Collecting and Leveraging a Benchmark of Build System Clones to Aid in Quality Assessments

Shane McIntosh[1], Martin Poehlmann[2], Elmar Juergens[2], Audris Mockus[3],
Bram Adams[4], Ahmed E. Hassan[1], Brigitte Haupt[5], and Christian Wagner[5]
[1]Queen's University, Canada; [2]CQSE GmbH, Germany; [3]Avaya Labs Research;
[4]Polytechnique Montréal, Canada; [5]Munich Re, Germany
[1]{mcintosh, ahmed}@cs.queensu.ca, [2]{poehlmann, juergens}@cqse.eu, [3]audris@avaya.com,
[4]bram.adams@polymtl.ca, [5]{bhaupt, cwagner}@munichre.com

ABSTRACT

Build systems specify how sources are transformed into deliverables, and hence must be carefully maintained to ensure that deliverables are assembled correctly. Similar to source code, build systems tend to grow in complexity unless specifications are refactored. This paper describes how clone detection can aid in quality assessments that determine if and where build refactoring effort should be applied. We gauge cloning rates in build systems by collecting and analyzing a benchmark comprising 3,872 build systems. Analysis of the benchmark reveals that: (1) build systems tend to have higher cloning rates than other software artifacts, (2) recent build technologies tend to be more prone to cloning, especially of configuration details like API dependencies, than older technologies, and (3) build systems that have fewer clones achieve higher levels of reuse via mechanisms not offered by build technologies. Our findings aided in refactoring a large industrial build system containing 1.1 million lines.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*restructuring, reverse engineering, and reengineering*

General Terms

Management, Measurement

Keywords

Build systems, clone detection, quality assessments

1. INTRODUCTION

Build systems transform the source code of a software system into deliverables. They describe the process by which software is assembled by orchestrating compilers, preprocessors, and other tools, allowing developers to focus on making source code changes. A stable build system is crucial to the timely delivery of software, boosting team productivity by: (1) testing code changes for regression by executing automated tests, (2) generating versions of the software for integration and feature testing, and (3) automatically packaging and deploying software with the correct versions of required libraries, documentation, and data files.

Build systems play a crucial role in contemporary software development techniques. For example, *continuous integration*, i.e., the practice of routinely downloading the latest source code changes onto dedicated servers to ensure that the code base is free of compilation and test failures would not be possible without a robust build system. Furthermore, the recent practice of *continuous delivery* [11], where new releases of a software system must be provided within minutes rather than days or months would not be possible either.

Yet to reap the most benefit, build systems must be carefully maintained to ensure that deliverables are assembled correctly. Since build systems tend to grow in terms of size and complexity as they age [1, 21], they also tend to become more difficult to maintain. Indeed, as Munich Re (one of the world's leading reinsurers) has shortened development cycles to yield more frequent releases, maintainers have noticed that change requests for the build system have increased in frequency and difficulty. The increased cost of build maintenance motivated management to contact CQSE (a software quality consultancy group) to investigate the root cause and propose methods of reducing the cost of build maintenance.

Through assessment of the Munich Re build system, we note that cloning (i.e., duplication) of build logic contributes to the increase in change- and error-proneness. Munich Re maintains roughly 30 custom business information systems implemented using C#, which share a common build system that exploits similarities among the applications. However, over the years, the build system has grown to roughly 1.1 million lines of build logic. Maintenance of the build system has been subcontracted to an external supplier who has allocated a team of three developers to the task. Changes to the build system often need to be repeated in as many as 30 locations. Defects may linger in the build system if changes are not propagated to all of the required locations.

Despite the perils of build logic cloning, it is not well understood. Hence, although build maintainers tend to agree that cloning is problematic, selecting a more maintainable solution is non-trivial. For example, it is not clear whether build logic cloning can be avoided, i.e., cloning may be an innate property of build systems. Moreover, it may be that

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICSE '14, May 31 - June 7, 2014, Hyderabad, India
Copyright 2014 ACM 978-1-4503-2768-8/14/05 ...$15.00.

certain technologies are more prone to cloning, which would suggest that migration to a less clone-prone technology could provide some relief.

In this paper, we set out to better understand build logic cloning by collecting and analyzing a benchmark comprising 3,872 open source build systems from Apache, GNU, Sourceforge, and Github. Through analysis of the benchmark, we address five research questions:

(RQ1) How much cloning is typical of build systems?
Although cloning rates in build systems are typically higher than those of other software artifacts, there are build systems with little cloning, indicating that there are measures one can take to reduce build logic cloning.

(RQ2) Does technology choice influence cloning in build systems?
The more recent CMake (C/C++) and Maven (Java) technologies tend to be more prone to cloning than the older Autotools (C/C++) and Ant (Java) ones.

(RQ3) Do benchmark-derived cloning thresholds vary among build technologies?
We use thresholds derived from quantiles in our benchmark to identify build systems with abnormal cloning characteristics. Technology-specific thresholds vary most for Java build systems with abnormally low amounts of cloning, and between CMake/-Autotools and Ant/Maven for build systems with abnormally high amounts of cloning.

(RQ4) What type of information is typically cloned in build specifications?
The more recent technologies are more susceptible to cloning of configuration details like API dependencies, while the older technologies are more susceptible to cloning of lower-level build logic.

(RQ5) How do build systems with few clones achieve low clone rates?
Build systems with little cloning leverage reuse mechanisms beyond those offered by build technologies themselves, suggesting that existing reuse mechanisms offered by build technologies are insufficient for avoiding build logic cloning.

Paper organization. The remainder of the paper is organized as follows. Section 2 describes build systems and clone detection in more detail. Section 3 provides details on build logic cloning at Munich Re. Section 4 describes the design of our benchmark collection and analysis study. Sections 5 and 6 present our findings with respect to our five research questions. Section 7 discloses the threats to the validity of our study. Section 8 surveys related work. Finally, Section 9 draws conclusions.

2. BACKGROUND

In this section, we provide a brief description of build systems and clone detection.

Build systems. The build process of a software system is typically broken down into four phases. First, the *configuration* phase selects build tools (e.g., compilers and linkers) and features (e.g., Windows or Android front-end). Next, the *construction* phase translates relevant source code and data files into deliverables by executing several order-dependent commands like compilers. The *certification* phase follows construction, where deliverables are checked for regression by executing suites of automated tests. Finally, the *packaging* phase bundles product documentation, data files, and deliverables for easy delivery to end users.

