" target in Figure 1). <**task**> tags represent specific system-level 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 lowest level of granularity 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.

ANT targets may "depend" on one another. In this sense, a build dependency graph may be constructed 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 however, 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.

III. Research Questions

Software evolution studies 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 [10, 11, 12]. Godfrey *et al.* found that the Linux source code grows super-linearly in size and complexity [13].

We conjecture that build systems also evolve in terms of size and complexity. 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 [5], measured in terms of lines of build code and the number of conditionally compiled code branches.

Inspired by these early studies on build system complexity, we are interested in studying whether their findings generalize to different build systems and technologies. We formulate the following two research questions:

RQ1) Do the static size and complexity of source code and build system evolve similarly?
Previous research has shown that by quantitatively analyzing release snapshots of source code, one may infer trends of software evolution [13]. Hence, we are interested in measuring build system specifications to find out whether they exhibit similar evolutionary trends in size and complexity. Moreover, how do these trends relate to the evolution of the source code.

RQ2) Does the perceived build-time complexity evolve?
Measuring dynamic properties yields insight into how complex a build system is as perceived by build system users. That is, how much build code is routinely exercised and how long a typical build takes. We are interested in investigating whether this perceived complexity exhibits evolutionary trends.

Table I
Studied Projects

	ArgoUML	**Tomcat**	**JBoss**	**Eclipse**
Domain	UML Editor	Web Container	App Server	IDE
Source Size (KSLOC)	≤ 176	≤ 277	≤ 731	$\leq 2,900$
Build System Size (KSBLOC)	≤ 6	≤ 11	≤ 29	≤ 200
Timespan	2002-09	1999-09	2002-09	2001-09
Number of Releases	12	90	25	25
Shortest Rel. Cycle	53 days	2 days	13 days	32 days
Longest Rel. Cycle	593 days	714 days	398 days	176 days
Average Rel. Cycle	228 days	95 days	130 days	110 days
Release Style	Single	Parallel	Parallel	Single

IV. Methodology

To track the progress of software build systems, we define and analyze build system metrics for release snapshots of a software project. 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 2. We now explain each step of our approach.

A. 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 any arbitrary time in the development cycle, a release snapshot is by nature a buildable and runnable version of a project. This decision is critical for our dynamic analysis in RQ2.

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 data, which was retrieved from a source code repository at the suggestion of the project documentation [14]. The released versions of ArgoUML were marked in the repository with annotated tags.

B. Evolution Metrics

In our study, we use various static and dynamic metrics to quantify a wide variety of build systems characteristics across the release snapshots. The metrics are summarized in Table II. SBLOC, 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 SBLOC, file count, DBLOC and the Halstead suite of complexity metrics are inspired by corresponding source code metrics, others such as target count and task count are inspired by [3]. Build graph depth and target coverage are new to this study.

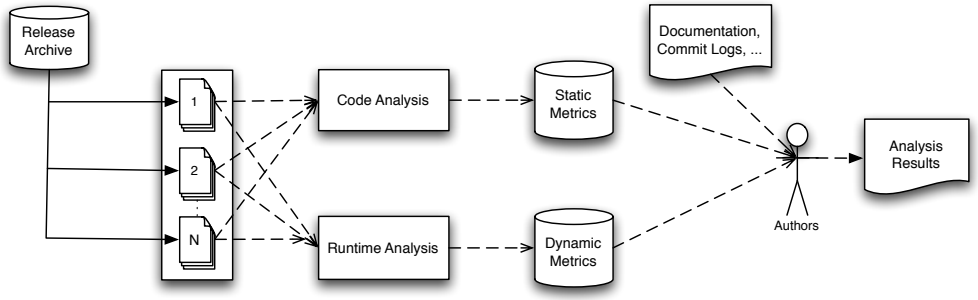

Figure 2. Overview of our approach to study the evolution of build systems.

Table II
METRICS USED IN BUILD SYSTEM ANALYSIS

Group	Metric	Description
Static	Static Build Lines of Code (SBLOC)	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.

