BuildEZ

Nehal Agarwal, Prerit Auti, and Antara Bhowmick

University of California, Davis
{naagarwal, pauti, abhowmick}@ucdavis.edu

ABSTRACT

Building is an essential step in the software development process. However, programmers frequently face build failures that reduce productivity and slow down the development process. Analyzing these errors can help build tools to automate the process of resolving them. In this paper, we analyze and categorize build failures of 100 open source Java projects by examining artifacts from the BugSwarm dataset. We select the most common category (dependency errors) and build scripts to automate the process of resolving the failures. We also evaluate these fixes on a set of new image-tags.

1. INTRODUCTION

Building is a process of compiling, linking and assembling the source code files into an executable form to run on the computer. This process is complex since it can involve multiple source code files (possibly in multiple languages), external dependencies in forms of libraries and platforms, make files and scripts to guide the process etc. With increasing popularity of open source, developers all around the world collaborate on projects using version control systems like Git, apache subversion etc. With multiple developers on board, issues in consistency and merge conflicts can increase chances of build failures.

According to a study conducted in [7], which examines over 26.6 million builds on projects across C++ and Java, the median percentage of build failures was 38.4% in C++ and 28.5% in Java. With such high rates of build failures, developers have to step away from the task at hand and dedicate extra time and resources to fix these errors. This reduces their productivity thereby hampering the development process. Sometimes the fixes are trivial that can be fixed without the intervention of the developer.

A better understanding of the causes of these frequent build errors can help develop tools that automate the process of resolving these errors. In this paper, we propose a study on the build failures of 100 open source Java projects by examining artifacts from the BugSwarm dataset. BugSwarm is a benchmark set for real world bugs and bug fixes. We analyzed these build errors and categorized them into seven categories. We further picked the category of dependency error and developed tools to automate the process of fixing these errors.

This paper is organized as follows. In section 2, we describe the motivating example behind this study. In section 3, we present in detail, our technical approach. Section 3.1 talks about data collection, 3.2 about data analysis and 3.3 about how we automated the process. In section 4 we present results of our analysis and tools and Section 5 talks about related work.

2. MOTIVATING EXAMPLES

While analyzing the dependency error category, we realized that even though the error messages from failed built were long and cryptic, most of the fixes were just minor changes in the POM file or package-info.java file that contain package level documentation. The probable fixes were hinted directly in the first of the many error messages printed. On discovering this pattern, we decided to go ahead and automate the process of fixing dependency errors. Some of the examples that helped us uncover this pattern are as follows:

1. SNAPSHOT errors: While building the projects, SNAPSHOT version of the dependencies could not be collected and hence the only way to resolve the error was to build the project using the non SNAPSHOT version of the dependency.

2. Version errors: The versions of the dependencies being used to build were not compatible and hence the fix was to change the version to a compatible one. In most cases, the error message specified the detected and desired version.

3. Unresolvable Dependencies: Builds failed because certain dependencies could not be collected and the way to resolve the error was to eliminate those dependencies from the build files.

3. TECHNICAL APPROACH

Our goal was to develop a system that can automatically resolve certain configuration errors for erroneous builds, thus saving developers' time in building and testing. In this section, we discuss in detail how we approached this problem. We performed the following high-level steps:

1. Build an initial dataset of failed builds and collect metadata about the errors.

2. Analyze the errors collected and organize them into classes. Use this data to choose classes for the next step.

3. For each selected class, identify the steps required to fix the error and automate the process

4. Test the fix for all the artifacts belonging to each class. (More details are provided in Section 4)

3.1 Building the Dataset

3.1.1 Data Collection

Since we had a limited amount of time and resources, we chose to restrict our data to Java projects. We used the BugSwarm [6] project which is a benchmark set of real-world failures as our base dataset. More specifically, we filtered the BugSwarm dataset browser to find artifacts which were in Java, were error-pass and had a stability score of 5/5. Error-pass here refers to the fact that they involved commits whose Travis builds failed, and then passed. This ensured that we were dealing with errors that could be fixed (and likely were fixed in the next commit) making it easier to find the fixes and automate the process. We obtained a list of 100 artifacts from BugSwarm that fulfilled all the requirements.

3.1.2 Building Artifacts

BugSwarm provides us with the Docker[3] image tag for each artifact, containing the passing and failed log and builds, along with metadata about the project. Since it's very time-consuming to do this for every artifact individually, we wrote a script that ran the Docker containers for each image-tag, and extracted all the relevant information into the local system.

We extracted the metadata that BugSwarm generates about the project using the bugswarm show command. From those we obtained the commit hashes for the failed and passed builds and generated the Github compare link. We also searched the build logs of both for errors and compared them. After running this script for all 100 artifacts, we populated a spreadsheet with the fore-mentioned details

3.2 Data Analysis

After we extracted the basic information about each artifact into a spreadsheet, we needed to manually go through the details to find the exact cause of the failed build and attempted to classify the error. We examined the error log for each artifact and compared it to the difference between the source code of the failed and subsequent passing build to identify the fix, and used this information to categorize the artifacts.

Many of the failures were due to easily fixable errors like wrongly named files, not adhering to camelcase, spelling errors in imported module-names. Some were slightly more complex with changes in versions of dependencies and compatibility with newer releases. Often the builds failed due to an error in the basic logic of the code which couldn't have been corrected by someone unfamiliar with the project, and hence couldn't be automated.

We categorized the errors into the following 7 categories:

- Code Errors
- Filename Change
- Buildconfig Error
- Dependency Error
- Quality Error
- Multiple Errors
- License Errors

3.3 Fixing the Errors by Automation

3.3.1 Automating the Process

The automation process will vary for each category and type of error, and may even differ for subcategories.

As an example, in case of filename-mismatch errors, the fix is to change the class name (or file/package name) wherever it is referenced in the project. One solution would be:

Algorithm 1 Fixing Filename Mismatch Error

```
1: procedure FIX_BUILD(ARTIFACT, ERROR_FILE)
2:     wrongClass ← incorrect class name from ERROR_FILE
3:     correctClass ← correct class name from ERROR_FILE
4:     buildStatus ← False
5:     while buildStatus ≠ True do
6:         for wrongClass in ARTIFACT do
7:             wrongClass ← correctClass
8:         buildStatus ← build(ARTIFACT)
9:     return buildStatus
```

For our project, we chose the dependency-error category for automation. This category had multiple sub-categories, each with a different resolution, so we took the following general approach, given an image-tag with a failed build (referred to as ARTIFACT in Algorithm 2).

Algorithm 2 General Driver Function

```
1: procedure FIX_BUILD(ARTIFACT, ERROR_FILE)
2:     buildStatus ← False
3:     for line in ERROR_FILE do
4:         for error_type in ERROR_TYPES do
5:             if match(error_type, line) then
6:                 fix_error_type(ARTIFACT)
7:                 buildStatus ← build(ARTIFACT)
8:             if buildStatus == True then
9:                 break
10:    return buildStatus
```

First, a copy of the failed build is made so that any changes made can be removed if we aren't able to fix the build. In Algorithm 2, ARTIFACT refers to the failed build and ERROR_FILE is the log generated on running the failed build. To further simplify the process (since log files are often over 5k lines), we extracted only the error messages from the log file and provided them as input to the main function. The program tries to detect which kind of error is mentioned in the log file. If a match is found (among the existing error types that have fixes), then the fix is applied to the ARTIFACT and it's re-built. If the build passes, it breaks out of the program or else it continues testing for other errors. This ensures that the artifact is tested for all errors, until the build passes. If no fixes are found it reverts to the original copy of the build.

3.3.2 Categories of Errors Automated

We automated fixes for the following kinds of errors:

1. **Version errors**: One common error is that the versions used in the file need to be updated to the next. (Eg. VERSION 2.1.0 to VERSION 2.2.0) Since the solution is often to shift to the next version, we replace the line in the file to reflect that.

   For example, image-tag: Adobe-Consulting-Services-acs-aem-commons-266576832

Error: @aQute.bnd.annotation.Version("1.**1**.0")

Fix : @aQute.bnd.annotation.Version("1.**2**.0")

The fix was to simply update the version. In cases where the version has trailing zeroes, we attempt to update the right-most non-zero digit. If that fails, we attempt to update the zeroes by one.

2. **Snapshot errors**: A Maven Snapshot indicates that the version hasn't been released. A very common error is that developers often forget to change dependencies from snapshot versions to the release versions; the fix is to simply remove "snapshot" from the relevant files.

   For instance, image-tag: charite-jannovar-78561983

   Error:
   <guava.version>20.0-**SNAPSHOT**</guava.version>

   Fix : <guava.version>20.0</guava.version>

3. **Import Errors/Unresolved Dependencies**: Another common dependency error is that the import for a particular package fails. This can be due to a variety of reasons (change in package name, package no longer existing, etc) but since we are operating without any familiarity with the code-base, we attempt to fix it by removing the import statement. For instance, image-tag: junkdog-artemis-odb-72035852

   Error:

   import com.sun.xml.internal.ws.api.wsdl.parser .MetaDataResolver;

   Fix : removing the import statement

   Clearly, if the error were due to some logic in code or naming error we wouldn't be able to fix it. In case this fix doesn't change the build status, we abort the method and mark it as non-fixable.

3.3.3 Challenges

For the remaining dependency errors we faced three main challenges:

1. **Inadequate Information**: The build logs don't have enough information regarding the error/ the specific files containing the error that it can be automated For instance, image-tag: Backendless-Android-SDK-151437488

   Error: <module>pom-gwt.xml</module>

   Fix : removing the module from the POM file

   However the error message from the build log doesn't indicate that the issue is in this particular module, so it's not possible for the fix to be automated.

2. **Require Knowledge of the Code-Base**: Some builds fail due to logical or semantic errors in the code, or due to missing files. These errors obviously can't be fixed without being familiar with the project so we didn't attempt to fix them. For instance, in image-tag: apache-commons-lang-159112598, the error is fixed by reverting an earlier commit.

3. **Multiple Errors**: There are instances where an artifact fails due to multiple reasons. If the first encountered error is an dependency error, it gets successfully fixed. However, the build fails because the projects runs into a different error which has not been automated yet. For instance, in image-tag: ProjectKorra-ProjectKorra-188207778, there are two errors. First one is a dependency error which is fixed successfully by our system and once this error is fixed, we encounter a new error which causes the build to fail.

4. EXPERIMENTAL EVALUATION

Firstly, as previously mentioned in Section 3.2 the errors were divided into categories and further into subcategories for ease of identification as given in Figure 1. The distribution of errors (in the 100 Java projects we chose) is given in Figure 2.

We chose Dependency Error, and then further analyzed those specific errors to classify them as 'fixable' or 'not fixable'. This can be seen in Figure 3

Once we automated the build process for a few sub-categories, we wanted to see how it performed against a new set of image-tags.

To evaluate our approach we wanted to run the main driver function on different artifacts and see if it resolves the failed build. Running the program can have the following potential outcomes:

1. Build Passed: The artifact was modified by the program and the build exited with return code 0, indicating success.

2. Build Failed: The artifact was not modified by any of our scripts and continues to fail.

3. New Build Failure: The artifact was modified by our program but it still failed or a new error was generated due to the changes.

4. Unknown: The build fails due to reasons not listed above.

Ideally, we expect that errors will fall into either outcome #1 (Build Passed) which indicates that our program works as expected or outcome #2 (Build Failed) which indicates that the error doesn't yet have a fix in our approach. Both of these indicate that the program is working correctly.

However, Outcome #3 (New Build Failure) and Outcome #4 (Unknown) both suggest an error in the code logic - either due to the bug being fixed incorrectly or due to an error in the structure of the code. On the other hand, outcome #3 may also occur due to complex artifacts (with multiple errors). Regardless, both indicate that program hasn't been run successfully on that specific artifact.

To implement this evaluation, we collected 67 image tags of projects with similar dependency errors from BugSwarm. However, there were only 2 image-tags that satisfied the requirements.