Recently, companies such as Google have adopted *continuous delivery* [11], a development approach that facilitates rapid distribution of newly produced versions of a software system. In order to support continuous delivery, build systems perform an additional *deployment* phase, where newly produced versions of a software system are automatically deployed to testing and production environments.

Clone detection. Clones are duplicated regions in software artifacts, typically created by copying and pasting. Clones tend to hinder maintenance, since changes to an artifact region often need to be performed consistently to all of its clones. Clone detection tools search for clones in software artifacts to support the maintenance of software artifacts that contain clones.

There are various types of clones used in research and practice [18, 30]. To the best of our knowledge, this is the first study to explore build logic cloning. Hence, for our measurements, we focus on *Type I clones*, i.e., exact copies ignoring the variations in whitespace and comments, and leave the exploration of higher level clone types to future work. We measure the extent of build logic cloning using:

Clone Coverage – The proportion of build logic lines that are cloned at least once in the build system. Values range between 0 (i.e., no detected clones) and 1 (i.e., each build logic line is cloned at least once).

Blow Up – The degree of inflation in build system size with respect to a hypothetical build system that does not contain any clones, i.e., $\frac{ActualSize}{RedundancyFreeSize} - 1$. Hence, a blow up value of 0 indicates that the system is not inflated by cloning, while values above 0 indicate the degree of inflation due to cloning.

3. BUILD LOGIC CLONING IN INDUSTRY

This section provides a motivational example to illustrate the reasons and impact of excessive build logic cloning.

Clone-based build system design. Most Munich Re business applications use a shared company-wide build infrastructure based on Microsoft Team Foundation Server (TFS) specified using MSBuild. Each business application has different build specifications for each build configuration (e.g., debug and release) and each application release (e.g., 2013.1 and 2013.2). For example, one business application has six build specifications representing debug and release configurations for its 2013.1, 2013.2, and 2013.3 releases.

These MSBuild specifications enhance the default TFS build process with unit testing, continuous code quality analysis, and packaging in preparation for automated deployment to testing, pre-production, and production environments. Build specifications range between 1,500-8,000 lines of build logic, with an average size of 3,800. The Munich Re build system currently contains more than 1.1 million lines of build logic spread across 295 build specifications.

To add a new release or a new application to the build system, the build specifications of a stable application are du-

plicated and customized. In the simplest case, the application name, as well as the application-specific directories and source file lists need to be customized. More complex applications have unique packaging requirements or need special interaction with the TFS. Yet, since the core build logic remains unchanged, build specifications are largely the same. Since new releases and new applications must be added to the build system regularly, one can easily see how the Munich Re build system has grown to the size it is today.

Clone-based build system maintenance. The effort required to maintain the Munich Re build system has steadily increased over the years. It now requires three full-time employees whose sole responsibility is to maintain the build system. These build maintainers are responsible for configuring new application releases, adding new applications to the build system, fixing build system defects, and adding new build system features.

Even with this dedicated team of build maintainers, defects fixes and new features take a long time to complete. In fact, due to time pressure, some build system changes are never completely propagated to all build specifications. For example, a build maintainer recently added the `ContinueOnError` flag (which prevents the build from failing) to one of three specifications that uninstall the same application. It was not until one week later that the flag was applied to the second of the three specifications. The flag has not yet been applied to a third instance.

Prolonged fixes and inconsistent changes are an often-observed clone-related problem in source code, too [14]. It is a novel observation, however, that build specifications are affected by these problems as well.

Shortcomings of the clone-based build system design. The clone-based build system design has been perceived by build maintainers as one of the fundamental causes of the build maintenance difficulties at Munich Re. Through discussions with the build maintainers, they report that: "You have to alter 15 occurrences [of a defect] and you have to be really careful not to introduce new [defects]" and "With over 270 or more versions [of a build specification], the [build system] is simply not maintainable anymore".

Indeed, with a minimum clone length of twenty lines, clone detection results indicate that the Munich Re build system has a clone coverage of 94.4% and a blow up of 1,023% (clone detector configuration details are found in Section 4). With a minimum clone length of five lines, clone coverage and blow up values increase to 99.1% and 2,335% respectively. In other words, the Munich Re build system: (1) is over 23 times larger than it would be without cloning, and (2) only contains roughly 50,000 unique lines of build logic.

Cloning between build specifications has also lead to dead build features. These features were copied when a specification was duplicated, but are not used during the build. This further inflates maintenance effort and increases the likelihood of introducing defects during maintenance, since one must first recognize whether a build feature is active or not before making modifications.

4. BENCHMARK COLLECTION AND ANALYSIS

In this section, we describe our benchmark collection and analysis approach. We structure our analysis by addressing five research questions. We present our rationale for selecting each research question below.

Table 1: Overview of the studied systems.

	Ant	Maven	Autotools	CMake	Total
Apache	51	56	18	3	128
Github	114	321	521	220	1,176
GNU	15	0	243	12	270
Sourceforge	593	125	1,517	63	2,298
Total	773	502	2,299	298	3,872
# w/ Clones	664	484	943	162	2,253
% w/ Clones	86%	96%	41%	54%	58%

Deriving baseline values. Intuitively, build logic cloning at Munich Re appears to be quite high. In order to ground our intuition empirically, we collect a benchmark of build logic clones from a large corpus of open source build systems. Through *quantitative* analysis of the benchmark, we address the following three research questions:

(RQ1) How much cloning is typical of build systems?
Little is known about build logic cloning. Hence, we are interested in first exploring what typical cloning rates are within the scope of build systems.

(RQ2) Does technology choice influence cloning in build systems?
There are numerous build technologies, each with its own nuances. A better understanding of the influence that technology choice has on build system quality metrics like cloning will allow practitioners to make more informed build technology choices.

(RQ3) Do benchmark-derived cloning thresholds vary among build technologies?
If technology-specific cloning benchmarks vary considerably, a single technology-independent benchmark would set a target that is unreasonably low for clone-prone technologies, and too lax for clone-resistant technologies.

Understanding cloned information. The build system describes up to five interdependent phases (*cf.* Section 2). Build specifications describe how each phase must be performed. It is not clear which of these phases are most susceptible to cloning. Through *qualitative* inspection of build logic clones, we address the following two research questions:

(RQ4) What type of information is typically cloned in build specifications?
We set out to better understand what phases of the build process tend to be cloned in each build technology with the intent to discover if cloning rates are affected by limitations of the technology itself or a lack of skill in applying it.

(RQ5) How do build systems with few clones achieve low clone rates?
We compare clone-prone and clone-resistant build systems to elucidate differences in cloning practices.

