Reuse (or Lack Thereof) in Travis CI Specifications: An Empirical Study of CI Phases and Commands

Puneet Kaur Sidhu
Department of Electrical and Computer Engineering
McGill University
Montreal, Canada
puneet.sidhu@mail.mcgill.ca

Gunter Mussbacher
Department of Electrical and Computer Engineering
McGill University
Montreal, Canada
gunter.mussbacher@mcgill.ca

Shane McIntosh
Department of Electrical and Computer Engineering
McGill University
Montreal, Canada
shane.mcintosh@mcgill.ca

***Abstract*—Continuous Integration (CI) is a widely used practice where code changes are automatically built and tested to check for regression as they appear in the Version Control System (VCS). CI services allow users to customize *phases*, which define the sequential steps of build jobs that are triggered by changes to the project. While past work has made important observations about the adoption and usage of CI, little is known about patterns of *reuse* in CI specifications. Should reuse be common in CI specifications, we envision that a tool could guide developers through the generation of CI specifications by offering suggestions based on popular sequences of phases and commands. To assess the feasibility of such a tool, we perform an empirical analysis of the use of different phases and commands in a curated sample of 913 CI specifications for Java-based projects that use Travis CI—one of the most popular public CI service providers. First, we observe that five of nine phases are used in 18%-75% of the projects. Second, for the five most popular phases, we apply association rule mining to discover frequent phase, command, and command category usage patterns. Unfortunately, we observe that the association rules lack sufficient support, confidence, or lift values to be considered statistically significantly interesting. Our findings suggest that the usage of phases and commands in Travis CI specifications are broad and diverse. Hence, we cannot provide suggestions for Java-based projects as we had envisioned.**

I. Introduction

Continuous Integration (CI) is a commonly adopted Software Engineering practice these days [2], [3], [4]. CI aims to ensure that each change to the software system is scanned by routine checks (e.g., automated compile, test, static code analysis) [10]. Automated CI services like Travis CI[1] integrate with GitHub[2] to facilitate this process by allowing users to script build routines for their project. Whenever a code change is pushed or a pull request is received, the contribution is checked by configuring machines and executing build scripts as expressed in the Travis CI specification (i.e., `.travis.yml`). Users are notified (e.g., via email, Slack message) of build results (e.g., when build jobs pass, fail, or either) as set in the Travis CI specification.

Travis CI is a hosted, distributed CI service used to build and test software projects hosted at GitHub. Travis CI provides free service for open source repositories on GitHub, and a paid service for private repositories.[3] According to a recent analysis conducted by GitHub, Travis CI is the most popular CI platform.[4]

The behavior of a project's build job is defined in its `.travis.yml` file, which is by convention located in the root directory of the project. This file contains build machine configuration details (e.g., which programming language toolchain needs to be installed), notification settings, and scripts that describe the order-dependent steps that must be executed to build, test, and deploy the project.[5]

Recent studies have analyzed CI adoption (and omission) trends [7], [9], the types of failures that occur during CI [1], [11], [14], [16], and the outcomes associated with CI adoption [15]. However, little is known about how unique CI specifications are. Given that recent qualitative studies suggest that a frequent reason for omission of CI is due to lack of familiarity with it [8], it seems reasonable that a CI template generation framework may be useful to ease the burden of CI adoption. As a step towards such a template generation framework, this paper performs an empirical analysis of which phases and commands are being used in real-world Travis CI specifications.

Based on the results of the analysis, we aim to build a tool that provides suggestions to users to choose phases and commands to automatically create a CI script for Java projects. Taking partial input provided by the user into account, the tool will first give suggestions to users about which phases to use and then will provide sequences of frequently used commands under each of the chosen phases. The tool would then generate a `.travis.yml` file according to the user's selections, which can be further customized by the user if required.

For this study, we analyze a corpus of 913 open source, active, large, non-forked and non-duplicate Java projects that were analyzed by Gallaba and McIntosh [6]. Using these projects, we set out to study the phases and commands that are frequently used together in the `.travis.yml` files. Using

[1]https://docs.travis-ci.com
[2]https://github.com
[3]http://docs.travis-ci.com/user/getting-started/
[4]https://blog.github.com/2017-11-07-github-welcomes-all-ci-tools/
[5]https://docs.travis-ci.com/user/customizing-the-build

978-1-7281-0591-8/19/$31.00 © 2019 IEEE

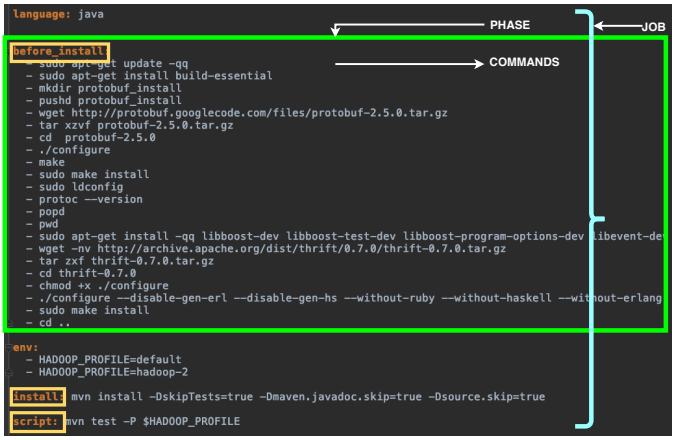

Fig. 1. An example `.travis.yml` specification from the Dlanza1 project.

an association rule mining approach, we make the following observations:

- Of the nine phases provided by Travis CI, the "script", "before_install", "install", "after_success", and "before_script" phases are the most frequently used phases by developers
- The mined association rules among phases, among commands, and among functionally-similar commands (i.e., command categories) lack sufficient interestingness scores (i.e., support, confidence, lift, or count) to be of use in the context of our tool

Thus, our vision to build a tool that provides suggestions to build CI specifications based on popular phases and commands in Java projects cannot be realized based on the findings of this empirical study. The study suggests that `.travis.yml` files are user-defined files with no generic structure beyond the already existing phases as each developer programs this file in a different way.

