A Reusable Concern for Continuous Integration Specifications

Puneet Kaur Sidhu

A thesis submitted to McGill University
in partial fulfillment of the requirements of the degree of

MASTER OF ENGINEERING

Department of Electrical and Computer Engineering

McGill University
Montreal

July 2019

Abstract

Continuous Integration (CI) is a broadly adopted 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 release engineers to customize *phases*, which define the sequential steps of build jobs that are triggered by changes to the project. While the 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 extending the framework for Concern-Oriented Reuse (CORE) to help developers reuse textual CI specifications based on popular sequences of phases and commands. To assess the feasibility of providing reuse suggestions in CORE, 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. 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 is broad and diverse. Hence, we cannot provide suggestions for Java-based projects as we had envisioned. However, we provide a proof-of-concept implementation for the reuse of CI specifications (without suggestions) by extending the feature model in CORE as well as the composition mechanism with the ability to handle textual CI specifications.

Abrégé

L'intégration continue (IC) est une pratique largement adoptée dans laquelle les modifications au code sont automatiquement construites et testées pour vérifier la régression telles qu'elles apparaissent dans un système de contrôle de versions (VCS). Les services IC permettent aux ingénieurs de mise en production de personnaliser des *phases* qui définissent les étapes séquentielles des tâches de construction déclenchées par des modifications apportées au projet. Bien que des travaux antérieurs fassent d'importantes observations sur l'adoption et l'utilisation de l'IC, on en sait encore peu sur les modèles de *réutilisation* dans les spécifications IC. Si la réutilisation est commune aux spécifications IC, nous envisageons d'étendre le cadre CORE (Concern-Oriented Reuse) pour aider les développeurs à réutiliser les spécifications textuelles IC basées sur des séquences fréquentes de phases et de commandes. Pour évaluer la faisabilité de proposer des suggestions de réutilisation dans CORE, nous effectuons une analyse empirique de l'utilisation de différentes phases et commandes à l'aide d'un échantillon de 913 spécifications de projets basées sur Java qui utilisent Travis CI, l'un des fournisseurs publics de services IC les plus populaires. Premièrement, nous observons que cinq phases parmi neuf sont utilisées dans 18% à 75% des projets. Deuxièmement, pour les cinq phases les plus populaires, nous effectuons un forage de règles d'associations afin de découvrir des motifs d'usages fréquents de phases, de commandes et de catégories de commandes. Nous observons que les règles d'association manquent d'évidence, de confiance ou de valeurs de montée suffisantes pour être considérées comme statistiquement significatives. Nos résultats suggèrent que l'utilisation de phases et de commandes dans les spécifications Travis CI est diversifiée. Par conséquent, nous ne pouvons pas fournir de suggestions pour les projets basés sur Java comme nous l'avions envisagé. Cependant, nous fournissons une implémentation démontrant la faisabilité de la réutilisation des spécifications IC (sans suggestions) en étendant le modèle de caractéristiques dans CORE et son mécanisme de composition avec la capacité de gérer des spécifications IC textuelles.

Related Publication

The part of the Empirical Analysis of Travis CI performed in this thesis has been published in the Proceedings of the IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER) 2019, pages 524-533:

Reuse (or Lack Thereof) in Travis CI Specifications: An Empirical Study of CI Phases and Commands [42] (Chapter 4). Puneet Kaur Sidhu, Gunter Mussbacher, and Shane McIntosh.

Acknowledgements

I use this opportunity to thank everyone who helped me towards the successful completion of my thesis. I would first like to thank my thesis supervisor Professor Gunter Mussbacher and co-supervisor Professor Shane McIntosh. The door to Prof. Mussbacher and McIntosh's office was always open for continuous guidance and support throughout my research and writing. They consistently pointed me in the right direction whenever they thought I needed their advice and allowed me to independently follow my research.

Other than my supervisors, I would like to thank my mates from Software Engineering and Software REBEL's lab. Keheliya Gallaba's continuous guidance and feedback helped me through a major part of my thesis. Ruchika Kumar and Nadine Bou Khzam's comments always helped me improve my work and kept me motivated. I am also immensely thankful to Nadine for working on the French version of the abstract of my thesis.

Finally, I must express immense gratitude to my parents and siblings for providing me with unfailing support and continuous encouragement throughout the two years of my study and through the process of researching and writing this thesis along with my friends in Montreal who made these two years great fun. This accomplishment would not have been possible without them.

Thank you.

Table of Contents

List of Tables

List of Figures

List of Programs

1 Introduction

Continuous Integration (CI) is a commonly adopted Software Engineering practice these days [9, 13, 15]. CI aims to ensure that each change to the software system is scanned by routine checks (e.g., automated compile, test, static code analysis) [21]. Automated CI services like Travis CI[1] integrate with GitHub[2] to facilitate this process by allowing release engineers 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`). Stakeholders 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 on GitHub. Travis CI provides free service for open source repositories on GitHub, and a paid service for

[1] `https://docs.travis-ci.com`
[2] `https://github.com`

private repositories.[3] According to a recent analysis conducted by GitHub, Travis CI is the most popular CI platform.[4] The `.travis.yml` 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]

Travis CI's `.travis.yml` is a technical specification that is subject to the same types of maintenance concerns as code [17]. One such key maintenance concern for `.travis.yml` files is reuse. Little is known about patterns of reuse in CI specifications, and in general, there is a lack of tool support for the reuse of Travis CI specifications. We envision extending the framework for Concern-Oriented Reuse (CORE) by building a CORE concern for Travis CI specifications to help developers reuse textual CI specifications.

CORE is a next-generation reuse technology inspired by multidimensional separation of concerns [33]. CORE builds on the fundamentals of Model Driven Engineering (MDE), Software Product Lines (SPL) [34], Goal modeling [3], and advanced modularization techniques offered by Aspect-Orientation [26] to define flexible software artifacts called *concerns* that promote model-based reuse.

A CORE concern [1, 2] is a unit of reuse that groups together modeling artifacts that describe properties and behavior of a domain of interest. Building a concern is a significant, time-consuming task, done by the concern designer, who is an expert of the concern's domain [40]. A concern provides a three-part interface comprised of the Variation, Customization, and Usage (VCU) interfaces which support and streamline the CORE reuse process [27].

1.1 Problem Statement

Thesis Statement. Despite the lack of tool support, reuse is a common activity when preparing and maintaining CI configuration files. The swaths of available CI configuration files can be mined to guide the extension of reuse frameworks to the CI configuration use case.

[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`

Recent studies have analyzed CI adoption (and omission) trends [18, 20], the types of failures that occur during CI [6, 25, 41, 47], and the outcomes associated with CI adoption [45]. Many CI "best practices" have been proposed.[6] However, despite the large scale adoption of CI, we know relatively little about the general practices followed in CI implementation and if developers are aligning their practices with "best practices".

Qualitative studies suggest that a frequent reason for the omission of CI is due to lack of familiarity with it [19]. Familiarity with the usage of CI can help developers to optimize their practices, project maintainers to make informed decisions about adopting CI, and researchers and tool builders to identify areas in need of attention [51]. There are many other systematic studies of CI systems, e.g., Vasilescu *et al.* [45] conducted a study on 246 projects, which compares several quality standards of projects that use and do not use CI. However, little is known about patterns and reuse in CI specifications. We submit that reuse of existing configuration snippets is a common activity, and better support for such activities is needed.

