Mining Co-Change Information to Understand when Build Changes are Necessary

Shane McIntosh*, Bram Adams†, Meiyappan Nagappan*, and Ahmed E. Hassan*
*School of Computing, Queen's University, Canada; {mcintosh, mei, ahmed}@cs.queensu.ca
†Polytechnique Montréal, Canada; bram.adams@polymtl.ca

Abstract—**As a software project ages, its source code is modified to add new features, restructure existing ones, and fix defects. These source code changes often induce changes in the build system, i.e., the system that specifies how source code is translated into deliverables. However, since developers are often not familiar with the complex and occasionally archaic technologies used to specify build systems, they may not be able to identify when their source code changes require accompanying build system changes. This can cause build breakages that slow development progress and impact other developers, testers, or even users. In this paper, we mine the source and test code changes that required accompanying build changes in order to better understand this co-change relationship. We build random forest classifiers using language-agnostic and language-specific code change characteristics to explain when code-accompanying build changes are necessary based on historical trends. Case studies of the Mozilla C++ system, the Lucene and Eclipse open source Java systems, and the IBM Jazz proprietary Java system indicate that our classifiers can accurately explain when build co-changes are necessary with an AUC of 0.60-0.88. Unsurprisingly, our highly accurate C++ classifiers (AUC of 0.88) derive much of their explanatory power from indicators of structural change (e.g., was a new source file added?). On the other hand, our Java classifiers are less accurate (AUC of 0.60-0.78) because roughly 75% of Java build co-changes do not coincide with changes to the structure of a system, but rather are instigated by concerns related to release engineering, quality assurance, and general build maintenance.**

I. Introduction

Build systems specify how source code is translated into deliverables. Often supporting hundreds of configurations [1] and specified using thousands of build files (e.g., `make` or `ant` files) [2], build systems are complex systems of their own. Such complex build systems require regular maintenance in order to continue functioning correctly. Our prior work shows that, from release to release, source code and build system tend to *co-evolve* [3, 4], i.e., changes to the source code can induce changes in the build system, and vice versa. This continual build maintenance generates considerable overhead on development activities [2, 5–7].

The overhead generated by the build system is exacerbated by the difficulty of identifying the code changes that require accompanying build system changes. Seo *et al.* show that 30%-37% of builds triggered by Google developers on their local copies of the source code are broken, with neglected build maintenance being the most commonly detected root cause [8]. If those build breakages are not fixed before the changes are committed to upstream repositories, then their team as a whole will be negatively impacted. For example, Kwan *et al.* find that 31% (60/191) of the studied IBM team builds were broken [9]. Furthermore, Hassan and Zhang find that 15% (209/1,429) of the studied IBM certification builds (i.e., builds that the development team believed were ready for testing) were broken [10]. These broken team builds prevent quality assurance teams from reproducing and testing actively developed versions of a system in a timely fashion, slowing development progress and the release process.

In order to avoid these costly build breakages, we set out to explore the following central question:

> *Can build changes be fully explained using characteristics of co-changed source and test code files?*

For this purpose, we construct random forest classifiers using language-agnostic and language-specific characteristics of source and test code changes to understand when build changes are required. Through case studies of the Mozilla system (primarily implemented using C++), and three Java systems, we address the following three research questions:

(RQ1) How often are build changes accompanied by source/test code changes?
Only a minority of the source/test code changes require accompanying build changes (4%-26%), with the majority of those build changes co-occurring with source/test code changes (53%-88%).

(RQ2) Can we accurately explain when build co-changes are necessary using code change characteristics?
Yes, our classifiers can explain the source and test code changes that require accompanying build changes with an AUC of 0.60-0.88. Our Java classifiers are less accurate than the C++ classifiers (AUC of 0.60-0.78 vs. 0.88) because 75% ($\pm$10%) of Java build changes are not related to changes to the structure of a system.

(RQ3) What are the most influential code change characteristics for explaining build co-changes?
Our Mozilla (C++) classifiers derive much of their explanatory power from indicators of structural changes like adding new source files. On the other hand, since Java build co-changes rarely coincide with these structural changes, our Java classifiers derive most of their explanatory power from the historical co-change tendencies of the modified files and deeper code change characteristics like the addition or removal of `import` statements that reference non-core APIs.

Paper organization. The remainder of the paper is organized as follows. Section II highlights the importance of consistency between source code and build system. Section III describes our case study design, while Sections IV and V present the results with respect to our three research questions. Section VI discusses threats to the validity of our study. Section VII surveys related work. Finally, Section VIII draws conclusions.

II. The Importance of Maintaining Consistency between Source Code and Build System

Build systems orchestrate order-dependent compiler and tool invocations in order to create project deliverables, and are often implemented in terms of: (1) a *configuration layer* for selecting tools and features (e.g., Android back-end vs. Windows back-end); and (2) a *construction layer* for producing deliverables based on configuration choices [3, 4].

Executing the complex builds of large software systems can be prohibitively expensive, often taking hours, if not days to complete. For example, Windows builds of the Firefox system take more than two and a half hours to complete on dedicated build machines.[1] Hassan and Zhang report that certification builds of a large IBM system take more than 24 hours to complete [10]. The lengthy and expensive build processes can slow development progress if the builds are frequently broken.

Such build breakages can often be linked to inconsistencies between the source code and the build system. Indeed, despite the strong link between the source code and the build system, developers in large teams are often not familiar with the internal structure of the build system [2]. This lack of familiarity is problematic, since it may cause developers to miss build changes when they are necessary, leading to build breakages [8] or even worse, producing incorrect deliverables. For example, Mozilla defect 417037[2] describes how end users in a networked environment were unable to use the web browser address and search bars. It was not until months later that the issue, an incorrect library version picked up by the build system, was fixed.

In order to avoid inconsistencies between the source code and the build system, we set out to better understand the code changes that require accompanying build changes. Specifically, we mine historical repositories to construct classifiers that use characteristics of source and test code changes to understand when accompanying build changes are required.

III. Case Study Design

In this section, we present our rationale for selecting our research questions, describe the studied systems, and present our data extraction and analysis approaches.

(RQ1) How often are build changes accompanied by source/test code changes?
If the majority of work items containing build changes do not contain accompanying source or test code changes, then code change characteristics would make poor indicators of build change. Hence, before building our classifiers, we want to know how frequently build and source/test code co-change.

