Automatically Repairing Dependency-Related Build Breakage

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

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

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

***Abstract*—Build systems are widely used in today's software projects to automate integration and build processes. Similar to source code, build specifications need to be maintained to avoid outdated specifications, and build breakage as a consequence. Recent work indicates that neglected build maintenance is one of the most frequently occurring reasons why open source and proprietary builds break.**

In this paper, we propose BUILDMEDIC, an approach to automatically repair Maven builds that break due to dependency-related issues. Based on a manual investigation of 37 broken Maven builds in 23 open source Java projects, we derive three repair strategies to automatically repair the build, namely *Version Update*, *Delete Dependency*, and *Add Repository*. We evaluate the three strategies on 84 additional broken builds from the 23 studied projects in order to demonstrate the applicability of our approach. The evaluation shows that BUILDMEDIC can automatically repair 45 of these broken builds (54%). Furthermore, in 36% of the successfully repaired build breakages, BUILDMEDIC outputs at least one repair candidate that is considered a correct repair. Moreover, 76% of them could be repaired with only a single dependency correction.

I. INTRODUCTION

With the recent growth of social coding platforms, such as GitHub[1] and Bitbucket,[2] Continuous Integration (CI) has also grown in popularity [1]. Development teams use a CI tool, such as Travis CI[3] or Jenkins,[4] to automatically build the project once a commit has been pushed to the source code repository [2]. CI improves the productivity of the team [3] because integration failures are immediately detected, and can be fixed while coding and design decisions are still fresh in the minds of developers [4].

However, Hilton *et al.* showed that developers are confronted with trade-offs when using CI [5]. For example, the benefit of immediate feedback comes with an additional burden in maintenance of build specifications [6] and furthermore, it has been shown that neglecting build maintenance can lead to build breakage [7]. Development teams are blocked by the build breakage and have to fix the build prior to proceeding with their main work. Kerzazi *et al.* [8] studied a project of a large software company and found that a minimum of 900 man-hours were spent on fixing build breakage over a six-month period.

The reasons for build breakage are manifold. Recent studies have focused on testing issues [9] and build specification issues [7], [10], [11]. Concerning build specification issues, the main reasons for build breakage are dependency-related issues, such as dependency resolution errors or outdated dependency configuration. Prior work investigated the frequency of different types of build breakage and found that dependency-related issues account for 39% [10], 58% [11], and 65% [7] of the analyzed breakages. However, to the best of our knowledge, there are no studies that have investigated the detailed reasons for the dependency-related build breakage and how such build breaks were repaired. We address this gap with our first research question:

(RQ1) How do developers repair dependency-related build breakage?

To answer this question, we use BUILDDIFF [12] to mine the changes in the Maven build specifications of 23 Java open source projects and introduce MAVENLOGANALYZER (MLA), a tool to extract details from the execution log of a Maven build. We then study these changes to find out which types of changes have been performed by developers in the past to repair dependency-related build breakage.

The results show that 17 of the 37 (46%) studied dependency-related build breakages were fixed by updating dependency versions (*e.g.*, promoting so-called "snapshot" dependencies by removing the "-SNAPSHOT" suffix). An additional five breakages (15%) were repaired by removing invalid dependencies. Furthermore, 30/37 (81%) of the breakages were repaired with only a single dependency correction.

Armed with an understanding of how developers fix dependency-related build breakage, we set out to automate the fixing process, which leads us to our second research question:

(RQ2) To what extent can we automatically repair dependency-related build breakage?

To address this RQ, we introduce BUILDMEDIC, an approach to automatically repair dependency-related build breakage. BUILDMEDIC implements three repair strategies that we derive from the most frequent developer fixes that we observe in RQ1.

[1]https://github.com
[2]https://bitbucket.org
[3]https://travis-ci.org
[4]https://jenkins.io

We evaluate BUILDMEDIC using an additional 84 randomly selected revision pairs that exhibit a build breakage, of which, BUILDMEDIC can automatically repair 45 (54%) of which 36% were found to be identical to the developer's repair actions. In 76% of the repairs, BUILDMEDIC only needs to correct a single dependency. Concerning performance, we find that BUILDMEDIC runs with an average total time of 22.8 minutes and an average overhead of only 8.6 minutes.

In summary, this work makes the following contributions: (1) an empirical investigation of how developers fix dependency-related build breakage, (2) MAVENLOGANALYZER (MLA), a tool to extract the build result and build details from a Maven build log, (3) BUILDMEDIC, an approach to automatically repair dependency-related errors, (4) a reference set of revision pairs that exhibit dependency-related build breakage and the corresponding fixes, and (5) a replication package that contains supplementary material.[5]

The remainder of the paper is organized as follows: Section II situates the paper with respect to related work. Section III describes the data set that we use for the studies in this paper. In Section IV, we present the study on how developers fixed dependency-related build breakage. The derived fix strategies are described in Section V. In Section VI, we propose BUILDMEDIC, our approach to automatically fix dependency-related build breakage. We evaluate BUILDMEDIC in Section VII and conclude the paper in Section VIII.

II. RELATED WORK

Build Maintenance. Prior studies have investigated maintenance of build systems and their specification. For instance, Adams *et al.* used MAKAO [13], a framework for re(verse)-engineering build systems, to study the Linux build system. They found that the complexity of the build system grows over time and identified maintenance as the main reason for evolution [6]. McIntosh *et al.* studied evolution of ANT build systems using a complexity metric [14] and found that the ANT build system is evolving over time as well. Beside studying maintenance and co-evolution, Hardt *et al.* developed Formiga, a tool to refactor ANT build scripts [15], which has been shown to reduce the build maintenance time and improve the correctness of refactorings [16]. Furthermore, Xia *et al.* [17] developed an algorithm that outperforms state-of-the-art algorithms to predict missing dependencies in make files. Moreover, other tools to support developers in maintaining build systems and their specification exist, such as Tamrawi *et al.* [18] who developed SYMake to visualize dependencies in make files. Extracting semantic changes from build files has also been addressed in prior work. Al-Kofahi *et al.* [19] extract semantics of build specification changes in make files with MkDiff.

Build systems for Java have also been studied. For example, McIntosh *et al.* [20] investigated the co-evolution of Java build systems, and production and test code. The study revealed that these parts of a project co-evolve. In a large-scale study, McIntosh *et al.* focus on understanding the relationship between build technology and maintenance effort [21]. They found that framework-driven build technologies like Maven need to be maintained more often. Since prior work [20] indicates that up to 27% of source code work items require an accompanying change to the build system, recent studies aim to support build maintenance by applying models to predict whether build changes are needed. For instance, McIntosh *et al.* [22] try to understand when build changes are necessary. They used code change metrics retrieved by mining the history of projects to train a model that can predict whether a commit lacks an accompanying build change. Xia *et al.* [23] extended this study by investigating the possibility of applying the model in a cross project setting. Moreover, Macho *et al.* [24] improved both models by adding fine-grained source code change information to the model. Similar to Al-Kofahi *et al.* for ANT systems, Macho *et al.* [12] apply BUILDDIFF to extract build changes from Maven build specification files.