Our approach to extracting and analyzing the build logic cloning benchmark is structured using the four steps suggested by Mockus for analyzing software repositories [24]. Figure 1 provides an overview of our approach. We describe each step in the approach below.

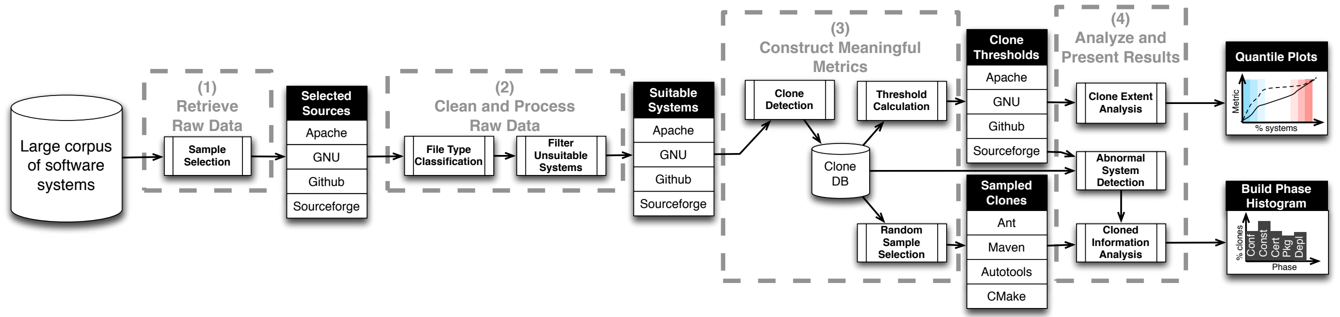


Figure 1: Overview of our data extraction and analysis approach.

4.1 Retrieve Raw Data

It is important that our benchmark contains a large sample of build systems in order to improve confidence in the conclusions that we draw. Hence, we select a sample of 3,872 build systems from the large corpus of open source systems of varying size, scope, and domain collected by Mockus [25]. We describe the corpus of build systems used in this study and explain our extraction and analysis approaches below.

Sample selection. The sample of build systems was obtained from four sources described in Table 1. The *Apache Software Foundation* provides organizational, legal, and financial support for a broad range of open source software systems. Savannah (*GNU*) is the software forge for people committed to free software. *Github* and *Sourceforge* are also popular software forges.

We select the build systems of Java and C/C++ systems for our benchmark, since they are among the most broadly adopted programming languages in our corpus [25]. We further narrow our study by selecting the two most frequently used build technologies for each studied language. In our corpus, C/C++ systems use GNU Autotools and CMake most frequently. Similarly, Ant and Maven are used most frequently to specify Java build systems. We extract the latest version of each software system that meets our selection criteria from the large corpus.

Figure 2 provides an overview of the benchmark by plotting the number of clones detected against size of the build system using hexbin plots [4]. Hexbin plots are scatterplots that represent several data points with hexagon-shaped bins. The darker the shade of the hexagon, the more data points that fall within the bin. The plot is logarithmically scaled in all dimensions to lessen the influence of outliers.

The relationship between number of clones and build system size is roughly linear on the log scale and quadratic on the linear scale. The hexagons in Figure 2 tend to appear in a positive upward diagonal direction. Similarly, the hexagons tend to deepen in shade along an upward diagonal trend in the Java build systems. This suggests that as a build system grows, so too does its proneness to cloning.

4.2 Clean and Process Raw Data

Prior to addressing our research questions, we must first ensure the extracted systems are suitable for analysis. This process is divided into two steps.

File type classification. In our prior work, we categorized files by type semi-automatically [22], however with a corpus of this scale, manual categorization is infeasible. To address this, we conservatively identify build files based on filename conventions. An overview of the filename conventions that we map to each technology is given in Table 2. Although our approach may miss some build specifications that do not follow filename conventions, the approach is lightweight enough to be applied to all files in the corpus.

Table 2: The adopted file name conventions for the studied build technologies.

Technology	Conventions
Ant	`build.xml, build.properties`
Maven	`pom.xml, maven([123])?.xml`
Autotools	`[Cc]onfigure.(ac|in), ac(local|site).m4, [Mm]akefile.(am|in), config.h.in`
CMake	`CMakeLists.txt, *.cmake`

Filter unsuitable systems. Software incubators such as Github and Sourceforge often contain systems that have not yet reached maturity. Neitsch *et al.* conjecture that IDE support for building software is sufficient for small systems [26]. Indeed, Smith suggests that build system maintenance does not become a problem until a system ages, requiring more configurability to expand market presence [31]. To reduce noise in the benchmark, we filter away systems with fewer than five build specification files or 100 lines of build logic.

4.3 Construct Meaningful Measures

Next, we apply clone detection to the surviving build systems using ConQAT [6]. Then, metric thresholds are derived from the benchmark. Finally, a random sample of clones are selected for detailed analysis.

Clone detection. The ConQAT clone detector reads all files of a system that match the pattern of the specified build technology from Table 2 into memory. The detection algorithm is configured to be line-based with varying minimum clone lengths of 5, 10, 15, and 20 lines. To handle file formatting differences, we trim the leading and trailing white space of each line. We omit empty lines and comments, since they do not have an impact on the build process. We also omit closing XML tags, since XML-based build specifications are more verbose. Although not strictly necessary for our analyses in this paper, controlling for XML verbosity helps to make XML and non-XML build logic cloning results more comparable. In this paper, we consider only Type I clones. For example, when the minimum clone length is set to five, clones must share at least five consecutive non-empty lines after applying the normalization described above.

Threshold calculation. Thresholds are used to identify

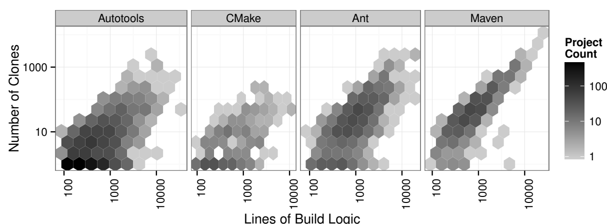


Figure 2: Number of clones detected vs. build system size (in lines of build logic).

entities with metric values that warrant further investigation. For example, build specifications with a blow up value above two may be worth inspection. Yet it is non-trivial to select effective thresholds that pinpoint abnormal entities while retaining low false positive and false negative rates. There are various threshold derivation techniques that can gauge a variable with unknown properties empirically. In order to address RQ3, we adopt the quantile-based technique suggested by Alves *et al.* [2], since (as they point out) other threshold derivation techniques (such as deviation analysis) often make invalid assumptions about the dataset (e.g., normally distributed), or require carefully tuned input parameters (e.g., number of clusters for clustering techniques).