We executed the driver function for each of them, which used the approach given in Algorithm 2 and additionally printed a message to indicate whether the (modified) build passed or failed (or any of the 4 categories mentioned above). Both image tags were successfully built. The results are given in Figure 4

5. RELATED WORK

There has been a considerable amount of research done in the analysis of build systems and build errors. Google

Error Category	Subcategory	Count	Description
Code Error	Incorrect Function Calls	3	Function call had incorrect parameters
	Incorrect (imported) Package Name	3	Incorrect package name
	Java Grammar edited	2	Changes made in .g file
	Compatibility Error (Java 7)	5	Compatibility errors with previous versions of JAVA
	Incorrect Exception handling	2	Errors in exception handling
	Function name error	2	Incorrect function name
	Misc	36	
Filename Changed	NA	5	Java class doesn't match with filename
	Code Logic	1	Filename was changed due to code logic
Dependency Error	Internal	8	Version changed for internal package import
	External	7	Version changed for external package import
	POM-changed module	4	Module changes in POM file
	Misc	2	
License Error	NA	1	License criteria not met (eg. missing headers)
Buildconfig Errors	Travis Error	4	Error in Travis specifications
	Merge Conflict Error	1	Error due to a merge conflict
	Gradle Error	1	Error in Gradle file
	POM error	1	Error in POM file
Multiple Errors	Multiple commits	3	Many changes over many files in multiple commits
	Combination of Errors	5	Multiple errors
Quality Error	Checkstyle Error	3	Error generated due to usage of checkstyle package

Figure 1: Error Categories and Subcategories

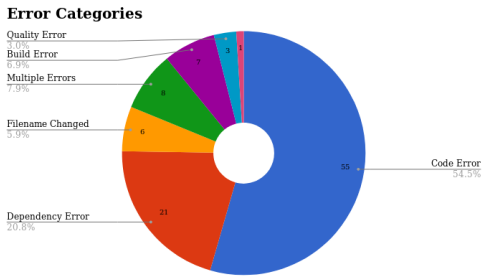

Figure 2: Distribution of Error Categories

presents a study [7] on 26.6 million builds on projects across C++ and Java. They analyze and categorize those build errors, their frequency, and their resolution. This study gives an insight on how a large organization's build process works, and pinpoints errors for which further developer support would be most effective. In further analysis of build failures, Ade Miller from Microsoft wrote the seminal paper [4] on build failures in Continuous Integration. This analyses the most common reasons for build failure (among other things).

Continuing with analysis of continuous integration and build failues, Carmine Vassallo et al's 2017 paper[8], goes deeper into the process, trying to find patterns in build failures using cluster analysis. His 2018 (pre-print) paper [1] further attempts to provide more information about the failure to the developer and also suggests some possible solutions found on the Internet.

Thomas et al.[5] present an empirical study of CI build failures in 14 Java-based OSS projects. They explore the data of CI builds to find what type of errors occur and what is the frequency of the error. Once they have all the errors, they classify these errors into different types. Based on the data, the paper tries to figure out what development practices lead to build failures.

On a similar vein to the work we've done, BuildMedic [2] is an approach to fix dependency errors on Maven builds (although we've not limited ourself to just Maven systems)

6. CONTRIBUTION

All three authors contributed equally to the project. Antara Bhowmick wrote the scripts for building the Docker artifacts and extracting the meta data. Each artifact was then individually examined to populate the spreadsheet - each author was responsible for approximately 33 artifacts. The first draft of the report was also written jointly by the authors. Together we analyzed the categories and selected one category to build automated tools for. Each author was assigned a few sub categories of the dependency errors to write python scripts for. The final report was also written jointly by the authors.

7. CONCLUSION

Despite the widespread use of continuous integration practices and the resulting problems faced by the build master to allow smooth collaboration of code, there isn't sufficient work done in the field of automating the process of fixing build errors. Through this project, we study and categorize these build errors and automate the fixes for a few of them. We then test the system against a new set of image-tags to evaluate our approach. The results indicate that the system was able to resolve build errors for the new image tags. To