The rest of this paper is organized as follows. Section II describes Travis CI terminology and situates this work with respect to the literature on CI. Section III describes the design of our analysis. In Section IV, we take "a devil's advocate" approach to describe the lack of evidence for useful reuse patterns in Travis CI specifications. Section V discusses the threats to the validity of our study. Finally, Section VI draws conclusions and outlines avenues for future work.

II. Background & Related Work

In this section, we describe terminology related to CI and situate this paper with respect to the prior work on CI specifications.

A. Terminology

In this paper, we use common build terminology in the Travis CI context. Figure 1 provides an example of how key build terminology maps onto the `.travis.yml` specification.

The Travis CI service listens for when an integration into the subscribing project is queued (i.e., a new pull request has been created) or performed (i.e., new commits have been pushed into the repository). When an integration is performed, Travis CI spawns a new *build*, i.e., a logical group of build jobs. The *status* of a build is dependent on the status of each of its jobs. A *build* is successful if all of its jobs are (a) labeled as successful; or (b) configured to be irrelevant [5].

A build *job* is an automated process that (1) downloads an up-to-date copy of the project under test to a testing (virtual) machine in the Travis CI environment; and (2) executes the specified steps to prepare the machine, compile and test the project, and optionally deploy a new release of the project to staging or production environments.

Build jobs are composed of a sequential series of *phases*, which are in turn composed of a sequence of *commands* to be invoked during phase execution. Travis CI phases fall into three categories. First, the *install* phase is responsible for preparing the job processing machine for the subsequent phases. Next, the *script* phase performs the tasks that are necessary to build and test the project. Finally, the optional *deploy* phase makes a new release of the project available for users. Each phase has a *before* variant for performing setup steps before executing the core phase logic. The *script* and *deploy* phases have *after* variants for cleaning up the execution environment after the phase has been performed.

B. Related Work

In this section, we situate our work with respect to the past work on CI along adoption and outcome dimensions.

Adoption of CI. An important trend of CI research has focused on adoption trends in software organizations. For example, Beller *et al.* [1] analyzed and shared a large, curated sample CI data from thousands of open source projects. Hilton *et al.* [9] studied the characteristics of projects that choose to adopt CI (or not), and barriers that developers face when adopting CI [7].

Moreover, the adoption of CI has been associated with project properties. Vasilescu *et al.* [15] found that CI adoption is often accompanied with a boost in team productivity. Hilton *et al.* [9] found that CI adoption was linked with project popularity, i.e., greater popularity implied a higher likelihood of CI adoption.

One of the most important barriers to CI adoption is the lack of support for specializing CI towards the process that a team is using [7]. Indeed, Widder *et al.* [17] observed that projects using language toolchains with limited support from Travis CI (e.g., C#) are more likely to abandon it. We envision a tool that could bridge that gap by supporting developers who are creating or editing CI specifications. To support the development of this tool, in this paper, we mine existing Travis CI specifications in search of common patterns of use that could be extrapolated into templates. These templates could be specialized during the development and maintenance phases of the CI specification.

Outcome of CI. The plethora of available CI data has enabled large-scale analyses of build processes. For example, Zampetti *et al.* [18] analyzed the usage of static analysis tools from

within the CI process. Beller *et al.* [1] studied testing practices in Java and Ruby projects using Travis CI, observing that build breakages (i.e., failing builds) are most often associated with test failures.

At its core, a CI cycle produces feedback about code changes. Failing builds in theory signal a problem that should be tackled immediately. Hence, researchers have set out to understand and predict failing builds in the CI context. For example, Rausch *et al.* [13] studied common types of build breakages in 20 Java open source systems. Vassallo *et al.* [16] compared build failures in the CI processes of open source organizations to those of a large financial institution. Gallaba *et al.* [5] analyzed the levels of noise in build outcome data, e.g., breakages that do not prompt quick responses (implying they did not need to be tackled immediately) or passing builds that include failing jobs.

Like other software artifacts, developers may make mistakes when specifying CI systems. Gallaba and McIntosh [6] formulated, quantified, and fixed patterns of misuse of CI features (i.e., anti-patterns) in a large sample of Travis CI specifications. Labuschagne *et al.* [12] observed that long chains of broken builds were often due to misconfigured CI specifications. Inspired by these prior studies, we set out to ease the burden of development and maintenance of CI specifications by learning from the plethora of existing specifications.

III. Experimental Design

In this section, we give an overview of the approach followed for the empirical analysis of the phases and commands used in Travis CI specifications, including details of the subject data-set as well as the outline of the data analysis and data validation processes. At the end of this section, we summarize the key findings of this study.

A. Data Extraction

In this study, we analyze 913 open source Java projects taken as a subset from the corpus of projects studied by Gallaba and McIntosh [6]. These projects are taken from GitHub with which Travis CI collaborates to implement continuous integration. These projects were selected to be analyzed because they are active, large, non-forked and non-duplicate projects that contain a valid Travis CI specification file in their root directory.

B. Data Analysis and Validation

Figure 2 shows an overview of the process followed for our analysis. As shown, we automatically and manually parse the `.travis.yml` files of the selected projects and store the parsed output. Then, we analyze and validate the same using Association Rule Mining (ARM).

The aim of ARM is to determine rules based on occurrence patterns in the source data-set. The metrics which define the strength of these rules are briefly explained in Table I.[6] The example in this table is explained in terms of transactions as it is an easily understood, general example.

TABLE I
Association rule metrics explained

Metrics	Meaning
support	This says how popular an item-set is, as measured by the proportion of transactions in which an item-set appears.
confidence	This says how likely item Y is purchased when item X is purchased, expressed as {X ->Y}.
lift	This says how likely item Y is purchased when item X is purchased while controlling for how popular item Y is. A lift value greater than 1 means that item Y is likely to be bought if item X is bought, while a value less than 1 means that item Y is unlikely to be bought if item X is bought.

C. Key Findings

Following the study design laid out in this section, we conclude that we cannot find useful patterns of phases and commands with ARM. We summarize our results below and discuss the details of the results of the study in Section IV.

Summary of Results. The results of our analysis suggests a *negative result*, i.e., we were not able to find patterns of phases and commands in Travis CI specifications for Java projects with ARM that are statistically significantly interesting. Based on these results, our vision of a tool that provides suggestions to build CI specifications based on popular sequences of phases and commands cannot be realized.