1.2 Thesis Methodology and Contribution

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 thesis performs an empirical analysis of which phases and commands are being used in real-world Travis CI specifications and provides a proof-of-concept implementation that uses the results of the empirical study to show feasibility for the envisioned reuse tool for Travis CI specifications. The proof-of-concept implementation defines a reusable concern for Travis CI to help release engineers automatically create specification files for Travis CI. For this, we create a feature model [24] using the dominant phases and commands of Travis CI, which we discover through empirical analysis of Travis CI specification files. Furthermore, we aim to provide suggestions on popular sequences of phases and commands based on our empirical study to help the release engineer with the specification of the desired Travis CI file. Once the release engineer has selected his desired features, which are realized by corresponding Travis CI specification files, our novel composition mechanism for Travis CI automatically merges the corresponding specification files into a partial Travis CI specification

[6] `https://martinfowler.com/articles/originalContinuousIntegration.html`

file. This partial file can then be customized further by the release engineer. The implementation of our Travis CI composition mechanism requires improvements to the CORE metamodel to support particularities of the feature selection process for Travis CI features.

For this empirical 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 [17]. Using these projects, we set out to study the phases and commands that are frequently used together in the `.travis.yml` files, i.e., Travis CI specification files. Using 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 envisioned tool

The study thus 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. Hence, irrespective of the patterns which would have been useful to provide suggestions in the feature selection, we still contribute to CORE by:

- Defining a feature model for Travis CI with all the dominant phases and commands as per our empirical study
- Extending the CORE metamodel to allow the same feature to be selected multiple times in a specific order from a feature model
- Introducing a metamodel for Travis CI specification files which depicts their generic structure
- Implementing a proof-of-concept composition mechanism for Travis CI specification files which allows to create `.travis.yml` files which can be further customized by the release engineer
- Extending the CORE metamodel with support for the selection of related features

1.3 Thesis Overview

The thesis is organized as follows:

- Chapter 2: Background — This chapter first discusses the terminology used in our thesis concerning CI and then describes the concept of CORE and related paradigms such as reuse, concerns, MDE, SPLs, and Feature Models.

- Chapter 3: The Design of the Empirical Study and Proof-of-Concept — This chapter gives details on how we conduct our analysis and all the different techniques and concepts we use to derive conclusions. We start with our analyses of 913 travis files from Java projects taken from GitHub. We try to parse these files and build a structure where we could understand how different phases and commands are used in these projects. We then interpret the usage of phases and commands to know if there are any patterns or similarities. Furthermore, we describe how the proof-of-concept implementation of our envisioned tool uses the results of our empirical study to support the reuse of Travis CI specification files.

- Chapter 4: Empirical Study Results (Advocatus Diaboli) — In this chapter, we take "a devil's advocate" approach to describe the lack of evidence for useful reuse patterns in Travis CI specifications. We address arguments that question our negative findings by explaining how we have taken care of these potential shortcomings to strengthen our analysis.

- Chapter 5: Concern-Oriented Reuse of CI Specification Files — This chapter explains how to create a concern for Travis CI specification files using the dominant features as in the phases and commands in Travis CI files. The release engineer can use this concern to select the required features. Using our proof-of-concept composition mechanism, the corresponding specification files of those features are then used to automatically create a partially completed `.travis.yml` file which may be further customized by the release engineer. The required improvements to the CORE metamodel for the reuse of the Travis CI specification files are discussed.

- Chapter 6: Related Work — In this chapter, we situate our thesis with similar prior research. Thus, we explain the relevance of our work in the field of Software Reuse and Continuous Integration.

- Chapter 7: Conclusions: This chapter summarizes the findings and contributions of our thesis along with a discussion of future work.

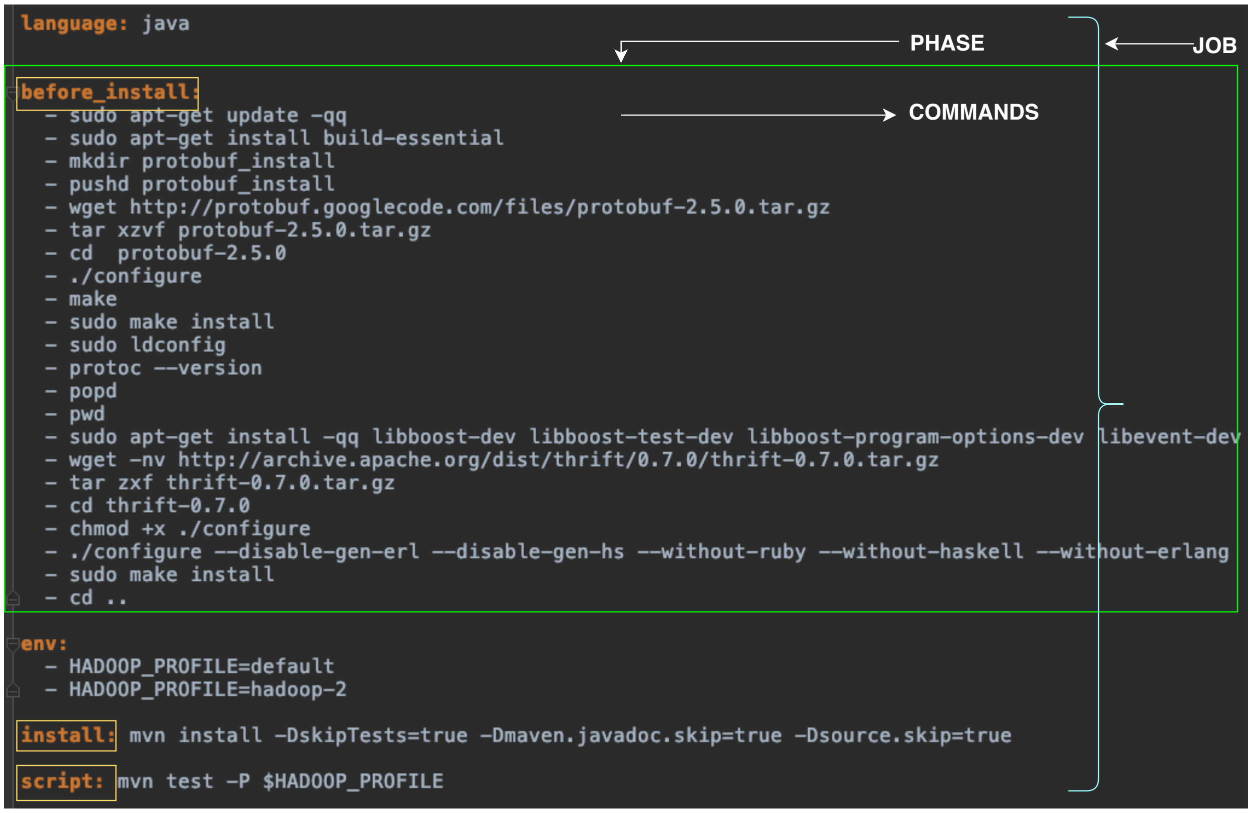
2
Background

In this chapter, we describe terminology related to Continuous Integration (CI) and Concern-Oriented Reuse (CORE).

2.1 Continuous Integration

In this thesis, we use common build terminology in the Travis CI context. Figure 2.1 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 [16].



```
language: java

before_install:
  - sudo apt-get update -qq
  - sudo apt-get install build-essential
  - mkdir protobuf_install
  - pushd protobuf_install
  - wget http://protobuf.googlecode.com/files/protobuf-2.5.0.tar.gz
  - tar xzvf protobuf-2.5.0.tar.gz
  - cd  protobuf-2.5.0
  - ./configure
  - make
  - sudo make install
  - sudo ldconfig
  - protoc --version
  - popd
  - pwd
  - sudo apt-get install -qq libboost-dev libboost-test-dev libboost-program-options-dev libevent-dev
  - wget -nv http://archive.apache.org/dist/thrift/0.7.0/thrift-0.7.0.tar.gz
  - tar zxf thrift-0.7.0.tar.gz
  - cd thrift-0.7.0
  - chmod +x ./configure
  - ./configure --disable-gen-erl --disable-gen-hs --without-ruby --without-haskell --without-erlang
  - sudo make install
  - cd ..

env:
  - HADOOP_PROFILE=default
  - HADOOP_PROFILE=hadoop-2

install: mvn install -DskipTests=true -Dmaven.javadoc.skip=true -Dsource.skip=true

script: mvn test -P $HADOOP_PROFILE
```

Figure 2.1: An example `.travis.yml` specification from the Dlanza1 project

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 the 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.