Image-Tag	Status	Obervation
Adobe-Consulting-Services-acs-aem-commons-266576832	Fixable	The error message for this build clearly mentioned the detected version and the suggested version. It also gave a sub path to the file that required the change.
apache-commons-lang-159112598	Not Fixable	The build failed because an entire commit was reverted. Hence, it cannot be automated.
Backendless-Android-SDK-151437488	Not fixable	The error message does not clearly indicate the cause of the build failure. Hence it is not fixable.
charite-jannovar-100092142, charite-jannovar-100092143, charite-jannovar-100092144	Not Fixable	The error message mentions the reason of build failure however, it is unclear how to determine the files where the failing plugin version needs to be updated.
charite-jannovar-78561983, charite-jannovar-78561984, charite-jannovar-78561985	Not Fixable	Discrepancy in error files generated.
google-closure-compiler-172459863	Fixable	The error message indicated an error in the snapshot version. This was easily fixed by switching to a non-snapshot version.
google-closure-compiler-97793256	Not fixable	This build is fixed by removing the reference to a pom file which causes the error. However, the error does not specify any information regarding the file.
HubSpot-Baragon-104701564, HubSpot-Baragon-104701567, HubSpot-Baragon-104701606	Not fixable	The fix for this build is done by updating the version of a artifact. However, the error message does not provide any information regarding the same.
HubSpot-Baragon-89269844	Not fixable	This build is fixed by adding a new dependency to the project. This cannot be automated as the error message does not indicate this lack of dependency.
joyent-java-manta-100589610	Fixable	The error message indicated an error in the snapshot version. This was easily fixed by switching to a non-snapshot version.
junkdog-artemis-odb-139129646	Fixable	The error message indicated the plugin which is not working. So this is fixed by removing the plugin.
junkdog-artemis-odb-72035852	Fixable	Log file specifically mentioned the file and dependency name which needed to be fixed.
konsoletyper-teavm-84626989	Not Fixable	Not fixable since the correction made in the commit and the error pointed by the log file are not the same.
petergeneric-stdlib-151206239	Fixable	All relevant information was given in the error log
ProjectKorra-ProjectKorra-188207778	Fixable	The expected file name and actual file name were both mentioned in the error message

Figure 3: Dependency Error: Further Analysis

Image Tags	Type	Results
Adobe-Consulting-Services-acs-aem-commons-266576832	Version Conflict	Fixed
google-closure-compiler-172459863	Snapshot error	Fixed
joyent-java-manta-100589610	Snapshot error	Fixed
junkdog-artemis-odb-139129646	Remove plugin	Fixed
junkdog-artemis-odb-72035852	Package not found	Fixed
petergeneric-stdlib-151206239	Could not resolve dependency	Fixed
ProjectKorra-ProjectKorra-188207778	Could not resolve dependency	Not Fixed*
joyent-java-manta-100589611	Snapshot error	Fixed
ProjectKorra-ProjectKorra-188207781	Could not resolve dependency	Not Fixed*

Figure 4: Final Results

continue the project further, one can write automated fixes for other categories and subcategories as well.

8. REFERENCES

[1] T. Z. Carmine Vassallo, Sebastian Proksch and H. C. Gall. Un-break my build: Assisting developers with build repair hints. *ICPC*, (5), May 2018. doi: 10.1145/3196321.3196350. URL `http://doi.org/10.1145/3196321.3196350`.

[2] C. Macho, S. McIntosh, and M. Pinzger. Automatically repairing dependency-related build breakage. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 106–117, March 2018. doi: 10.1109/SANER.2018.8330201.

[3] D. Merkel. Docker: Lightweight linux containers for consistent development and deployment. *Linux J.*, 2014 (239), Mar. 2014. URL `http://dl.acm.org/citation.cfm?id=2600239.2600241`.

[4] A. Miller. A hundred days of continuous integration. In *Agile 2008 Conference*, pages 289–293, Aug 2008. doi: 10.1109/Agile.2008.8.

[5] T. Rausch, W. Hummer, P. Leitner, and S. Schulte. An empirical analysis of build failures in the continuous integration workflows of java-based open-source software. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 345–355, May 2017. doi: 10.1109/MSR.2017.54.

[6] C. Rubio-GonzÃąlez. Bugswarm project, 2018. URL `http://www.bugswarm.org`.

[7] 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*, ICSE 2014, pages

724–734, New York, NY, USA, 2014. ACM. doi: 10.1145/2568225.2568255. URL http://doi.acm.org/10.1145/2568225.2568255.

[8] C. Vassallo, G. Schermann, F. Zampetti, D. Romano, P. Leitner, A. Zaidman, M. D. Penta, and S. Panichella. A tale of ci build failures: An open source and a financial organization perspective. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 183–193, Sept 2017. doi: 10.1109/ICSME.2017.67.