Alves *et al.* suggest that values that fall above the 70th, 80th, and 90th percentiles are abnormal to a moderately high, high, and very high degree respectively. We extend this concept by arguing that values that fall below the 30th, 20th, and 10th percentiles are abnormal to a moderately low, low, and very low degree respectively. Values that appear at quantile boundaries are considered thresholds.

Random sample selection. In order to address RQ4, we need to select a representative sample of clones of each studied build technology for deeper analysis. We randomly select a sample of clones that is large enough to achieve a 95% confidence level.

4.4 Analyze and Present Results

Finally, we use the derived thresholds to detect and analyze build systems with abnormal amounts of cloning.

Clone extent analysis. We use quantile plots to indicate whether the amount of cloning in a system is abnormal. These plots show the cumulative proportion of systems that have clone coverage and blow up metrics below a given value.

Abnormal system detection. To better understand good and bad cloning practices, we analyze the most and the least clone-prone systems. We first identify common cloning pitfalls of the most clone-prone systems. Then, we analyze the least clone-prone systems to understand how these pitfalls can be avoided.

Cloned information analysis. We manually analyze the information cloned in a random sample of clones for each studied technology (RQ4), and all of the clones in the highly clone-prone build systems (RQ5). To address RQ4, we assess each clone to determine which of the five build phases (*cf.* Section 2) are impacted.

The configuration phase can be broken down into three subcategories. *Dependency probing* checks for the existence of an appropriate version of a third-party dependency (e.g., build tools, APIs). *Dependency resolution* probes for, downloads, and deploys third-party dependencies in a local cache in preparation for use in later build phases. *Tool configuration* selects the necessary options to prepare tools for use in later build phases (e.g., compiler flags).

The construction phase is comprised of two subcategories. *Build* either describes: (1) internal source dependencies (e.g., `foo.o` should be compiled before linking it into `foo.so`), or (2) how input files are translated into output files (e.g., `gcc` should be executed on `foo.c` to produce `foo.o`). *Filesystem* logic handles the creation of output directories, or implements so-called "clean" targets that remove intermediate and output files to force the build system to start from scratch.

The certification phase is most often comprised of *Unit testing* logic that configures, compiles, or executes unit tests. Similarly, *Packaging* logic describes how deliverables should be bundled together for end user consumption.

The deployment phase not only comprises *Installation* logic that describes how deliverables are deployed on a target machine, but also *Execution* logic that describes how deployed deliverables should be executed in testing environments.

5. DERIVING BASELINE VALUES

In order to ground our intuition about the extent of build cloning empirically, we perform a quantitative analysis of the benchmark. In this section, we present the results of this analysis with respect to RQ1-RQ3.

(RQ1) How much cloning is typical of build systems?

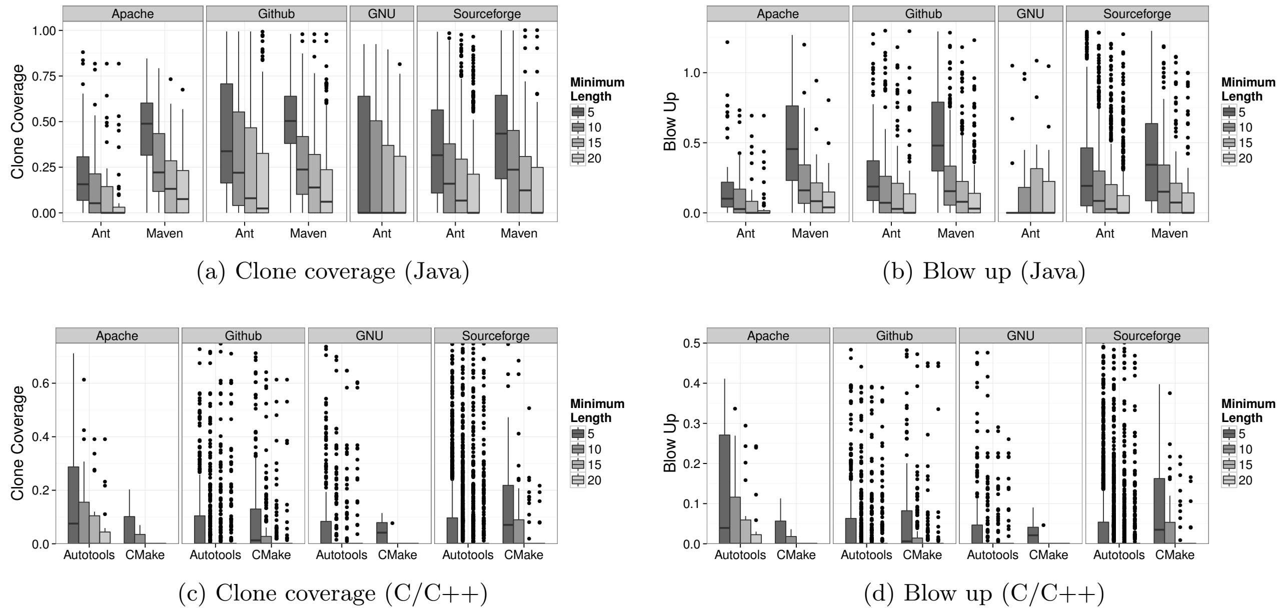
In order to address RQ1, we analyze the distributions of clone coverage and blow up in the benchmark using boxplots.

In general, build logic clones tend to be small. Figure 3 shows that clone coverage and blow up values decrease drastically when the minimum clone length is set to ten or higher, indicating that many of build specification clones cover five to nine lines. This is consistent with clones in other software artifacts, where short clones are also more frequent than long ones [13, 15].

Manual analysis of randomly selected clones with a minimum length five reveals few false positives. Hence, to simplify the remaining analyses, we only discuss the results with respect to a minimum length of five.

Cloning is much more prevalent in Java build systems than other software artifacts. Prior work shows that large software systems are expected to contain 7%-23% duplicated code [3, 17, 19], with rare cases reaching 59% [8]. Requirements documents have an average clone coverage of 13.6%, with one reported case of 71.6% [13]. Conversely, our benchmark values indicate that a clone coverage of 50% occurs rather frequently for Java build systems. Figure 3a shows that the studied Maven build systems have a median clone coverage ranging between 47%-50%. While Ant build systems have medians below 50%, the top of the box (indicating the 75th percentile) extends beyond 50% for Github, GNU, and Sourceforge build systems, indicating that more than one quarter of Ant build systems have clone coverage values that exceed 50%.