2.2 Concern-Oriented Reuse

We strive to increase productivity by building reusable software artifacts on the basis of the empirical study of phases and commands of Travis CI. By definition, software *reuse* is the process of creating software systems from existing software rather than building software systems from scratch [29]. Alam *et al.* [1] suggest that in order for reuse to be maximally effective, a new, broader unit of reuse that incorporates all design solutions targeted at solving a design problem is needed. They call this new unit of reuse a *concern* and the reuse technique Concern-Oriented Reuse (CORE). Concern Driven Development (CDD) [1] seeks to address the challenge of how to enable broad-scale, model-based reuse. This thesis applies the CORE techniques supported by Model Driven Engineering (MDE) [37] and Software Product Lines (SPLs) [34]. As CI strives to achieve automation, so does *MDE*. One approach for MDE is the Model Driven Architecture (MDA) which is a venture of the Object Management Group (OMG).[7] MDA provides guidelines for structuring software specifications that are expressed as models. In general, MDE aims to simplify how systems are built from scratch by allowing domain concepts to be captured with the most appropriate modeling formalisms, describing properties of the application domain at the right abstraction level, and then transforming models to lower-level, more detailed models or executable code.

On the other hand, an *SPL* is a set of software intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular domain of interest and that are developed from a common set of core assets in a prescribed way. The set of systems described by an SPL is commonly called a family of systems and a particular system is called a member of this family. Common examples of SPLs are mobile operating systems and the Linux Kernel. An SPL generally consists of three kinds of artifacts, representing the problem space, the solution space, and the mappings between problem and solution spaces [14]. Artifacts in the solution space represent design and implementation of all members of the family. The problem space, on the other hand, is comprised of all the features for the family members. Typically, the problem space is captured with feature models [24]. A feature is a characteristic which shows the behavior of a software component. Features are used in SPL engineering to specify and communicate commonalities and differences of the systems in the software family among stakeholders and to guide the structure,

[7] `https://www.omg.org`

reuse, and variations of the family. SPL engineering promotes development for reuse using domain analysis.[8] Domain analysis, the systematic exploration of software systems to define commonalities and differences, comprehends the features and capabilities of a class of a related software system [24] and is often captured in feature models.

Feature Models are widely accepted means of capturing commonality and managing variability within SPLs. They are modeled as feature diagrams, i.e., tree-like structures consisting of nodes that represent features of a modeled SPL and their relationships [10]. Features are those attributes of a system that directly affect end-users. The relationships among the features determine if the features are mandatory or optional and mutually exclusive or inclusive. An example of a feature model for *Authentication* is shown in Figure 2.2. A selection of features that describes a particular member of the software family is called a *configuration*. Each feature has its realizations associated with it which describe how that feature is implemented in the formalisms chosen for the problem domain. In the context of CORE, these realizations are called *Realization Models*. In our case in the context of Travis CI, we have a textual specification file thus each feature is realized by a 'string' component.

A *CORE concern* consists of all relevant features of a problem domain, is a software artifact that enables broad-scale model-based software reuse, and groups together software artifacts describing properties and behavior related to any area of interest to a developer at different levels of abstraction. Extensive knowledge of the character of the concern is needed to be able to establish its user-relevant features, to model the common properties and variation of all features of a concern in any respective level of abstraction, and to specify the impact of variants on high-level stakeholder goals and qualities. This is often ensured by creating requirements, design, and implementation models that (i) realize the features of the concern using the most appropriate modeling notations and programming languages, and (ii) are eventually refined into executable specifications [8].

A *concern* provides three interfaces [1] that represent different stages of the reuse process. The term, interfaces, is used because people with different roles interact with the artifact during different activities of the development process through the appropriate interface to achieve a desired result. Each interface targets a different dimension of reuse, and together they streamline the reuse process [27]. The three interfaces employed in the reuse process are called VCU interfaces and

[8] `http://stg-tud.github.io/sedc/Lecture/ws16-17/6-SPL.pdf`

explained below:

- *The Variation Interface* allows us to see the impact of different variations of a concern on high-level goals, qualities, and requirements. This is facilitated by a *feature model* that specifies all features of the concern and connected goal models that capture the impact of features. In this thesis, we build a feature model for Travis CI specification files. The definition of goal models is left for future work.

- *The Customization Interface* describes the way each variant of a concern can be adapted to the needs of a specific application, i.e., a specific reuse context. A concern is built as generally as possible to increase reusability. This means that the concern includes partial elements that can only be completed with application-specific information available at the time of a reuse. The customization interface clearly specifies all partial elements that need to be completed.

- *The Usage Interface* describes how the reusing application can finally access the structure and behavior provided by the reused concern, similar to what the set of public operations represents for a class in the object-oriented methodology.

Consequently to use the Travis CI concern, a release engineer must (i) select the feature(s) from the variation interface (the Travis CI feature model; in our case we do not cover impact analysis of features, hence the release engineer selects features as per his needs), then (ii) adapt the generated specification file models to the application context by mapping generic customization interface elements to application-specific model elements (mostly parameters in the Travis CI specification file), and finally (iii) use the behaviour provided by the complete Travis CI specification file as defined in its usage interface (i.e., run the Travis CI specification file).

As a concrete example of a CORE concern, consider the example of *Authentication*. The concern user opens the *Authentication* concern [44] and selects the desired features from the feature model shown in the left part of Figure 2.2. While interacting with the feature model, the impacts resulting from the current selection are constantly updated. An example is shown in the right part of Figure 2.2 where we see the impact of the current selection of features from the feature model on security. Additional impact models capture other system qualities and non-functional requirements. When a satisfactory selection has been made, a composition algorithm composes all design models of the selected features together to produce a detailed design model for this specific configuration. The result of this composition is shown at the bottom of Figure 2.3. In this case, the features `Blocking`, `PasswordExpiry`, and `Password` are selected. The composed model is still generic as it contains partial elements (e.g., |`ProtectedClass` and |`Authenticatable`). The concern user is then presented with a mapping view as shown in Figure 2.3 that allows the concern user to customize the generic *Authentication* concern to his specific needs by establishing mappings between the partial model elements in the concern and the application model. In this case, the software is a simple Banking application, and the concern user wants to enforce authenticated access to accounts. Therefore, |`Authenticatable` maps to the `Customer` class, |`ProtectedClass` to `Account`, and |`protectedMethod()` to `withdraw()`, `deposit()`, and `transfer()`. Once the customization is completed, the composition algorithm composes the application model with the *Authentication* concern to yield the combined structure and behaviour of the system. Now, the functionality of the *Authentication* concern (e.g., `changePassword()` or `authenticate()` in the `AuthenticationManager`) is available in the Banking application.