IV. Advocatus Diaboli

This section states the proceedings of the **Advocatus Diaboli** (AD), Latin for **Devil's Advocate**, who represents the domain expert for our area of study. The concept of the AD is inspired from a former process followed in the Catholic Church where the AD would support a prosecution case against the candidates for canonization to sainthood. The AD was required to question every reason for the candidate's elevation and reject the same. Proponents for canonization would then build a defense case to answer each of the points raised by the AD.[7] In our case, we are the proponents who analyze Travis CI specification files and state a negative result, i.e., relationships based on ARM among phases and commands in Travis CI files of Java projects cannot be used to create a prediction system for building CI specifications. Thus in the following sub-sections, the AD questions our methodology and results and we refute all arguments put forward by the AD. Consequently, we state the details of our approach, the precautions, and actions taken to support our analysis.

A. Experiment Approach

Below the AD presents arguments related to the experiment setup.

[6]https://algobeans.com/2016/04/01/association-rules-and-the-apriori-algorithm/

[7]http://www.enase.org/CallForPapers.aspx?y=2013#ADF

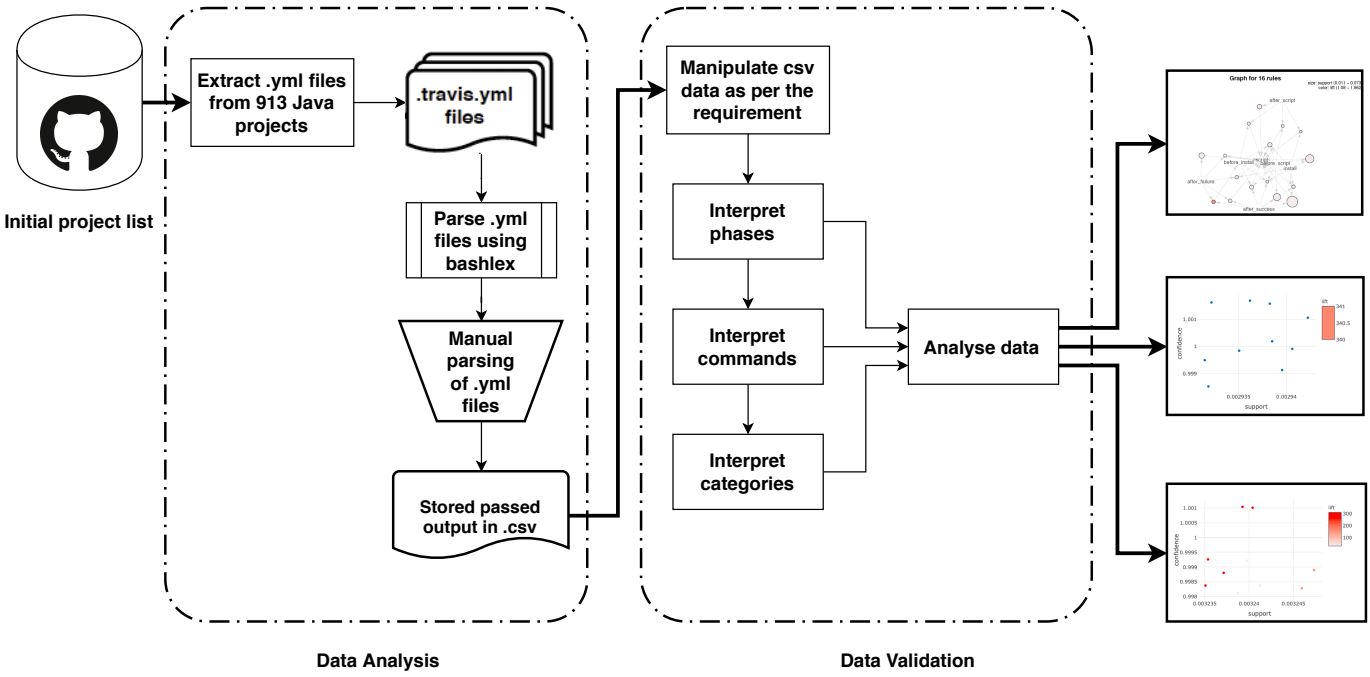

Fig. 2. An overview of our approach to study the `.travis.yml` files and analyze the phases and commands inside them

Argument A.1: The right projects were not selected for the analysis. The AD wants to understand if we selected the correct data-set for our analysis as that might affect the results adversely. We refute the AD's concern by stating that we selected only valid and active Java projects taken from GitHub. These 913 projects use Travis CI which is verified manually by checking each project as during analysis we inspect all `.travis.yml` files in the root directory of the project. These are non-forked and non-duplicated projects as backed up by Gallaba and McIntosh [6] who state that forked projects should not be included as their analysis leads to duplicated outcomes.

In addition, we studied the domain of the projects to check the heterogeneity of the sample. We do this by iteratively going through the project documentation on GitHub until we find no more new domains for the last 20 studied projects. In our case, we needed to look at 44 projects and our observations are shown in Table II. Thus, we see that we study different types of projects and also that the sample is not too heterogeneous which also might lead to inconsistent results. Note that we are still using all 913 projects for all other analyses.

At this point, the AD questions whether we conducted the experiment in the right way as the results can be invalid if the right approach is not followed. Thus, we explain our approach below.

Argument A.2: The study design is flawed. We first clone the projects from the initial project list of 913 projects with just the `.travis.yml` files in them from GitHub. We iterate over the project list and parse the `.travis.yml` files kept in the root directory of the projects. We use the bashlex parser library of python to do this and we parse each line word by word to extract the phases and commands with their prefixes and parameters separately. To get a refined list of commands as per the related phases, we apply various regular expressions to avoid getting irrelevant data for the study such as hard-coded arguments and conditional statements. Also, there are lines in `.travis.yml` files which contain more than one command. For handling this, we use the bashlex parser as it creates an abstract syntax tree which maintains the inherent structure of phases and the commands under them.

We verify the output of our python script by checking that the parser has read the prefixes, commands, and arguments of commands correctly for a small set of 10 projects and after the initial validation, we move ahead with running our script over the entire data-set for the "install" phase. We choose the "install" phase, because it is one of the main phases and is frequently used. After verification of this output, we see that the parser does not work quite well on lines containing multiple brackets and commands having more than one option. We also see that there is no definite structure of the Travis CI

TABLE II
DOMAIN SPECIFICATION OF SUBSET OF SAMPLE PROJECTS

Type of Project	No. Of Projects	Percentage%
Application Framework/Library	15	34.09
Development Tool	10	22.72
Mobile Application	8	18.18
Web Application	4	9.10
DevOps	3	6.81
Games/Game Engine	2	4.54
Communication/Collaboration Tool	1	2.27
Other	1	2.27
Total	**44**	**100.00**

files. Therefore, we manually inspect each line after running the script for all phases on the entire data-set.

The resulting output is saved in a csv file. We create the output such that it denotes the project name, the phase, the prefix of the command, the command, the sub-command, the option(s), and the parameter. If a command has more than one parameter we write the same line again but with the other parameter and so on. We also categorize the commands using our output from the csv and we write the category in the same csv under which each command falls. For example, we say that "mvn" and "gradle" fall under the "builders" category. We create these categories based on the functionality provided by the command. The parsing approach is built based on multiple discussions among all three authors and on multiple `.travis.yml` files with different styles of commands.

We started our analysis with the phases mentioned in Table III. We also state the number of projects out of the total projects in which the mentioned phases are used respectively. We observe that the top five phases are used in 18%-75% of the projects. Thus, we study only the top five phases in greater detail as we expect to find patterns in the phases that are the most common.

TABLE III
PHASES STUDIED FROM `.TRAVIS.YML` FILES

No. Of Projects	Project Percentage	Phase
681	74.6	script
397	43.5	before_install
329	36.0	install
254	27.8	after_success
162	17.7	before_script
36	3.9	after_failure
27	3.0	after_script
21	2.3	before_deploy
3	0.3	after_deploy

Table IV shows the details of the various categories of commands used in Travis files across all projects. We further find the command usage as per categories in the top five phases respectively which is stated in Table V. We show only the top five categories occurring under each phase to reduce clutter. Now, as we know the top categories in each of the phases, we study the frequency of the top most used category in each phase. Table VI shows the most popular category per phase and its usage as per the number of projects. For example, 'builders' is the most popular command category in the 'script' phase and it is used 14 times, 10 times, and 9 times in one project, 7 times in three other projects, and so on. We understand the dominance of the various categories per phase and proceed with our analysis accordingly by studying the most frequent category per phase. Again, we expect patterns to appear in the most used categories.

We analyze all the above information as the source for ARM. Our scripts and parsed data are available online at https://figshare.com/s/96287a53e4ad7e55a906 which can be used for attestation of our analysis.

However, the AD is not satisfied with the description of our parsing approach and wants to dig deeper into details of how we decided on the parsing of the commands as that is at the core of our analysis. Incorrect parsing can altogether affect the outcome drastically. This concern is addressed below.

TABLE IV
ALL PROJECTS: CATEGORIZATION OF COMMANDS

No. Of Projects	Category
746	builders
277	not_mutate
255	env_setup
194	interpreter
180	pkg_mgr
136	fs
123	internet
101	security
96	execute_script
96	vcs
80	compress
54	database
45	text_manipulate
38	travis_command
28	sca
25	process_mgmt
15	mobile_framework
7	compiler
3	browser_env

TABLE V
TOPMOST FREQUENTLY USED CATEGORIES OF COMMANDS PER PHASE

Frequency	script	before_install	install	after_success	before_script
Top-1	builders	env_setup	builders	builders	env_setup
Top-2	not_mutate	pkg_mgr	not_mutate	interpreter	not_mutate
Top-3	env_setup	not_mutate	pkg_mgr	not_mutate	database
Top-4	execute_script	interpreter	travis_command	internet	interpreter
Top-5	interpreter	internet	vcs	sca	builders

TABLE VI
USAGE OF THE MOST FREQUENT CATEGORY PER PHASE

Frequency	14	10	9	7	6	5	4	3	2	1
script: builders	1	1	1	3	3	1	9	30	70	493
before_install: env_setup			8	1	4	4	3	11	37	82
install: builders							1	4	20	217
after_success: builders							4	6	40	95
before_script: env_setup					1	1	1		10	47

Argument A.3: The parsing of lines in `.travis.yml` files is not valid. To efficiently and clearly analyze commands, we segregated commands to have prefixes, sub-commands, and options. We explain how and why we did this with an example. We take this line from one of the Travis files and try to understand its structure: *"./gradlew –console=plain –no-daemon -S –scan check test integrationTest functionalTest jacocoTestReport jacocoIntegrationTestReport jacocoFunctionalTestReport jacocoRootReport -x :sample-javafx-groovy:jfxJar -x :sample-javafx-java:jfxJar -x clirr"*. In this line, there are four options used for the "gradlew" command. The sub-commands "check" and "test" are used later with multiple parameters like "integrationTest" and three "-x" options to uninstall and update

jars. Thus, in this case, it is difficult to define a parser that goes through the indefinite number of options and parameters with options present in between the parameters. The case gets more complicated with multiple commands in a single line.

We also segregate commands with prefix such as "sudo" to focus on the functionality of the commands. We divided commands into main command and sub-command. This was mainly done for commands which work in pairs. For example, *'pip', 'apt-get', 'git', 'clean', 'npm', 'bower', 'nvm', 'go', 'xargs', 'pip3', 'bundle', 'service', 'time', 'bash', 'sh', 'travis_retry', 'travis_wait', 'travis_terminate', 'ant', 'android', 'mvn', 'mvnw', 'gradle', 'gradlew', 'bash64',* and *'python'* would have a second part to them such as *'git push'*. In this case, *'git'* is the main command and *'push'* is the sub-command. We restricted our analyses to the main command to get considerable association rules.

Thus, we tried to customize the parser as much as possible but the random structure of the `.travis.yml` files made it unfeasible to create a generic parser that could produce a perfectly traversed output with no data loss. To counter this, we manually check the parsed output of all top five phases and make the required changes. The manual inspection is done by the first author following a discussion of the parsing approach among all three authors.