These prior studies mainly focus on empirically gaining knowledge about build maintenance. Our work aims at going one step further by providing a prototype that implements our findings and supports developers during maintenance of build systems and their specification by automatically repairing broken build specifications.

Automatic Repair. Automatic Repair has already been addressed by prior work mainly to fix bugs in programs' source code. Forrest *et al.* applied Genetic Programming to the problem of bug fixing in C programs [25]. Moreover, Weimar *et al.* present GenProg [26], a generic approach to automatically repair bugs in source code files using a genetic algorithm. GenProg was used in subsequent studies, such as Fast *et al.* [27] which aims to improve fitness functions for the genetic algorithm and Le Goues *et al.* [28] which improves the efficiency and precision of the fitness function. Furthermore, Le Goues *et al.* also evaluate the fixing capability of their approach and estimate the cost to automatically repair each bug is on average $8 [29]. In the same study they could fix 55 of 105 bugs in 8 open source programs. Le *et al.* use historical information to prioritize the application of changes for fixing bugs [30]. Martinez *et al.* use the Defects4J dataset to investigate the real correctness of fixes performed by automatic program repair techniques [31].

However, to the best of our knowledge, there is no approach that is capable of automatically repairing broken builds. We address this gap in this paper and propose BUILDMEDIC, an approach to automatically repair dependency-related build breakage.

III. DATA PREPARATION

In this section, we describe the studied data set. We first describe the selection of the projects under investigation and then provide an overview of the data retrieval process.

A. Project Selection

For our studies, we considered open source Java projects of different sizes, vendors, and purposes. To minimize selection

[5] https://mitschi.github.io/tools/

bias, we used GitHub, a popular social coding platform, as our sample population and queried the GitHub API to receive a list of the top-1000 Java projects with the most stars. Similar to prior studies [12], [24], [32], we then selected the projects that fulfill the following criteria (the criteria thresholds are chosen conservative in order to minimize false positives):

Maven as build tool. We restrict our selection to Maven projects for two reasons. First, Maven is a broadly adopted technology for building software systems [21]. Second, Maven has been the focus of many prior studies [12], [20], [21], [24].

More than 500 commits. We select projects that contain at least 500 commits to avoid including projects that have not yet reached maturity because projects usually change differently in the starting phase and in the maintenance phase [33]. Including such projects could bias our conclusions. We use GitHub to obtain the total number of commits for each candidate project.

Actively developed. Furthermore, projects that are discontinued are also excluded. We identify discontinued or abandoned projects in two ways. First, we check the date of the last commit. If the project does not have at least one commit in 2017, we assume the project is no longer under active development. Second, we manually checked if the project description contains any sign of discontinuity (*e.g.*, "project no longer maintained/discontinued").

Build without manual setup or intervention. Build systems often rely on the environment that they are used in. To avoid flagging builds as broken when the build environment was not properly configured, we restrict our analysis to projects that do not require any manual setup or environmental configuration. Furthermore, we manually investigate the build results of the candidate projects and remove projects that build incorrectly without manual intervention. We consider projects to build incorrectly if they fail because installation of a program is necessary or other additional steps need to be performed (*e.g.*, Apache Hadoop requires that `npm` is installed prior to executing the build).

After applying the criteria, we were left with 23 projects consisting of a total of 183,466 commits including 32,100 build-changing commits. Table I lists the selected projects and shows the total number of commits, the number of build-changing commits, and the number of build changes that we extracted with BUILDDIFF. We use this list to create the data set that we describe in the next section.

TABLE I
JAVA PROJECTS USED IN OUR STUDIES PLUS VALUES FOR THEIR BCC (BUILD-CHANGING COMMITS) AND BC (BUILD CHANGES); *INDICATES THAT THE PROJECT DID NOT HAVE A `DEPENDENCY_RESOLUTION_FAILED` TO `SUCCESS` REVISION PAIR

Project	#Commits	#BCC	#BC
Activiti*	8,019	1,333	22,013
alluxio	24,982	3,289	38,889
async-http-client	5,064	1,030	3,789
closure-compiler	10,367	91	230
cucumber-jvm	3,454	1,186	19,878
druid	8,586	2,524	28,795
fastjson	2,807	322	855
hazelcast	24,819	2,556	16,549
immutables	1,378	396	4,731
jersey	3,232	1,496	66,754
keycloak	10,918	3,318	125,691
libgdx	12,983	557	7,249
mapdb*	2,407	333	653
pinot*	5,188	530	2,656
retrofit	1,564	392	1,823
solo	1,791	298	857
storm	11,402	2,228	27,902
swagger-core	3,271	853	12,232
symphony	5,052	260	423
undertow	5,550	726	7,825
vavr	4,131	291	1,350
wildfly	25,276	7,631	127,045
YCSB	1,225	460	5,636
Total	183,466	32,100	523,825

B. Data Preparation Process

The data preparation process consists of the following five steps: First, we build all of the revisions of the selected projects that are within the studied time period and save the build output. Second, we extract the build details, such as the build result, from the build output. Third, we pair revisions according to their child-parent relation in the source code management system and only keep pairs that fix dependency-related build breakage. As a fourth step, we extract the build changes [12] that were performed between the parent and the child revisions. In the fifth step, we create two separate data sets for investigating RQ1 and RQ2. Below, we describe each step in detail.

Build Revisions. The first step of the data retrieval process builds all of the revisions of the selected projects that have been committed after December 31, 2014 and before July 13, 2017 (day of data retrieval) including revisions on branches. We only consider revisions within this time period to mitigate ecosystem-related build failures, *e.g.*, those that stem from libraries that are outdated or are no longer available. A preliminary analysis of the build results showed that revisions before December 31, 2014 are more likely to have ecosystem-related build failures.

We are aware that data sets, such as TravisTorrent [34], do exist. However, as pointed out by Zolfagharinia *et al.* [35], build results can be unreliable and difficult to reproduce because they depend on the environment, *e.g.*, build command parameters and system configuration parameters. We build each studied revisions using our own execution parameters with the same build command parameters to avoid inconsistency in the flagged build failures.

For each revision, we run the Maven command `mvn -U clean package -DskipTest=true` to invoke the packaging of the project. The toggle `-U` "forces a check for missing releases and updated snapshots on remote repositories".[6] This is necessary because the `mvn` command waits for some time after the download of a dependency has failed. This timeout can affect the execution of the next build and even cause the next build to fail (even though it would not fail under normal circumstances). We enable the `-U` toggle to avoid such errors.

[6] `See mvn --help`

Furthermore, we set the `skipTests` parameter to `true` to exclude the testing phase from the build. We exclude test execution for two reasons. First, we are only interested in dependency-related errors in this study. Second, the execution of tests can take a large amount of time. The output of the Maven build is saved in a build log file that contains the build details, such as the build result, the failing module, and a description of the error.

Extract Build Details. We developed MAVENLOGANALYZER (MLA), a tool that uses a set of regular expressions to extract the build result and details of the build from the build logs, such as the failing module or the missing dependency that caused the breakage. We refer the reader to the replication package[5] for a detailed description of the MLA approach.

The list of possible build errors is derived from the list of standard Maven errors.[7] We use a different set of categories than proposed by Vasallo *et al.* [36] because their categorization is based on the lifecycle phases of Maven, whereas we need a categorization that is based on the error types of Maven to find fixing revisions. The following list shows all of the possible build results that we extract with MLA:

- `SUCCESS`
- `DEPENDENCY_RESOLUTION_FAILED`
- `CYCLIC_DEPENDENCIES_FAILED`
- `TEST_EXECUTION_FAILED`
- `COMPILATION_FAILED`
- `GOAL_FAILED`
- `NO_PARENT_FAILED`
- `NO_CHILD_FAILED`
- `POM_PARSING_FAILED`
- `NO_POM_AVAILABLE`
- `NO_LOG`
- `UNKNOWN_FAILED`

`SUCCESS` is the only build result that indicates that a build is free of errors. `DEPENDENCY_RESOLUTION_FAILED` occurs if a dependency cannot be downloaded (Figure 1). `CYCLIC_DEPENDENCIES_FAILED` occurs if dependencies form a cycle (*e.g.*, A depends on B and B depends on A). `TEST_EXECUTION_FAILED` occurs if at least one test fails. Although we exclude the test execution phase with the `skipTests` parameter, it is possible that some tests are still executed because Maven allows any plugin to be bound to any phase in the build lifecycle. One could bind a testing plugin to the packaging phase for example. `COMPILATION_FAILED` occurs if the code does not compile cleanly. `GOAL_FAILED` occurs when build errors stem from a failed execution of a plugin *e.g.*, for deployment, such as the `maven-deploy-plugin`. `NO_PARENT_FAILED` and `NO_CHILD_FAILED` occur if the parent-child relationship between two modules is corrupted, *e.g.*, a child module is referenced but not defined. `NO_POM_AVAILABLE` occurs when the `pom.xml` file is missing, whereas `NO_LOG` indicates the absence of the build log file. If MLA could not extract a build result, the build result is set to `UNKNOWN_FAILED`. Furthermore, MLA extracts important details, such as the missing dependency and the module that is failing in the case of `DEPENDENCY_RESOLUTION_FAILED`, for each of the possible build results.

Figure 1 shows the bottom lines of the build log of revision 601cee7[8] of the Apache Storm project. MLA correctly extracts the build result as `DEPENDENCY_RESOLUTION_FAILED` and further recognizes that the missing dependency is `org.apache.storm:storm-hbase:jar:0.11.0-SNAPSHOT`. To ensure that MLA can correctly extract build details from build logs, we manually evaluate 580 randomly selected build logs from the Travis Torrent [34] data set which ensures 95% confidence level with 5% margin of error. We can use the Travis Torrent build logs because we are not verifying whether the build result is correct, but instead are verifying whether MLA can extract the build result that is represented by the build log. We found that MLA correctly extracts the build details for all of the evaluated revisions. The full evaluation details can be found in the replication package.[5]

```
[INFO] BUILD FAILURE
[INFO] ------------------------------------------------------------
[INFO] Total time: 10.362 s
[INFO] Finished at: 2017-08-04T08:52:44+02:00
[INFO] Final Memory: 25M/264M
[INFO] ------------------------------------------------------------
[ERROR] Failed to execute goal on project flux-core: Could
not resolve dependencies for project com.github.ptgoetz:
flux-core:jar:0.2.3-SNAPSHOT: Could not find artifact org.
apache.storm:storm-hbase:jar:0.11.0-SNAPSHOT in sonatype-
nexus-snapshots (https://oss.sonatype.org/content/
repositories/snapshots) -> [Help 1]
```

Fig. 1. Excerpt of a Maven build log showing a dependency resolution error

Filter pairs. This step connects parent revisions to their child revisions in the source code management system and only keeps pairs that fix dependency-related build breakage. For the context of this paper, we define a build as failing with a dependency-related error if MLA extracts `DEPENDENCY_RESOLUTION_FAILED` as the build result. Hence, a pair that fixes such errors is a tuple $\langle r1, r2 \rangle$ where $r1$ is the parent revision of $r2$, $r2$ is the child revision of $r1$, $r1$ yields `DEPENDENCY_RESOLUTION_FAILED` as its build result, and $r2$ yields `SUCCESS`. We are aware that also other build errors (*e.g.*, `COMPILATION_FAILED`) can be ascribed to a dependency-related error. However, we leave the analysis of these less frequently occurring types of build breakage [11] to future work.

Extract Build Changes. We use BUILDDIFF [12] to extract build changes from Maven build files and use them to analyze the changes that fix dependency-related build breakage. BUILDDIFF parses two versions of a `pom.xml` file and represents them as two trees. It then uses GumTree [37] to extract tree edit actions that represent the differences between the two trees. Finally, BUILDDIFF maps the edit actions to build change types that are defined in the taxonomy of Macho *et al.* [12]. Table II lists the change types that BUILDDIFF

[7] https://cwiki.apache.org/confluence/display/MAVEN/Errors+and+Solutions

[8] http://goo.gl/Tohyq9

TABLE II
THE BUILDDIFF CHANGE TYPES [12] THAT WE FOCUS ON IN THIS PAPER

Abbr.	Change Type
PAVU	PARENT_VERSION_UPDATE
GPU	GENERAL_PROPERTY_UPDATE
DVU	DEPENDENCY_VERSION_UPDATE
DD	DEPENDENCY_DELETE
MI	MODULE_INSERT
PU	PROFILE_UPDATE
DU	DEPENDENCY_UPDATE
PRVU	PROJECT_VERSION_UPDATE
GPD	GENERAL_PROPERTY_DELETE
GPI	GENERAL_PROPERTY_INSERT
MD	MODULE_DELETE
PI	PLUGIN_INSERT
MDI	MANAGED_DEPENDENCY_INSERT
RD	REPOSITORY_DELETE
MDU	MANAGED_DEPENDENCY_UPDATE
MDVU	MANAGED_DEPENDENCY_VERSION_UPDATE
MU	MODULE_UPDATE
PCU	PLUGIN_CONFIGURATION_UPDATE

extracted during our analysis. In this paper, we focus on those change types for the sake of clarity. We pass each of the revision pairs $\langle r1, r2 \rangle$ of the prior step to BUILDDIFF and save the extracted changes.

Create Data Sets. We collect the changes and the build results from all 125 revision pairs that we were left after applying the filters to the 20 Java open source projects. Since our approach to address RQ2 is based on the results of RQ1, we randomly split the data set into two subsets. We use 30% (37 pairs) of the instances for investigating RQ1 and the remaining 70% (88 pairs) for RQ2.

IV. DEPENDENCY RESOLUTION FIXES

In this section, we investigate how developers have repaired builds that broke because of dependency-related issues. We quantitatively and qualitatively analyze 37 revisions (30%) of our data set. We study the frequency of the change types that were made in build-fixing revisions and which change types were causal for the fix. Based on these frequencies, we derive the repair strategies that we present in Section V. Below, we present the approach and results of the quantitative and qualitative analysis.

A. Quantitative Analysis

Our first investigation is performed using a quantitative analysis of changes in revisions that fix dependency-related issues. Counting the changes per change type, we obtain the results depicted in Figure 2.

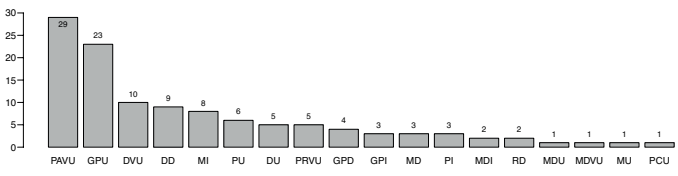

Fig. 2. Frequencies per change type occurring in revisions that fix dependency-related build breakage

The results show that PARENT_VERSION_UPDATE (29) is the most frequently occurring change type for fixing dependency-related build breakage followed by GENERAL_PROPERTY_UPDATE (23), DEPENDENCY_VERSION_UPDATE (10), and DEPENDENCY_DELETE (9).

Since several change types have a similar purpose, we group them into seven categories. For example PARENT_VERSION_UPDATE and DEPENDENCY_VERSION_UPDATE both update a version number and hence, we group them into the category Version Change. Our seven change type categories are:

- **Property Change:** contains all changes to property definitions, such as GENERAL_PROPERTY_UPDATE and GENERAL_PROPERTY_INSERT.
- **Version Change:** contains all changes to version elements in the `pom.xml`, such as DEPENDENCY_VERSION_DELETE and PROJECT_VERSION_UPDATE.
- **Repository Change:** contains all changes to the repository entries, such as REPOSITORY_UPDATE and REPOSITORY_INSERT.
- **Dependency Delete:** contains all changes concerning the deletion of dependencies, such as DEPENDENCY_DELETE and MANAGED_DEPENDENCY_DELETE.
- **Dependency ID Change:** contains all changes that belong to the identification of dependencies, namely the `groupId` and the `artifactId`, such as DEPENDENCY_UPDATE and MANAGED_DEPENDENCY_UPDATE.
- **Dependency Insert:** contains all insertions of dependencies, such as DEPENDENCY_INSERT and MANAGED_DEPENDENCY_INSERT.
- **Others:** contains all change types that were not assigned to any of the categories above.

Our categorization is inspired by the categorization presented by Macho *et al.* [12]. While they group the change types according to their context in the `pom.xml` file, such as all changes of the dependency specification are grouped into the Dependency Changes category, we group the change types according to their purpose. The full change type categorization was validated with a PhD student and can be found in the replication package.[5]

Figure 3 shows the results of counting the changes per change type category. The results are similar to the results obtained from counting the changes per change type. Changes of the category Version Change (45) occurred most frequently in revisions that fix dependency-related build breakage followed by the category Property Change (30) and the category Others (22).

We find that some change types, such as PARENT_VERSION_UPDATE, only occur in a few revisions but when they occur, they occur in several locations at once. This might overemphasize the importance of those change types. To address this possible bias, we also study the change type frequencies per revision. We count the revisions per change type and change type category, meaning that only the number

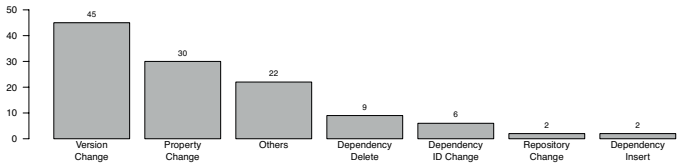

Fig. 3. Frequencies per change type category occurring in revisions that fix dependency-related build breakage

of revisions that a change type occurs in is counted. Figure 4 and Figure 5 show the results of this approach.

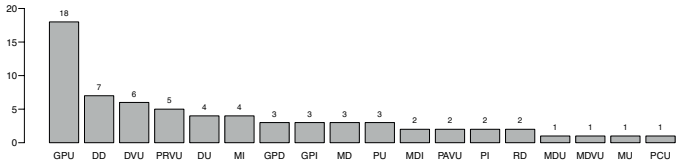

Fig. 4. Number of revisions per change type that fix dependency-related build breakage

As expected, Figure 4 shows a slightly different picture compared to Figure 2. GENERAL_PROPERTY_UPDATE changes occur in 18 of 37 revisions (49%), DEPENDENCY-_DELETE changes occur in 7 of 37 revisions (19%), and DEPENDENCY_VERSION_UPDATE changes in 6 of 37 revisions (16%).

Figure 5 shows the same analysis for our change type categories. Changes of the category Property Change were performed in 24 of the 37 (65%) revisions while changes of the categories Version Change and Others were performed in 14 revisions (38%) each.

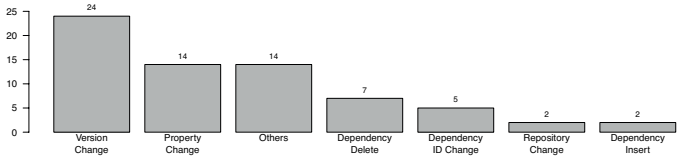

Fig. 5. Number of revisions per change type category that fix dependency-related build breakage

B. Qualitative Analysis

Our quantitative analysis only reports the frequencies of change types in revisions that fix dependency-related build breakage, but does not show which specific change or combination of changes were performed to actually fix the breakage. As stated by Dias *et al.* [38], commits are often tangled, meaning that a commit may not solely consist of changes of a single purpose (*i.e.*, only changes that fix dependency-related build breakage). To identify which changes fixed the dependency-related build breakage, we manually analyze the 37 revisions.

We analyze each revision pair as follows: First, we checkout the parent revision that contains the dependency-related issue. Second, we apply each change (or combination of changes) extracted by BUILDDIFF separately to the parent version and execute the build. For combining the changes, we start with each single change instance and then proceed with the various combinations of changes (first combining two changes, then combining three changes, and so on). If the build is successful after applying a change (or combination of changes), we have identified the change(s) that fix the dependency-related build breakage. If we cannot obtain a successful build after applying all the combinations of build changes, the fix might require other changes, such as changes in the Java source code, that we currently do not support. Note that it is possible that there exist multiple ways to fix the build, *e.g.*, two changes lead to a fix independently.

Figure 6 shows the results of our qualitative analysis for the change type categories considering only the 27/37 revisions that can be fixed with a *single* change. 14/27 revisions fix a dependency-related build breakage with a single change of the category Property Change, followed by changes of the category Dependency Delete that fix 5/27 revisions. The remaining 8 revisions are fixed by the categories Version Change, Repository Change, and Dependency ID Change (3, 3, and 2, respectively).

The fixes contained by the 10/37 revisions that need more than a single change to fix the build breakage break down as follows: 3/10 revisions of the `keycloak` project are fixed with either a single GENERAL_PROPERTY_-UPDATE change or a single REPOSITORY_INSERT change. Finally, 7/10 revisions required at least multiple changes of at least two categories. For instance, one revision of the project `hazelcast` involves 10 changes of 4 different types and 2 change type categories to fix the dependency-related build breakage.

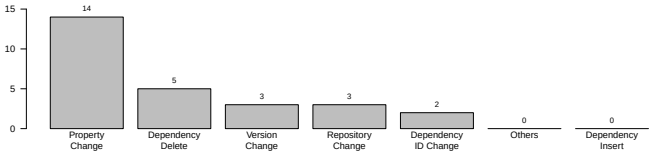

Fig. 6. Number of revisions per change type category that fix a dependency-related build breakage with a single change

We further investigate the changes of the category Property Change in detail because properties in Maven can be used in all parts of the build specification. We observe that all observed changes modify properties that are referenced in version elements of dependencies, thus, they transitively modify the version number of these dependencies. In total, 17 revisions are fixed with a single change of a version number. Furthermore, we make two observations: First, 13 out of the 17 (76%) changes update the version from a snapshot dependency to a regular dependency *e.g.*, by removing the "-SNAPSHOT" suffix. Second, the updated version numbers are usually close to the original version number (*e.g.*, from 0.7.2-mmx4 to 0.7.2).

Another observation resulting from our qualitative analysis is that in 30 out of 37 revisions (81%) the issue can be fixed with only one change. This indicates that the fixes of

dependency-related issues in Maven build specifications are typically small compared to fixes in other areas, such as program repair (*e.g.*, the fixes listed in Weimar *et al.* [39] range from 2-11 lines in terms of size).

Based on the results, we can answer RQ1 as follows:

Developers fix dependency issues in Maven build specifications by updating the version in 17/37 (46%) cases. A common update is to remove the "-SNAPSHOT" suffix to promote the dependency (13/17). Furthermore, 81% of the fixes contain only a single change.

V. BuildMedic Repair Strategies

In this section, we present three strategies to automatically repair dependency-related errors in Maven build files. The strategies are: Version Update, Dependency Delete, and Add Repository. We derive these strategies from the results of our quantitative and qualitative analyses of RQ1 (see Section IV, especially Figure 6). The implementation of other strategies is left to future work because they require additional data. This is because, for example, adding dependencies to libraries requires knowledge of the classes contained in new dependencies, *e.g.*, in the case of a compilation error, one needs to know which libraries could provide the missing classes.

A. Version Update

The first strategy is derived from our finding that dependency-related build breakage is most frequently fixed with Version Update changes. The overall goal of this strategy is to modify the version element or referenced property in a way that the build succeeds. To achieve this goal, the following steps are performed: First, the failing dependency is identified using MLA. Second, the corresponding dependency definition in the Maven build file is located. Third, we determine how the version of the dependency is specified. In Maven, this can be done in four ways:

- **Literal:** The version is literally specified in the version element of the dependency definition.
- **Property:** The version element references a property that is defined in the current or a parent `pom.xml`.
- **Dependency Management:** The version is inherited from the dependency management specification. The current or a parent `pom.xml` defines the dependency in its dependency management area and the current build configuration file only specifies the usage of that dependency.
- **Parent Dependency:** The dependency and its version element are declared in a parent build specification file and inherited from the current `pom.xml`.

Fourth, the correct version of the dependency is determined using the following procedure: First, we query the four popular Maven repositories that we also use for the Add Repository Strategy (see Section V-C) and retrieve all of the candidate versions for the missing dependency. We then remove all candidate versions that have already been used in previous applications of the Version Update strategy to avoid double usage of a single version. If no candidate version for the dependency remains (*i.e.*, all versions have already been tried or there is no version available at all), we finish this strategy without modifying the specification. If we could find at least one available version, we analyze the current version that caused the build to break. If it ends with "`-SNAPSHOT`" or "`-BUILD-SNAPSHOT`", we remove this extension and check whether the candidate versions list contains the resulting version. If it does, we choose that version. If the candidate versions list does not contain that version, we select the version that is most similar to the version of the failing dependency.

We make use of the concept of semantic versioning which is also used by Raemaekers *et al.* [40]. They analyzed the Maven repository and found that most versions follow the pattern "Major.Minor.Patch". Based on that finding, we compute the distance between two versions with the following formula:

$$distance = abs(10000 * (V1_{maj} - V2_{maj}) + 100 * (V1_{min} - V2_{min}) + (V1_{pat} - V2_{pat}))$$

The greater the distance value, the greater the difference between the two versions. We use the factors 10000 and 100, respectively, to impose a greater penalty for a mismatch on a major or a minor version than that of a patch version. These values worked best in a sensitivity analysis. We compute the distance of the current version to each of the candidate versions. The candidate list is then sorted by the following criteria: (1) whether the version was retrieved from the standard Maven repository (versions from this repository first), (2) by the distance (low distances first), and (3) by the date when the version was added to the repository (closer dates to current version first, no dates last). After these steps, we obtain a list of versions sorted by the similarity to the original version. We choose the first element of this list as new candidate version.

Finally, if the failing dependency did not specify any version, we choose the most recent version with respect to the date on which the dependency was added to a repository.

B. Dependency Delete

The results for RQ1 revealed that the second most frequently applied change type to fix dependency-related build breakage is Dependency Delete. Based on this finding, we derive the Dependency Delete strategy that scans all of the `pom.xml` files in the current project and removes all definitions of the dependency that cause the broken build. This ensures that the dependency management system of Maven no longer tries to retrieve this dependency, and thus, fixes the error.

C. Add Repository

Maven repositories can be configured in many ways, such as in the Maven build specification (`pom.xml`) or in the global Maven user settings. In the latter case, the specification might be missing when cloning a repository because the global Maven settings are usually not stored in the source code management system.

To address this issue, we designed the Add Repository strategy that adds the four common Maven repositories to the build specifications. Maven can then use these repositories to

resolve the missing dependencies. The repositories that this strategy adds are:

- The public repository of JBoss (https://repository.jboss.org/nexus/content/repositories/public-jboss/)
- The Spring repository (http://repo.spring.io/libs-milestone/)
- The Atlassian repository (https://maven.atlassian.com/3rdparty/)

Note that the Maven Central repository will always be queried first. The list can easily be adapted by the user. We implement the three strategies in the BUILDMEDIC approach that we present and evaluate in the next sections.

VI. BUILDMEDIC

In this section, we first present BUILDMEDIC, our approach to repair dependency-related build breakage in Maven build specification files. Second, we present an empirical evaluation of BUILDMEDIC on 88 revision pairs to demonstrate its ability to repair such breakage.

A. Approach

The input for BUILDMEDIC is the Maven build log expressing the dependency-related build error, and all of the files of the failing revision (including Java source code files and Maven build specification files). BUILDMEDIC extracts the build information using MLA and generates candidate repair plans. Each candidate is executed by applying a repair strategy and running the build one after another. We consider a candidate to be successful, if the build can be successfully executed (*i.e.*, MLA extracts `SUCCESS` from the resulting build log file). The overall approach is depicted in Figure 7. In the following, we describe the four steps in detail.

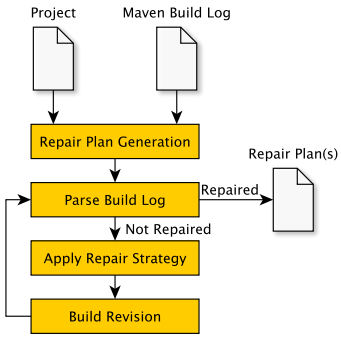

Fig. 7. Overview of the BUILDMEDIC approach

Repair Plan Generation. The first step concerns the generation of candidate repair plans. A repair plan consists of n repair strategies. For this paper, we use $n = 3$ to limit the execution time of the evaluation because as seen in Section IV, fixes are usually small. Combinatorial, we obtain 3^n plans.

However, some candidate repair plans might contain unreasonable steps, such as adding repositories more than once. We optimize the plan generation in two ways. First, we allow the Add Repository strategy to occur only once and only as the first repair action. Second, we sort the repair plans according to their expected ability to fix an issue. Plans starting with Add Repository are placed first. Then, we calculate a weighted sum of the repair plan strategies using $DeleteDependency = 1$ and $VersionUpdate = 2$. We choose these numbers because we found in Section IV that Version Change is the most promising strategy, directly followed by Delete Dependency. Plans with higher sums are ranked first. In the end, we obtain a list of $2^{n-1} + 2^n = 12$ candidate plans for $n = 3$ that are sorted according to their expected ability to repair the build breakage.

Next, we execute the following steps for each repair plan until the build succeeds or no more strategies are available.

Parse Build Log. In this step, BUILDMEDIC parses the most recent build log with MLA. At the start of the repair, the most recent build log is the input build log and during the repairing procedure it is the most recent log that ended in `DEPENDENCY_RESOLUTION_ERROR` because logs of other intermediate results, such as `COMPILATION_FAILED` do not include details about the failing dependency anymore. If the build result is `SUCCESS`, BUILDMEDIC has found a repair plan and the repair actions that have already been performed within this plan are marked as a successful repair candidate. Otherwise, the approach continues.

Apply Fix Strategy. We apply the next repair strategy as specified in the current repair plan. If the current repair plan starts with the same repair strategies as an already known successful repair candidate, BUILDMEDIC skips the current repair plan. For example if we already found that a Version Update repairs the build, we do not need to try any plans that start with the Version Update strategy.

Build Revision. This step executes the Maven build with the modified build specification and saves the build log. We use the same Maven command as described in Section III-B to invoke the build.

The procedure is repeated until a build ends with `SUCCESS` or no more strategies are left in the current plan.

B. Evaluation

We evaluate our approach with the 88 revision pairs of the RQ2 data set depicted in Section III. A preliminary check of the pairs revealed that 8 pairs share 4 parent revisions because the parent revisions were branched. To avoid double counting the repair results of these pairs, we removed 4 pairs with the duplicated parent and finally used 84 pairs for the evaluation. For each revision pair, we checkout the parent revision that fails because of a dependency-related issue. We then apply BUILDMEDIC to repair the build and record the successful repair candidates.

Table III shows the results per project of applying BUILDMEDIC to the 84 revision pairs. BUILDMEDIC could successfully repair 45/84 (54%) studied breakage pairs. Considering projects, BUILDMEDIC could repair 50% or more of the failing builds in 13/19 (68%) of the studied projects. In only 2/19 (11%) projects, BUILDMEDIC could not find a repair candidate for any failing revision. Investigating the number of changes needed to repair a build, we found that in 34/45 (76%) of the successfully repaired revisions only a single change is

TABLE III
RESULTS OF USING BUILDMEDIC TO REPAIR 84 BUILDS IN 19 JAVA OPEN SOURCE PROJECTS; ID AND SIM: IDENTICAL AND SIMILAR REPAIR, RESP.

Project	Fixed	Not Fixed	$n = 1$	ID	SIM
async-http-client	1 (100%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
immutables	1 (100%)	0 (0%)	1 (100%)	1 (100%)	0 (0%)
closure-compiler	1 (100%)	0 (0%)	1 (100%)	1 (100%)	0 (0%)
symphony	4 (57%)	3 (43%)	4 (100%)	2 (50%)	2 (50%)
cucumber-jvm	1 (100%)	0 (0%)	1 (100%)	1 (100%)	0 (0%)
fastjson	1 (100%)	0 (0%)	1 (100%)	1 (100%)	0 (0%)
undertow	1 (50%)	1 (50%)	0 (0%)	0 (0%)	1 (100%)
solo	2 (67%)	1 (33%)	2 (100%)	0 (0%)	2 (100%)
vavr	2 (100%)	0 (0%)	2 (100%)	1 (50%)	1 (50%)
wildfly	0 (0%)	1 (100%)	- (-)	- (-)	- (-)
YCSB	1 (20%)	4 (80%)	1 (100%)	0 (0%)	1 (100%)
retrofit	2 (67%)	1 (33%)	0 (0%)	0 (0%)	1 (50%)
storm	2 (100%)	0 (0%)	2 (100%)	0 (0%)	2 (100%)
jersey	2 (100%)	0 (0%)	2 (100%)	0 (0%)	0 (0%)
keycloak	2 (40%)	3 (60%)	2 (100%)	0 (0%)	0 (0%)
druid	0 (0%)	1 (100%)	- (-)	- (-)	- (-)
alluxio	10 (37%)	17 (63%)	3 (30%)	1 (10%)	7 (70%)
libgdx	5 (100%)	0 (0%)	5 (100%)	1 (20%)	3 (60%)
hazelcast	7 (50%)	7 (50%)	7 (100%)	7 (100%)	0 (0%)
Total	45 (54%)	39 (46%)	34 (76%)	16 (36%)	20 (44%)

needed (see column $n = 1$). This observation confirms the findings of the qualitative analysis in Section IV.

We also evaluated the quality of the successful repair candidates. We compared the successful candidates with the repairs that have been performed by the developers in the repairing revision. We consider three levels: (1) identical (ID) expresses a repair that is exactly the same as in the repository (*e.g.*, a Version Update is performed with the identical new version), (2) similar (SIM) represents that the repair type is the same but details are different (*e.g.*, a different version), and (3) different (DIFF) otherwise. Table III shows the results for ID and SIM per project. We observe that 16/45 (36%) revisions contain at least one repair plan that is identical to the repair by the developers and 20/45 (44%) contain at least one repair that is similar.

Table III shows that 39 out of 84 (46%) revisions could not be automatically repaired by BUILDMEDIC. A manual investigation of these broken builds reveals that:

- The step restriction ($n = 3$ in this study) limited BUILDMEDIC to continue with the repair. A solution could be found if the maximum number of steps is increased.
- The missing dependency was removed from the repositories and needs to be compensated by inserting other dependencies. Adding different dependencies or modifying the ID of the current dependency can fix this issue.
- The fix also requires changes to the source code which are currently not supported by our approach.
- BUILDMEDIC does not have a strategy to fix the build breakage. A solution is to implement new strategies, such as Insert Dependency and Update Dependency IDs. We plan to provide these strategies in future work.

In addition to analyzing the repairs themselves, we also analyze the runtime of BUILDMEDIC. It is important that the fixes can be retrieved within an reasonable amount of time (*e.g.*, Martinez *et al.* [41] set the timeout in their study to three hours). Addressing this, we measure the total time until BUILDMEDIC finishes, and the overhead that BUILDMEDIC adds to the build duration. We consider the overhead as the time of running BUILDMEDIC excluding the time that is need to execute the build. We found that the total time ranges between 4 and 61 minutes with an average of 22.8 minutes. The overhead of BUILDMEDIC ranges between 1.5 and 35 minutes with an average of 8.6 minutes.

With these results, we can answer RQ2 as follows:

> BUILDMEDIC *is able to repair 54% of dependency-related build breakage adding an average overhead of 8.6 minutes. 36% of the repairs are identical to the developer's repair actions. In 76% of the repairs, a single change could repair the breakage.* BUILDMEDIC *runs with an average total time of 22.8 minutes and an average overhead of only 8.6 minutes.*

VII. DISCUSSION

In this section, we discuss the applications and implications of our results on research as well as on development. Furthermore, we discuss the possible threats to validity of our results and how we addressed them.

A. Applications and Implications

For researchers. The answers to RQ1 and RQ2 have several implications on research. First, the results of RQ1 (Section IV) show that most dependency issues are fixed by updating the version of the dependencies. Moreover, the removal of a dependency fixed the breakage in five cases. Furthermore, the results of RQ1 show that 81% of the developer repairs consist of a single change. This demonstrates that the detailed changes that are extracted by BUILDDIFF can be used to analyze Maven build specifications in detail, *e.g.*, for deriving repair strategies. Hence, future research should use BUILDDIFF for the analysis of changes in Maven build specifications. Second, the results of RQ2 (Section VI) show that it is possible to automatically repair many Maven builds that suffer from dependency-related breakage. Compared with other areas of automatic repair, such as program repair, the repair rate is similar. However, Qi *et al.* [42] stated that the well known GenProg approach can generate patches for 55/105 bugs (52%) [29] but only two (4%) were correct. In our case, the results of RQ2 show that 16/45 (36%) of BUILDMEDIC-generated repairs are identical to the developer-performed repair and 20/45 (44%) are similar to the developer-performed repair. We suspect that this is because of two reasons. First, we limit the search space for the repairs by the nature of our repair strategies and second, we do not rely on machine learning but instead derive empirically-informed strategies based on developer-performed repairs. Moreover, our manual analysis of the 39 revisions that could not be repaired by BUILDMEDIC showed that it needs additional strategies to cover other types of build breakage. Future research should address this issue and develop further strategies.

For developers. Our approach for repairing dependency-related build breakage is fully automated by BUILDMEDIC. Developers can use BUILDMEDIC to automatically repair

dependency-related build breakage which typically accounts for 39% to 65% of build breakage [7], [10], [11].

Ideally, BUILDMEDIC can be added as a post-build reaction in the CI environment to suggest repair candidates or to automatically repair Maven builds that broke because of a dependency-related issue. The evaluation of the performance of BUILDMEDIC shows that it is applicable for this job as we observed that BUILDMEDIC only adds an overhead of 8.6 minutes on average.

For smaller projects, BUILDMEDIC can be used as a standalone tool or as a plugin for the integrated development environment. Whenever a build fails, the developer can run BUILDMEDIC to suggest and/or execute a repair candidate to repair the build. Both applications of BUILDMEDIC aim at supporting developers with repairing broken builds in order to be able to stay focused on their current programming tasks.

B. Threats to Validity

Internal Validity. We split the data randomly into two data sets to use one for each research question. The resulting samples might be biased. We mitigate this threat by adding a qualitative analysis of the data. Furthermore, we only study data within a date range (December 31, 2014 to July 13, 2017). This restriction might exclude data that could affect our findings. We argue that this restriction actually supports the validity because it reflects the current changes instead of old changes. In our preliminary analysis, we found that the build results before this threshold are likely to incorrectly flag builds as failing because of environmental changes. Moreover, we perform an analysis of historical data. We cannot analyze rebased or deleted revisions which could bias our results. Another threat to internal validity concerns our observation of fixing changes. We only investigate the changes of the directly preceding revision. We mitigate this threat by analyzing the fixes and identifying the change that caused the fix. Moreover, the repairing revision might be a tangled commit [38] containing changes that are unrelated to the repair. We mitigate this threat by performing a manual investigation to identify unrelated changes. In our data set, we include build results that we retrieved by building each revision in our own stable environment with a single build command. Excluding the tests in the build command makes it possible that the build is successful, but produces a semantically different result. We counter this threat by analyzing the produced repairs and comparing the fixes with the developer-performed repairs.

External Validity. The findings of this study might not generalize to other projects. We mitigate this threat by choosing projects of different sizes, vendors, and purposes. Furthermore, we only consider projects that use Maven to build. Hence, the results might not generalize to other build tools, such as Gradle and Ant. We use a common versioning pattern (MAJOR.MINOR.PATCH) to measure the distance of versions. Other projects may use a different versioning pattern which threatens the generalizability. We decided to use this pattern because it is a common versioning pattern and is also used in prior studies [40].

Construct Validity. We construct our repair strategies based on qualitative observations. This might bias the selection of the strategies because changes that are unrelated to fixes might be counted. We address this threat by qualitatively studying the cause for the fixes to identify the responsible changes. Another threat to construct validity is that we consider a repair to be successful if the build yields `SUCCESS`. However, the build result can be successful without fixing the underlying issue (*e.g.*, deleting dependencies that are only needed at runtime). We counter this threat in two ways. First, we do not only find one repair candidate but we show all possible repair candidates that our approach found. Second, we investigated all repair candidates and compared them to the fixes that developers performed.

VIII. CONCLUSIONS

Build specifications need to be maintained to avoid outdated specifications, and build breakage as a consequence. Recent studies found that dependency-related issues are the main cause of build breakage. In this paper, we analyze 37 builds that were broken due to dependency-related issues and study the changes that were performed to repair the build breakage. Furthermore, we perform a qualitative study of the fixing revisions to identify the change or the changes that caused the fix. Using this knowledge, we derive three strategies based on frequently occurring repair types. We then propose BUILDMEDIC, an approach to automatically fix dependency-related build breakage. We evaluate BUILDMEDIC on 84 builds that were broken because of dependency-related issues. More specifically, our study answers the research questions as follows:

(RQ1) Developers often (46%) repair dependency-related build breakage with changes to versions (*e.g.*, removing the "-SNAPSHOT" postfix) and dependencies themselves. 81% of the fixes contain only a single change. Furthermore, property changes are changes to version identifiers in all of the studied revisions.

(RQ2) BUILDMEDIC fixes 54% of dependency-related build breakage, which generates an overhead of 8.6 minutes on average. 36% of repairs are identical to developer-performed repairs and 76% of the repairs only consist of a single step. BUILDMEDIC runs with an average total time of 22.8 minutes and an average overhead of only 8.6 minutes

Future work. We plan to extend BUILDMEDIC with further strategies to cover other types of build breakage, such as `COMPILATION_FAILED`, and to cover breakage that BUILDMEDIC currently cannot repair. Furthermore, we will improve the fix time by investigating models to predict the best next strategy. Concerning dependency issues, we plan to extend this work by implementing other fixing strategies, such as adding dependencies. We also aim at extending the study to other projects, in particular to projects from industry. Furthermore, we plan to integrate BUILDMEDIC directly into Maven (via a plug-in) to immediately invoke the repair, once an issue is detected.

REFERENCES

[1] B. Vasilescu, S. van Schuylenburg, J. Wulms, A. Serebrenik, and M. G. J. van den Brand, "Continuous integration in a social-coding world: Empirical evidence from github," in *Proceedings of the International Conference on Software Maintenance and Evolution*. IEEE, 2014, pp. 401–405.

[2] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig, "Usage, costs, and benefits of continuous integration in open-source projects," in *Proceedings of the International Conference on Automated Software Engineering*, 2016, pp. 426–437.

[3] B. Vasilescu, Y. Yu, H. Wang, P. Devanbu, and V. Filkov, "Quality and productivity outcomes relating to continuous integration in github," in *Proceedings of the Joint Meeting on Foundations of Software Engineering*. ACM, 2015, pp. 805–816.

[4] B. Adams and S. McIntosh, "Modern Release Engineering in a Nutshell: Why Researchers should Care," in *Proceedings of the International Conference on Software Analysis, Evolution, and Reengineering*, 2016, pp. 78–90.

[5] M. Hilton, N. Nelson, T. Tunnell, D. Marinov, and D. Dig, "Trade-offs in continuous integration: assurance, security, and flexibility," in *Proceedings of the Joint Meeting on Foundations of Software Engineering*. ACM, 2017, pp. 197–207.

[6] 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.

[7] 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.

[8] 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.

[9] M. Beller, G. Gousios, and A. Zaidman, "Oops, my tests broke the build: An analysis of travis ci builds with github," PeerJ Preprints, Tech. Rep., 2016.

[10] M. Sulír and J. Porubän, "A quantitative study of Java software buildability," in *Proceedings of the International Workshop on Evaluation and Usability of Programming Languages and Tools*, 2016, pp. 17–25.

[11] M. Tufano, F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, "There and back again: Can you compile that snapshot?" *Journal of Software: Evolution and Process*, vol. 29, no. 4, 2017.

[12] C. Macho, S. McIntosh, and M. Pinzger, "Extracting Build Changes with BuildDiff," in *Proceedings of the International Conference on Mining Software Repositories*, 2017, pp. 368–378.

[13] 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.

[14] 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.

[15] R. Hardt and E. V. Munson, "Ant build maintenance with formiga," in *Proceedings of the International Workshop on Release Engineering*. IEEE, 2013, pp. 13–16.

[16] ——, "An empirical evaluation of ant build maintenance using formiga," in *Proceedings of the International Conference on Software Maintenance and Evolution*, 2015, pp. 201–210.

[17] 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.

[18] A. Tamrawi, H. A. Nguyen, H. V. Nguyen, and T. N. Nguyen, "Build code analysis with symbolic evaluation," in *Proceedings of the International Conference on Software Engineering*. IEEE, 2012, pp. 650–660.

[19] 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 *Proceedings of the International Conference on Software Maintenance*. IEEE, 2012, pp. 150–159.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] S. Forrest, T. Nguyen, W. Weimer, and C. Le Goues, "A genetic programming approach to automated software repair," in *Proceedings of the Annual conference on Genetic and evolutionary computation*. ACM, 2009, pp. 947–954.

[26] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *Proceedings of the International Conference on Software Engineering*. IEEE, 2009, pp. 364–374.

[27] E. Fast, C. Le Goues, S. Forrest, and W. Weimer, "Designing better fitness functions for automated program repair," in *Proceedings of the Annual Conference on Genetic and evolutionary computation*. ACM, 2010, pp. 965–972.

[28] C. Le Goues, T. V. Nguyen, S. Forrest, and W. Weimer, "GenProg: A generic method for automatic software repair," *Transactions on Software Engineering*, vol. 38, no. 1, pp. 54–72, 2012.

[29] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each," in *Proceedings of the International Conference on Software Engineering*. IEEE, 2012, pp. 3–13.

[30] X. B. D. Le, D. Lo, and C. Le Goues, "History driven program repair," in *International Conference on Software Analysis, Evolution, and Reengineering*. IEEE, 2016, pp. 213–224.

[31] M. Martinez, T. Durieux, R. Sommerard, J. Xuan, and M. Monperrus, "Automatic repair of real bugs in java: A large-scale experiment on the defects4j dataset," *Empirical Software Engineering*, vol. 22, no. 4, pp. 1936–1964, 2017.

[32] C. Marsavina, D. Romano, and A. Zaidman, "Studying fine-grained co-evolution patterns of production and test code," in *Proceedings of the International Working Conference on Source Code Analysis and Manipulation*. IEEE, 2014, pp. 195–204.

[33] E. Bouwers, J. P. Correia, A. van Deursen, and J. Visser, "Quantifying the analyzability of software architectures," in *Proceedings of the Working Conference on Software Architecture*. IEEE, 2011, pp. 83–92.

[34] M. Beller, G. Gousios, and A. Zaidman, "Travistorrent: Synthesizing travis ci and github for full-stack research on continuous integration," in *Proceedings of the International Conference on Mining Software Repositories*. IEEE, 2017, pp. 447–450.

[35] M. Zolfagharinia, B. Adams, and Y.-G. Guéhéneuc, "Do not trust build results at face value: an empirical study of 30 million cpan builds," in *Proceedings of the International Conference on Mining Software Repositories*. IEEE, 2017, pp. 312–322.

[36] C. Vassallo, G. Schermann, F. Zampetti, D. Romano, P. Leitner, A. Zaidman, M. Di Penta, and S. Panichella, "A tale of ci build failures: an open source and a financial organization perspective," in *International Conference on Software Maintenance and Evolution*. IEEE, 2017, pp. 183–193.

[37] 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.

[38] M. Dias, A. Bacchelli, G. Gousios, D. Cassou, and S. Ducasse, "Untangling fine-grained code changes," in *International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 2015, pp. 341–350.

[39] W. Weimer, S. Forrest, C. Le Goues, and T. Nguyen, "Automatic program repair with evolutionary computation," *Communications of the ACM*, vol. 53, no. 5, pp. 109–116, 2010.

[40] S. Raemaekers, A. V. Deursen, and J. Visser, "Semantic Versioning versus Breaking Changes : A Study of the Maven Repository," in

Proceedings of International Working Conference on Source Code Analysis and Manipulation, 2014, pp. 215–224.

[41] M. Martinez, T. Durieux, J. Xuan, R. Sommerard, and M. Monperrus, "Automatic repair of real bugs: An experience report on the defects4j dataset," *arXiv preprint arXiv:1505.07002*, 2015.

[42] Z. Qi, F. Long, S. Achour, and M. Rinard, "An analysis of patch plausibility and correctness for generate-and-validate patch generation systems," in *Proceedings of the International Symposium on Software Testing and Analysis*. ACM, 2015, pp. 24–36.