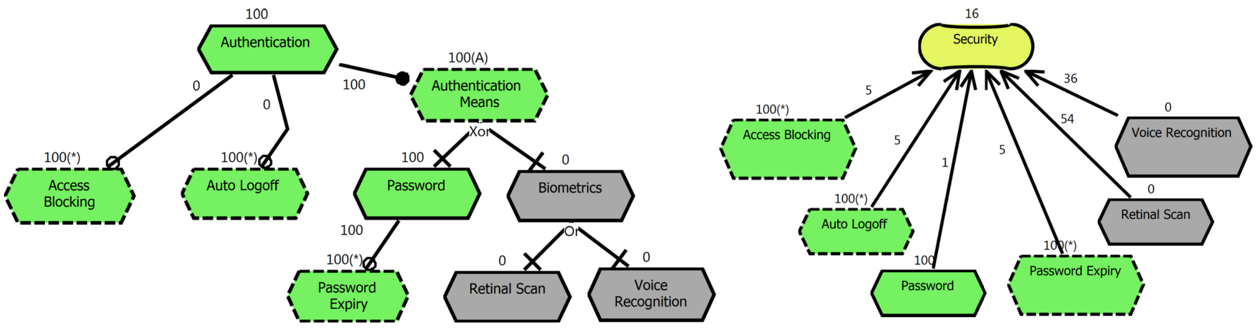


Figure 2.2: Feature and impact modelling and analysis of *Authentication* concern [44]

As can be seen from the above example, CORE requires a composition mechanism for two situations. First, the realization models of all selected features need to be combined to automatically generate a generic, reusable artifact. This artifact is tailored, because it only includes the realization models of selected features but not those of unselected features. Furthermore, this reusable artifact is generic, because it still contains partial elements. Once the partial elements are mapped to elements from the application domain, the composition mechanism needs to combine the generic artifact with the application artifact. In this thesis, we build a proof-of-concept implementation of the composition mechanism for Travis CI specification files.

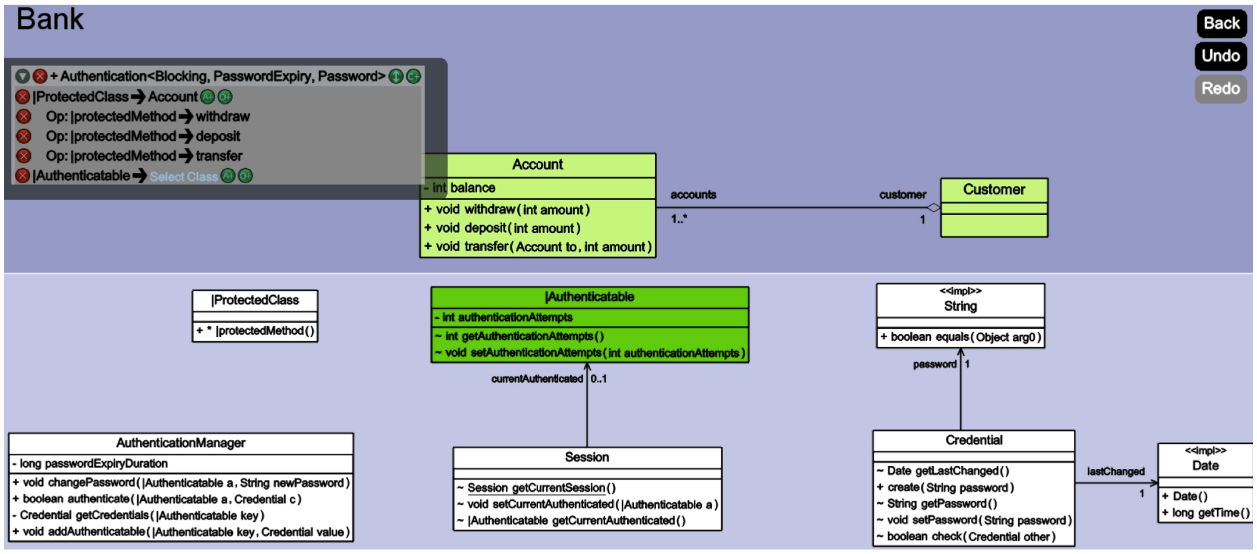

Bank
Back
Undo
Redo
+ Authentication<Blocking, PasswordExpiry, Password>
|ProtectedClass → Account
Op: |protectedMethod → withdraw
Op: |protectedMethod → deposit
Op: |protectedMethod → transfer
|Authenticatable → Select Class
Account
- int balance
+ void withdraw(int amount)
+ void deposit(int amount)
+ void transfer(Account to, int amount)
accounts
1..*
customer
1
Customer
|ProtectedClass
+ * |protectedMethod()
|Authenticatable
- int authenticationAttempts
~ int getAuthenticationAttempts()
~ void setAuthenticationAttempts(int authenticationAttempts)
currentAuthenticated
0..1
<<impl>>
String
+ boolean equals(Object arg0)
password
1
AuthenticationManager
- long passwordExpiryDuration
+ void changePassword(|Authenticatable a, String newPassword)
+ boolean authenticate(|Authenticatable a, Credential c)
- Credential getCredentials(|Authenticatable key)
+ void addAuthenticatable(|Authenticatable key, Credential value)
Session
~ Session getCurrentSession()
~ void setCurrentAuthenticated(|Authenticatable a)
~ |Authenticatable getCurrentAuthenticated()
Credential
~ Date getLastChanged()
+ create(String password)
~ String getPassword()
~ void setPassword(String password)
~ boolean check(Credential other)
lastChanged
1
<<impl>>
Date
+ Date()
+ long getTime()

Figure 2.3: Reusing *Authentication* in `TouchRAM` [44]

The combination of the generic artifact with the application artifact constitutes the actual reuse, which results in a clear *reuse hierarchy* because smaller concerns are combined with larger concerns, which are then combined with even larger concerns until eventually the application is complete.

Hence, for implementing Concern Driven Development (CDD), it is desirable to restrict the expressiveness of the modeling language to only allow the construction of correct and relevant models. The relevance of a modeling language is, to a large degree, determined by its abstract syntax. As a result, the abstract syntax of modeling languages ought to be constrained in agreement with the problem domain. The abstract syntax of well known conceptual modeling languages like User Requirements Notation (URN) [4, 22], Business Process Model and Notation (BPMN) [48], Unified Modeling Language (UML) [32], and Entity Relationship (ER) model [12] is depicted by the language's metamodel. As the name suggests, a *metamodel* is a model of an other model, i.e., it defines a modeling language. For example, the UML is itself defined by a Meta Object Facility (MOF) model.[9] MOF is a language standardized by OMG for the specification of metamodels.[10] Similarly, the metamodel for CORE defines the abstract syntax required to follow CDD [1]. This thesis extends the CORE metamodel to support new functionality required for the feature model of Travis CI.

[9] `https://www.omg.org/spec/UML/2.5.1/PDF`
[10] `https://www.omg.org/spec/MOF/2.5`

Finally, to create realization models for Travis CI specification files, this thesis defines a metamodel for the specification files to be able to create a `.travis.yml` specification file with our proof-of-concept composition mechanism, which composes the realization models of all selected features. This composition is often called *weaving*.

2.3 Summary

In this chapter, we discuss the key technologies and concepts related to Travis CI and CORE used and mentioned in this thesis. We move forward to the next chapter and explain how we conduct the empirical study of Travis CI phases and commands and how we use the results of our empirical study to support reuse of Travis CI specification files with the help of our proof-of-concept implementation.

3 The Design of the Empirical Study and Proof-of-Concept

In this chapter, we provide an overview of the design of our empirical study 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. Furthermore, we describe the proof-of-concept of our envisioned concern-oriented tool for Travis CI specification files.

3.1 Data Extraction

In this study, we analyze 913 open source Java projects, which are a subset of the corpus of projects studied by Gallaba and McIntosh [17]. Gallaba and McIntosh collected GitHub repositories that use Travis CI to implement continuous integration. These projects were selected for analysis because they are active, large, non-forked and non-duplicate projects that contain a valid Travis CI specification file in their root directory.