With the manually inspected, parsed output now matching the `.travis.yml` files, the AD questions whether we chose the right set of values of support and confidence to filter out association rules and ensure that these values are not too strict.

Argument A.4: The criteria for association rule mining are too strict. Our goal is to find strong rules for phases, commands and categories in a single phase, and commands and categories across phases based on the computation metrics of support, confidence, and lift. We researched what can be the minimum values of these metrics to be considered for making solid rules and proceeded with our analysis accordingly.[8] We understand that we have to do sensitivity analyses by changing the values for support and confidence to generate rules that are complemented by a high number of projects.

We chose different values of support starting from 0.5 and confidence as 1 for which we got no rules. We then subsequently lowered the value of support to find more frequent rules but keeping the value of confidence at 1. We went to as low a value as 0.001 which means minimum 1 out of 1000 projects will have this rule. We found 10 projects which satisfied the strongest rule for phase-phase relationship. We did the same for commands and categories. The strongest rule for commands that appear together in the most used "script" phase with this criteria resulted in 32 projects and the second strongest rule resulted in 6 projects. Similarly the strongest rule for categories occurring together in the "script" phase resulted in 3 projects. We could not decrease the value of support further as we were analyzing approximately 1000 projects thus support lower than 0.001 would not have made sense.

Thus, we started decreasing the value of the confidence metric. We started with 0.9 keeping support at 0.001. We found 12 projects for the strongest rule for phases that occur together, 32 projects for commands, and 3 projects for categories in the "script" phase. We continued to tweak these metric values for all phases and then finalized the minimum value for support to be 0.001 and confidence to be 0.7 for each of the ARM done on phases, commands, and categories to reinforce well-built rules with considerable lift value. We show the results of the same in the remainder of the paper.

Finally, the AD questions the categorization of commands.

Argument A.5: The command-category mapping is incorrect. Two authors independently checked the functionality provided by each command and categorized accordingly. We match the categorization results of the two authors and verify the coherence by calculating Cohens Kappa which resulted in 0.46. Since the result is not as high as desired, the differences are discussed among all three authors and consensus is reached for each of the distinct 245 commands.

B. Experiment Conduct

After we refuted the arguments against our experiment setup, the AD now focuses on the conduct of the experiment by stating a series of arguments about coverage of the right set of relationships while doing ARM. If we do not compare the right relationships, then obviously we may miss some commonality in the data. Below, we address these concerns.

Argument B.1: The relationships among phases are not checked. We started our analysis with the phases as they are at the highest level of generalization. We state our results in Table VII and Figure 3.

TABLE VII
ASSOCIATION MINING RULES FOR PHASES

rules	support	confidence	lift	count
{} =>{**script**}	0.7459	0.7459	1.0000	**681**
{**install**} =>{**script**}	0.2673	0.7416	0.9943	**244**
{**before_script**} =>{**script**}	0.1413	0.7963	1.0676	**129**
{**before_install,install**} =>{**script**}	0.1314	0.7595	1.0182	**120**
{**after_success,install**} =>{**script**}	0.0734	0.8072	1.0822	**67**
{**before_script,install**} =>{**script**}	0.0537	0.8596	1.1525	**49**
{**before_install,before_script**} =>{**script**}	0.0471	0.7679	1.0294	**43**
{**after_success,before_script**} =>{**script**}	0.0449	0.8200	1.0994	**41**
{**after_success,before_install,install**} =>{**script**}	0.0383	0.7778	1.0427	**35**
{**after_failure**} =>{**script**}	0.0318	0.8056	1.0800	**29**
{**after_script**} =>{**script**}	0.0263	0.8889	1.1917	**24**
{**before_install,before_script,install**} =>{**script**}	0.0241	0.8462	1.1344	**22**
{**after_success,before_install,before_script**} =>{**script**}	0.0219	0.8333	1.1172	**20**

Approach. We identify the percentage of the projects having the phases mentioned in Table III. We find this by querying our csv file so as to find the number of projects containing the respective phase. We also run ARM on the top 5 popular phases mentioned in Travis CI files across projects to find patterns in their usage. We find all combinations of frequently occurring phases. We create a data-set for all these combinations. We feed this data to the arules and arulesViz libraries of the R language and using apriori algorithm we find out the relationship pattern between these phases. We

[8]https://www.quora.com/How-do-I-pick-appropriate-support-confidence-value-when-doing-basket-analysis-with-Apriori-algorithm
http://r-statistics.co/Association-Mining-With-R.html

find out the support, confidence, and lift values for all these relationship rules. We derive various rules as shown in Table VII and Figure 3 shows the plots that depict the same trends.

Results. The percentages of projects in Table III clearly show that there is a difference in usage of all the phases. We can gather that all phases are not equally used by all developers and there are phases like "script", "before_install", "install", "after_success", and "before_script" which are dominantly used. Among these common phases, we found 13 rules of multiple phases occurring together in a `.travis.yml` file and with a minimum support value of 0.001 and a minimum confidence value of 0.7. We see in Table VII that the topmost rule is the one showing that "script" phase on its own is used in 681 projects (i.e., 75%). This can also be verified from Table III. However, this is not a very interesting rule for our purpose of building a tool that suggest phases and commands based on already specified phases and commands. The subsequent rules hold true in 27%, 14%, 13%, 7% and so on with respect to the total number of projects. Although, we find 1 rule which occurs in more than 20% of the projects, the rest of the rules have low existence. We also plot these rules in Figure 3 where we can see that the usage of the "script" phase alone is the most as inferred from the size of the associated circle. This rule has lower lift which can be clearly seen from the light color of the circle as compared to, e.g., the rule between "after_script" and "script" which has a darker color. Similarly, we can understand the intensity of the rules using this figure. As we did not find any prominent, useful rule among phases, we move to find patterns between commands in a phase to determine if the developers use similar functionality for their projects in Travis CI specifications.

Summary for Phase-Phase relationship. Among all the phases taken into consideration, we find out that "script", "before_install", "install", "after_success", and "before_script" are the most frequently used phases by the developers in their Travis CI files. We find 1 rule among two phases that is present in 27% of the total projects, but the occurrences of further rules are rare. It shows that although phases are at the top level of generalization in a `.travis.yml` file, even then there is no strong coherence in their usage.

Argument B.2: The relationships among commands in a single phase are not checked. To consolidate our analyses to derive patterns in the usage of Travis CI specifications which could help developers to specify them faster, we try to deduce repetitions in the command usage in a single phase. Commands specify the functions performed by the CI script, thus it is important that we examine their usage.

Approach. To study this, we pick up all combinations of commands as per phase for all projects. We follow the same approach of finding association rules and then plotting them to understand if the commands co-occur.

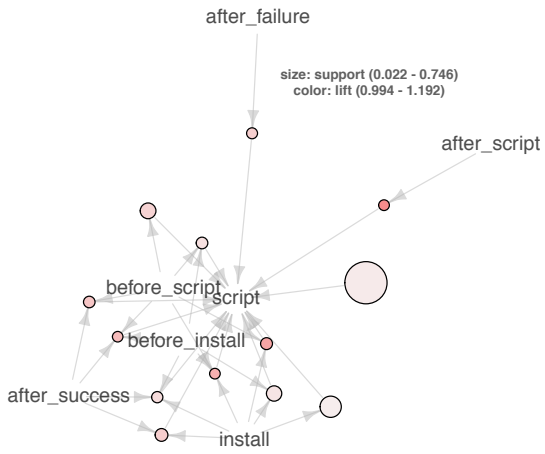

Fig. 3. Association rules for phases used in all projects

Results. We find various rules for all the five phases that we study. We state our results in Figure 4 which shows that for each of the studied phases there is a maximum of 6 rules with confidence of more than 1. Also, the color of the rules depicts the lift value, therefore the darker the color, the more the lift of the rule. Rules which do not have a lift value specified in the legend are shown in blue color. Similar to phases, we do not find any patterns in the usage of commands in the Travis CI specifications.

Summary for Command-Command relationship in a single Phase. We do not find any common patterns in the way CI is implemented by developers even on the most popular CI service on GitHub. We find a maximum of 6 rules supported by strong values of confidence but low values of support and count, i.e., even these strong rules occur at the most in 4.3% of the projects. Furthermore with lower confidence values, the most frequent rule in the "script" phase still occurs in only 9% of the total projects and this percentage further falls for other phases.

Argument B.3: The relationships among command categories in a single phase are not checked. Expanding our analysis deeper into the phases and their commands and to be able to find solid rules between commands in a phase, we increase our level of analysis and study categories of commands. We strive to understand what type of functionality different developers tend to perform using Travis CI. Performing ARM on categories after phases and commands shall finally give a clear picture of the implementation of Travis CI specifications.

Approach. We analyze all the commands stated in the Travis files and categorize them according to the function they perform. In Table V, we show which categories of commands appear per phase across all projects. More details are shown in Figure 5.

Results. Although, we see that the category usage across projects is significant as stated in Table IV, we do not find

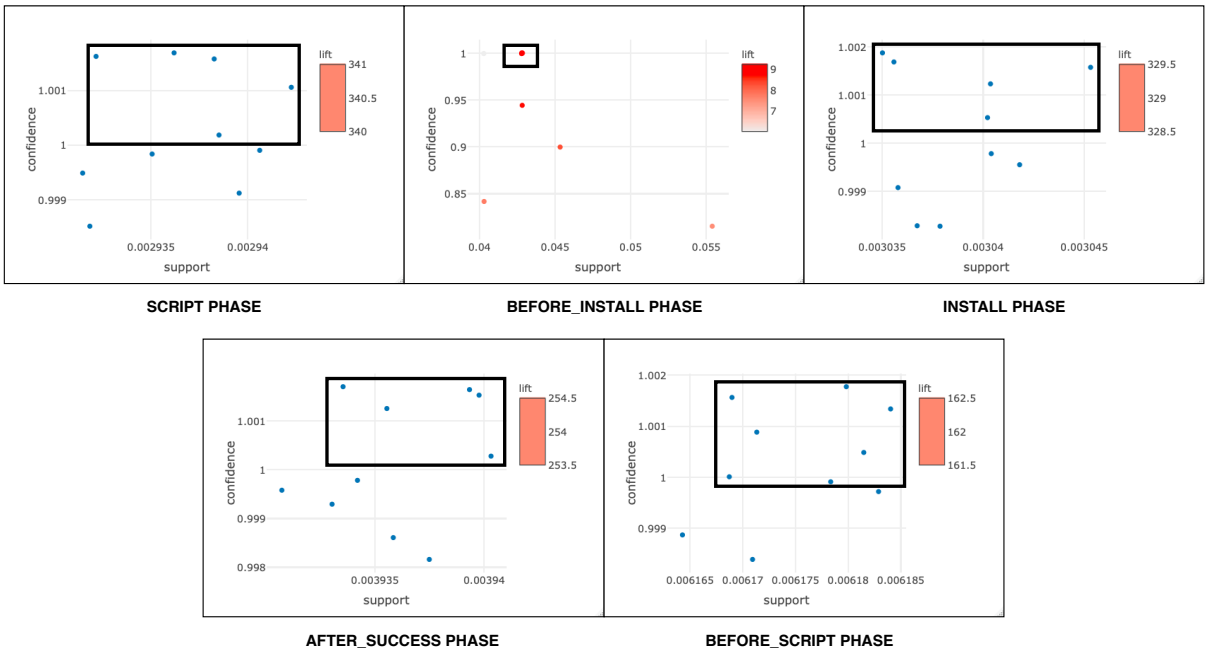

Fig. 4. Association rules for commands used in all projects per phase

strong association rules supported by high values of support, confidence, and lift. By analyzing Figure 5, we can see that, for all phases, they are just 4-6 rules that have the confidence value of at least 1. However, the strongest support for any of these rules is less than 1%, i.e., there is no combination of categories of commands that frequently occurs together.

Summary for Category-Category relationship in a single Phase. Among all the defined categories, we find that different phases have different functionality that they dominantly perform in terms of categories of commands. This is inferred from Table V and VI. We find a maximum of 6 rules supported by strong values of confidence but low values of support and count, i.e., even these strong rules occur in less than 1% of the projects. Furthermore with lower confidence values, the most frequent rule in the most frequent phase, i.e., the "script" phase, still occurs in just 8.5% of the total projects and the usage falls further for categories in other phases.

Argument B.4: The relationships among commands across phases as well as command categories across phases are not checked. To not leave any shred of doubt, we inspect commands and command-categories, respectively, across phases, that is irrespective of the phases in which they are predominately used. Although it is expected that patterns of commands and categories occur more likely within their respective phases, we perform this analysis to be sure of the fact that there are no useful patterns of commands and command categories in Travis CI specifications.

Approach. We collect all distinct commands and categories for all projects with no filter on phase and apply ARM on them. As with all other ARM, we find rules with a minimum support value of 0.001 and a minimum confidence of 0.7. Our results are shown in Table VIII and IX. Table VIII shows only the top 5 most frequent rules to reduce clutter. We look over these trends generated by ARM and analyze the results.

TABLE VIII
ASSOCIATION MINING RULES FOR COMMANDS ACROSS PHASES

rules	support	confidence	lift	count
{sh} =>{export}	0.0788	0.8275	3.7221	72
{tar} =>{wget}	0.0558	0.9622	10.5849	51
{tar} =>{export}	0.0493	0.8491	3.8186	45
{tar,wget} =>{export}	0.0482	0.8627	3.8802	44
{export,tar} =>{wget}	0.0482	0.9778	10.7556	44

Results. As expected, we do not find rules that are applicable to a large number of projects. In Table VIII, we have rules for commands occurring together across phases sorted by the number of projects in which they appear. The most common occurs in 72 out of 913 projects which is only 8% of the projects. While there are other rules with high values of confidence and lift, their support and count are low.

A similar inference can be made for command categories occurring together across phases when we look at the data in Table IX. The "builders" category is the most common category as we already know from Table IV and by itself forms the most common rule. Again, however, this is not a very interesting rule for our purpose of building a tool that suggest phases and commands based on already specified phases and commands. The second most frequent rule occurs in approximately only 25% of the total projects, which

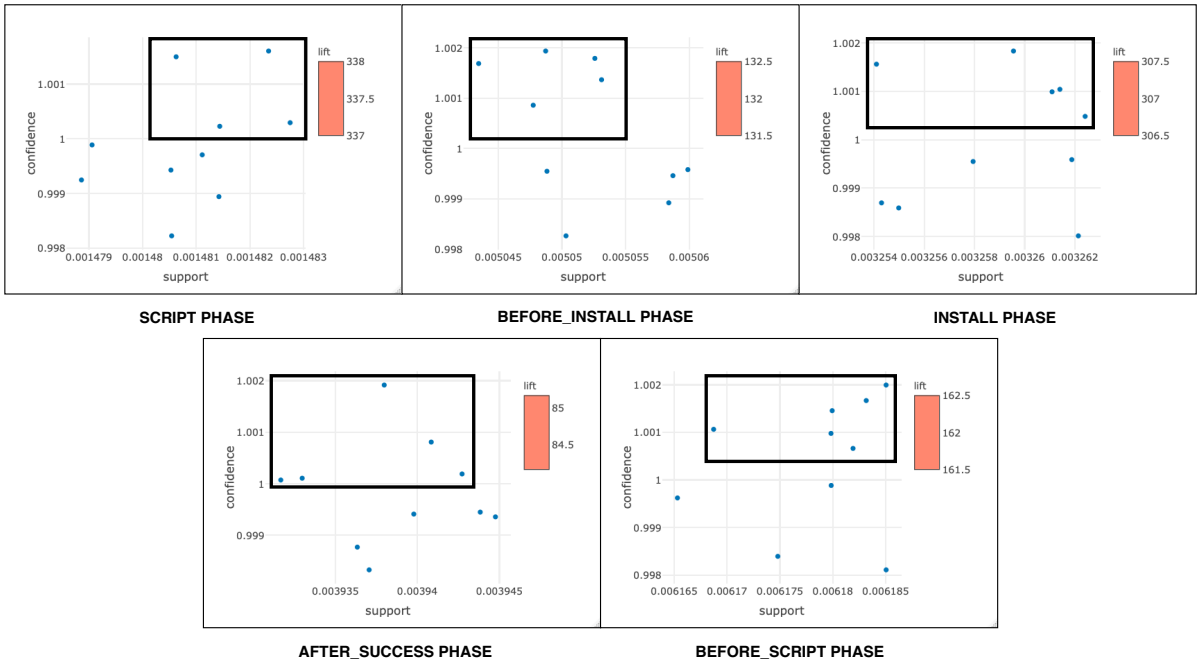

Fig. 5. Association rules for categories used in all projects per phase

again is not high enough to be useful for suggestions in our envisioned tool.

Summary for Command-Command relationship across Phases and Category-Category relationship across Phases. As anticipated, we do not find any rules with high enough confidence, lift, and support values which could help us provide suggestions based on existing Travis CI specifications.

TABLE IX
ASSOCIATION MINING RULES FOR CATEGORIES ACROSS PHASES

rules	support	confidence	lift	count
{} =>{builders}	0.8171	0.8171	1.0000	746
{not_mutate} =>{builders}	0.2530	0.8339	1.0206	231
{env_setup} =>{builders}	0.2180	0.7804	0.9551	199
{interpreter} =>{builders}	0.1698	0.7990	0.9778	155
{env_setup,not_mutate} =>{builders}	0.1369	0.8503	1.0407	125
{fs} =>{builders}	0.1183	0.7941	0.9719	108
{internet} =>{env_setup}	0.0953	0.7073	2.5325	87
{env_setup,interpreter} =>{builders}	0.0887	0.8100	0.9913	81
{security} =>{builders}	0.0865	0.7822	0.9573	79

V. THREATS TO VALIDITY

This section discusses the threats to the validity of our study.

A. External Threats

External threats to validity concern the generalization of our study.

We are only studying Java projects. We focus on one language because of, e.g., tool-chain differences. This may hinder generalization to other languages. Other languages are out of scope for this study and we hence leave the analysis of other languages to future work. We also only study .travis.yml files and the contents of any build scripts (e.g., Maven/Gradle) are also out of scope.

We studied only the publicly visible part of Travis CI and GitHub and not the private projects available on GitHub which might deviate the results. Furthermore, we presume that the .travis.yml files are largely written by development teams even though user guides exist that provide some code snippets.

B. Internal Threats

Internal threats to validity concern the correctness of our design.

We analyze `.travis.yml` files from 913 different projects but we are not sure if all those projects even use those files for building and testing purpose. It might be a possibility that in the initial stages of the project development, the developer decided to use CI but later on did not end up using it. We assume that the projects that have a valid `.travis.yml` configuration file use the CI service and we continue our analysis accordingly.

The list of five phases and underlying commands and categories per phase and across phases that we analyze in this paper is not exhaustive. We did not represent all phases and commands as our objective was limited to studying only the frequent phases and commands to understand the usage of commands in phases. However, we did not find high usages of commands and categories even for the most frequently used one, which makes it unlikely to find patterns in low-frequency phases and commands.

Similarly, we did not analyze the parameters of the commands for the same reason as stated above, i.e., high usages

were not found even for more general concepts. Thus, we kept the analysis of arguments out of scope for our current study.

C. Construct Threats

Construct threats to validity concern the link between theory and real observation.

The phases and commands are analyzed using our scripts written in Python. These scripts may themselves contain defects which would affect our analysis. To address this threat, we first tested our scripts on a smaller data-set, then after verification we ran them on the full data-set, and checked and corrected the results manually to ensure conformance to the Travis CI specification files.

Our scope of analysis is specific to ARM. However, other techniques may find different results.

VI. Conclusions

Travis CI started recently in 2010 and was established as a company in 2012. As discussed in our paper, Travis CI is a commonly used CI service for open-source projects because of which we consider it an appropriate target for our study.

We envision a tool that provides suggestions based on partial input to users to choose phases and commands to automatically create a Travis CI script for Java projects. The suggestions are to be based on sequences of commands that commonly occur together in Travis CI scripts. To do this, we analyzed the phases and commands mentioned in the Travis files as there is a lack of studies on their usage. In this paper, we see that there are five phases that are majorly used in Travis CI specifications. When analyzed, these phases have 245 different commands that are executed under them. These commands can be categorized based on the similarity of their functionality.

We show that although the source sample was large with 913 projects, we find no strong coherence in the manner in which the phases, commands, or command categories tend to appear together in Travis CI specification files. To support our argument, we analyzed relationships among phases, among commands in a single phase as well as across phases, and command categories in a single phase as well as across phases in our heterogeneous sample of projects using association rule mining (ARM). Based on these findings, our vision to build a tool that provides suggestions for CI specifications considering popular phases and commands in Java projects cannot be realized. Useful patterns could not be found in the user-defined `.travis.yml` files beyond the already existing phases. However, techniques other than ARM may find different results. Furthermore, we plan to take the context of a project into account in future work (e.g., does the project use a database?), which may also influence the existence of interesting patterns.

While our findings are negative, we hope that our work is helpful for researchers to understand the usage of phases and commands of Travis CI specifications and they can build on it to further study Travis CI specifications.

References

[1] M. Beller, G. Gousios, and A. Zaidman, "Oops, my tests broke the build: An explorative analysis of travis ci with github," in *Mining Software Repositories (MSR), 2017 IEEE/ACM 14th International Conference on*. IEEE, 2017, pp. 356–367.

[2] G. Booch, "Object-oriented design; redwood city," 1991.

[3] M. A. Cusumano and R. W. Selby, "Microsoft secrets," 1997.

[4] M. Fowler and M. Foemmel, "Continuous integration," *Thought-Works) http://www. thoughtworks. com/Continuous Integration. pdf*, vol. 122, p. 14, 2006.

[5] K. Gallaba, C. Macho, M. Pinzger, and S. McIntosh, "Noise and Heterogeneity in Historical Build Data: An Empirical Study of Travis CI," in *Proc. of the International Conference on Automated Software Engineering (ASE)*, 2018, pp. 87–97.

[6] K. Gallaba and S. McIntosh, "Use and misuse of continuous integration features: An empirical study of projects that (mis) use travis ci," *IEEE Transactions on Software Engineering*, 2018.

[7] M. Hilton, N. Nelson, D. Dig, T. Tunnell, D. Marinov *et al.*, "Continuous integration (ci) needs and wishes for developers of proprietary code," Corvallis, OR: Oregon State University, Dept. of Computer Science, Tech. Rep., 2016.

[8] M. Hilton, N. Nelson, T. Tunnell, D. Marinov, and D. Dig, "Trade-offs in continuous integration: assurance, security, and flexibility," in *Proceedings of Joint Meeting of the European Software Engineering Conference and the International Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2017, pp. 197–207.

[9] 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 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2016, pp. 426–437.

[10] J. Humble and D. Farley, *Continuous delivery: reliable software releases through build, test, and deployment automation*. Addison-Wesley Boston, 2011.

[11] N. Kerzazi, F. Khomh, and B. Adams, "Why do Automated Builds Break? An Empirical Study," in *Proc. of the 30th Int'l Conf. on Software Maintenance and Evolution (ICSME)*, 2014, pp. 41–50.

[12] A. Labuschagne, L. Inozemtseva, and R. Holmes, "Measuring the cost of regression testing in practice: a study of Java projects using continuous integration," in *Proc. of the International Symposium on the Foundations of Software Engineering (FSE)*, 2017, pp. 821–830.

[13] T. Rausch, W. Hummer, P. Leitner, and S. Schulte, "How Open Source Projects Use Static Code Analysis Tools in Continuous Integration Pipelines," in *Proc. of the International Conference on Mining Software Repositories (MSR)*, 2017, pp. 345–355.

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

[15] 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 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 2015, pp. 805–816.

[16] 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 *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*, 2017, pp. 183–193.

[17] D. G. Widder, M. Hilton, C. Kästner, and B. Vasilescu, "Im Leaving You, Travis: A Continuous Integration Breakup Story," in *Proc. of the International Conference on Mining Software Repositories (MSR)*, 2018, pp. 165–169.

[18] F. Zampetti, S. Scalabrino, R. Oliveto, G. Canfora, and M. D. Penta, "How Open Source Projects Use Static Code Analysis Tools in Continuous Integration Pipelines," in *Proc. of the International Conference on Mining Software Repositories (MSR)*, 2017, pp. 334–344.