Extracting Build Changes with BUILDDIFF

Christian Macho
Software Engineering Research Group
University of Klagenfurt
Klagenfurt, Austria
Email: christian.macho@aau.at

Shane McIntosh
Dept. of Electrical and Computer Eng.
McGill University
Montréal, Canada
Email: shane.mcintosh@mcgill.ca

Martin Pinzger
Software Engineering Research Group
University of Klagenfurt
Klagenfurt, Austria
Email: martin.pinzger@aau.at

Abstract—**Build systems are an essential part of modern software engineering projects. As software projects change continuously, it is crucial to understand how the build system changes because neglecting its maintenance can lead to expensive build breakage. Recent studies have investigated the (co-)evolution of build configurations and reasons for build breakage, but they did this only on a coarse grained level.**

In this paper, we present BUILDDIFF, an approach to extract detailed build changes from MAVEN build files and classify them into 95 change types. In a manual evaluation of 400 build changing commits, we show that BUILDDIFF can extract and classify build changes with an average precision and recall of 0.96 and 0.98, respectively. We then present two studies using the build changes extracted from 30 open source Java projects to study the frequency and time of build changes. The results show that the top 10 most frequent change types account for 73% of the build changes. Among them, changes to version numbers and changes to dependencies of the projects occur most frequently. Furthermore, our results show that build changes occur frequently around releases.

With these results, we provide the basis for further research, such as for analyzing the (co-)evolution of build files with other artifacts or improving effort estimation approaches. Furthermore, our detailed change information enables improvements of refactoring approaches for build configurations and improvements of models to identify error-prone build files.

Keywords-**Maintenance, Build Systems, Software Quality**

I. INTRODUCTION

Large software projects use build tools, such as MAVEN, GRADLE, or ANT, to automate the assembling and testing process of their software products. The configuration of such build systems can often be complex [1], which also complicates their maintenance. Seo *et al.* [2] showed that up to 37% of builds at Google fail, stating neglected build maintenance as the most frequent cause. The development team is then blocked and obliged to fix the build first. Kerzazi *et al.* [3] found a similar ratio of up to 18% of build breakage and estimated the total costs for the breakages in their study to be more than 336 man-hours.

As a software system evolves, changes are applied to the source code. Developer teams also need to maintain the build and hence, subsequent changes need to be applied to the build configuration. Adams *et al.* [4] and McIntosh *et al.* [1] found evidence of a co-evolutionary relationship between source and build code. Hence, omitting changes to the build configuration that are needed to remain synchronized with the source code, can lead to build breakage. To that end, it is important to know when build changes should be applied. McIntosh *et al.* [5] and Macho *et al.* [6] studied this problem and found that their models can predict whether a source code change should have an accompanying build change. However, both studies lack detailed information about the type of build change that is needed.

In this study, we investigate changes to the build configuration in detail. We are interested in which types of changes are typically made to the build configuration and when they are performed. Prior studies [5] consider the build configuration to be changed if the build file changes but do not investigate the detailed type of the change. We claim that a more detailed view on build configuration changes can improve studies of the build system and its configuration. Thus, we introduce BUILDDIFF, an approach to extract build changes from MAVEN build files. Our approach is inspired by ChangeDistiller [7], [8], which extracts source code changes from Java source files. To the best of our knowledge, we are the first to investigate build changes in this granularity. We also propose a taxonomy of build changes consisting of 95 build change types and 5 categories that our approach can extract. We evaluate BUILDDIFF in a manual investigation of 400 randomly selected build changing commits and find that it yields an average precision and recall of 0.96 and 0.98, respectively.

Armed with an approach to extract build changes from MAVEN build files, we extract build changes from 30 open source Java projects from different vendors, of different sizes, and with different purposes. We study the extracted data in two ways. First, we study the frequency of build changes. We explore which change types are the most frequent ones and which change types are rarely applied. We analyze the frequencies also in terms of change categories. Second, we study the time at which build changes have been recorded and investigate whether build changes are equally distributed over a project or if there are periods in the projects where they occur more frequently. With the two studies, we address the following two research questions:

(RQ1) Which build change types occur the most frequently?

The most frequent build change type is PARENT_VERSION_UPDATE followed by PROJECT_VERSION_UPDATE, and DEPENDENCY_INSERT. The most frequent build change category is `General`

`Changes`, directly followed by `Dependency Changes`, and `Build Changes`. Changes to version properties and to dependencies can be found frequently among the top 10 most frequent change types. The top 10 change types account for 73% of all of the changes.

(RQ2) When are build changes recorded?
Build changes are not equally distributed over the projects' timeline. There are particular phases that contain significantly more build changes than others. We observe that especially around releases, the frequency of build changes is high.

This work makes the following contributions: (1) an approach and a corresponding prototype implementation to extract fine-grained build changes from MAVEN build files, (2) a dataset containing historical build changes of 30 open source Java projects, (3) an evaluation of the performance of this tool, (4) two empirical studies of the frequency and time of build changes, and (5) a replication package that contains supplementary material.[1]

The remainder of the paper is organized as follows: Section II situates the paper with respect to the related work. Section III presents our BUILDDIFF approach. Section IV describes the data that we used for this study and evaluates the performance of BUILDDIFF, and discusses its strengths and weaknesses. Section V presents the first study on the frequency of build changes and Section VI shows the second study on when build changes occur. Section VII discusses the implications of our results and threats to validity. Section VIII concludes the paper.

II. RELATED WORK

Build Maintenance. Related work on build maintenance includes the co-evolution of build systems with other artifacts of the development process. For instance, Adams *et al.* introduced MAKAO [9], a framework for re(verse)-engineering build systems, and studied the co-evolution of the Linux build system [4] using MAKAO. They found that the build system itself evolves and its complexity grows over time. Furthermore, they identified maintenance as the main factor for evolution. McIntosh *et al.* investigated the evolution of the ANT build system [10] from a static and a dynamic perspective. They defined a metric for measuring the complexity of build systems and found that the complexity of ANT build files evolves over time, too. In follow-up work, McIntosh *et al.* investigated Java build systems and their co-evolution with production and test code [1]. The results of a large-scale study showed a relationship between build technology and maintenance effort [11]. In addition to these studies, Hardt and Munson developed Formiga, a tool to refactor ANT build scripts [12] [13].

Concerning the co-evolution of build configurations with other software artifacts, existing studies investigated models to predict build co-changes based on various metrics. For instance, McIntosh *et al.* [5] used code change characteristics

to predict build co-changes within a software project. Xia *et al.* [14] extended this study by building a model for predicting build co-changes across software projects. Macho *et al.* [6] showed that they can improve both studies by using fine-grained source code changes. Furthermore, Xia *et al.* [15] investigated missing dependencies in build files using link prediction. They showed that their algorithm outperforms state-of-the-art link prediction algorithms for this problem.

Change Extraction. Many previous studies used changes that were extracted from different versions of source files to investigate various aspects of the evolution of software systems. Miller *et al.* [16] and Myers *et al.* [17] performed their studies on the level of text simply by counting the number of added or deleted lines of text. One advantage of these approaches is that they do not need a parser or a grammar to output the differences between the different versions of source files. However, one important shortcoming is that these approaches have difficulty mapping the changed lines of text to actual changes in source files, such as the change of the return type of a method or the addition of an else branch.

Modern approaches overcome this issue by performing the differencing on the level of Abstract Syntax Trees (ASTs). For instance, Hashimoto and Mori [18] developed Diff/TS, which is working on the raw AST created from parsing two versions of a source file. The most prominent approach in this area is ChangeDistiller of Fluri *et al.* [7]. Their approach extracts differences from two consecutive versions of a Java file and maps the differences to 48 change types [19]. Falleri *et al.* [20] improved the differencing algorithm by applying a combined method for matching equal subtrees and showed that their approach outperforms ChangeDistiller. Concerning build changes, Désarmeaux *et al.* [21] mapped line-level changes to MAVEN lifecycle phases and investigated the maintenance effort of each phase. They found out that the compile phase accounts for most of the maintenance.

In summary, we find that several approaches exist to study build maintenance, build systems, and their configuration. However, these studies are primarily based on coarse-grained metrics. Previous work showed that for programming languages, such as Java, the usage of a finer granularity of changes can help to improve prediction models [22], [23], [24], [25] or support the understanding of (co-)evolution [26]. To that extent, it is important to also investigate build changes on a fine-grained level. To the best of our knowledge, we are the first to present an approach to extract detailed changes from MAVEN build files.

III. EXTRACTING BUILD CHANGES WITH BUILDDIFF

In this section, we describe our approach to extract build changes from build files. Currently, we focus on the extraction of MAVEN build files. MAVEN build files are named `pom.xml` following the naming convention of MAVEN. First, we define a taxonomy of build changes and provide our rationale for the defined changes. Second, we describe BUILDDIFF, our approach to extract build changes of two consecutive MAVEN build file revisions.

---

[1] https://figshare.com/articles/BuildDiff_Supplementary_Material/4786084

A. Taxonomy

MAVEN build files are specified using a special type of XML. Hence, we can easily read, parse, and transform their content into a tree that corresponds to the MAVEN schema[2] that defines the various XML elements and attributes used for configuring a MAVEN build. Having the content of a MAVEN build file represented as a tree, we then can use tree differencing algorithms, such as ChangeDistiller [7] or GumTree [20], to extract differences between two build files. We use the modified version of the GumTree implementation of Dotzler *et al.* [27] to extract edit operations that transform one tree into the other. In the remainder of the paper, we refer to this implementation as GumTree. We describe the extraction procedure in more detail in Section III-B.

Similar to ChangeDistiller, we defined the change types of our taxonomy based on the edit operations extracted by the tree differencing algorithm whereas the structure of the tree and its different elements correspond to the MAVEN schema. For defining the taxonomy, we started with the top level elements of the MAVEN schema and moved down the schema until we reached the bottom-most child elements. For each element (*i.e.*, XML tag), we defined change types for inserting (*_INSERT), deleting (*_DELETE), and updating (*_UPDATE) that element. For some particular tags, such as `artifactId` and `groupId`, we only created the *_UPDATE change type because they are mandatory for the definition of particular tags, such as `dependency`, and we assume that they are inserted and deleted with their parent tag. This is further described in detail in Section III-B. The resulting taxonomy currently comprises 95 that we validated with two expert developers using MAVEN.

We also grouped the change types into categories. We retrieved the categories and the respective change type assignments by performing card sorting [28]. First, we gave the list of change types to the two developers who validated the changes types separately and asked them to group the change types. Second, we asked the developers to assign names to the created groups. In a third step, we asked both developers to discuss their categories and assignments to arrive at a common categorization. If the developers assigned a change to different categories they discussed with one another to arrive at a consensus.

The two developers developed the following 5 categories: (1) `Dependency Changes` contain all changes that are related to dependencies of the project, (2) `Build Changes` cover the changes that directly affect or modify the build process, (3) `Team Changes` comprise all modifications to the list of team members, (4) `Repository Changes` hold changes that are performed to the distribution and repository locations, and (5) `General Changes` contain changes that are made to the general items of a MAVEN project. Table I shows an excerpt of the taxonomy with examples of change types for each category. The full taxonomy can be found online in the supplementary material.[1]

---

[2]http://maven.apache.org/xsd/maven-4.0.0.xsd

TABLE I
EXCERPT OF OUR TAXONOMY OF BUILD CHANGES

Category	Change Types (Excerpt)
Dependency Changes	DEPENDENCY_INSERT DEPENDENCY_VERSION_UPDATE MANAGED_DEPENDENCY_DELETE
Build Changes	PLUGIN_INSERT PLUGIN_CONFIGURATION_UPDATE TEST_RESOURCE_DELETE
Team Changes	DEVELOPER_INSERT CONTRIBUTOR_DELETE
Repository Changes	PLUGIN_REPOSITORY_INSERT DIST_SNAPSHOT_REPOSITORY_UPDATE REPOSITORY_DELETE
General Changes	MODULE_INSERT PARENT_VERSION_UPDATE GENERAL_PROPERTY_DELETE

In the following, we provide two examples of frequently occurring build changes. The first example depicted in Listing 1 (old version of the build file) and Listing 2 (new version of the build file) shows a change of the version of a dependency to the `spring-core` library from `4.2.5.RELEASE` to `4.2.6.RELEASE`. This change is extracted and classified by BUILDDIFF as DEPENDENCY_VERSION_UPDATE.

Listing 1
DEPENDENCY VERSION UPDATE - OLD VERSION

```
<dependency>
 <groupId>org.springframework</groupId>
 <artifactId>spring-core</artifactId>
 <version>4.2.5.RELEASE</version>
</dependency>
```

Listing 2
DEPENDENCY VERSION UPDATE - NEW VERSION

```
<dependency>
 <groupId>org.springframework</groupId>
 <artifactId>spring-core</artifactId>
 <version>4.2.6.RELEASE</version>
</dependency>
```

The second example depicted in Listing 3 (old version of the build file) and Listing 4 (new version of the build file) shows the insertion of the `maven-jar-plugin` plugin. This change is extracted and classified by BUILDDIFF as PLUGIN_INSERT.

Listing 3
PLUGIN INSERTION - OLD VERSION

```
<build>
 <plugins>
 </plugins>
</build>
```

```xml
<build>
 <plugins>
  <plugin>
   <groupId>org.apache.maven.plugins</groupId>
   <artifactId>maven-jar-plugin</artifactId>
   <version>2.6</version>
  </plugin>
 </plugins>
</build>
```

B. Approach

This section presents our BUILDDIFF approach to extract 95 types of changes from MAVEN build files. Our approach is mainly motivated and inspired by the work of Gall *et al.* [8] and Fluri *et al.* [7], who showed that detailed information on source code changes can aid in understanding the evolution of software projects, and the work of Macho *et al.* [6], who showed that this information can be used for computing models to predict when build configurations should be updated.

Concerning changes in build configuration files, in particular MAVEN build files, the finest level of analysis that has been performed was on line level. Désarmeaux *et al.* [21] mapped lines of a MAVEN pom.xml to the respective build lifecycle phase. To the best of our knowledge, we are the first to present an approach to extract changes in MAVEN build files on the level of MAVEN configuration elements, that we refer to as fine-grained build changes.

BUILDDIFF first reads two versions of a MAVEN build file (*i.e.*, pom.xml) and represents each version as a tree. Then, it uses the GumTree [20] implementation of Dotzler *et al.* [27] to extract the differences between the two trees in terms of edit operations to transform one tree into the other. The list of edit operations is then mapped to the 95 change types that are defined in our taxonomy. In the following, we present each step in detail:

Preprocess Build Files. The first step of BUILDDIFF preprocesses the two versions of a MAVEN build file. MAVEN build files are descriptive, meaning that the order of the elements in the file can be changed without changing its semantics. We observed that GumTree can match elements of the same level more accurately if they are sorted. Hence, BUILDDIFF first sorts the elements on the same level according to their content.[3] For example, the tag <module>MySubmodule</module> appears before <module>TheModule</module>. Furthermore, BUILD-DIFF removes comments and attributes. Attributes, such as combine.children and combine.self for plugin configuration inheritance, affect the build configuration at execution time. We only analyze the build configuration from a static point of view and hence, we remove attributes.

Extract Edit Operations. Next, BUILDDIFF parses the two preprocessed versions of a MAVEN build file into two trees and passes them to the GumTree differencing algorithm. GumTree

---

[3]Strings are sorted alphabetically and numbers in ascending order

provides a TreeGenerator for XML files. Unfortunately, this implementation does not handle values of tags in XML documents. Therefore, we implemented our own TreeGenerator that transforms XML files into GumTree trees. We use the prominent Java XML library *jdom* to read the XML file, and methods provided by GumTree to create the tree.

GumTree then uses a Matcher instance to find mappings between two trees. BUILDDIFF extends the GumTree's default matcher by adding a mechanism to ensure that only tags with the same name will be matched, and by modifying the similarity calculation of two nodes. Tags that have a child tag named id are matched if the Levenshtein similarity of the id value exceeds a threshold t. The matcher chooses the node with the highest similarity exceeding the threshold. Tags that have the MAVEN triplet (groupId, artifactId, version) as child nodes are matched by applying the Levenshtein distance for groupId and artifactId. Two nodes are matched if the Levenshtein similarity exceeds a threshold t. The matcher chooses the node with the highest similarity exceeding the threshold. Experiments with different t values suggest that $t = 0.65$ yields the best performance.

Given the matcher, GumTree outputs a list of tree edit operations comprising added, deleted, updated, and moved elements in the tree that transform the source tree (previous version of the MAVEN build file) into the target tree (subsequent version of the MAVEN build file).

Sort Edit Operations. BUILDDIFF considers a particular order to process the changes in MAVEN files. We process the operations of the edit script in a top down order according to their level in the build file (parent nodes first). BUILDDIFF applies this order to prevent the extraction of additional changes that result from the insertion and deletion of composite MAVEN tags that also insert or delete their children at the same time. For instance, when a new dependency is inserted, BUILDDIFF only records a DEPENDENCY_INSERT, skipping the insertion of the child tags of that dependency (*e.g.*, groupId, artifactId, and version).

Map Build Changes. In this step, BUILDDIFF maps the tree edit operations that are generated by GumTree to the 95 change types of our taxonomy. We consider insertions, deletions, and updates. We do not consider moves, since MAVEN build files are descriptive, meaning that the order of the elements in the file can be changed without changing its semantics.

To map edit operations to change types, BUILDDIFF iterates over the sorted edit operations mapping each edit operation to at most one change type. Changes to child elements are handled by first checking whether the change is part of an insertion, deletion, or update of its parent. In that case, the change to the child element is not mapped, since it is already part of the parent change. For instance, the insertion of a dependency is mapped only to the change DEPENDENCY_INSERT while the insertions of its child elements groupId, artifactId, and version are skipped.

As a result, BUILDDIFF outputs a list of build changes that have been performed between two versions of a MAVEN file.

IV. EVALUATING BUILDDIFF

In this section, we describe the evaluation of BUILDDIFF. First, we describe how we selected the projects and how we extracted the data that we used for the evaluation of BUILDDIFF and the experiments. Second, we present the evaluation of our prototype implementation of the BUILDDIFF approach in terms of precision and recall. Finally, we discuss examples of correctly and incorrectly extracted changes.

A. Data Preparation

The projects that we selected for our experiments stem from two origins. First, we selected the list of projects that Macho *et al.* used in their prior study [6] because they cover a wide range of different vendors, project sizes, and purposes. Second, we extended this list with popular projects from GitHub to improve the variety of the selected projects. We retrieved a list of Java projects ordered by their star rating[4] and removed projects that do not use MAVEN as build system and projects with less than 3500 commits in the repository or rated with less than 1000 stars. We calculated a ranking metric by adding the number of commits and the star rating to evenly balance the user rating and the number of commits of a project. Then, the list of projects was sorted according to the ranking value in a descending order. From this list, we selected the top 20 projects and added them to our list of projects. Table II shows the full list of selected projects and several descriptive statistics of commits and build changes.

For each project, we extracted the build changes as follows: First, we cloned the repository and iterated over each commit, including commits on branches. Second, we checked for modifications in MAVEN build files (pom.xml) indicated by Git. For each of the modified build files, we determined its preceding version and provided both versions to BUILDDIFF to extract MAVEN build changes. The extracted changes were stored in a database called the ChangeDB. Due to its size, we provide the database for other researchers on request. If a commit was a merging commit (*e.g.*, pull requests that were merged into the master branch), we did not extract the changes of this commit because we already extracted these changes in the respective commits of the branch.

B. Evaluation

We evaluated BUILDDIFF in two ways: first, we used extensive JUnit testing to check whether our approach can detect each single build change type. Second, we performed a manual evaluation with 400 build-changing commits that were randomly selected from the build files of the 30 open source Java projects.

1) JUnit Tests: For each build change type, we first constructed a pom.xml pair that contains exactly one instance of a particular change type and the corresponding JUnit test case. In addition, we selected subsequent versions of actual pom.xml files from open source Java projects that contain multiple changes and created JUnit tests for those as well.

These pairs have been selected by an expert with more than 7 years of MAVEN experience to cover changes that are often performed in practice. In total, we developed 250 JUnit tests showing that BUILDDIFF is capable of correctly extracting and classifying each single build change in isolation, as well as co-occurring build changes.

2) Manual Investigation: To show that BUILDDIFF is also working on real world projects and that it can extract build changes as they are understood by software developers, we invited two PhD students to evaluate 400 build-changing commits. Both students are studying software engineering and have more than 4 years of experience with MAVEN. They received the same set of 400 pom.xml pairs containing 745 build changes. Prior to the experiment, we briefly explained our taxonomy of build changes to them. In the experiment, we asked each student to label the changes in the pom.xml pairs according to our change taxonomy and compared the output among the participants and with the output of BUILDDIFF.

Data Selection. Table II shows the list of open source Java projects from which we randomly selected 400 commits that contained changes to a pom.xml build file. We calculated the sample size based on a population size of 66,984 commits that contain build changes (Table II, sum of column #BCC), with a margin of error of 5% and a confidence level of 95%. The minimum sample size is 382 commits[5] and we finally decided to randomly select 400 commits to exceed the minimum sample size.

The total amount of build changes in the data set is 641,056. By randomly selecting the commits for the evaluation, we missed to evaluate 37 of the 95 change types (or 44%). However, the analysis of these missing change types in our subject systems showed that these changes only represent 0.9% of the total build changes (5,760 out of 641,056). Hence, we can safely assume that our sample set sufficiently covers the majority of build changes in our data set.

Evaluation. For each of the 400 commits, we provided the study participants with the original and modified version of the build file. We asked each participant to go through all of the selected commits and assign each change to the corresponding change type from our taxonomy. We then compared the changes that were assigned by the participants with the list of build changes extracted by BUILDDIFF. With the results from each participant, we calculated *precision* and *recall* to measure the performance of our approach. Following the approach of Fluri *et al.* [7] used for evaluating ChangeDistiller, we calculated precision and recall as:

$$\text{precision} = \frac{\#\text{relevant changes found}}{\#\text{changes found}}$$

$$\text{recall} = \frac{\#\text{relevant changes found}}{\#\text{changes expected}}$$

Precision measures how many of the changes that were extracted by our approach were also detected by a study participant. *Recall* measures how many of the changes that

---

Vendor	Name	Rank	Stars	#Commits	#BCC	BCCR	#NBCC	NBCR	BC	#R
neo4j	neo4j	49,663	3,344	46,319	9,684	0.21	36,635	0.79	78,551	170
hazelcast	hazelcast	24,995	1,850	23,145	1,306	0.06	21,839	0.94	7,914	126
SonarSource	sonarqube	23,989	1,431	22,558	1,746	0.08	20,812	0.92	18,332	109
Alluxio	alluxio	23,545	2,787	20,758	1,694	0.08	19,064	0.92	12,973	25
languagetool-org	languagetool	21,949	1,021	20,928	285	0.01	20,643	0.99	5,027	17
netty	netty	21,443	8,637	12,806	1,395	0.11	11,411	0.89	7,791	160
orientechnologies	orientdb	20,458	2,769	17,689	1,326	0.07	16,363	0.93	8,866	94
spring-projects	spring-boot	20,356	9,312	11,044	2,922	0.26	8,122	0.74	29,699	68
h2oai	h2o-2	19,071	2,116	16,955	74	0.00	16,881	1.00	194	324
google	guava	18,643	13,676	4,967	221	0.04	4,746	0.96	850	64
deeplearning4j	deeplearning4j	16,891	5,044	11,847	1,097	0.09	10,750	0.91	7,454	42
stanfordnlp	CoreNLP	16,688	2,685	14,003	847	0.06	13,156	0.94	3,365	0
eclipse	jetty	16,357	1,148	15,209	2,965	0.19	12,244	0.81	82,153	272
Graylog2	graylog2-server	15,822	2,600	13,222	1,465	0.11	11,757	0.89	5,464	141
prestodb	presto	15,052	5,455	9,597	1,069	0.11	8,528	0.89	10,832	178
apache	storm	14,055	3,791	10,264	1,244	0.12	9,020	0.88	9,945	25
apache	flink	13,466	1,869	11,597	1,343	0.12	10,254	0.88	11,022	25
druid-io	druid	12,427	4,215	8,212	1,947	0.24	6,265	0.76	20,329	391
naver	pinpoint	11,976	2,864	9,112	1,012	0.11	8,100	0.89	5,950	11
google	closure-compiler	11,899	2,948	8,951	60	0.01	8,891	0.99	168	75
apache	activemq	-	-	11,135	1,754	0.16	9,381	0.84	12,988	51
apache	camel	-	-	35,649	8,003	0.22	27,646	0.78	110,150	107
apache	hadoop	-	-	48,582	2,395	0.05	46,187	0.95	25,057	240
apache	hbase	-	-	29,097	2,161	0.07	26,936	0.93	10,059	538
apache	karaf	-	-	15,953	5,853	0.37	10,100	0.63	54,655	60
apache	wicket	-	-	31,456	1,527	0.05	29,929	0.95	13,322	243
hibernate	hibernate-search	-	-	5,976	1,177	0.20	4,799	0.80	6,251	105
jenkinsci	jenkins	-	-	26,286	4,551	0.17	21,735	0.83	28,138	483
spring-projects	spring-roo	-	-	6,440	675	0.10	5,765	0.90	10,173	35
wildfly	wildfly	-	-	23,370	5,186	0.22	18,184	0.78	43,384	74
	Sum	-	-	543,127	66,984	-	476,143	-	641,056	4,253
	Average	-	-	18,104	2,233	0.12	15,871	0.88	21,369	141.77

a study participant has found have also been found by our approach. Similar to the evaluation of Dintzner *et al.* [29], we are able to evaluate the correctness and the completeness of our approach with these performance measures. The results of the evaluation show a high precision and recall of 0.9513 and 0.9796, respectively for Participant 1. For Participant 2, the results show a precision of 0.9601 and a recall of 0.9844. Averaging the values of both participants, we obtain a mean precision of 0.9557 and a mean recall of 0.9820. The detailed results of the evaluation are provided in the supplementary material.[1]

We also evaluated the errors per build change type according to our taxonomy. We used the classification of the two manual evaluations to calculate precision and recall per change type. We found that 36 out of the 58 change types could be detected among the randomly selected evaluation commits with precision and recall of 1. This group contains changes, such as DEPENDENCY_VERSION_UPDATE and GENERAL_PROPERTY_INSERT. 9 change types showed a precision and recall between 0.80 and 1. The other 13 change types that were contained in the evaluation scored lower than 0.80, *e.g.*, PLUGIN_UPDATE. We found that especially PLUGIN_UPDATE is a change type that is difficult to detect

properly by our approach, because it is tightly coupled with PLUGIN_CONFIGURATION_UPDATE.

Besides the quantitative evaluation, we performed a qualitative evaluation to find out in which scenarios BUILDDIFF shows a good performance and in which it does not. We present an example where BUILDDIFF did not achieve a proper change extraction compared to a human evaluation. We refer the interested reader to the supplementary material[1] to find more types of wrong classification. The example is taken from the flink project[6] where a dependency definition was changed. Listing 5 shows the dependency in the old version and Listing 6 shows the updated version of the dependency. BUILDDIFF extracted two changes, DEPENDENCY_DELETE and DEPENDENCY_INSERT. In fact, this is an update of the same dependency where the `groupId` and the `artifactId` changed simultaneously. Hence, the correct classification would be a DEPENDENCY_UPDATE. Our approach could not detect this change correctly because we use a distance measure to match the nodes of two build files. In this case, the measure indicated that the two dependency definitions are not close enough to be considered the same

[6]http://goo.gl/rWPFDy

dependency and consequently, BUILDDIFF extracted the two changes wrongly.

Listing 5
EXAMPLE OF AN INCORRECT CLASSIFICATION - OLD VERSION

```
<dependency>
 <groupId>com.typesafe.akka</groupId>
 <artifactId>akka-testkit_${scala.binary.
     ↪ version}</artifactId>
 <scope>test</scope>
</dependency>
```

Listing 6
EXAMPLE OF AN INCORRECT CLASSIFICATION - NEW VERSION

```
<dependency>
 <groupId>com.data-artisans</groupId>
 <artifactId>flakka-testkit_${scala.binary.
     ↪ version}</artifactId>
 <scope>test</scope>
</dependency>
```

In conclusion, we observe that:

> BUILDDIFF is capable of extracting changes from MAVEN build files with an average precision of 0.96 and an average recall of 0.98. DEPENDENCY_VERSION_UPDATE and GENERAL_PROPERTY_INSERT are among the change types that achieve the best performance, whereas PLUGIN_UPDATE is among the change types with the highest rate of error.

V. BUILD CHANGE FREQUENCY (RQ1)

Our first experiment investigates the frequency of build changes. We aim to gain knowledge of which change types are frequently performed in projects and how often they are performed. This information can help to understand the evolution of build files similar to the study of Gall *et al.* [8]. With this experiment, we aim to answer RQ1: *Which build change types occur the most frequently?*

Approach. First, we checked the projects for their number of build changes. The projects h2o-2 (last commit: Nov 4, 2014), guava (last commit: Jan 1, 2017, only 11 build changing commits in 2016), and closure-compiler (last commit: Nov 15, 2016, only 4 build changing commits in 2016) contain less than 1000 build changes each. Hence, we excluded them from the experiment because we assume that they do not use MAVEN actively and only keep the MAVEN configuration in the source code management system for legacy reasons.

Second, starting with the change data stored in the ChangeDB, we iterated over the remaining 27 projects and counted the occurring changes. We counted the number of each change type per project and aggregated the numbers also per change category. As depicted in Table II, the selected projects differ in their size, and hence, contain a different amount of total build changes (column BC). Given this variance in the number of build changes, we normalized the change counts

to allow a fair comparison between the studied projects. We divided each aggregated change count by the number of total build changes in the project. For example, project spring-roo contains a total amount of 9,534 build changes and 222 instances of the change type MODULE_INSERT. We calculated the relative occurrence of this change type with $222/9534 = 2.33\%$ and used this relative value instead of the absolute value for our experiment. We then analyzed this data in two ways. First, we analyzed the relative occurrence of each build change type, and second, we analyzed the number of build changes per change type category.

Results. The most frequently occurring change type is PARENT_VERSION_UPDATE with a relative frequency of 0.41 on average, meaning that on average 41% of the build changes are of this type. The second most occurring change type is PROJECT_VERSION_UPDATE having an average relative frequency of 0.08. We observe a large drop (0.33) in the relative frequency of those two types of changes underlining that PARENT_VERSION_UPDATE is the most frequently occurring change type by far. These two change types are followed by DEPENDENCY_INSERT (0.06) and GENERAL_PROPERTY_UPDATE (0.03).

We see that the top 10 most frequent change types consist of four change types that modify dependencies, and two change types concerning version changes, plugin changes and changes to properties, respectively. This indicates that the dependency management system, which is a core part of the build system, is changed frequently. We also observe that the configurations of plugins are frequently changed which indicates the importance of this part of the build configuration. Furthermore, we observe that the top 10 change types account for 73% of all changes. Figure 1 shows boxplots of the top 10 most frequent change types.
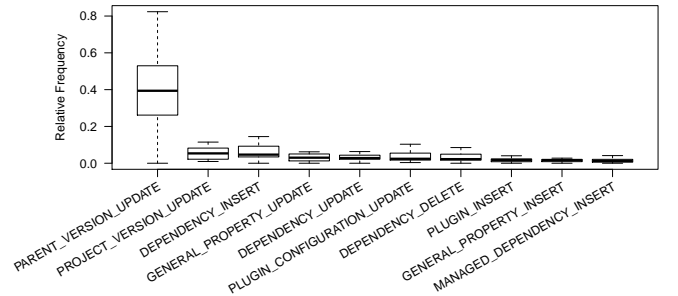

Fig. 1. Boxplots of the relative build change frequency for the top 10 most frequent change types (sorted by median).

The next step of the analysis deals with the frequencies of build changes per change category that we have defined in Section III-A. Figure 2 shows the relative frequencies of the build changes per build category. We observe that the General Changes category accounts for 0.64 (64%) of all changes on average. We argue that this ratio is as expected because changes to the properties, parent changes, and changes to the project metadata, such as project version, are aggregated in this category.

Furthermore, we can see that `Dependency Changes` are the second most frequent change category (0.24). This is in line with the observations of the single change types. As mentioned above, the dependency management system is a core part of the build system and is frequently updated. The third most frequently occurring category contains the changes to the `Build Changes` category (0.11). Lastly, changes to the `Repository Changes` and to the `Team Changes` are rare (0.008 and 0.004, respectively).
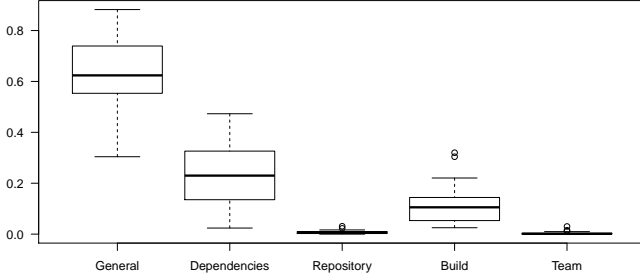

Fig. 2. Boxplots of the relative frequency of build changes per change type category.

With these results, we can answer research question RQ1:

> Among the top 10 most frequently occurring change types, we find version changes and dependency changes frequently. The most frequent change type is PARENT_VERSION_UPDATE followed by PROJECT_VERSION_UPDATE, and DEPENDENCY_INSERT. The most frequently occurring change category is `General Changes` directly followed by `Dependency Changes`, and `Build Changes`.

VI. WHEN ARE THE CHANGES RECORDED (RQ2)

In this section, we investigate when the build changes occur. We suppose that build changes are not equally distributed over the project, but have phases in which they occur significantly more frequently than in other phases. Hence, we used the build change data that we extracted using BUILDDIFF to check whether our hypothesis holds and answer RQ2: *When are build changes recorded?*

Approach. We started with the aggregated change data that we created in Section V. This data contains for each commit of a project the number of changes per change type that have been performed in the commit. For this research question, we added the date on which the commit was performed and summed up all build changes to a single value per day, *i.e.*, one row of our dataset contains the ID of the commit, the number of build changes that were performed in that commit, and the date of the commit. Based on this information, we investigated the data in two ways, as a single day value and with a sliding window approach.

The first investigation treats each day as a single data point, and hence, adds the number of build changes of commits that were made on the same day. For example, if exactly two commits were made on 23th June 2016 with 10 and 15 build changes, respectively, we created a single data point with 25 build changes. The second investigation uses this data and applies a sliding window approach, similar to the approach of Maarek *et al.* [30]. We summed the number of build changes of k days to increase the context of the build changes.

As our hypothesis for this research question states, we suppose that build changes are not equally distributed over the project, but occur more frequently in some time periods of the project. We further suppose that one special period in the project that shows a significantly higher amount of build changes, is the time around releases. Thus, we extracted the release data of each of the studied projects provided by the GitHub API. In particular, we extracted the commit ID of the release, the day of the release, and its name. Column #R of Table II shows the number of releases per project that we could extract. We can see that for the project `CoreNLP`, we could not retrieve release data. Hence, we also excluded this project and performed the experiment with the remaining 26 projects.

To substantiate our claim, we will show that days that are close to a release contain statistically significantly more build changes than days that are not close to a release. We consider a single day as well as a sliding window approach. To that extent, we consider a day to be close to a release if it is in between k days before the release. For the analysis on a daily basis, we consider $k = 1$ and for the sliding window approach, we consider $k \in \{5, 7, 9\}$ days. We choose different values for k to investigate the influence of the size of the window on the results. We did not try with larger window sizes because we argue that changes that happen more than 9 days before a release should not be considered close to a release. This argumentation is in line with the release data that we used because the average number of days between two consecutive releases is 14 days. Hence, selecting larger k values would possibly cover more than one release. We performed the study with all of the k values and the results were similar. Thus, we only present results for $k = 7$ in the paper. The results for $k \in \{5, 9\}$ can be found in the supplementary material.[1]

Next, we checked if the distributions are significantly different with a Mann-Whitney-Wilcoxon test ($\alpha < 0.01$) and calculated the effect size d using Cliff's Delta [31]. We used Mann-Whitney-Wilcoxon and Cliff's Delta since the number of build changes is non-normal distributed. The effect size is considered negligible for $d < 0.147$, small for $0.147 \leq d < 0.33$, medium for $0.33 \leq d < 0.47$, and large for $d \geq 0.47$ [32].

Results. Figure 3 shows the distribution of build changes across the project `spring-roo`. We refer the reader to the supplementary material[1] for the figures of all of the studied projects. The black line depicts the number of build changes according to the sliding window approach ($k = 7$). Each vertical red line indicates a release. We can see that most of the peaks of the black line (number of build changes) appear near a vertical red line (release). This suggests that our hypothesis is correct.
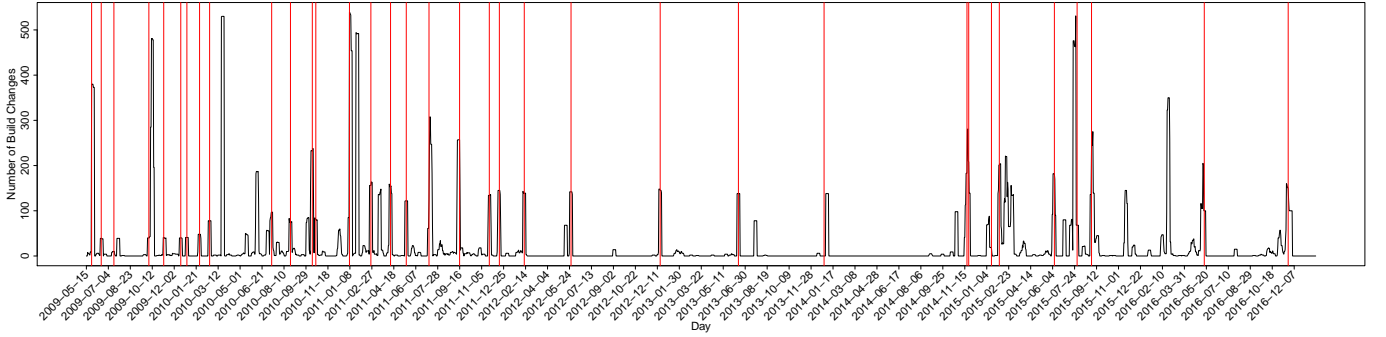
Fig. 3. Number of build changes over time in `spring-roo` using a sliding window ($k = 7$). Releases are depicted as vertical red lines.
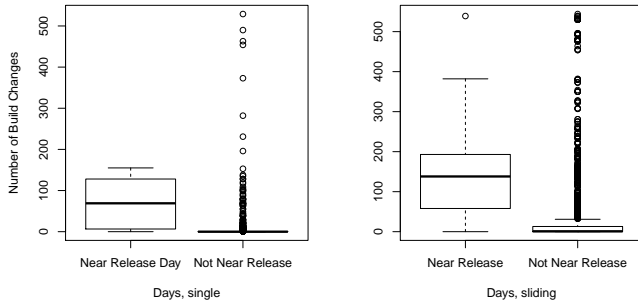

Fig. 4. Boxplots showing the distributions of the counts of near release and non-near release build changes of `spring-roo` as computed with the single day approach and sliding window approach using $k = 7$.

Furthermore, looking at the distribution of the number of build changes in days near releases and comparing it with the distribution of build changes in days that are far from releases, we can see that the distributions appear to be different. Figure 4 shows the boxplots for both approaches and both distributions. The left two boxplots represent the single day approach whereas the right two boxplots represent the sliding window approach using $k = 7$. In each of the two plots, the respective boxplot on the left depicts the distribution of build changes on days near a release whereas the respective boxplot on the right depicts the distribution of build changes on days that are not near a release.

Table III presents the p-value (p) of the Mann-Whitney-Wilcoxon test and Cliff's Delta d for each project and approach in detail. The p-values show that the frequency of build changes near and not near a release differ significantly (all $p < 0.01$). Furthermore, the effect size can be considered large in all projects except `hadoop` (small) and `wicket` (medium). The `jenkins` project achieves a large effect size with the single day approach but only shows a small effect size with the sliding window approach. The `presto` project also shows a large effect size with the single day approach but only a medium effect size with the sliding window approach. We find that these lower effect sizes are caused by the release information. `hadoop` and `wicket` show a dense release plan

TABLE III
RESULTS OF THE MANN-WHITNEY-WILCOXON TEST (P: P-VALUE) AND CLIFFS DELTA d OF THE DISTRIBUTIONS OF THE NUMBER OF BUILD CHANGES NEAR AND NON-NEAR RELEASES USING THE SINGLE DAY APPROACH AND SLIDING WINDOW APPROACH WITH $k = 7$.

Project	Single Day		Sliding ($k = 7$)	
	p	d	p	d
activemq	2.73E-35	0.75	3.31E-15	0.64
alluxio	3.17E-15	0.79	7.93E-06	0.52
camel	3.03E-57	0.89	1.20E-29	0.64
deeplearning4j	1.33E-34	0.96	2.00E-17	0.77
druid	8.57E-225	0.97	2.42E-92	0.68
flink	8.78E-21	0.89	3.59E-10	0.73
graylog2-server	2.47E-117	0.84	3.21E-21	0.47
hadoop	1.43E-26	0.30	1.21E-16	0.30
hazelcast	1.53E-58	0.61	1.86E-25	0.54
hbase	1.49E-194	0.63	2.38E-102	0.57
hibernate-search	1.14E-177	0.98	3.53E-42	0.76
jenkins	5.28E-235	0.86	5.10E-08	0.15
jetty	2.17E-200	0.96	8.73E-72	0.66
karaf	1.79E-48	0.98	1.34E-26	0.80
languagetool	2.26E-99	0.94	3.27E-24	0.94
neo4j	2.07E-68	0.70	1.23E-34	0.55
netty	1.55E-203	0.94	2.05E-50	0.69
orientdb	8.71E-93	0.94	2.68E-28	0.66
pinpoint	1.94E-14	0.95	6.59E-07	0.86
presto	1.27E-145	0.98	2.07E-23	0.46
sonarqube	8.37E-85	0.93	1.46E-25	0.59
spring-boot	2.81E-50	0.97	3.45E-26	0.76
spring-roo	8.27E-42	0.78	8.57E-17	0.77
storm	9.55E-15	0.70	4.20E-06	0.53
wicket	1.19E-59	0.37	3.94E-29	0.41
wildfly	2.61E-45	0.93	9.97E-26	0.71

in the beginning of the data and this can influence the sliding window approach. Similarly, the changes of the `jenkins` and `presto` projects are performed on the release day and hence, the inclusion of additional days, as done by the sliding window approach, lowers the effect size.

With these results, we can answer research question RQ2 as follows:

Build changes are not equally distributed over the projects' timeline. There are particular phases which show significantly higher build change frequencies than others. Especially around releases, a high build change frequency is observed.

VII. DISCUSSION

In this section, we first discuss a number of implications of our results on recent and ongoing research of build systems and their configuration. Next, we discuss implications for developers who use MAVEN as build system. Finally, we discuss the threats to the validity of our results.

A. Implications of the Results

On Research. First, compared to the state-of-the-art our fine-grained build changes enable a more detailed analysis of the co-evolution of source code and build files. Second, studies on effort estimation, such as that from Sarro *et al.* [33], can be refined by taking into account our 95 types of build changes. Third, refactoring approaches, such as MAKAO [9] and Formiga [12] can be applied to MAVEN build files and enriched with our detailed change information to improve the refactoring process of MAVEN build files. Fourth, studies of build complexity [1] can also benefit from our detailed analysis of build changes by including dynamical information, such as our detailed build change information, to the calculation of the metrics. Finally, our build changes can be used to improve the models to predict bug-prone build files [22] or suggest potentially missing changes to build configurations [6] that might lead to a build breakage.

On Development. We observed that build changes occur more frequently near releases. This observation can help developers to avoid build breakage by increasing the awareness that each change to the build configuration can possible break the build. Furthermore, project managers can use this finding to consider the peak of build changes near releases in their planning of releases and work load. We also give insight into the type of build changes that are frequently made. This can be used by developers, for instance, to identify and refactor plugins that often change their configuration.

B. Threats to Validity

Regarding the validity of our results, we identified the following threats to construct, internal, and external validity.

Construct Validity. One threat is that our taxonomy may not cover all possible change types or change categories that could be theoretically made to a build file. We mitigated this threat in two ways. First, we compared the taxonomy with the XML schema of MAVEN build files to cover all important changes. Second, we asked two experienced MAVEN users to verify the taxonomy and create the categories, including a discussion if necessary. Furthermore, we retrieved the release data of the 30 open source Java projects from GitHub as the only resource. To that extent, we could miss possible releases if they are not covered by the GitHub data. However, we mitigated this threat by manually checking if the data is compliant with the data provided by the source code management system.

Internal Validity. A threat to internal validity concerns whether BUILDDIFF can extract the changes to a build configuration file accurately. We mitigated this threat by covering all changes of the taxonomy with JUnit tests and a manual evaluation comparing against the opinions of two experienced MAVEN users. Concerning the evaluation of BUILDDIFF, a threat is that the randomly selected commits do not include all change types. We mitigated this threat by calculating the proportion of actually missed changes due to the selection. We observed that we only miss 0.9% of the changes and hence, we can safely assume that the majority of the changes will be covered by BUILDDIFF.

External Validity. The main threat to external validity stems from the selection of projects that we used in our study. We mitigated this threat by selecting 30 open source Java projects of different vendors, sizes, and purposes. However, additional experiments with projects using other build systems and from industrial settings are needed to further generalize our results. Another threat to external validity is that our taxonomy is tailored for MAVEN build configurations. While we designed the taxonomy to be usable for other build tools as well, the taxonomy may not generalize to all other build systems.

VIII. CONCLUSIONS

Build systems are an essential part in the engineering process of modern software systems. In this paper we introduced BUILDDIFF, an approach for extracting fine-grained build changes from MAVEN build files. In a manual evaluation, we showed that BUILDDIFF is capable of extracting build changes with an average precision and recall of 0.96 and 0.98, respectively. With the build changes extracted from 30 open source Java projects we performed two empirical studies to investigate the frequency and time of build changes. The results of the two studies showed:

- (RQ1) The most frequent change type is PARENT_VERSION_UPDATE followed by PROJECT_VERSION_UPDATE, and DEPENDENCY_INSERT. The most frequent change category is `General Changes` directly followed by `Dependency Changes`, and `Build Changes`. The top 10 change types account for 73% of all changes.
- (RQ2) Build changes are not equally distributed over the projects' timeline. We observed that especially near releases build changes occur more frequently.

Our results benefit research on build configurations and developers using MAVEN as their build system.

Future Work. We plan to extend BUILDDIFF to support other build systems, such as Gradle,[7] and compare them with MAVEN. Furthermore, we will investigate build changes to find frequent change patterns among commits and work items that affect the build result. Finally, we plan to perform a more detailed analysis of the co-evolution between build changes and source code changes.

REFERENCES

[1] 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, 2012.

[7] https://gradle.org

[2] H. Seo, C. Sadowski, S. Elbaum, E. Aftandilian, and R. Bowdidge, "Programmers' build errors: a case study (at google)," in *International Conference on Software Engineering*. ACM, 2014, pp. 724–734.

[3] N. Kerzazi, F. Khomh, and B. Adams, "Why do automated builds break? an empirical study." in *International Conference on Software Maintenance and Evolution*. IEEE, 2014, pp. 41–50.

[4] B. Adams, K. D. Schutter, H. Tromp, and W. D. Meuter, "The evolution of the linux build system," *Electronic Communication of the European Association of Software Science and Technology*, vol. 8, 2007.

[5] S. McIntosh, B. Adams, M. Nagappan, and A. E. Hassan, "Mining co-change information to understand when build changes are necessary," in *Proceedings of the International Conference on Software Maintenance and Evolution*. IEEE, 2014, pp. 241–250.

[6] C. Macho, S. McIntosh, and M. Pinzger, "Predicting Build Co-Changes with Source Code Change and Commit Categories," in *Proceedings of the International Conference on Software Analysis, Evolution, and Reengineering*. IEEE, 2016, pp. 541–551.

[7] B. Fluri, M. Würsch, M. Pinzger, and H. C. Gall, "Change distilling: Tree differencing for fine-grained source code change extraction," *Transactions on Software Engineering*, vol. 33, no. 11, pp. 725–743, 2007.

[8] H. C. Gall, B. Fluri, and M. Pinzger, "Change analysis with evolizer and changedistiller," *IEEE Software*, vol. 26, no. 1, pp. 26–33, 2009.

[9] B. Adams, H. Tromp, K. D. Schutter, and W. D. Meuter, "Design recovery and maintenance of build systems," in *Proceedings of the International Conference on Software Maintenance*. IEEE, 2007, pp. 114–123.

[10] S. McIntosh, B. Adams, and A. E. Hassan, "The evolution of ANT build systems," in *Proceedings of the International Working Conference on Mining Software Repositories*. IEEE, 2010, pp. 42–51.

[11] S. McIntosh, M. Nagappan, B. Adams, A. Mockus, and A. E. Hassan, "A large-scale empirical study of the relationship between build technology and build maintenance," *Empirical Software Engineering*, vol. 20, no. 6, pp. 1587–1633, 2015.

[12] R. Hardt and E. V. Munson, "An empirical evaluation of ant build maintenance using formiga," in *Proceedings of the International Conference on Software Maintenance and Evolution*, 2015, pp. 201–210.

[13] ——, "Ant build maintenance with formiga," in *Proceedings of the International Workshop on Release Engineering*. IEEE, 2013, pp. 13–16.

[14] X. Xia, D. Lo, S. McIntosh, E. Shihab, and A. E. Hassan, "Cross-project build co-change prediction," in *Proceedings of the International Conference on Software Analysis, Evolution, and Reengineering*. IEEE, 2015, pp. 311–320.

[15] X. Xia, D. Lo, X. Wang, and B. Zhou, "Build system analysis with link prediction," in *Symposium on Applied Computing*. ACM, 2014, pp. 1184–1186.

[16] W. Miller and E. W. Myers, "A file comparison program," *Software: Practice and Experience*, vol. 15, no. 11, pp. 1025–1040, 1985.

[17] E. W. Myers, "Ano (nd) difference algorithm and its variations," *Algorithmica*, vol. 1, no. 1-4, pp. 251–266, 1986.

[18] M. Hashimoto and A. Mori, "Diff/TS: a tool for fine-grained structural change analysis," in *2008 15th Working Conference on Reverse Engineering*. IEEE, 2008, pp. 279–288.

[19] B. Fluri and H. C. Gall, "Classifying change types for qualifying change couplings," *IEEE International Conference on Program Comprehension*, vol. 2006, pp. 35–45, 2006.

[20] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *Proceedings of the International Conference on Automated Software Engineering*. ACM, 2014, pp. 313–324.

[21] C. Désarmeaux, A. Pecatikov, and S. McIntosh, "The Dispersion of Build Maintenance Activity across Maven Lifecycle Phases," in *International Conference on Mining Software Repositories*. ACM, 2016, pp. 492–495.

[22] E. Giger, M. Pinzger, and H. C. Gall, "Comparing Fine-grained Source Code Changes and Code Churn for Bug Prediction," in *International Working Conference on Mining Software Repositories*. ACM, 2011, pp. 83–92.

[23] ——, "Can we predict types of code changes? an empirical analysis," in *Working Conference of Mining Software Repositories*. IEEE, 2012, pp. 217–226.

[24] E. Giger, M. D'Ambros, M. Pinzger, and H. C. Gall, "Method-level bug prediction," in *International Symposium on Empirical Software Engineering and Measurement*. ACM, 2012, pp. 171–180.

[25] D. Romano and M. Pinzger, "Using source code metrics to predict change-prone Java interfaces," in *Proceedings of the International Conference on Software Maintenance*. IEEE, 2011, pp. 303–312.

[26] B. Fluri, M. Würsch, E. Giger, and H. C. Gall, "Analyzing the co-evolution of comments and source code," *Software Quality Journal*, vol. 17, no. 4, pp. 367–394, 2009.

[27] G. Dotzler and M. Philippsen, "Move-optimized source code tree differencing," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2016, pp. 660–671.

[28] J. Nielsen, "Card sorting to discover the users model of the information space," 1995.

[29] N. Dintzner, A. Van Deursen, and M. Pinzger, "Extracting Feature Model Changes from the Linux Kernel Using FMDiff," in *International Workshop on Variability Modelling of Software-Intensive Systems*. ACM, 2014, pp. 22:1–22:8.

[30] Y. S. Maarek, D. M. Berry, and G. E. Kaiser, "An information retrieval approach for automatically constructing software libraries," *Transactions on Software Engineering*, vol. 17, no. 8, pp. 800–813, 1991.

[31] N. Cliff, "Dominance statistics: Ordinal analyses to answer ordinal questions." *Psychological Bulletin*, vol. 114, no. 3, p. 494, 1993.

[32] R. J. Grissom and J. J. Kim, *Effect Sizes for Research: A Broad Practical Approach*. Lawrence Erlbaum Associates, 2005.

[33] F. Sarro, A. Petrozziello, and M. Harman, "Multi-objective software effort estimation," in *Proceedings of the 38th International Conference on Software Engineering*. ACM, 2016, pp. 619–630.