3.2 Data Analysis and Validation

Figure 3.1 shows an overview of the process that we followed for our analysis. As shown, we automatically and manually parse the `.travis.yml` files of the selected projects and store the parsed output. Bashlex is a python parser for bash that creates an Abstract Syntax Tree (AST) for the input data provided to it.[11] We parse through the AST to collect the node information that corresponds to the CI phases and commands. We notice that even after customizing the parser as much as possible, the output produced is not clean and complete because of the indefinite and random structure of the Travis files. Therefore, we manually inspect each line after running the parser script for all phases on the entire data-set. Then, we analyze and evaluate reuse patterns using Association Rule Mining (ARM).

The aim of ARM is to determine rules based on co-occurrence patterns in the data-set. Table 3.1 defines the commonly used ARM measures[12] that we use to estimate the interestingness of mined association rules.

Table 3.1: Measures of the interestingness of association rules

Measure	**Definition**	**Formula**
support(X, Y)	An estimate of the popularity of a rule. The proportion of projects in which an association (i.e., X and Y) is present.	$support(X, Y) = \frac{\lvert X \cap Y \rvert}{\lvert Dataset \rvert}$
confidence(X, Y)	An estimate of the strength of the implication of the rule. In projects that include X, what is the proportion that also include Y?	$confidence(X, Y) = \frac{support(X,Y)}{support(X)}$
lift(X, Y)	An estimate of the likelihood of the rule due to a spurious correlation. More specifically, the ratio of the measured rate of co-occurrence of X and Y (i.e., support(X, Y)) to that which would be expected due to random chance (i.e., support(X) x support(Y)).	$lift(X, Y) = \frac{support(X,Y)}{support(X) \times support(Y)}$

[11] `https://github.com/idank/bashlex/blob/master/README.md`

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

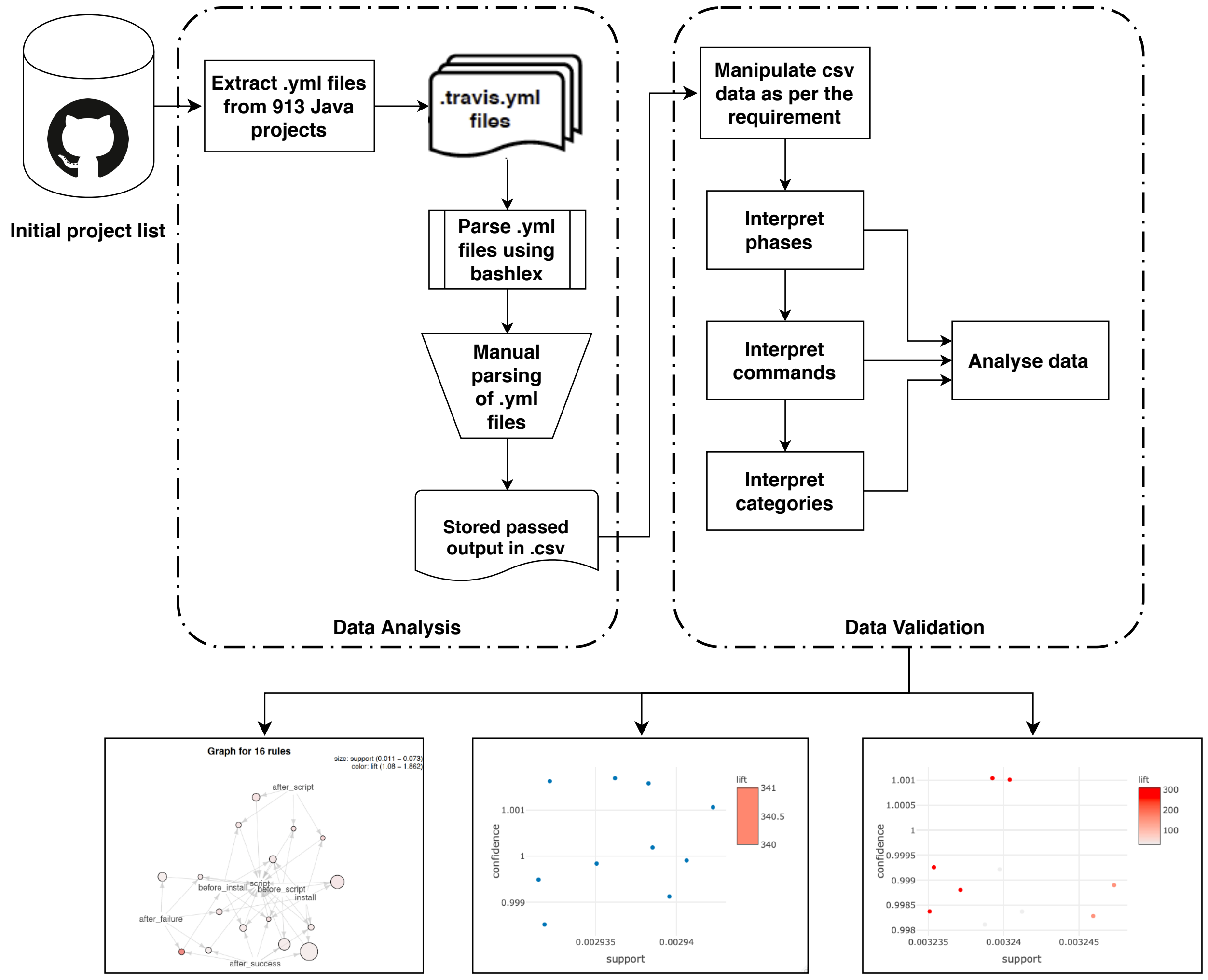


Figure 3.1: An overview of our approach to study the `.travis.yml` files and analyze the phases and commands inside them

3.3 Reuse

Figure 3.2 provides an overview of our approach to implement the reuse of `.travis.yml` specification files. Considering the results of the empirical study, we create a feature model of CI specification files, which contains the *commonalities* and *variations* of the functions provided by Travis CI. We also extend the CORE metamodel to support additional functionality in feature models, since Travis CI requires the ability to (i) choose the same feature multiple times and in an ordered fashion; and (ii) provide suggestions for features that occur together as per the observed patterns of reuse in Travis CI specifications. Thus, a feature must support having co-occuring features from the feature model with a certain likelihood. We also create a metamodel for Travis CI specification files that describes the relevant components of a `.travis.yml` specification file (i.e., phases, commands, and arguments). With this in place, we create a proof-of-concept composition mechanism that binds the features selected by the release engineer from the feature model and creates a partially completed `.travis.yml` file to which the release engineer may add details if required.