On the other hand, cloning in C/C++ build systems is less prevalent. Figure 3c shows that the median clone coverage for Autotools build systems only exceeds 0 in the Apache organization, indicating that half of the studied Autotools build systems in the Github, GNU, and Sourceforge organizations do not contain any clones. In fact, Table 1



(a) Clone coverage (Java)

(b) Blow up (Java)

(c) Clone coverage (C/C++)

(d) Blow up (C/C++)

Figure 3: Cloning metrics gathered from the studied systems. Note: scales differ among the plots.

shows that while 86%-96% of the studied Java build systems contain clones, only 41%-54% of C/C++ build systems do. Furthermore, Figure 3c shows that 75th percentile of C/C++ build systems does not exceed 30% clone coverage.

While the magnitude of the observed C/C++ build clone coverage values pale in comparison to the observed Java ones, there are still many C/C++ build systems that have plenty of clones. For example, Figure 3c shows that 25% of Autotools build systems in Apache have a clone coverage between 27%-66%. In addition, 25% of CMake build systems in Sourceforge have a clone coverage between 21%-48%.

Java build systems often have 50% clone coverage, rates that have only been observed in extreme cases when studying other software artifacts. While cloning in C/C++ build systems is less pervasive, there are still several systems that have a substantial number of clones.

(RQ2) Does technology choice influence cloning in build systems?

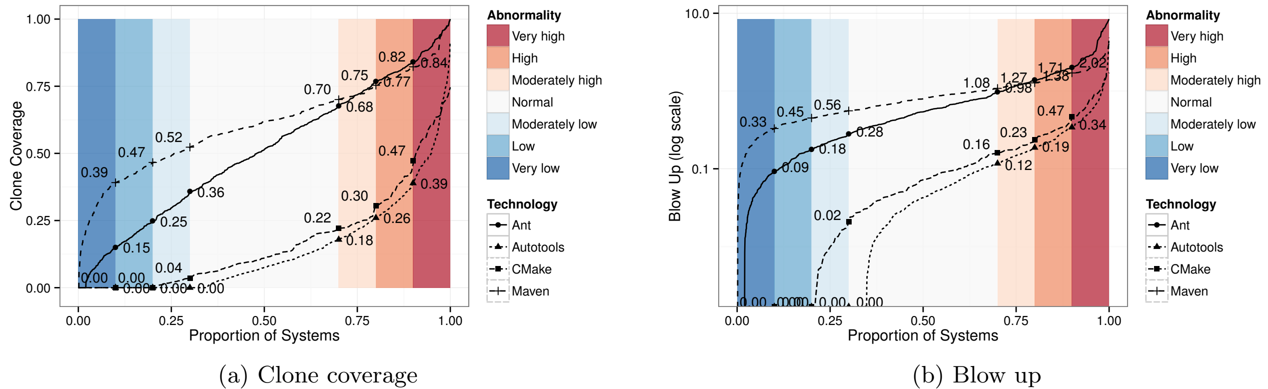
To address RQ2, we compare the distributions of Figure 3. **For Java systems, cloning is more prominent when using the more recent Maven technology than Ant.** Figures 3a and 3b show that Maven build systems tend to have higher clone coverage and blow up values than Ant ones do. Mann-Whitney U-tests (an alternative to the Student t-test with greater resiliency to non-normal distributions) confirm that the differences in clone coverage and blow up are statistically significant ($p < 0.01$) in Apache, Github, and Sourceforge. Table 1 shows that none of the studied GNU systems use Maven, so no comparison can be made.
For C/C++ systems, cloning is more prominent when using the more recent CMake build technology than Autotools. Although Figure 2 indicates that there are more clones in Autotools than CMake build systems, clone coverage and blow up statistics tend to favour CMake. First, Figure 3c shows that clones tend to cover more CMake lines than Autotools ones do. Second, Figure 3d shows that CMake clones tend to inflate build systems more than Autotools clones do. Indeed, with a minimum clone length of five, the median clone coverage and blow up of CMake exceeds that of Autotools in Github, GNU, and Sourceforge.

Mann-Whitney U-tests confirm that the differences in clone coverage and blow up are statistically significant ($p < 0.01$) in GNU and Sourceforge, however they cannot confirm a statistically significant difference in Github ($p = 0.06$). Furthermore, although the median for Autotools build systems exceeds that of CMake in Apache, Table 1 shows that our sample of three CMake systems in Apache is too small for statistical comparisons. In general, CMake build systems tend to be covered and inflated more by cloning than Autotools ones are.

The more recent CMake (C/C++) and Maven (Java) build technologies tend to be more prone to cloning than the older Autotools (C/C++) and Ant (Java) ones are.

(RQ3) Do benchmark-derived cloning thresholds vary among build technologies?

To address RQ3, Figure 4 shows the clone coverage and blow up quantile plots derived from our benchmark. We discuss the differences in thresholds for the studied Java and C/C++ technologies below.
Maven build systems have much higher thresholds for low values than Ant ones do. Complementing our RQ2 findings, Figure 4 shows that normal cloning rates in Maven are higher than those of Ant. In fact, Figure 4a shows that Maven build systems with a clone coverage below 52%, 47%, or 39% are considered moderately low to very low in our benchmark. On the other hand, Ant build systems with a clone coverage of 36%, 25%, or 15% are considered low. Similarly, Figure 4b shows that blow up values of 56%, 45%,



(a) Clone coverage

(b) Blow up

Figure 4: Quantile plots of system-level cloning metrics.

and 33% are also considered low for Maven build systems, while values of 28%, 18%, and 9% are considered low for Ant build systems. In other words, clone coverage and blow up values up to and exceeding the 30th percentile of Ant would still be beneath the 10th percentile of Maven build systems.

On the other hand, there is very little difference between the low thresholds of C/C++ build systems. Figure 4 shows that the CMake and Autotools clone coverage and blow up thresholds differ at most by four percentage points.

High thresholds are similar between Ant and Maven, and between Autotools and CMake. Figure 4a shows that the high clone coverage thresholds for Java build systems differ by two percentage points at the 70th, 80th, and 90th percentiles. C/C++ systems differ by four to eight percentage points.

Similarly, Figure 4b shows that blow up thresholds at the 70th, 80th, and 90th percentiles of the C/C++ build systems differ by four to seven percentage points. However, blow up thresholds of the studied Java build systems cover a broader range of 10 to 31 percentage points. The largest difference in blow up thresholds for Java build systems (31 percentage points) is at the 90th percentile, and is likely due to extreme blow up values in outlier systems.

Munich Re build system is indeed unusual. Regardless of the technology, clone coverage or blow up values of the same magnitude as Munich Re are not observed beneath the 90th percentile. Hence, our intuition about the Munich Re build system is empirically confirmed by the benchmark.

> *Technology-specific thresholds vary most for Java build systems with abnormally low amounts of cloning, and between CMake/Autotools and Ant/Maven for build systems with abnormally high amounts of cloning.*

6. UNDERSTANDING CLONED INFORMATION

While our quantitative analysis of Section 5 can be used to identify build systems with abnormal amounts of cloning, it does not help us to understand how cloning can be avoided (without migrating to a different technology). To address this, we perform a qualitative inspection of build logic clones. In this section, we present the results of this analysis with respect to RQ4 and RQ5.

Table 3: Clones pertaining each subcategories. Phase totals are not the sum of each subcategory because a clone may pertain to many subcategories.

Clone Counts		Ant	Maven	Autotools	CMake
All clones		56,521	71,543	23,723	3,746
Sample size (95% ± 5%)		382	382	378	349
Phase	**Subcategory**	**Ant**	**Maven**	**Autotools**	**CMake**
Config.	Deps. Probing	3%	0%	1%	8%
	Deps. Resolution	1%	54%	0%	4%
	Tool Configuration	29%	32%	21%	32%
	Phase Total	**32%**	**79%**	**22%**	**40%**
Cons.	Build	47%	16%	48%	65%
	Filesystem	32%	1%	19%	1%
	Phase Total	**64%**	**17%**	**56%**	**66%**
Cert.	Unit Testing	12%	4%	13%	11%
Pkg.	Packaging	25%	21%	21%	2%
Depl.	Installation	8%	1%	9%	7%
	Execution	3%	0%	0%	0%
	Phase Total	**11%**	**1%**	**9%**	**7%**

(RQ4) What type of information is typically cloned in build specifications?

In order to address RQ4, we need to analyze a representative sample of clones in each studied technology. As the total number of clones is very large, and there was no automatic means of determining the build phase of the clone, we sampled the clones for manual inspection. To obtain the proportion estimates that are within 5% bounds of the actual proportion with 95% confidence level, we use the sample size calculation of $s = \frac{z^2 p(1-p)}{0.05^2}$, where p is the proportion we want to estimate and $z = 1.96$. Since we did not know the proportion in advance, we use $p = 0.5$. We, further, correct for the finite population of clones to obtain 382 Ant, 382 Maven, 379 Autotools, and 349 CMake clones. Table 3 shows the proportion of randomly selected clones that are associated with each build subcategory.

Cloning focus shifts from construction to configuration in Maven. Table 3 shows that the majority of Ant clones impact the construction phase (64%). 47% of clones are associated with the build category and 32% with the filesystem category. The next most frequently cloned phase (configuration) appears in half as many clones (32%).

Many of these construction clones replicate entire Ant targets that create or delete temporary output directories or compile Java source code. Since a single invocation of the Java compiler will automatically resolve dependencies between input source files [7], Ant targets that compile Java code often invoke the Java compiler specifying all impacted

Java files as inputs using wildcards. Hence, the compile target is generic, and often cloned in several Ant specifications.

While construction accounts for many of the clones in Ant build systems (64%), most Maven clones impact configuration (79%). The next most frequently cloned phase (packaging) appears in less than a third as many clones (21%).

We observed that many of the Maven clones replicate third-party dependency lists or plugin configuration among subsystems. While this ensures that each subsystem can be built independently of the others, it imposes a heavy load on maintainers, who will need to update several `pom.xml` files in order to modify third-party dependency lists or update plugin configurations.

Construction is the most heavily cloned build phase in C/C++ build systems. Table 3 shows that the build subcategory represents 48% and 65% of Autotools and CMake clones respectively. The filesystem category is also detected in 19% of Autotools and 1% of CMake clones. All in all, the construction phase accounts for 56% of Autotools clones and 66% of CMake clones.

While the Autotools construction phase is most frequently cloned (56%), Table 3 shows that packaging details are also cloned often (21%). Yet packaging details are rarely cloned in CMake (2%). Many of these Autotools packaging clones have to do with repetition of data file lists among subsystems. Since Autotools build specifications generate recursive `make` build systems, variables are not shared among subsystems [23]. Although Autotools offers developers an `include` directive, we have observed that rather than place shared variables in a header-like file, developers often clone variables that have a shared scope. Similar to Maven, where dependency lists and plugin configuration were replicated, developers likely duplicate shared variables to facilitate subsystem independence. However, this makes system-wide changes more difficult.

Conversely, we observe that CMake packaging details are rarely cloned. We observe that developers leverage built-in CPack functionality of CMake [20], where packaging details are typically specified in a single location: `CPackConfig.cmake`. This eliminates the need to replicate packaging details in subsystem specifications.

There are more configuration clones in the more recent build technologies. Table 3 shows that there are more than twice as many configuration clones in Maven build systems (79%) than Ant ones (32%). Similarly, there are almost twice as many configuration clones in CMake (40%) than there are in Autotools (22%). In Section 5, we report that these more recent technologies are more prone to cloning (RQ2). The shift of cloning tendencies towards configuration likely contributes to the inflated cloning values we observe in the more recent technologies.

Configuration details are cloned more often in the more recent CMake and Maven build technologies. For Java build systems, Maven clones favour the configuration phase, while Ant clones (and clones in C/C++ builds) favour construction. CMake packaging support (CPack) helps to reduce cloning in the packaging phase.

(RQ5) How do build systems with few clones achieve low clone rates?

To address RQ5, we analyze clones in the systems with the lowest and highest cloning rates for each studied technology.

```
<?xml version="1.0"?>

<!-- Define references to files containing common targets -->
<!DOCTYPE project [
    <!ENTITY modules-common SYSTEM "../modules-common.ent">
]>

<!-- the project element's name attribute will be used in the name of the module's jar file -->
<project name="hee" default="all">
    <!-- Include the file containing common targets. -->
    &modules-common;
</project>
```

Figure 5: Using XML entity expansion to import common build code in the Keel system.

Much Java build cloning can be avoided by exploiting the underlying XML representation. We observe that entire files are duplicated in the Ant and Maven systems with the highest clone coverage. In these cases, development teams duplicate existing build specifications to rapidly develop new subsystems. However, developers have referred to maintaining such build systems fraught with clones as a "nightmare" (*cf.* Section 3). Defect fixes or updates to dependency lists, tool configuration, and packaging details must be carefully replicated among the clones to ensure that builds continue to assemble deliverables correctly.