Most of the metrics are self-explanatory, except for the Halstead complexity metrics, as we had to adapt its 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 ANT files because build specification files share many similarities with source code implemented in an interpreted programming language. Case in point, the SCons build language [15] is entirely based on the Python 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 paper, we only focus on the Halstead suite of complexity metrics [16]. In future work, we plan to examine the McCabe cyclomatic complexity [17] and how it applies to build systems, although results of our case study indicate that (similar to source code [18, 19]), 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 ANT build systems, we consider an operator as any target or task. Next, we must tally the number of *operands* used in the source code. Within the scope of ANT build systems, we consider operands as the parameters passed to a target or task tag. 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 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:

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

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

$$\text{Effort} = D \times V \tag{3}$$

C. Analysis Methodology

We analyze each release snapshot using two perspectives. First, build system files and program source files of each release are examined statically. We measure source code size (in SLOC) so that we may compare it against build system size (in SBLOC). SLOC was measured using David A. Wheeler's `sloccount` utility [20]. 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 SBLOC count also discards them using a `sed` script and the remaining lines are tallied using `wc`.

Second, the build system of each release was exercised (using the default build configuration) and the results were logged. The ANT output was exported to an XML log using the built-in ANT XML logger (-logger=XmlLogger). This log 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.

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

D. Studied Projects

We selected four open source projects of different size, domain, and release style. Table I 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 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.

V. Case Study

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

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

We explored the evolution of build system specification files from three angles. First, we use Figure 3 to show a general trend of increasing size in the four projects, then we use Table III and IV 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 3 and Table III again to show that the build system and source code evolve similarly.

Growth in the Size of Build Systems: In Figure 3, we plot the standardized SBLOC and SLOC metrics so that we may compare these two metrics in one graph, as SLOC values have a much higher scale than SBLOC values. This standardization is calculated by weighting each data point in terms of its distance from the average SBLOC or SLOC across all releases of a system, measured in units of standard deviation (i.e., $Y = \frac{n-\mu}{\sigma}$). In projects with parallel releases, 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 SBLOC of ArgoUML shows a clearly increasing trend with the exception of one period in between releases 0.18.1 and 0.20 (Figure 3(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 SBLOC. In the 4.0.x releases, the build system was initially subject to a rapid increase in SBLOC (Figure 3(d)). This was due to extensive work in the Catalina subproject. 568 lines of SBLOC 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 3(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 SBLOC 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 3(f) and Figure 3(g) skew the first half of the results in JBoss, which otherwise point to an increasing trend in SBLOC. During Figure 3(f), an entire rewrite of the enormous testsuite build specification file resulted in the removal of approximately 5,000 SBLOC. During Figure 3(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 SBLOC.

In tracking the progress of the Eclipse build system size over time, we see an exponentially increasing trend in SBLOC (Figure 4). 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. We calculated the Pearson correlation between the number of plugins in each release and SBLOC to be 0.99. This suggests that