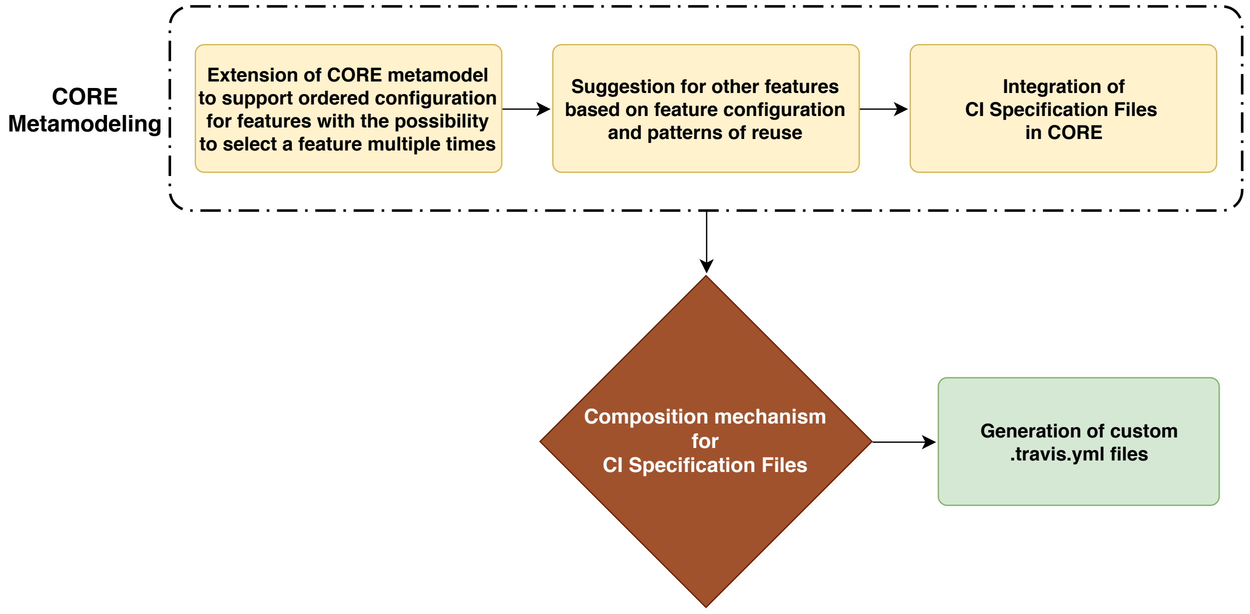

CORE Metamodeling
Extension of CORE metamodel to support ordered configuration for features with the possibility to select a feature multiple times
Suggestion for other features based on feature configuration and patterns of reuse
Integration of CI Specification Files in CORE
Composition mechanism for CI Specification Files
Generation of custom .travis.yml files

Figure 3.2: Overview of our approach to implement the reuse of `.travis.yml` files in CORE

3.4 Summary

This chapter gives an overview of our empirical study (i.e., data extraction from 913 open source Java projects from GitHub, analysis, and validation using association rule mining required to perform empirical analysis of Travis CI phases and commands) along with how we use the results of the empirical analysis to implement support for the reuse of Travis CI specification files by creating a Travis CI CORE concern.

In the next chapter, we present results of our empirical analysis and address arguments that may arise with respect to our negative finding.

An earlier version of the work in this chapter appears in the proceedings of the International Conference on Software Analysis, Evolution, and Reengineering (SANER) 2019 [42].

4 Empirical Study Results (Advocatus Diaboli)

Following the study design laid out in Chapter 3, we do not observe any patterns of phases and commands of sufficient interestingness to provide suggestions in our envisioned tool. Nonetheless, support for the reuse of Travis CI specifications can still be provided by integrating CI specification files into CORE and providing a composition mechanism for them. We summarize our results below and discuss the details in the remainder of this chapter.

Summary of Results. The results of our analysis suggest a *negative result*, i.e., we are not able to find patterns of phases and commands in Travis CI specifications for Java projects with ARM that are statistically 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. However, using the results of our empirical study, we create a feature model of Travis CI. Furthermore, we enhance the metamodel of CORE and create a new metamodel for Travis CI specification files to enable the integration of Travis CI specifications in CORE. Finally, we build a composition mechanism that generates Travis CI specification files based on the feature selection in the CORE feature model of Travis CI.

To structure our empirical analysis, we follow the **Advocatus Dioboli** (AD, i.e., **Devil's Advocate**) approach. The AD 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. Proponents for canonization would then build a defense case to answer each of the points raised by the AD.[13]

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 sections, the AD questions our methodology and results, and we address those arguments. Consequently, we state the details of our approach, the precautions, and actions taken to support our analysis.

4.1 Experiment Approach

Below the AD presents arguments related to the experiment setup.

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

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

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 [17] 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 4.1. 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.

Table 4.1: 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**

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 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 4.2. 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 4.3 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 4.4. 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 4.5 shows the most popular category per phase and its usage as per the number of projects. For example, 'builders' is the most popular command

Table 4.2: 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

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.

Table 4.3: 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

We analyze all the above information as the source for ARM. Our scripts and parsed data are available online at the following address: `https://figshare.com/s/96287a53e4ad7e55a906`

Table 4.4: 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

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 4.5: 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.[14] 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

[14]`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`

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 lower values of confidence going as low as 0.4 for all phases. To reinforce well-built rules with considerable lift value, we 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 as with confidence lower than 0.7 we only find rules with lift values that are too low. We show the results of the same in the remainder of this chapter.

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 Cohen’s 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.

4.2 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 4.6 and Figure 4.1.

Table 4.6: Association 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 4.2. 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 find out the support, confidence, and lift values for all these relationship rules. We derive various rules as shown in Table 4.6 and Figure 4.1 shows the plots that depict the same trends.

Results. The percentages of projects in Table 4.2 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 4.6 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 4.2.

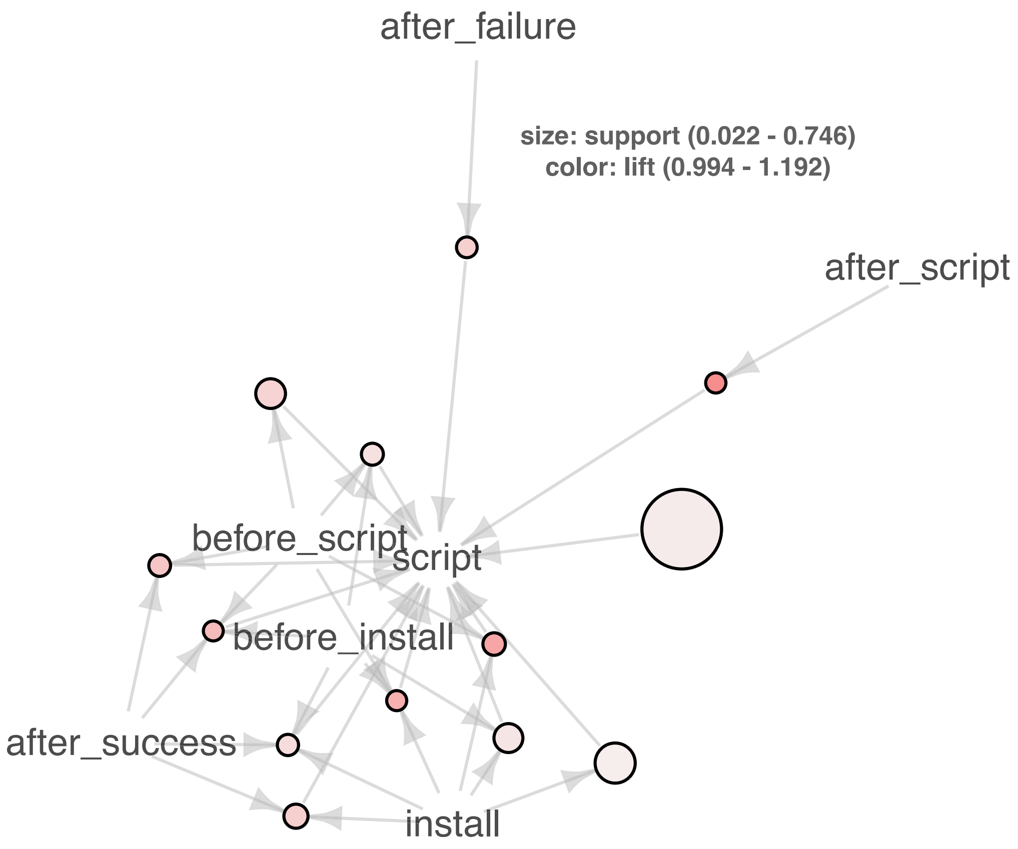


Figure 4.1: Association rules for phases used in all projects

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 4.1 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.

Results. We find various rules for all the five phases that we study. We state our results in Figure 4.2 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.