(RQ2) Can we accurately explain when build co-changes are necessary using code change characteristics?
Prior work has shown that classifiers can be built to accurately explain phenomena in software engineering [10–14]. We conjecture that such classifiers can be built to accurately explain when a build change is necessary using code change characteristics.

(RQ3) What are the most influential code change characteristics for explaining build co-changes?
Knowing which code change characteristics are influential indicators of build change could help practitioners to identify code changes that require accompanying build changes.

A. Studied Systems

In order to address our research questions, we study one large system primarily implemented using C++ and three systems primarily implemented using Java. The studied systems are of different sizes and domains in order to combat potential bias in our conclusions. More importantly, the studied systems carefully record co-change data at the work item level (see below), which is a critical precondition for our co-change analysis. The scarcity of carefully recorded work item data in practice prevents us from analyzing a larger sample of systems.

Table I provides an overview of the studied systems. Mozilla is a suite of internet tools including the Firefox web browser. Eclipse is an Integrated Development Environment (IDE), of which we studied the core subsystem. Lucene is a library offering common search indexing functionality. IBM Jazz$^{\text{TM}}$[3] is a proprietary next-generation IDE.

B. Data Extraction

Software projects evolve through continual change in the source code, test code, build system, and other artifacts. Changes to a file are often collected in *file patches* that show the differences between subsequent revisions of a single file. These file patches are typically logged in a Version Control System (VCS). In addition to logging file patches, modern VCSs track *transactions* (a.k.a., *atomic commits*), i.e., collections of file patches that authors commit together.

A *work item* is a development task such as fixing a bug, adding a new feature, or restructuring an existing feature. Several transactions may be required to complete a work item, since developers from different teams may need to collaborate. Work items are often logged in an Issue Tracking System (ITS) like Bugzilla or IBM Jazz and branded with a unique identifier. This ID helps to identify the transactions that are associated with a work item.

We extract work item data from each of the studied systems in order to address our research questions. Figure 1 provides an overview of our approach, for which the data extraction

[1] http://tbpl.mozilla.org/
[2] https://bugzilla.mozilla.org/show_bug.cgi?id=417037
[3] http://www.jazz.net. IBM and Jazz are trademarks of IBM Corporation in the US, other countries, or both.

TABLE I
CHARACTERISTICS OF THE STUDIED PROJECTS.

Project	**Mozilla**		**Eclipse-core**		**Lucene**		**Jazz**	
Domain	Internet Suite		IDE		Search Indexing Library		IDE	
Timeframe	1998 – 2010		2001– 2010		2010 – 2013		2007 – 2008	
# Project Files	123,175		5,490		18,811		67,357	
Source Files (#, % of total)	43,952	35%	2,391	43%	8,879	47%	45,275	67%
Test Files (#, % of total)	30,835	25%	1,211	22%	4,898	26%	14,738	22%
Build Files (#, % of total)	10,709	9%	477	9%	421	2%	5,967	9%
Other Files (#, % of total)	37,679	31%	1,411	26%	4,613	25%	1,377	2%
# Transactions	210,400		6,391		9,856		36,557	
# Work Items	55,199		2,452		3,280		11,611	
# Transactions with Work Items	79,242	38%	4,092	64%	6,046	61%	22,485	62%
Source Work Items (#, % work items)	45,815	83%	2,130	87%	2,553	78%	9,869	85%
Test Work Items (#, % work items)	9,383	17%	765	31%	2,084	64%	2,786	24%
Build Work Items (#, % work items)	14,477	26%	427	17%	443	14%	608	5%
Other Work Items (#, % work items)	5,275	10%	165	7%	254	8%	973	8%
Source-Build Co-Change Work Items (#, % source, % build)	12,450	27%, 86%	350	16%, 82%	194	6%, 44%	437	4%, 72%
Test-Build Co-Change Work Items (#, % test, % build)	4,198	45%, 29%	154	20%, 36%	183	9%, 41%	219	8%, 36%
Source- or Test-Build Co-Change Work Items (#, % source and test)	12,698	26%	382	17%	234	7%	468	4%
Build without Source or Test Work Items (#, % build)	1,779	12%	82	19%	209	47%	140	23%

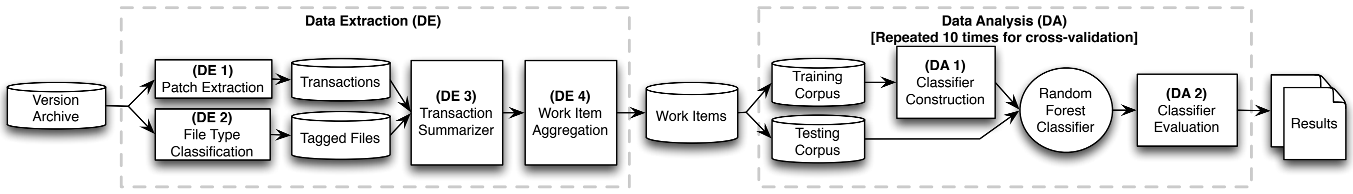

Data Extraction (DE)
Version Archive
(DE 1) Patch Extraction
(DE 2) File Type Classification
Transactions
Tagged Files
(DE 3) Transaction Summarizer
(DE 4) Work Item Aggregation
Work Items
Data Analysis (DA) [Repeated 10 times for cross-validation]
Training Corpus
Testing Corpus
(DA 1) Classifier Construction
Random Forest Classifier
(DA 2) Classifier Evaluation
Results

Fig. 1. An overview of our data extraction and analysis approaches.

component is broken down into four steps. We briefly describe each step of our data extraction approach below.

(DE 1) Patch extraction. After gathering the VCS archives for each studied project, we extract all transactions as well as authorship, timestamps, and commit message metadata. Although the studied systems use different VCSs (i.e., Git and Mercurial), we wrote scripts to extract transactions and metadata in a common format.

(DE 2) File type classification. In order to assess whether a transaction (and hence, a work item) impacts the build system, we use the file type classification process from our prior work [2], which tags each file in a project history as either a source, test, or build file. Build system files include helper scripts, as well as construction and configuration layer specifications (such as `make` or `ant` files). Source code files implement software logic. Test code files contain automated tests that check the software for regressions.

The file type classification process was semi-automatic. Table I lists the number of files classified under each category for the studied systems. Most files could be classified using filename conventions, e.g., file extensions. However, many extensions were ambiguous, e.g., ".xml". After classifying unambiguous file types, the remaining files were classified manually. For example, of the 123,175 Mozilla files, approximately 20,000 files remained unclassified after all known filename conventions were exhausted. Through manual inspection, we found project-specific extension types that could be classified automatically, further reducing the number of unclassified files to roughly 5,000. The remaining 5,000 or so files were classified manually.