On the other hand, we have observed that in addition to abstraction mechanisms provided by Ant and Maven (e.g., the `include` and `import` tasks), XML-based build systems avoid cloning by leveraging the underlying XML representation. In prior work, we note that the JBoss build system leverages XML entity expansion in Ant to implement a framework-driven build system referred to as "buildmagic" [21]. Indeed, Figure 5 shows how one can use XML entity expansion to avoid duplicating shared build code in subsystem build specifications. We also find that the Ant test suite includes regression tests to ensure that entity expansion continues to work, suggesting that it is not a workaround, but instead is intentionally supported functionality.

Many C/C++ build logic clones can be avoided by duplicating templates automatically when building. Many of the studied C/C++ systems provide development APIs. As such, they ship examples of how to use various API functionality with their deliverables. These examples include accompanying build specifications. However, in the C/C++ systems with the highest clone coverage, we find that many of these example build specifications are file clones of each other, which poses maintainability problems.

One of the studied systems with a low clone coverage avoids these clones by duplicating and specializing template build specification automatically using shell scripts during the construction phase. Using this approach, cloning shared build code in example build specifications can be avoided.

XML entity expansion can be used to avoid cloning shared build code in Java build systems. Cloning of build logic shipped with API usage examples can be avoided by automatically deriving specifications at build-time.

7. THREATS TO VALIDITY

We now discuss the threats to the validity of our analysis.

Construct validity. Our clone detection tool is configured to only detect Type I (exact) clones. Since we do not detect Type II, III, or IV clones, our cloning results should be interpreted as lower bounds rather than exact values.

Internal validity. We assume that large values of cloning metrics suggest maintenance problems illustrated in our Mu-

nich Re example. Yet, recent research suggests that despite the inherent maintainability issues, cloning may not always be harmful [16, 27]. Nonetheless, we find that developers complain about maintainability problems in heavily cloned build systems, suggesting that excessive cloning makes build system maintenance more difficult. Furthermore, our prior work shows that unintentional inconsistent changes do occur in large industrial systems [14].

Reliability validity. We conservatively detect build specifications using filename conventions. Although our classification tool is lightweight enough to iterate over all files in our large corpus, we may miss files that are build-related that do not conform to filename conventions.

External validity. Although our benchmark covers a large corpus of 3,872 systems, a limited number of open source organizations are covered. As such, our results may not generalize to other open source or even proprietary build systems. However, since any build system needs to implement the phases outlined in Section 2, we believe that our benchmark is a sound starting point. We plan to extend our benchmark to include proprietary systems in future work.

There are hundreds of build technologies and of these, we only include four in our benchmark. Our findings are entirely bound to the studied technologies. However, it is our experience that the technologies that we have selected are quite popular in open source communities and are frequently used in industry. Furthermore, Ant shares many similarities with MSBuild. Specifications for both technologies are expressed using abstract targets and tasks specified in an XML format. Hence, we suspect that the characteristics of cloning we observed in Ant will also appear in MSBuild systems. We plan to inspect this suspicion by expanding the scope of our benchmark to include MSBuild in future work.

8. RELATED WORK

In this section, we discuss the related work with respect to clone detection and build maintenance.

Clone detection. Clone detection for source code is a mature research area. A plethora of detection tools have been suggested and many studies have been investigated the impact of code clones on software maintenance. Comprehensive surveys of prior work on clone detection have been published by Koschke [18] and Roy *et al.* [29, 30].

Yet, as Robles *et al.* emphasize, a software system comprises artifacts other than source code that also require research [28]. Our prior work highlights research areas for clone detection beyond source code [12]. While researchers have recently extended clone detection research to other software artifacts, such as requirements specifications [13], models [5], and test cases [9], to the best of our knowledge, this work is the first to analyze clones in build systems.

Build maintenance. Recent research has shown that build system maintenance imposes a non-trivial "tax" on software development [10, 22]. Indeed, our prior work shows that source code and build system tend to *co-evolve*, i.e., changes in the source code often induce changes in the build system and vice versa [1, 21]. Indeed, Neitsch *et al.* find that abstractions tend to "leak" between source code and build system [26]. Suvorov *et al.* find that when the maintenance of the build system grows unwieldy, development teams take on large (and costly) build refactoring and migration projects [32]. This paper analyzes build logic cloning in a benchmark of open source build systems. This benchmark is intended to help identify if and where developers should focus build refactoring effort.

9. CONCLUSIONS

Build systems play a crucial role in software development. They tend to grow in terms of complexity as a software project ages [1, 21]. When build system complexity grows unwieldy, build maintenance becomes difficult, and development teams refactor build systems to restore order.

In order to determine if and where build refactoring should be applied, CQSE performs quality assessments of build systems. In this paper, we discuss how a benchmark of build logic clones can empirically ground metrics used in these assessments. Through analysis of the benchmark, we make the following observations:

- 50% clone coverage rates, which have only been recorded in rare cases in other software artifacts [8], frequently occur in Java build systems.
- The more recent CMake and Maven build technologies tend to be more prone to cloning, especially of configuration details like API dependencies, than the older Autotools and Ant technologies respectively.
- While build logic cloning can be difficult to avoid, it is not a necessity, i.e., we have observed build systems with little cloning using each studied technology.
- Templating and inclusion mechanisms beyond those provided by build technologies are employed to reduce build logic cloning, suggesting that the mechanisms provided by build technologies are insufficient.

Future work. While we have seen the shortcomings of a clone-based build system design at Munich Re, we do not know to what extent it can be generalized. For example, it could be that different types of build specification information have differences in change-proneness. Clones in some areas (e.g. construction) could be more problematic than in others (e.g. configuration). Analysis of the evolution of clones in build systems could help to further our knowledge.

Refactoring to reduce cloning at Munich Re. The benchmark-derived thresholds confirm that the clone-based build system design at Munich Re is unusual. Munich Re has decided to restructure the build system. To facilitate this, we are creating reusable build logic components that can be shared among build specifications (without cloning).

The analysis we performed in this paper helped us in designing the solution. First, to work around the limitations of the MSBuild abstraction mechanisms, we adopt a practice that we observed in C/C++ build systems, where common build logic is stored in a template that is copied and specialized automatically during an initial phase in the build process. Second, similar to Maven build systems, our solution divides the core build logic that drives the different build phases into individual plugins that enable automated testing, packaging, and deployment.