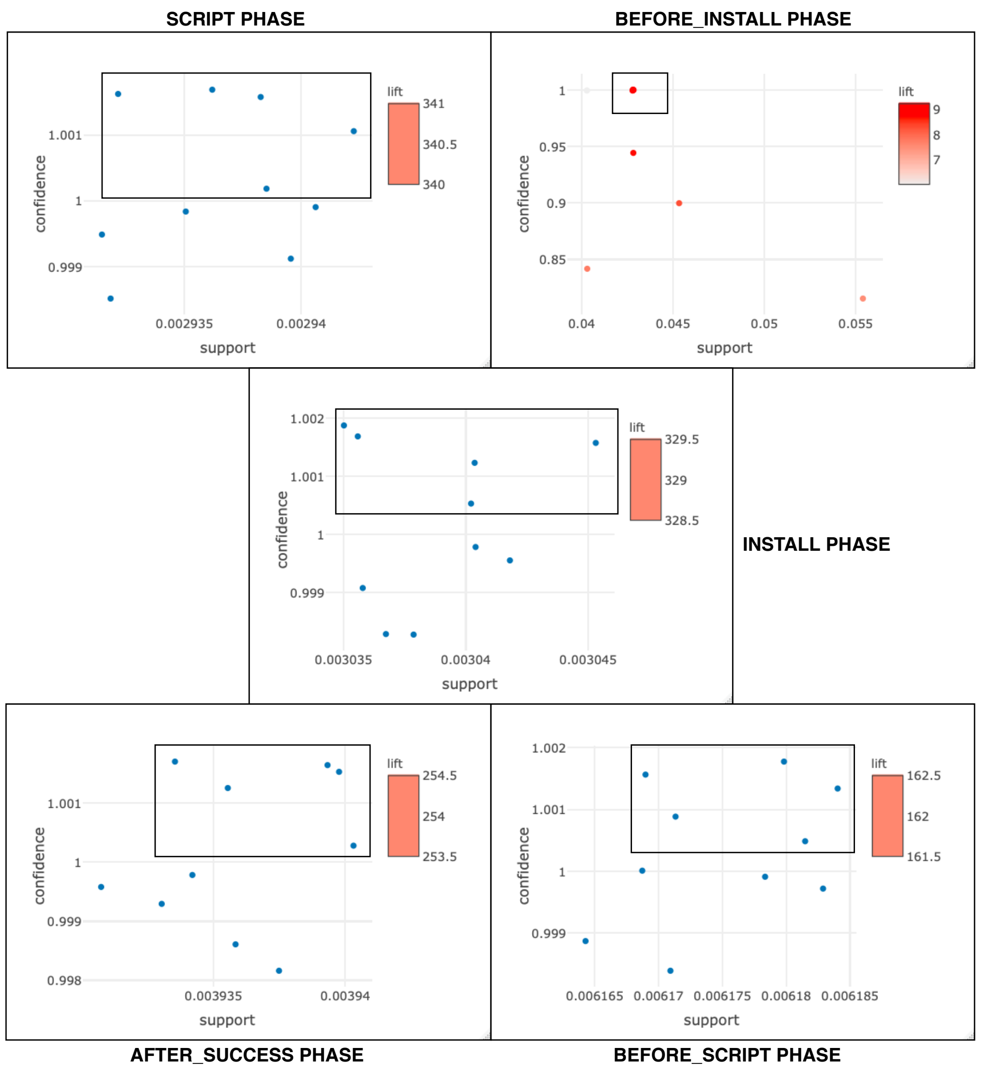


Figure 4.2: Association rules for commands used in all projects per phase

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 4.4, we show which categories of commands appear per phase across all projects. More details are shown in Figure 4.3.

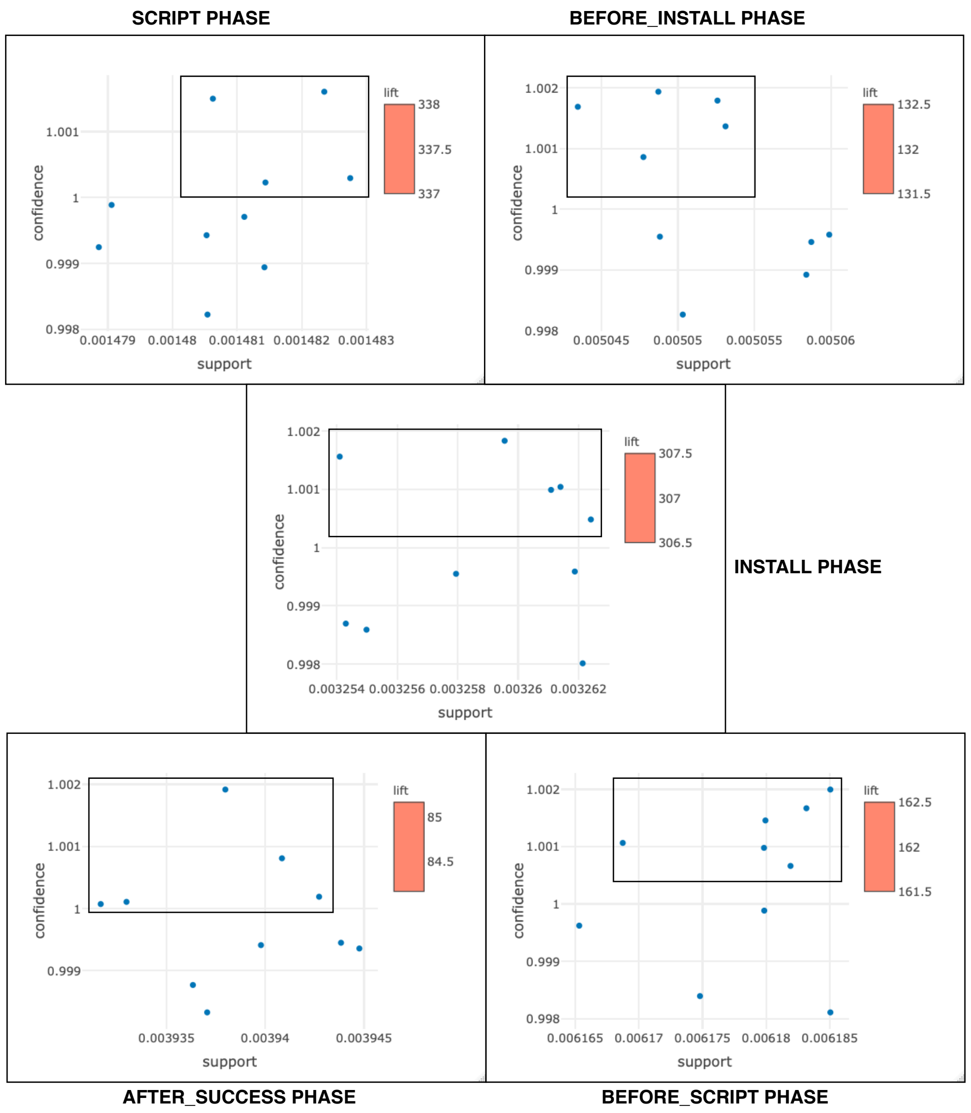


Figure 4.3: Association rules for categories used in all projects per phase

Results. Although, we see that the category usage across projects is significant as stated in Table 4.3, we do not find strong association rules supported by high values of support, confidence, and lift. By analyzing Figure 4.3, 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 4.4 and 4.5. 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 4.7 and 4.8. Table 4.7 shows only the top 5 most frequent rules to reduce clutter. We look over these trends generated by ARM and analyze the results.