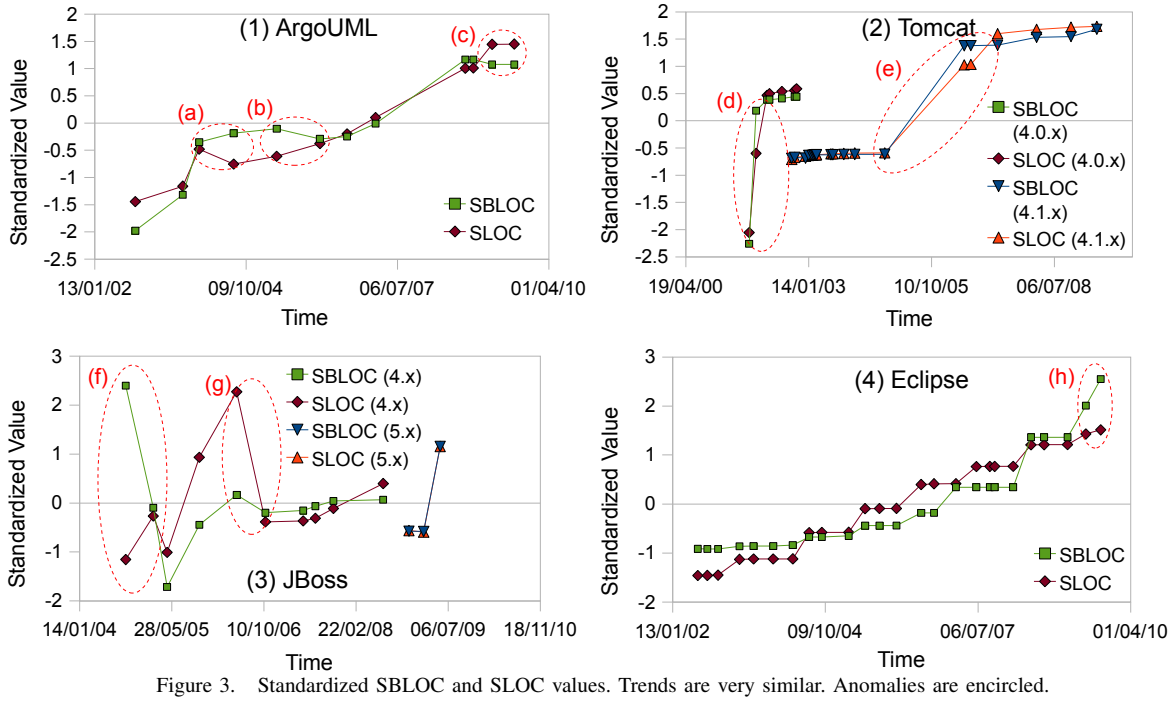

Figure 3. Standardized SBLOC and SLOC values. Trends are very similar. Anomalies are encircled.

Table III
CORRELATION OF STATIC SIZE METRICS (ARGOUML, TOMCAT, JBOSS, ECLIPSE)

	Task Count				File Count				SBLOC				SLOC			
	A	T	J	E	A	T	J	E	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	0.98	0.99	**0.40**	0.98	0.78	0.97	0.89	0.95
Task Count					0.95	0.98	**0.64**	0.99	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
SBLOC													0.89	0.98	**0.40**	0.99

the exponentially rising trend in build system size strongly correlates with the trend in the number of plugins per release.

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.

The Evolution of Build System Static Complexity: Table III shows the Pearson correlation between the static size metrics for each studied system. With the exception of the JBoss project, the high correlation values indicate that SBLOC is a good approximation for build system size.

We also found that the Halstead complexity metrics follow trends similar to SBLOC. Table IV shows, for each studied system, the Pearson correlation between each Halstead complexity metric and the SBLOC. With the exception of the JBoss project, the results indicate that build specification complexity is highly correlated with build specification size (SBLOC). This finding seems to agree with similar findings from research in the source code domain [18, 19].

The JBoss build system deviates from the trend, as it is implemented with a different style. It leverages the underlying XML roots of ANT specification files to introduce a system of abstraction. The <!ENTITY> macro substitution

Table IV
PEARSON CORRELATION BETWEEN HALSTEAD METRICS (ROWS) AND SBLOC (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

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 as we study SBLOC 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 (SBLOC), indicating that SBLOC is a good approximation of build system complexity.

Correlation of Source Code and Build System Growth: 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 SBLOC and SLOC; and (2) visually compare the

trends of SBLOC and SLOC in Figure 3.

Table III shows that SBLOC 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 SBLOC and SLOC for the four subject systems is illustrated in Figure 3. In most cases, the characteristics of the source code and build specification curves are very similar, which suggests that SBLOC 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 3(a), (b), and (c). During Figure 3(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 SBLOC and a sharp decrease in SLOC. During Figure 3(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 project source code. During Figure 3(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 3(d) and (e) are explained above.

For the two parallel release branches of the JBoss project, it would appear that there is little correlation between the SBLOC and SLOC trends. During the rewrite of the build specification file in the testsuite module in the Figure 3(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 3(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 SBLOC and SLOC trends.

In Eclipse, the trends in SBLOC and SLOC are very similar. However, in between releases 3.5 and 3.5.1 (Figure 3(h)), we observe a sharp increase in SBLOC and a moderate increase in SLOC. The SBLOC 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 SBLOC.

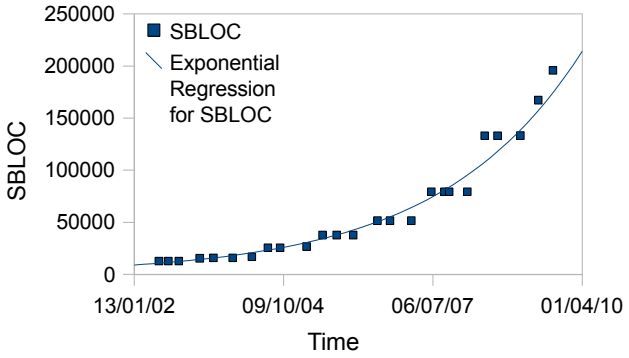

Figure 4. The exponential trend in Eclipse SBLOC. The trend line has an R^2 value of 0.98.

> *In most projects, SBLOC 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 5 to show growth of build graph length and depth in the four studied build systems, then we use Table V to examine the build recursion complexity, and finally we analyze changes in target coverage: a measure of perceived build system complexity.

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 5(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 5(b) interval corresponds to the Figure 3(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 5(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 hardcoded URLs of the archive release locations. The hardcoded 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 5(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. In the 6.x release branch, Tomcat switched from an ANT build system to a Maven build system because Maven has support for maintaining third party library dependencies and avoiding

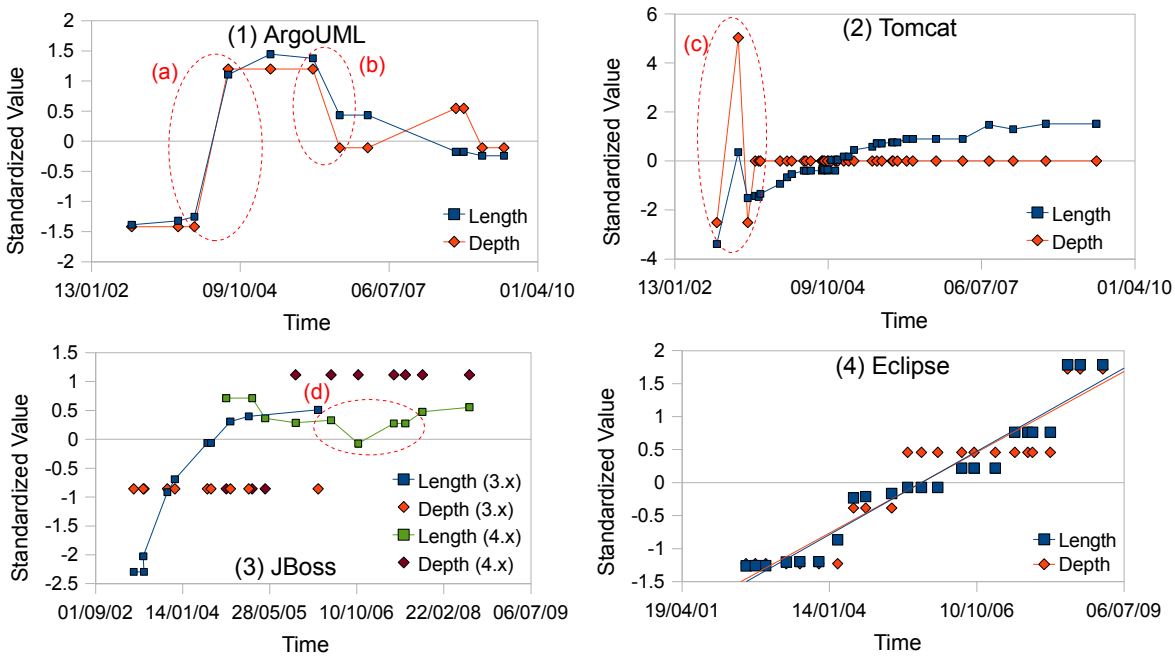

Figure 5. Standardized build graph dimensions. Build graph length (in targets) and depth have an R^2 value of 0.94 and 0.88.

hard-coded external URLs. As we only consider ANT build systems, we did not measure the builds of the 6.x branch. The inability to build Tomcat 3.x and 4.x and the move to Maven show that managing third-party dependencies is an important driver for build system evolution.

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. The correlation between these two dimensions are 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.*

Constant Depth vs. Varying Depth: Figure 5 shows two distinct trends in the build graph depth: (1) a near-constant depth (Tomcat and JBoss); and (2) a depth which seems to vary in trends similar to build graph length (ArgoUML and Eclipse). Table V shows the Pearson correlation between build graph depth and length metrics. The table indicates that the ArgoUML and Eclipse builds are recursive in design while Tomcat and JBoss are not. 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 limited use of recursion, only ever reaching a maximum depth of two. These projects only grow in length.

Table V
PEARSON CORRELATION BETWEEN DYNAMIC METRICS (ROWS) AND BUILD GRAPH DEPTH (COLUMNS)

	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

> *The studied build systems are either recursive in design or not. Once a design has been selected, the studied projects do not switch.*

Build Coverage Behaviour Analysis: To study the dynamic coverage of a typical build, we calculate the proportion of code exercised in a typical build 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 project restructuring periods (Figure 5(b) and (c)). The SBLOC shrank during the restructuring, which implies that they were

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 $\pm 3\%$ 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.

> *Target coverage remains more or less constant for each project. Major fluctuations of $\pm 10\%$ correspond with major project events such as restructuring efforts or major releases.*

Threats to Validity

Our case studies are based on open source projects and more specifically, open source projects built using ANT. Our results may not generalize to commercial systems or even open source systems that are built using a different build system language. To combat this limitation, we considered projects of differing size, domain, and release style.

Our analysis focuses on the release level. At this resolution, we may miss some 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.

Similar to [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 consistent platform for comparison. By only exploring a single configuration, we may have left areas of the build unexplored.

The SBLOC metric measures lines of build specification code and does not consider build task implementation code. As such, custom ANT task implementations 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.

VI. Related Work

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

In [21], Robles *et al.* argue that software artifacts other than source code also require research. In our paper, we present a study of the evolution of build systems, an entity which co-exists with project source code.

In [3], Adams *et al.* conjecture that not only do build systems evolve but also that they co-evolve with the program source code. They present their study of the Linux kernel build system, implemented using `make`, in which they found 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 ANT build systems while the Linux build system is implemented in `make`.

In [22], Miller presents his study of `make` build systems implemented using the common recursive paradigm. 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 the practice of build recursion in ANT is a good design choice for build systems.

In [2], Kumfert and Epperly investigate the development overhead involved with maintaining the build system. 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 CVS 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 [11, 12], Lehman *et al.* discuss their laws of program evolution. 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 their environment, we focus on the evolution of build systems and changes in their environment.

In [5], Zadok studied the effect of introducing the GNU Autotools build infrastructure on the complexity of the Berkeley Automounter build system. We observe the effect that environmental factors such as the source code and development libraries have on the evolution of build systems.

VII. 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. That is, our case study indicates that build systems continuously change, especially due to changes in their environment (i.e., source code and development libraries). Furthermore, build systems grow in complexity as a side effect of the changes induced by Lehman's first law. The evolution of a build system is often in sync with the project's source code.