(DE 3) Transaction summarizer. Next, we produce transaction summaries for all transactions that contain source, test, and/or build file changes, which consist of: (1) measured characteristics that describe the code change, and (2) a boolean value noting whether or not at least one build file was changed. A summary of the measured code change characteristics and the rationale for their use is given in Table II.

(DE 4) Work item aggregation. Our prior work has shown that transactions are too fine-grained to accurately depict development tasks [2]. It may take several transactions to resolve a work item. In such cases, build changes often appear in different transactions than the corresponding source or test code changes. To avoid missing cases of co-change, we group transactions that address the same work item together by examining the transaction commit messages for work item IDs.

Bias Assessment. As shown in Table I, the aggregation to work items is lossy, since it relies heavily on developer behaviour to link transactions to work items. Overall, 38%-64% of the transactions can be connected to work items. The lack of well-linked work item data is a known problem [15, 16]. Hence, we first evaluate whether the lossy nature of work item aggregation introduces bias in our dataset. We are primarily concerned with two types of bias:

1) Time periods in project history may be missing due to the lossy nature of work item aggregation, i.e., we only have work item data for certain time periods.
2) Work item linkage may be a property of project experience [15], i.e., experienced developers might be more likely to provide the links to work items in their commits.

To study the extent of these biases, we compare the number

TABLE II
A TAXONOMY OF THE STUDIED CODE CHANGE CHARACTERISTICS. EACH IS MEASURED ONCE FOR SOURCE CODE AND ONCE FOR TEST CODE.

	Attribute Name	Type	Definition	Rationale
Language-Agnostic	File added	Boolean	True if a given work item adds new source or test files.	Adding new source files changes the structural layout of the codebase, which may require accompanying build changes to include the new file.
	File deleted	Boolean	True if a given work item deletes old source or test files.	Deleting old source files changes the structural layout of the codebase, which may require accompanying build changes to disregard the dead file.
	File renamed	Boolean	True if a given work item renames source or test files.	Renaming a source file alters the structural layout of the codebase, invalidating prior dependencies while creating new ones, which may require accompanying build changes.
	File modified	Boolean	True if a given work item modifies existing source or test files.	With the exception of the special language-specific cases (see below), modification of source code should rarely require build changes, since modifications do not alter the structure of a system.
	Prior build co-changes*	Numeric	For each source and test file involved in a given work item, we compute the proportions of prior work items that were build co-changing. We select the maximum proportion of the work item's changed files.	Historical co-change tendencies may provide insight into future co-change trends.
	Number of files*	Numeric	The number of source and test files that were involved in a given work item.	Changes that impact more files may be more likely to require accompanying build changes.
Language-Specific	Changed dependencies	Boolean	True if a given work item adds or removes dependencies on other code through `#include` preprocessor directives for C++ code or `import` statements in Java code.	Dependency changes may need to propagate to the build system.
	Added/removed dependencies	Boolean	True if the dependency being: (1) added does not appear in any other source or test file, or (2) removed has been completely removed from all source and test files.	Adding or removing dependencies indicates that a new dependency may have been introduced or an old one relaxed. Such changes may need to propagate to the build system.
	Added/removed non-core dependencies	Boolean	True if the conditions listed for Added/removed dependencies are satisfied by a dependency that is not part of the core language API.	Adding or removing dependencies on core language APIs will not have an impact on the build process, and hence may introduce noise in the Added/removed dependencies metric.
	Changed conditional compilation (C++ only)	Boolean	True if a given work item adds new or removes old #if[n][def] preprocessor directives.	Conditional compilation is often used to control platform- or feature-specific functionality in the source or test code. The conditions for these blocks of code often depend on configuration layer settings.

* Could not be calculated for Jazz due to privacy concerns.

of transactions per month to the number of work items per month and study how these measures evolve over time. We also compare developer contributions in terms of the number of transactions and work items. Figure 2 visualizes these distributions using beanplots [17]. Beanplots are boxplots in which the vertical curves summarize the distributions of different datasets. The horizontal black lines indicate the median values. Due to differences in scale, we separate the Java beanplots (Figure 2b) from the Mozilla one (Figure 2c).

Figures 2a and 2b show that Eclipse-core, Lucene, and Jazz share highly symmetrical beanplots, indicating that transactions and work items share similar temporal and developer contribution characteristics. The median lines in Jazz and Lucene are almost identical, while the median of the work items is higher than that of the transactions in the Eclipse-core project. The slight difference in medians indicates that the work item granularity introduces minimal skew with respect to the transaction data.

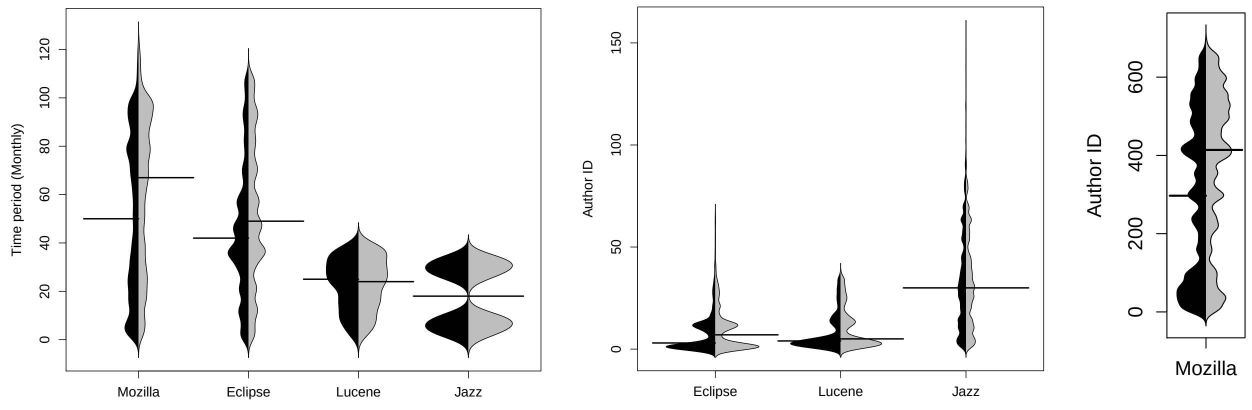
The asymmetrical nature of the Mozilla plot in Figure 2a shows that there is skew introduced, i.e., very few transactions could be linked to work items in the initial Mozilla development months. Once the practice of recording the work item ID in the commit message was more firmly established, the symmetry of the beanplot increases, indicating that the temporal characteristics between the two datasets are similar from that point on. To resolve this, we removed the initial 12 development months of Mozilla prior to performing our case study. Figure 2c shows that this filtering also makes the distribution of Mozilla developer contributions less skewed, i.e., the bias in our data has been controlled.

C. Data Analysis

Figure 1 provides an overview of our data analysis approach. The work items are split into training and testing corpora. Classifiers are constructed using the training corpus, and their performance is evaluated on work items in the testing corpus. We briefly describe each step in our analysis below.

(DA 1) Classifier Construction. We use the random forest technique to construct classifiers (one for each studied system) that explain when build changes are necessary. The random forest technique constructs a large number of decision trees at training time [18]. Each node in a decision tree is split using a random subset of all of the attributes. Performing this random split ensures that all of the trees have a low correlation between them [18]. Since each tree in the forest may report a different outcome, the final class of a work item is decided by aggregating the votes from all trees and deciding whether the final score is higher than a chosen threshold.

(DA 2) Classifier Evaluation. To evaluate the performance of a classifier, we use it to classify work items in a testing



(a) Monthly changes (b) Changes made by each developer (c) Changes made by each developer

Fig. 2. Comparison of the time and developer distribution of transactions (black) and work items (grey).

corpus and compare its deduction against the known result. To obtain the testing corpus and evaluate the performance of our classifiers, we use tenfold cross-validation. Cross-validation splits the data into ten equal parts, using nine parts for the training corpus, setting aside one for the testing corpus. The process is repeated ten times, using a different part for the testing corpus each time.

Table III shows the confusion matrix constructed based on the cross-validation classification results. The performance of the decision tree is measured in terms of recall, precision, F-measure, and AUC. We describe each metric below.

- **Recall**: Of all known build co-changing work items, how many were classified as such, i.e., $\frac{a}{a+b}$.
- **Precision**: Of the work items that are classified as build co-changing, how many actually did co-change, i.e., $\frac{a}{a+c}$.
- **F-measure**: The harmonic mean of precision and recall, i.e., $2 \cdot \frac{precision \cdot recall}{precision + recall}$.
- **Area Under the Curve (AUC)**: The area under the curve that plots true positive rate ($\frac{a}{a+b}$) against the false positive rate ($\frac{c}{c+d}$), for various values of the chosen threshold used to determine whether a work item is classified as build co-changing. Values of AUC range between 0 (worst classifier performance) and 1 (best classifier performance).

We first construct classifiers using only the language-agnostic characteristics from Table II. We then add language-specific characteristics to the classifiers.

Handling imbalanced categories. Table I shows that build co-changing work items are the minority category (4%-26%). Classifiers tend to favour the majority category, since it offers more explanatory power, i.e., classification of "no build change needed" will likely be more accurate than classification of "build change needed". To combat the bias of imbalanced categories, we re-balance the training corpus to improve minority category performance [11, 19]. Re-balancing is not applied to the testing corpus.

TABLE III
AN EXAMPLE CONFUSION MATRIX.

	Classified As	
Actual Category	Change	No Change
Change	a	b
No Change	c	d

We chose to re-balance the data using a re-sampling technique, which removes samples from the majority category (under-sampling) and repeats samples in the minority category (over-sampling). We chose to re-sample rather than apply other re-balancing techniques like re-weighing (i.e., assigning more weight to correctly classified minority items) because we found that re-sampling yielded slightly better results, which is consistent with findings reported in the literature [11, 20].

Re-sampling is performed with a given bias β towards equally distributed categories ($\beta = 1$). No re-sampling is performed when $\beta = 0$. Values between $0 < \beta < 1$ vary between unmodified categories and equally distributed categories. We report findings for different values of β.

IV. Mozilla Case Study Results (C++)

In this section, we present the results of our Mozilla case study with respect to our three research questions. For each research question, we present our approach for addressing it followed by the results that we observe.

(RQ1) How often are build changes accompanied by source/test code changes?

Approach. We measure the rate of build and source/test co-change as a percentage of all build changes. Specifically, we

TABLE IV
THE MEDIAN OF THE RECALL, PRECISION, F-MEASURE, AND AUC VALUES OF THE TEN CLASSIFIERS CONSTRUCTED AT RE-SAMPLING BIAS (β) LEVELS OF 0, OPTIMAL, AND 1. THE FIRST ROW SHOWS THE RAW VALUES WHILE THE SECOND ROW SHOWS THE IMPROVEMENT OF ADDING LANGUAGE-SPECIFIC CHARACTERISTICS TO LANGUAGE-AGNOSTIC CLASSIFIERS.

	Mozilla			**Eclipse-core**			**Lucene**			**Jazz**		
Bias (β)	0.0	0.40	1.0	0.0	0.59	1.0	0.0	0.73	1.0	0.0	0.32	1.0
Recall	0.57	0.63	0.67	0.30	0.39	0.43	0.14	0.31	0.39	0.24	0.31	0.40
	+0.05***	+0.00	-0.02	+0.01	+0.00	+0.01	+0.03	-0.05	+0.00	+0.04	+0.08**	+0.15***
Precision	0.74	0.63	0.53	0.50	0.39	0.34	0.38	0.31	0.33	0.36	0.31	0.24
	+0.06***	+0.10***	+0.15***	+0.09*	+0.07**	+0.09**	+0.09	+0.11**	+0.14***	-0.12	-0.06	-0.13
F-measure	0.64	0.63	0.60	0.37	0.39	0.38	0.20	0.31	0.36	0.29	0.31	0.30
	+0.05***	+0.06***	+0.10***	+0.02	+0.02	+0.05*	+0.04	+0.05*	+0.10**	+0.1	+0.05	+0.0
AUC	0.86	0.88	0.88	0.68	0.69	0.68	0.75	0.78	0.79	0.61	0.60	0.59
	+0.03***	+0.04***	+0.05***	+0.02	+0.03*	+0.04*	+0.09**	+0.07**	+0.10***	+0.05**	+0.03*	+0.04*

Statistical significance of the improvement achieved through language-specific characteristics (One-tailed Mann-Whitney U-test)
* $p < 0.05$; ** $p < 0.01$; *** $p < 0.001$

report the percentage of build-changing work items that also contain source or test code changes.

Results. **Most Mozilla build changes co-occur with source or test code**. While Table I shows that Mozilla build co-change is the minority category with respect to all source and test changes (27%), source/test co-change is the majority category with respect to all build changes. Indeed, 86% of Mozilla build-changing work items also change source code, and 29% also change test code. Altogether, 88% of Mozilla build changes co-occur with source/test code changes.

Co-occurrence alone does not indicate that there is a causal relationship between build changes and source/test changes. However, the inflated rates of co-occurrence that we observe suggest that there is likely information in these co-changes that we can leverage to better understand the types of source and test changes that require accompanying build changes.

While build co-changing work items are the minority category with respect to all source and test changes, source/test co-changing work items are the vast majority of all build changes in Mozilla. This suggests that source and test change characteristics may help to explain when build changes are necessary.

(RQ2) Can we accurately explain when build co-changes are necessary using code change characteristics?

Approach. Table IV shows performance values with $\beta = 0, 1, \theta$, where θ is the value where recall and precision values are equal. We refer to θ as the optimal β value, since we value precision (are build co-change classifications reliable?) and recall (are we finding all of the build co-changes?) equally.

Results. **Our Mozilla classifiers vastly outperform random classifiers**. The source- or test-build co-change work items row of Table I shows that a random classifier would achieve 0.26 precision at best. Table IV shows that our Mozilla classifiers more than double the precision of random classifiers, achieving a recall and precision of 0.63 ($\beta = \theta$). Moreover, since the AUC metric is designed such that a random classifier would achieve an AUC of 0.5, Table IV shows that our Mozilla classifier outperforms a random classifier by 0.38, achieving an AUC of 0.88.

Language-specific characteristics improve classifier performance. Table IV shows that when language-specific characteristics are added to our classifiers, the overall performance improves. Indeed, despite slight decreases in recall, the precision, F-measure, and AUC values improve. To test whether the observed improvement is statistically significant, we performed one-tailed Mann-Whitney U-tests ($\alpha = 0.05$). Test results indicate that the improvements in precision, F-measure, and AUC are statistically significant.

Using language-specific metrics, we can improve Mozilla classifier performance, achieving an AUC of 0.88.

(RQ3) What are the most influential code change characteristics for explaining build co-changes?

Approach. To study the most influential code change characteristics in our random forest classifiers, we compute Breiman's variable importance score [18] for each studied characteristic. The larger the score, the greater the importance of the code change characteristic.

Figure 3 shows the variable importance scores for the studied code change characteristic in each of the ten folds using boxplots. Since analysis of variable importance scores at $\beta = 0$ and $\beta = 1$ show similar trends, Figure 3 shows only the variable importance scores for the classifier trained with $\beta = \theta$ to reduce clutter.

Results. **Source and test changes that modify the structure of a system and prior build co-change are important explanatory factors of build changes in Mozilla**. Figure 3 shows that activities that alter the structure of a system like adding/deleting source code and adding/removing non-core libraries through `#include` statements are among the most important variables used by the Mozilla classifiers. Furthermore, prior build co-change is also an important indicator of future build co-changes. While renaming operations also modify the structure of a system, their low importance scores are likely due to the relative infrequency of rename operations in the Mozilla VCS history.

Our Mozilla classifiers derive much of their explanatory power from frequently occurring structural changes like adding source files, as well as historical co-change tendencies.

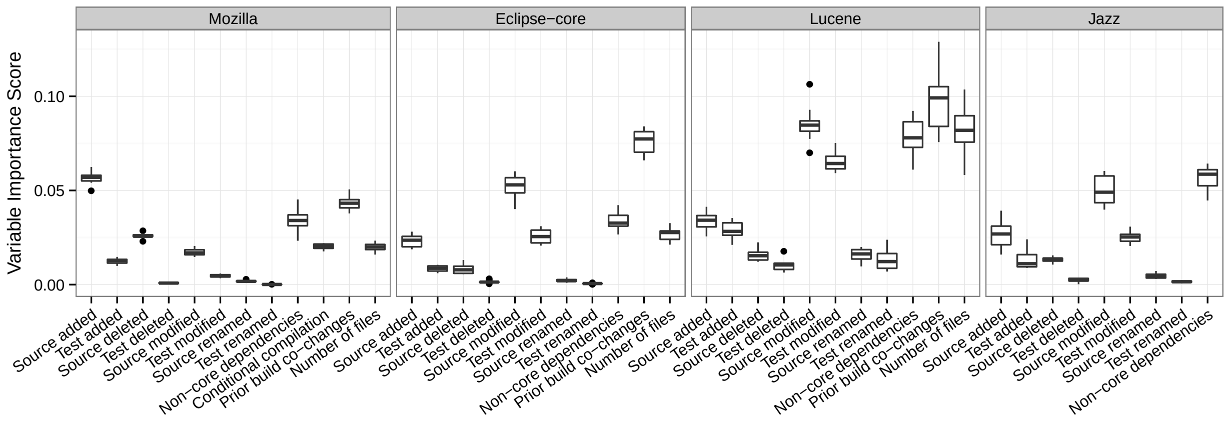


Fig. 3. Variable importance scores for the studied code change characteristics ($\beta = \theta$).

Discussion

While our classifiers perform well for Mozilla in general, we wondered whether the nature of the programming languages used in a subsystem (i.e., top-level directory) will have an impact on classifier performance. While Mozilla primarily consists of C++ code, it also contains subsystems implemented using several other programming languages (e.g., Javascript and PHP). To evaluate our conjecture, we construct and analyze directory-specific Mozilla classifiers.

Approach. In order to study classifier performance on a subsystem basis, we mark each work item with a listing of directories that are impacted by source and test changes within the work item. We then build classifiers for each directory separately. We ignore directories with fewer than 50 work items because we want our tenfold cross validation approach to test on at least five work items (10% of 50 work items).

Results. **As could be expected, Mozilla classifier performance is the weakest in subsystems primarily implemented using web technologies**. We find that most Mozilla subsystems have classifier performance that exceeds 0.7 AUC. However, the Mozilla `webtools` subsystem has subpar classifier performance when compared to the other subsystems (0.31-0.45 AUC). We observe similar weak classifier performance in the test subdirectories of the `js` subsystem. The code in these subsystems is written using web technologies, such as Javascript for testing the Mozilla Javascript engine, and PHP and Perl CGI for implementing tools like Bugzilla. Web technologies differ in terms of build tooling from the C++ code for which our classifiers perform well. While C++ code must be compiled and linked by the build system, the web code must only be tested, packaged, and deployed. We constructed special classifiers that detect the PHP `require` keyword as a dependency change, but it did not improve performance. The changes that induce build changes for web technologies are less code-related, and are thus more difficult to explain.

While coarse-grained file modifications and dependency information explain build changes in C++ subsystems reasonably well, they do not explain build changes in subsystems with web application code.

V. Java Case Study Results

Our findings in Section IV show that since Mozilla build changes are frequently accompanied by source and test changes (RQ1), we can derive information from the source and test changes to accurately explain when build changes are necessary (RQ2). This confirms common wisdom among C/C++ developers. However, we find that the programming languages used in a subsystem seem to influence the performance of our classifiers, i.e., it is harder to explain build changes in the subsystems that are implemented using web technologies than those implemented using C++. Furthermore, our prior work has shown that there are differences in the evolution of Java and C build systems, likely due to the built-in dependency management performed by the Java compiler [4].

To further investigate whether those environment changes have an impact on our co-change classifiers, we replicate our Mozilla case study on three Java systems. In this section, we present the results of our Java case study with respect to our three research questions. Since we use the same approaches as were discussed in Section IV, we only discuss the results that we observe with respect to each research question below.

(RQ1) How often are build changes accompanied by source/test code changes?

Similar to Mozilla, Java build systems frequently co-change with source or test code. Table I shows that between 44% (Lucene) and 82% (Eclipse-core) of Java work items that contain build changes also change source code. Furthermore, between 36% (Jazz, Eclipse-core) and 41% (Lucene) of work items that change the build also change test code. Altogether, between 53% (Lucene) and 81% (Eclipse-core) of Java build changes co-occur with source or test changes.

Similar to C++ build systems, most Java build changes are accompanied by source or test changes, suggesting that Java source and test change characteristics may also help to explain when Java build changes are necessary.

(RQ2) Can we accurately explain when build co-changes are necessary using code change characteristics?

Similar to Mozilla, our Java classifiers outperform random classifiers. Table IV shows that Eclipse-core, Lucene, and Jazz classifiers achieve recall and precision of 0.31-0.39 ($\beta = \theta$) and AUC values of 0.60-0.78. Our classifiers for Java systems outperform random classifiers that would only achieve between 0.04 (Jazz) and 0.16 (Eclipse-core) precision by a minimum factor of two.

We achieve the lowest performance in our Jazz classifiers. Unfortunately, the prior build co-changes and number of files characteristics could not be calculated for Jazz due to limitations of the provided dataset. We suspect that adding these metrics would bring the Jazz classifier performance up to match the performance of the other Java case studies.

Similar to Mozilla, language-specific characteristics improve classifier performance, especially in terms of precision and AUC. Table IV shows that the AUC of our Java classifiers improves by 0.03-0.07 when language-specific characteristics are added ($\beta = \theta$). Mann-Whitney U tests indicate that these AUC improvements are significant.

On the other hand, our Java classifiers under-perform with respect to our Mozilla classifier. The difference in performance is substantial – a reduction of roughly 33% in most of the performance metrics. We hypothesize that such a consistent difference in the performance of Mozilla and the Java classifiers is related to fundamental differences in the C++ and Java compile and link tools. For example, when using a C++ compiler, developers often rely on external build tools like `make` to manage dependencies amongst source files, while Java compilers automatically resolve these dependencies [21]. Since Java compilers are more intelligent in this regard, build changes are rarely required to track file-level dependencies.

To evaluate our hypothesis, we selected a representative sample of work items for manual analysis, since the full set of work items is too large to study entirely. To obtain proportion estimates that are within 10% bounds of the actual proportion with a 95% confidence level, we use a sample size of $s = \frac{z^2 p(1-p)}{0.1^2}$, where p is the proportion that 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 build co-changing work items in Eclipse-core (i.e., 382, see Table I) using $ss = \frac{s}{1+\frac{s-1}{382}}$ to obtain a sample size of 77. Table V shows the percentage of randomly selected work items that are associated with each change category.

The majority of Eclipse-core build changes are unrelated to the structure of the system. Table V shows that release engineering tasks (e.g., expanding platform support) and build maintenance tasks (e.g., compiler flag settings) account for $43\% \pm 10\%$ and $28\% \pm 10\%$ of build change respectively, a larger portion than structural changes ($25\% \pm 10\%$). Indeed, $75\% \pm 10\%$ of the studied build-changing work items were unrelated to the structure of the system (i.e., build maintenance, release engineering, and test maintenance).

For example, we studied a defect (ID 226462) where Eclipse was crashing when operating in a specific environment. The source code was fixed to prevent the crash, however the assigned developer discovered that a particular compiler warning could have notified the team of the issue prior to release. The work item fix included the build change to enable the compiler warning to prevent regression. Our classifiers fail to explain these sorts of build changes that do not directly link to source code changes, and in general, most source code changes in the Java systems do not require accompanying build changes (see Table I). Hence, the factors that drive Java build change are more elusive and difficult to isolate based on code change characteristics alone, which might provide an additional difficulty for developers to realize when they need to make a build system change.

Furthermore, a large proportion of source/build co-change requires expertise from different team roles. For example, the source code maintenance tasks require developer expertise, while release engineering and build maintenance tasks require release engineering expertise. This finding complements those of Wolf *et al.*, who find that team communication is a powerful predictor of build outcome [22].

Nonetheless, our Java classifiers can explain the build changes that are relevant to a developer. Indeed, Table V shows that 8 of the 19 work items that alter system structure were identified by our Eclipse-core classifier. In contrast, our classifiers only identified 5 of the 32 release engineering work items and no build or test maintenance work items. Since our classifiers are based on code change characteristics, they cannot assist release engineers, build maintainers, or quality assurance personnel. We plan to expand the scope of our classifiers to assist these practitioners in future work.

Our Java classifiers outperform random classifiers, achieving an AUC of 0.60-0.78. Yet, they under-perform with respect to the Mozilla classifier (0.88 AUC), since Java build co-changes are mostly related to release engineering activities rather than being purely code-based.

TABLE V
CATEGORIES OF IDENTIFIED ECLIPSE-CORE BUILD CHANGES WITH A 95% CONFIDENCE LEVEL AND A CONFIDENCE INTERVAL OF $\pm$ 10%.

Category	Task	Total #	%	# correctly classified
System structure	Refactorings	19	25%	8
Build maintenance	Build tool configuration	15	20%	0
	Build defects	6	8%	0
Release engineering	Add platform support	12	16%	2
	Packaging fixes	12	16%	3
	Library versioning	8	11%	0
Test maintenance	Test infrastructure	3	4%	0

(RQ3) What are the most influential code change characteristics for explaining build co-changes?

Source and test changes that alter system structure are not good indicators of build changes in studied Java systems. Figure 3 shows that source code modifications that do not alter the structure of a system (i.e., Source/Test modified) are more important indicators of build changes in the Java systems than those that do. This finding complements Table V, indicating that structural changes are not very important indicators of build change in Java systems. The relative infrequency of structural co-change for the Java build systems is likely due to the Java compiler's built-in support for dependency resolution. **Since structural co-changes are of little value for our Java classifiers, these classifiers need to derive co-change indications from other code change characteristics**. Figure 3 shows that adding or removing non-core dependencies (specified by Java `import` statements) helps to fill the void left by the missing structural cues. Although omitted from Figure 3 due to space constraints, the less detailed versions of the dependency characteristic (see Table II) have lower variable importance scores, suggesting that narrowing the scope of the dependency characteristic to only detect non-core API changes improves its performance in our classifiers. Furthermore, the prior build co-changes characteristic is the most important indicator of build co-change in our Eclipse-core and Lucene classifiers. Prior build co-changes also plays an important role in our Mozilla classifiers, indicating that historical co-change tendencies are consistent indicators of future build co-changes.

Since Java build changes rarely coincide with changes to the structure of a system, Java build changes are more effectively explained by historical co-change tendencies and changes to non-core Java API import statements.

VI. Threats to Validity

We now discuss threats to the validity of our case studies.

Construct validity. We make an implicit assumption that the collected data is correct, i.e., in the data used to build our classifiers, developers always commit related source, test, and build changes under the same work item when necessary. On the other hand, our work item data is robust enough to handle cases where developers did forget to change the build in the same transaction as a corresponding code change.

Our bias analysis in Section III shows that work item aggregation skews the developer contributions in Mozilla. To combat this bias, we remove the skewed early development period from the dataset prior to performing our case studies.

Internal validity. We use code change characteristics to explain build changes because most of the build changes coincide with code changes. We selected metrics that cover a wide range of change characteristics that we felt would induce build changes. However, other metrics that we have overlooked may also improve the performance of our classifiers.

Our file classification approach is subject to the authors' opinion and may not be 100% accurate. The authors used their best judgement to classify files that could not be automatically classified using filename conventions. The authors rely on their prior experience with build systems to classify files that may have fit several categories [3, 4, 7, 23]. We have also used this classification approach in our prior work [2] and made the classified files available online[4] to aid in future research.

[4]http://sailhome.cs.queensu.ca/replication/understanding_build_changes/

External validity. Despite the difficulty of collecting linked work item data, we study four software systems. However, our sample size may limit the generalizability of our results. To combat this limitation, we study systems of different sizes and domains. Moreover, we augment our study of three open source systems with the proprietary IBM Jazz system.

We suspect that the differences in the C++ and Java build change classifiers are due to differences in the dependency support of C++ and Java build tools. However, there are likely several confounding factors that we could not control for in such a small sample. For example, we observed variability in the performance of the three studied Java systems. Thus, the differences that we observe among C++ and Java systems may simply be due to natural variability among the systems rather than indicative of differences between the C++ and Java build systems. While deeper manual analysis seems to support the latter case, further replication of our results in other (particularly C++) systems could prove fruitful.

VII. Related Work

In this section, we discuss the related work with respect to build system maintenance, tool support for build maintenance, and change prediction.

Co-evolution of source code and build system. Prior work shows that build systems generate substantial project maintenance overhead. Kumfert *et al.* [6] and Hochstein *et al.* [5] find that there is a "hidden overhead" associated with maintaining build systems. In our prior work, we show that build systems tend to co-evolve with source code from release to release [3, 4]. We have also shown that the build system evolution imposes a non-trivial overhead on software development [2], e.g., up to 27% of source code changes require accompanying build changes. Indeed, Neitsch *et al.* find that abstractions can leak between source and build domains, likely due to the co-dependent nature of source and build changes [24]. These findings motivate our use of source and test code change characteristics for training classifiers that explain when build changes are necessary.

Tool support for build maintenance. Recent research has proposed several tools to assist developers in maintaining the build system. Adams *et al.* develop the MAKAO tool to visualize and reason about build dependencies [23]. Tamrawi *et al.* propose a technique for visualizing and verifying build dependencies using symbolic dependency graphs [25]. Al-Kofahi *et al.* extract the semantics of build specification changes using MkDiff [26]. Nadi *et al.* develop a technique for reporting anomalies between source and build system layers as likely defects in Linux [27, 28]. By contrast, our work explores indicators that may be used to create a tool to help developers avoid neglecting build changes when they are required.

Change prediction. Researchers have used machine learning classifiers to explain or predict software engineering phenomena. For example, Hassan and Zhang use classifiers to predict whether a build would pass a certification process [10]. Ibrahim *et al.* use classifiers to predict whether a developer should contribute to an email discussion [11]. Shihab *et al.* use classifiers to predict whether a bug will be reopened [14]. Knab *et al.* use classifiers to predict the defect density of source code files [12], while Ratzinger *et al.* use decision trees and other learned classifiers to predict when refactoring is required [13]. We construct classifiers to explain when build changes are necessary using source and test code change characteristics.

VIII. Conclusions

Build systems age in tandem with the software systems that they are tasked with building. Changes in source and test code often require accompanying changes in the build system. Developers may not be aware of changes that require build maintenance, since build systems are large and complex. Since realistic builds take a long time to run, neglecting such build changes can cause build breakages that slow development progress, or worse can cause the build system to produce incorrect deliverables, impacting end users. Hence, in this paper, we set out to answer this central question:

Can build changes be fully explained using characteristics of co-changed source and test code files?

Through a case study of four large software systems, we found that the answer is no:

- While 4%-26% of work items that change source/test code also change the build system, 53%-88% of build-changing work items also contain source/test changes, suggesting that there is a strong co-change relationship between the build system and source/test code.
- Our Mozilla build co-change classifiers achieve an AUC of 0.88, with these co-changes being most effectively indicated by structural changes to a system and historical build co-change tendencies of the modified files.
- However, classifier performance suffers in systems composed of Java and web application code due to a shift in the usage and design of build technology from requiring build changes for structural code changes (e.g., adding a file) to enabling cross-disciplinary activities related to release engineering and general build maintenance.

Future work. Our findings suggest that most C++ build changes and at least the code-related Java build changes can indeed be predicted using characteristics of corresponding changes to source and test code. To assist release engineers, build maintainers, and quality assurance personnel in interacting with Java-based build systems, we plan to explore metrics related to build structure and platform configuration.

Acknowledgments

This research was partially supported by the Natural Sciences and Engineering Research Council of Canada (NSERC).

References

[1] C. AtLee, L. Blakk, J. O'Duinn, and A. Z. Gasparnian, "Firefox Release Engineering," in *The Architecture of Open Source Applications*, A. Brown and G. Wilson, Eds., 2012, vol. 2, pp. 23–38.

[2] 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)*, 2011, pp. 141–150.

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

[4] S. McIntosh, B. Adams, and A. E. Hassan, "The evolution of Java build systems," *Empirical Software Engineering*, vol. 17, no. 4-5, pp. 578–608, August 2012.

[5] L. Hochstein and Y. Jiao, "The cost of the build tax in scientific software," in *Proc. of the 5th International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2011, pp. 384–387.

[6] 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, 2002.

[7] S. McIntosh, M. Poehlmann, E. Juergens, A. Mockus, B. Adams, A. E. Hassan, B. Haupt, and C. Wagner, "Collecting and Leveraging a Benchmark of Build System Clones to Aid in Quality Assessments," in *Proc. of the 36th Int'l Conf. on Software Engineering (ICSE)*, 2014, pp. 115–124.

[8] H. Seo, C. Sadowski, S. Elbaum, E. Aftandilian, and R. Bowdidge, "Programmers' Build Errors: A Case Study (at Google)," in *Proc. of the 36th Int'l Conf. on Software Engineering (ICSE)*, 2014, pp. 724–734.

[9] I. Kwan, A. Schröter, and D. Damian, "Does Socio-Technical Congruence Have An Effect on Software Build Success? A Study of Coordination in a Software Project?" *Transactions on Software Engineering (TSE)*, vol. 37, no. 3, pp. 307–324, May/June 2011.

[10] A. E. Hassan and K. Zhang, "Using Decision Trees to Predict the Certification Result of a Build," in *Proc. of the 21st Int'l Conf. on Automated Software Engineering (ASE)*, 2006, pp. 189–198.

[11] W. Ibrahim, N. Bettenburg, E. Shihab, B. Adams, and A. E. Hassan, "Should I contribute to this discussion?" in *Proc. of the 7th working conf. on Mining Software Repositories (MSR)*, 2010.

[12] P. Knab, M. Pinzger, and A. Bernstein, "Predicting Defect Densities in Source Code Files with Decision Tree Learners," in *Proc. of the 3rd Int'l Workshop on Mining Software Repositories (MSR)*, 2006, pp. 119–125.

[13] J. Ratzinger, T. Sigmund, P. Vorburger, and H. Gall, "Mining Software Evolution to Predict Refactoring," in *Proc. of the 1st Int'l Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2007, pp. 354–363.

[14] E. Shihab, A. Ihara, Y. Kamei, W. M. Ibrahim, M. Ohira, B. Adams, A. E. Hassan, and K. ichi Matsumoto, "Predicting Re-opened Bugs: A Case Study on the Eclipse Project," in *Proc. of the 17th Working Conf. on Reverse Engineering (WCRE)*, 2010, pp. 249–258.

[15] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu, "Fair and Balanced? Bias in Bug-Fix Datasets," in *Proc. of the 7th joint meeting of the European Software Engineering Conf. and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2009, pp. 121–130.

[16] T. H. D. Nguyen, B. Adams, and A. E. Hassan, "A Case Study of Bias in Bug-Fix Datasets," in *Proc. of the 17th Working Conf. on Reverse Engineering (WCRE)*, 2010, pp. 259–268.

[17] P. Kampstra, "Beanplot: A boxplot alternative for visual comparison of distributions," *Journal of Statistical Software, Code Snippets*, vol. 28, no. 1, pp. 1–9, 2008. [Online]. Available: http://www.jstatsoft.org/v28/c01/

[18] L. Breiman, "Random Forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001.

[19] R. Barandela, J. Sánchez, V. García, and F. Ferri, "Learning from Imbalanced sets through resampling and weighting," in *Proc. of the 1st Iberian Conf. on Pattern Recognition and Image Analysis (IbPRIA)*, 2003.

[20] A. Estabrooks and N. Japkowicz, "A mixture-of-experts framework for learning from imbalanced data sets," in *Proc. of the 4th Int. Conf. on Advances in Intelligent Data Analysis (IDA)*, 2001, pp. 34–43.

[21] 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)*. ACM, 2002.

[22] T. Wolf, A. Schröter, D. Damian, and T. Nguyen, "Predicting build failures using social network analysis on developer communication," in *Proc. of the 31st Int'l Conf. on Software Engineering (ICSE)*, Washington, DC, USA, 2009, pp. 1–11.

[23] 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, pp. 114–123.

[24] 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*, 2012, pp. 140–149.

[25] A. Tamrawi, H. A. Nguyen, H. V. Nguyen, and T. Nguyen, "Build Code Analysis with Symbolic Evaluation," in *Proc. of the 34th Int'l Conf. on Software Engineering (ICSE)*, 2012, pp. 650–660.

[26] J. M. Al-Kofahi, H. V. Nguyen, A. T. Nguyen, T. T. Nguyen, and T. N. Nguyen, "Detecting Semantic Changes in Makefile Build Code," in *Proc. of the 28th Int'l Conf. on Software Maintenance (ICSM)*, 2012, pp. 150–159.

[27] S. Nadi and R. Holt, "Make it or Break it: Mining Anomalies in Linux Kbuild," in *Proc. of the 18th Working Conf. on Reverse Engineering (WCRE)*, 2011, pp. 315–324.

[28] ——, "Mining Kbuild to Detect Variability Anomalies in Linux," in *Proc. of the 16th European Conf. on Software Maintenance and Reengineering (CSMR)*, 2012, pp. 107–116.