However, the new build solution also requires a more structured change process. Since changes to shared build components affect all build specifications that rely on them, they must be more carefully maintained than the prior clone-based solution was. To this end, Munich Re has created a dedicated test bed in which build component changes can

be evaluated before they are deployed to production builds. Furthermore, we are creating a dashboard that displays nightly clone detection results as an early-warning system against proliferation of cloning in the new build system.

Acknowledgments

This research was partially supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) and the German Federal Ministry of Education and Research (BMBF), grant "EvoCon, 01IS12034A".

References

[1] B. Adams, K. D. Schutter, H. Tromp, and W. D. Meuter. The Evolution of the Linux Build System. *Electronic Communications of the ECEASST*, 8, 2008.

[2] T. L. Alves, C. Ypma, and J. Visser. Deriving Metric Thresholds from Benchmark Data. In *Proc. of the 26th Int'l Conf. on Software Maintenance (ICSM)*, pages 1–10, 2010.

[3] B. S. Baker. On Finding Duplication and Near-Duplication in Large Software Systems. In *Proc. of the 2nd Working Conf. on Reverse Engineering (WCRE)*, pages 86–95, 1995.

[4] D. B. Carr, R. J. Littlefield, W. L. Nicholson, and J. S. Littlefield. Scatterplot Matrix Techniques for Large N. *Journal of the American Statistical Association*, 82(398):424–436, 1987.

[5] F. Deissenboeck, B. Hummel, E. Juergens, B. Schaetz, S. Wagner, J.-F. Girard, and S. Teuchert. Clone Detection in Automotive Model-Based Development. In *Proc. of the 30th Int'l Conf. on Software Engineering (ICSE)*, pages 603–612, 2008.

[6] F. Deissenboeck, E. Juergens, B. Hummel, S. Wagner, B. M. y Parareda, and M. Pizka. Tool Support for Continuous Quality Control. *IEEE Software*, 25(5):60–67, 2008.

[7] M. Dmitriev. Language-Specific Make Technology for the Java Programming Language. In *Proc. of the 17th Conf. on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)*, pages 373–385, 2002.

[8] S. Ducasse, M. Rieger, and S. Demeyer. A Language Independent Approach for Detecting Duplicated Code. In *Proc. of the 15th Int'l Conf. on Software Maintenance (ICSM)*, pages 109–118, 1999.

[9] B. Hauptmann, M. Junker, S. Eder, E. Juergens, and R. Vaas. Can clone detection support test comprehension? In *Proc. of the 20th Int'l Conf. on Program Comprehension (ICPC)*, pages 209–218, 2012.

[10] L. Hochstein and Y. Jiao. The cost of the build tax in scientific software. In *Proc. of the 5th Int'l Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 384–387, 2011.

[11] J. Humble and D. Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation.* Addison-Wesley, 2010.

[12] E. Juergens. Research in cloning beyond code: a first roadmap. In *Proc. of the 5th Int'l Workshop on Software Clones (IWSC)*, pages 67–68, 2011.

[13] E. Juergens, F. Deissenboeck, M. Feilkas, B. Hummel, B. Schaetz, S. Wagner, C. Domann, and J. Streit. Can Clone Detection Support Quality Assessments of Requirements Specifications? In *Proc. of the 32nd Int'l Conf. on Software Engineering (ICSE)*, volume 2, pages 79–88, 2010.

[14] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner. Do Code Clones Matter? In *Proc. of the 31st Int'l Conf. on Software Engineering (ICSE)*, pages 485–495, 2009.

[15] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code. *Transactions on Software Engineering (TSE)*, 27(7):654–670, 2002.

[16] C. J. Kasper and M. W. Godfrey. "Cloning considered harmful" considered harmful: patterns of cloning in software. *Empirical Software Engineering*, 13(6):645–692, 2008.

[17] K. A. Kontogiannis, R. Demori, E. Merlo, M. Galler, and M. Bernstein. Pattern Matching for Clone and Concept Detection. *Automated Software Engineering*, 3(1-2):77–108, 1996.

[18] R. Koschke. Survey of research on software clones. In *Duplication, Redundancy, and Similarity in Software.* Dagstuhl Seminar Proceedings, 2007.

[19] B. Laguë, D. Proulx, E. M. Merlo, J. Mayrand, and J. Hudepohl. Assessing the Benefits of Incorporating Function Clone Detection in a Development Process. In *Proc. of the 13th Int'l Conf. on Software Maintenance (ICSM)*, pages 314–321, 1997.

[20] K. Martin and B. Hoffman. *Mastering CMake, 5th Edition.* Kitware Inc., Clifton Park, NY, USA, 2009.

[21] S. McIntosh, B. Adams, and A. E. Hassan. The evolution of Java build systems. *Empirical Software Engineering*, 17(4-5):578–608, 2012.

[22] S. McIntosh, B. Adams, T. H. D. Nguyen, Y. Kamei, and A. E. Hassan. An Empirical Study of Build Maintenance Effort. In *Proc. of the 33rd Int'l Conf. on Software Engineering (ICSE)*, pages 141–150, 2011.

[23] P. Miller. Recursive make considered harmful. In *Australian Unix User Group Newsletter*, volume 19, pages 14–25, 1998.

[24] A. Mockus. Software support tools and experimental work. In *Proc. of the Int'l Conf. on Empirical Software Engineering Issues: Critical Assessment and Future Directions*, pages 91–99, 2007.

[25] A. Mockus. Amassing and indexing a large sample of version control systems: Towards the census of public source code history. In *Proc. of the 6th Working Conf. on Mining Software Repositories (MSR)*, pages 11–20, 2009.

[26] A. Neitsch, K. Wong, and M. W. Godfrey. Build System Issues in Multilanguage Software. In *Proc. of the 28th Int'l Conf. on Software Maintenance (ICSM)*, pages 140–149, 2012.

[27] F. Rahman, C. Bird, and P. Devanbu. Clones: what is that smell? *Empirical Software Engineering*, 17(4-5):503–530, 2012.

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

[29] C. Roy, J. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 2009.

[30] C. K. Roy and J. R. Cordy. A survey on software clone detection research. Technical Report 541, Queen's University at Kingston, 2007.

[31] P. Smith. *Software Build Systems: Principles and Experience.* Addison-Wesley, 1st edition, March 2011.

[32] R. Suvorov, M. Nagappan, A. E. Hassan, Y. Zou, and B. Adams. An Empirical Study of Build System Migrations in Practice: Case Studies on KDE and the Linux Kernel. In *Proc. of the 28th Int'l Conf. on Software Maintenance (ICSM)*, pages 160–169, 2012.