Through a case study of four open source projects, we made the following important observations:

- Both the static and dynamic size and complexity of build systems show differing patterns of growth over time that correlate with their project source code.
- The exponential growth of Eclipse's build system is highly correlated with the project plugin count.
- Build systems are either recursive in design or not. 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 (SBLOC).
- 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.

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 more resources to the maintenance and testing of the source code and build system.

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] R. Abreu and R. Premraj, "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)*. New York, NY, USA: ACM, 2009, pp. 153–158.

[2] G. Kumfert and T. Epperly, "Software in the DOE: The Hidden Overhead of "The Build"," Lawrence Livermore National Laboratory, CA, USA, Tech. Rep. UCRL-ID-147343, February 2002.

[3] B. Adams, K. D. Schutter, H. Tromp, and W. D. Meuter, "The Evolution of the Linux Build System," *ECEASST*, vol. 8, 2007.

[4] A. Neundorf, "Why the KDE project switched to CMake – and how (continued)," http://lwn.net/Articles/188693/, last viewed: 06-Mar-2010.

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

[6] B. Adams, K. De Schutter, H. Tromp, and W. Meuter, "Design Recovery and Maintenance of Build Systems," in *Proc. of the 23rd Int'l Conf. on Software Maintenance (ICSM)*, 2007.

[7] S. Feldman, "Make-A Program for Maintaining Computer Programs," *Software - Practice and Experience*, vol. 9, no. 4, pp. 255–265, 1979.

[8] Apache Foundation, "Apache ANT Manual," http://ant.apache.org/manual/.

[9] A. Neagu, "What is Wrong with Make," http://freshmeat.net/articles/what-is-wrong-with-make, last viewed: 26-Feb-2010.

[10] L. Belady and M. Lehman, "A Model of Large Program Development," *IBM Systems Journal*, vol. 15, no. 3, pp. 225–252, 1976.

[11] M. Lehman, "On Understanding Laws, Evolution and Conservation in the Large Program Life Cycle," *Journal of Systems and Software*, vol. 1, no. 3, pp. 213–221, 1980.

[12] M. Lehman, J. Ramil, P. Wernick, D. Perry, and W. Turski, "Metrics and Laws of Software Evolution – The Nineties View," in *Proc. of the 4th Int'l Software Metrics Symposium (METRICS)*, 1997.

[13] M. W. Godfrey and Q. Tu, "Evolution in Open Source Software: A Case Study," in *Proc. of the Int'l Conf. on Software Maintenance (ICSM)*. Washington, DC, USA: IEEE Computer Society, 2000, p. 131.

[14] ArgoUML Team, "ArgoUML Documentation," http://argouml-downloads.tigris.org/argouml-0.28.1/, last viewed: 26-Feb-2010.

[15] S. Knight, "SCons Design and Implementation," in *Tenth Int'l Python Conf.*, 2002.

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

[17] T. J. McCabe, "A Complexity Measure," in *Proc. of the 2nd int'l conf. on Software engineering (ICSE)*. IEEE Computer Society Press, 1976, p. 407.

[18] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy, "Predicting fault incidence using software change history," *IEEE Trans. Softw. Eng.*, vol. 26, no. 7, pp. 653–661, 2000.

[19] G. Robles, J. Amor, J. Gonzalez-Barahona, and I. Herraiz, "Evolution and Growth in Large Libre Software Projects," in *Proc. of the Int'l Workshop on Principles of Software Evolution (IWPSE)*, 2005, pp. 165–174.

[20] D. A. Wheeler, "Sloccount," http://www.dwheeler.com/sloccount/, last viewed: 26-Feb-2010.

[21] G. Robles, J. M. Gonzalez-Barahona, and J. J. Merelo, "Beyond Source Code: The Importance of Other Artifacts in Software Development (A Case Study)," *J. Syst. Softw.*, vol. 79, no. 9, pp. 1233–1248, 2006.

[22] P. Miller, "Recursive Make Considered Harmful," in *Australian Unix User Group Newsletter*, vol. 19, 1998, pp. 14–25.