Table 4.7: Association 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 4.7, 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 4.8. The "builders" category is the most common category as we already know from Table 4.3 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. While the second most frequent rule occurs in approximately 25% of the total projects and the sixth most frequent rule still occurs in more than 10% of the total projects, the vast majority of projects still cannot benefit from these rules and they are so abstract that the benefits derived would be small in our envisioned tool. For example, if we were to use the second most frequent rule in our tool, the developer could select a command from the "not mutate" category ('echo', 'cd', 'ls', 'pwd', 'sleep', ...) and the tool could then tell the developer that it is likely that a command from the "builders" category ('mvn', 'gradlew', 'ant', 'gradle', 'make', ...) should be used in the same or some other phase. Hence, the rule is of limited use for the developer, because the tool cannot inform the developer about which exact commands in which exact phases are likely to appear together.

Table 4.8: Association 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

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.

4.3 Summary

This chapter presents the results of our empirical study, the precautions, and actions taken to support our analysis. We find out that 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 and 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 envisioned tool.

In the next chapter, we discuss the details of our envisioned tool and how we use the results of our empirical study to create a reusable concern for Travis CI and implement the VCU interfaces to support generation and reuse of Travis CI specification files.

5

Concern-Oriented Reuse of CI Specification Files

This chapter states our work towards creating a reusable concern for CI specification files. We proceed with the results of our empirical study and keeping in mind the concept of reuse. We start by presenting a feature model for Travis CI specification files, which is comprised of the five most frequently occurring phases, their topmost frequently used category of commands and their five most frequently occurring commands. We show this feature model in Figure 5.2. We state all the stages of the creation of the feature model in Section 5.1.

In Section 5.2, we move further in the process of reuse and extend the metamodel of CORE to support our need to choose the same feature (i.e., same command) multiple times in a feature model, because one command can be used more than once in each phase (e.g., 'npm install' in the 'before_install' phase of Program 5.1). We also add to the CORE metamodel the functionality of providing suggestions for features that are selected together as per association rule mining (see Chapter 4).

After extending the metamodel of CORE, we specify the metamodel for CI specification files as shown in Figure 5.9. We give the details of the process followed in Section 5.3.

Finally, after building the infrastructure in terms of metamodels for our study, we work towards creating a composition (weaving) mechanism explained in Section 5.4 to compose the models of CI specification files to create custom CI specification files.

5.1 Feature Model of Travis CI and its Realization Models

Feature models are a widely accepted means of capturing commonality and variation in SPLs [10]. They consist of a tree like structure with nodes that represent features of a particular concern and their interrelationships. Figure 5.1 shows a sample feature model of the *mobile phone domain* [7]. From this feature model, we understand that the *nodes* represent the features/functionality offered in a mobile phone. According to the model, all phones must have a *Screen* and *Calls* feature, which is denoted by the *Mandatory* relationship (denoted by a filled black circle). A phone may or may not have a *GPS* and *Media* feature which is represented by the *Optional* relationship (denoted by a non-filled white circle). The *Alternative* relationship (denoted by a non-filled white quarter-circle) among features under *Screen* denotes that if *Screen* is selected then exactly one of the three underlying features must be selected (i.e., it is an XOR relationship). The *Or* relationship (denoted by a filled black quarter-circle) under *Media* explains that either one or both of *Camera* and *MP3* features must be selected if *Media* is selected (i.e., it is an IOR relationship). Also, if the *Camera* feature exists in a phone, then that phone must have a *High Resolution* Screen. This is indicated by the *Requires* relationship between *Camera* and *High Resolution* features. Lastly, we get to know from the *Excludes* relationship that if a phone has the *GPS* feature, then that phone must not have a *Basic* Screen and vice versa. Thus, a mobile phone will have different features depending on its configuration.

Similarly, we construct a feature model to describe the commonalities and variations in the features of Travis CI specification files. We capture the most prominent phases and commands of Travis CI in the feature model. However, since the release engineer may freely choose which phases and commands to include in a specification file, all features in the feature model are optional.

We start by defining the *Variation Interface* for Travis CI. Since, an investigation of the impact

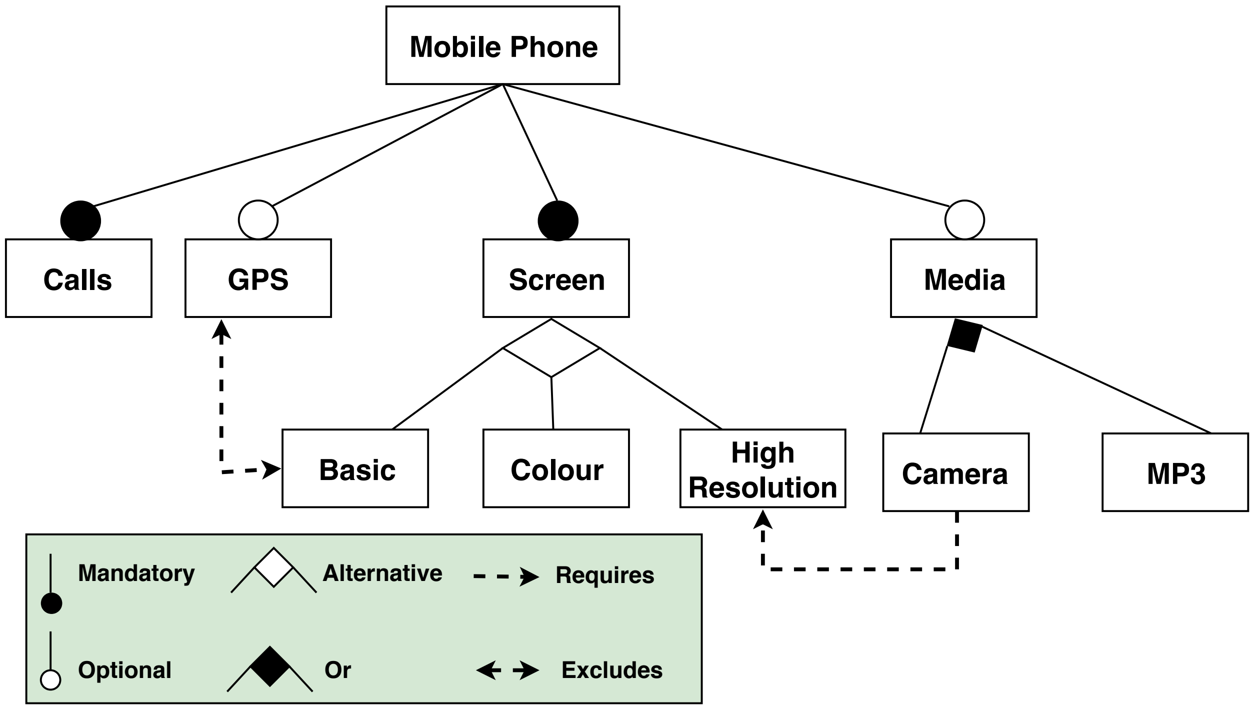


Figure 5.1: A sample feature model [7]

of Travis CI features on system characteristics such as security or performance is outside the scope of this thesis, we focus on the feature model part of the Variation Interface and leave the goal model part for future work. We first add two features, *JobProcessing* and *JobReporting*, as per the functions performed by the underlying phases. A *JobProcessing* feature is responsible for processing build jobs and a *JobReporting* feature is responsible for reporting on the status of build jobs. We see from the most frequently used phases as described in Table 4.2 that the 'script', 'before_script', 'install', and 'before_install' phases fall under the *JobProcessing* category and that 'after_success' falls in the *JobReporting* category. We separate the *JobProcessing* category into *RunScript* and *RunInstall* because of the nature of phases under this category. Under the *RunScript* category, we allow the release engineer to choose the 'script' phase. Under the 'script' phase, as determined from Table 4.4, we know the most used command type is 'builders'. