Tackling Build Failures in Continuous Integration

Foyzul Hassan
University of Texas at San Antonio, San Antonio, USA
Homepage: http://foyzulhassan.github.io/
Advisor: Xiaoyin Wang
Email: foyzul.hassan@my.utsa.edu

***Abstract*—In popular continuous integration(CI) practice, coding is followed by building, integration and system testing, pre-release inspection, and deploying artifacts. This can reduce integration risk and speed up the development process. But large number of CI build failures may interrupt the normal software development process. So, the failures need to be analyzed and fixed quickly. Although various automated program repair techniques have great potential to resolve software failures, the existing techniques mostly focus on repairing source code. So, those techniques cannot directly help resolve software build failures. Apart from that, a special challenge to fix build failures in CI environment is that the failures are often involved with both source code and build scripts. This paper outlines promising preliminary work towards automatic build repair in CI environment that involves both source code and build script. As the first step, we conducted an empirical study on software build failures and build fix patterns. Based on the findings of the empirical study, we developed an approach that can automatically fix build errors involving build scripts. We plan to extend this repair approach considering both source code and build script. Moreover, we plan to quantify our automatic fixes by user study and comparison between fixes generated by our approach and actual fixes.**

***Index Terms*—build failures, build repair, continuous integration.**

I. Introduction

Due to ever-increasing nature of software requirements and rigorous release practice, Continuous Integration (CI) [1] is gaining more and more popularity. In CI environment, developers merge code in the code base, followed by the automatic software build process, test execution, and artifact deployment. This practice allows developers to detect more software faults earlier and also request developers to resolve the detected faults in a timely manner to make the release pipeline flawless. A study on Google CI [2] finds that more than 5,500 code commits are merged to the codebase and 100 million test cases are executed per day for validation.

Within CI environment, dedicated infrastructure with different build systems such as Make, Ant, Maven, Gradle, Bazel, etc. are used to automate CI tasks like compilation, test invocation, etc. For continuous integration, developers describe the build process through build scripts such as “build.xml” for Ant, “pom.xml” for Maven, and “build.gradle” for Gradle. But with the growing functionality and CI requirement, build scripts can be very complex and may require frequent maintenance [3]. Our study [4] on TravisTorrent dataset [5] finds that around 29% of CI trials fail. Seo et al. [6] also addressed a similar issue with 37% build failure proportion at Google CI environment. Rausch et al.[7] finds that dependency resolution, compilation, and configuration phases account for 22% of CI failures. On the other hand, 65% failures are categorized as test failures and 13% as quality checking errors. Our study on 1,187 CI failures on TravisTorrent Dataset also confirms that 10.8% of CI-failure fix contains only build script revisions, and 25.6% CI-failure contains both build script and source code.

On the opportunity side, the scenario of CI provides rich code commit history and build logs from previous passing builds and current failing builds. Such information sources are often not available in other application scenarios of repair. But existing program repair techniques cannot be directly applied to build failures. First, build-script repair often involves open knowledge that does not exist in the current project, such as adding a new dependency or incorporate static analysis. Second, build-script repair involves build domain knowledge on CI, build management tools and external tools. Third, many build failures are due to environment issues and might require a fix in multiple types of files such as source code, configuration files, etc.

In this paper, we describe the systematic categorization of CI build failures and techniques to avoid and repair build failures. Through our research work, we tried to answer the following research questions:

- ***(RQ1)*** How build failures can be classified to different categories?
- ***(RQ2)*** Can we predict CI build failures based on commit change analysis?
- ***(RQ3)*** To what extent we can automatically fix build configuration files?
- ***(RQ4)*** Is it feasible to localize and repair faults in CI environment involving heterogeneous resources?

II. Related Works

A. Study on Build Failures

On the study of building errors, Hyunmin et al. [6] performed an empirical study to categorize build errors at Google. While Sulír et al. [7] applied text analysis techniques to categorize build failures. McIntosh et al. [8] found interesting insight that for modern build systems build maintenance effort on external dependency is higher than internal dependency management. These studies mainly focus on build failure; but for automatic build failure resolution, we need to have a detailed study of failure root cause and feasibility analysis of the automatic resolution of build failures.

B. Automatic Program Repair

In recent years automatic program repair techniques have been active research to reduce bug fixing effort. Le Goues et al. [9] proposed GenProg, which is one of the earliest and promising genetic programming based program repair approach. Later D. Kim proposed PAR [10] to generate fix candidates based on templates generated from human fixes. Prophet [11] used a probabilistic model learned from human-written patches to generate a new patch. Nopol [12] proposes a test-suite based repair technique using SMT solvers. These repair techniques can be applied to source related bug fixes. But due to different functionality and domain, these techniques cannot be applied directly to build repair.

III. Research Approach

Research in the area can be roughly divided into two broad categories: 1) Empirical analysis of build failures, and 2) Toolset preparation to overcome the challenges of fixing build failures. Our RQ1 falls into the first category and the rest RQs fall into the second category. In the following, we describe how we answer each research question in detail.

A. Analysis of Build Failure Taxonomy and Resolution Feasibility (RQ1)

Advanced build systems like Maven, Gradle, etc. provide functionality for faster compilation, integration, and testing in CI environment and also in a standalone environment. Despite these functionalities, there are still difficulties to maintain the build configuration management and fixing issues. To answer RQ1, we plan to perform empirical studies from three aspects: i) build failure taxonomy, ii) failure analysis from supporting files, and iii) feasibility of automatic fix generation.

Build Failure Taxonomy: For build failure taxonomy generation, we execute build process on top Java projects maintained by Ant, Maven,, and Gradle build management systems. With default build command we build these projects to find out failure cases. Based on build failure types, we classify the build failures into three general categories: environment issues, process issues, and project issues. Environment issues are build failure causes due to change of build environment like operating system dependency, third party dependency issue, etc. From our identified environment issues, we divide the failures into three sub-categories: platform version issues, dependency issues, and external tools issues. Apart from environment issues, we will also identify failures related to process issues and project issues. Process Issues are build failures caused by the requirement of additional steps in the building process like required file setup and non-trivial build command execution. Project issues are build failures caused by defects in the project itself. These defects can either be code defects that prevent the software from being compiled anyway, or generalization defects that prevent the software from being built on another machine.

Failure Analysis From Supporting Files: After identifying build failure taxonomy, we measure for how many build failures the root causes are explicitly mentioned supporting files: build log file and readme file [13]. With expression-based string analysis we can answer how many build failure causes we can identify from these two types of supporting files.

Feasibility of Automatic Fix Generation: Based on build failure taxonomy and fail analysis from supporting files, we perform a feasibility study on build failure resolve techniques. To resolve the build failures we can apply natural language processing techniques to identify build steps and the process to generate proper builds. Apart from that, version reverting based on commit history can be applied to overcome platform issues.

B. CI Build Failure Prediction Model(RQ2)

With the growing popularity of data-driven approach, researchers have been working on defect prediction models for testing and quality assurance. Xia et al. [14] proposed build a co-change prediction model to avoid build breakage. While Bird and Zimmerman [15] an overview of software build error prediction model, but they did not provide detailed analysis. These works are based on the isolated development work environment. CI build environment is different from than isolated environment due to distributed system in nature and a large number of code change in tandem by different developers. Prior work [2] on build task decomposition reports that on average, the Google code repository receives over 5,500 code commits per day and executes 100 million test cases to validate code changes. So, a single build breakage can lead delay to integrate new feature or bug fix and requires another code commit. For build prediction model in CI environment we use TravisTorrent [5] dataset. TravisTorrent dataset provides build execution information of TravisCI and also provides commit change data. As part of the feasibility study of CI build prediction model, we perform analysis of commit frequency, build execution frequency and build execution duration to find out whether build failure is creating a bottleneck in CI environment. Based on the analysis, we use build commit information, build error type and code change metric to predict build outcome prediction in CI environment. Since build process involves linking of class or object files, we also apply AST level code change analysis to generate a better model that can predict build outcome considering recent code changes.

C. Build Configuration File Repair(RQ3)

Automatic program repair techniques have been active research for many years. Most of the research works focus on source code repair. We planed to extend program repair work in the area of the build configuration file. But build script repair has challenges over source code repair due to build specific domain issues like dependency management, plug-in management, etc. Apart from that build script fault localization, fix pattern generation and patch validation can be different due to the functionality of build scripts. We divide this research problem into three subcategories: i) Build Fix Template Generation, ii) Patch Candidate Generation and iii) Patch Evaluation.

Build Fix Template Generation: To repair build failures we need have to fix templates that can be used to resolve failures. To get fix templates, we planned to apply history driven technique. For a given build failure we planned to perform failed log similarity to identify which failures are similar in nature and with similar build failures, we can extract AST level build script change to identify how developer fixed those issue. With AST level change history, we generate fix templates with node-level modification and hierarchical relationship of code changes.

Patch Candidate Generation: After generating fix templates, we need to localize which build script caused the failure. We deploy lightweight fault localization based on text similarity of failed text in build log and gradle build scripts. Based on lightweight fault localization result, we try to fit the fix templates and this will give has concrete patch candidates. To improve repair time performance we also merge multiple patch candidate and prioritize the patch candidates based on probability of how many cases a patch candidate can be applied in a script. Once patch candidate generation and the ranking process will compete, we can apply those patch candidates.

Patch Evaluation: Automatic program repair patch validation mostly depends on test case execution result. But for build repair, we might not have a test case. So, we leverage the technique of build the repair validation process. After applying each patch candidate, we validate two conditions: i) all the source code files are converted to object file or class file and ii) there is no error message in the log file. For a given candidate patch if the validation conditions are successful, we will consider that the patch is correct. Otherwise, we will try with another candidate patch to fix the failure. We also provide a time threshold to ensure that our approach can generate build fixes within a justifiable time limit. If it takes more time than the threshold time then we will consider than our proposed approach may not generate build fix in a timely manner.

D. Fault Localization of Test and Build Failures in CI Environment(RQ4)

Existing fault localization (FL) techniques either rely on bug reports or test failures to locate bugs in source code [16] [17]. Apart from that, current fault localization techniques are based on single-source type and find fault in a single type of files (i.e., source code). But in CI build result depends on build script, source code, configuration files, data files, etc. FL techniques are even more complex if the application developed in multi-language code like a mix of C and Java. To overcome this challenge, we plan to develop a technique that will be more generic in nature to find faults in CI environment and this will also help us to apply build repair technique in a more broader way. For FL we plan to use both IR-based technique and spectrum based technique to find out fault location.

IR-Based FL: Existing IR-based techniques use bug reports as query and source code as documents. But for build fault localization we do not have bug report and build logs are very lengthy in size to use as search query. Apart from that, we want to apply FL technique to heterogeneous file types. To overcome these challenges, we plan to apply both query optimization and search space optimization. Query optimization will find the critical error text from build log files to identify the exact search query. Using static build dependency analysis we also plan to reduce IR search space so that we can find the fault localization in an effective manner. We also plan to AST level code entities to reduce terms in search documents.

Spectrum-Based FL: IR-Based FL has a limitation of not considering run time execution information. So, we also plan to explore Spectrum-base FL techniques to build failures. Existing Spectrum-base fault localization techniques depend on test case coverage to identify fault covered instruction. Since build executions may have may and may not have test cases, we cannot directly apply existing Spectrum-based fault localization techniques. We plan to apply dynamic instrumentation techniques to identify which source file or configuration files are involved for the error. Based on the instrumented execution trace we will identify potential fault locations.

IV. Preliminary Results and Contributions

To evaluate **RQ1** we used the default build commands on 200 Java projects from GitHub. Among these 200 projects, 86 projects failed with the default build attempt. We manually examined correct build sequences to build each of these projects. Five projects had more than one failure. In total, we had 91 failures to analyze. We present a detailed build failure taxonomy from these build results. Among these 91 build failures we identified that 31 failures are due to Environment Issues, 46 failures are for Process Issue and project issue creates 14 failures. We also performed an analysis to resolve these failures. Our analysis showed that at least 57% of build failures can be automatically resolved.

As part of **RQ2**, we developed a build prediction model that uses TravisTorrent data set with build error log clustering and AST level code change modification data to predict whether a build will be successful or not without attempting actual build so that the developer can get early build outcome result. With the proposed model we can predict build outcome with an average F-Measure over 87% on all three build systems (Ant, Maven, Gradle) under the cross-project prediction scenario.

Our work HireBuild (History-Driven Repair of Build) Script on automatic repair of build scripts tried to solve **RQ3** research problem. From TravisTorrent dataset, we extracted 175 build failures for our analysis. Among these 175 build failures, we used the 135 earlier build fixes for automatic fix-pattern generation and the more recent 40 build failures (fixes) for evaluation of our approach. Our experiment shows that our approach can fix 11 of 24 reproducible build failures, or 45% of the reproducible build failures, within a comparable time of manual fixes. This work purely based on build scripts and to handle more complex build failures we need to address

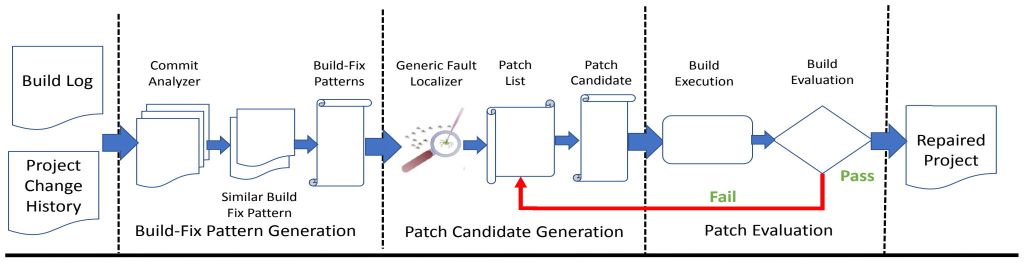

Fig. 1: Overview of our planned approach

research challenges mentioned at **RQ4**. We are working on the challenges of RQ4. Apart from heterogeneous build error fix, we also planned to extend build fix data and also planned to perform a qualitative analysis of generated fix patches.

As part of the research contribution, we made RQ2 and RQ3 data set publicly available. We are also working to make a repair tool available once our proposed plan on handling build failures involving source code and build script will be handled and evaluated. Figure 1 shows an overview of our planned approach that can generate a build failure patch for more complex build failures with heterogeneous file type code changes.

V. Prior Publications

My prior papers, organized by conference, are published in ESEM 2017 (feasibility study of automatic building [18] and prediction of build stalls [19], ICSE Poster Track 2017 (mining readme files to guide software build [20]), and ICSE 2018 (automatic build repair [21])

ACKNOWLEDGEMENTS. Special thanks to my Ph.D. advisor Xiaoyin Wang for his guidance on this research. This work was supported by the NSF Awards CCF-1464425, CNS-1748109 and CCF-1846467.

References

[1] P. M. Duvall, S. Matyas, and A. Glover, *Continuous integration: improving software quality and reducing risk*. Pearson Education, 2007.

[2] M. Vakilian, R. Sauciuc, J. D. Morgenthaler, and V. Mirrokni, "Automated decomposition of build targets," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 123–133. [Online]. Available: http://dl.acm.org/citation.cfm?id=2818754.2818772

[3] S. McIntosh, B. Adams, T. H. Nguyen, Y. Kamei, and A. E. Hassan, "An empirical study of build maintenance effort," in *Software Engineering (ICSE), 2011 33rd International Conference on*. IEEE, 2011, pp. 141–150.

[4] F. Hassan and X. Wang, "Hirebuild: An automatic approach to history-driven repair of build scripts," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: ACM, 2018, pp. 1078–1089. [Online]. Available: http://doi.acm.org/10.1145/3180155.3180181

[5] M. Beller, G. Gousios, and A. Zaidman, "Travistorrent: Synthesizing travis ci and github for full-stack research on continuous integration," in *Proceedings of the 14th working conference on mining software repositories*, 2017.

[6] H. Seo, C. Sadowski, S. Elbaum, E. Aftandilian, and R. Bowdidge, "Programmers' build errors: A case study (at google)," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 724–734. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568255

[7] M. Sulír and J. Porubän, "A quantitative study of java software buildability," in *Proceedings of the 7th International Workshop on Evaluation and Usability of Programming Languages and Tools*, ser. PLATEAU 2016. New York, NY, USA: ACM, 2016, pp. 17–25. [Online]. Available: http://doi.acm.org/10.1145/3001878.3001882

[8] 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.

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

[10] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic Patch Generation Learned from Human-written Patches," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 802–811.

[11] F. Long and M. Rinard, "Automatic patch generation by learning correct code," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '16. New York, NY, USA: ACM, 2016, pp. 298–312. [Online]. Available: http://doi.acm.org/10.1145/2837614.2837617

[12] J. Xuan, M. Martinez, F. DeMarco, M. ClÃľment, S. L. Marcote, T. Durieux, D. Le Berre, and M. Monperrus, "Nopol: Automatic repair of conditional statement bugs in java programs," *IEEE Transactions on Software Engineering*, vol. 43, no. 1, pp. 34–55, Jan 2017.

[13] F. Hassan and Xiaoyin Wang, "Mining readme files to support automatic building of java projects in software repositories," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, May 2017, pp. 277–279.

[14] X. Xia, D. Lo, S. McIntosh, E. Shihab, and A. E. Hassan, "Cross-project build co-change prediction," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, March 2015, pp. 311–320.

[15] C. Bird and T. Zimmermann, "Predicting software build errors," Feb. 20 2014, uS Patent App. 13/589,180. [Online]. Available: http://www.google.ch/patents/US20140053135

[16] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '05. New York, NY, USA: ACM, 2005, pp. 273–282. [Online]. Available: http://doi.acm.org/10.1145/1101908.1101949

[17] J. Zhou, H. Zhang, and D. Lo, "Where should the bugs be fixed? - more accurate information retrieval-based bug localization based on bug reports," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 14–24. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337223.2337226

[18] F. Hassan, S. Mostafa, E. S. L. Lam, and X. Wang, "Automatic building of java projects in software repositories: A study on feasibility and challenges," in *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, Nov 2017, pp. 38–47.

[19] F. Hassan and X. Wang, "Change-aware build prediction model for stall avoidance in continuous integration," in *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, Nov 2017, pp. 157–162.

[20] F. Hassan and Xiaoyin Wang, "Mining readme files to support automatic building of java projects in software repositories," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, May 2017, pp. 277–279.

[21] F. Hassan and X. Wang, "Hirebuild: An automatic approach to history-driven repair of build scripts," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, May 2018, pp. 1078–1089.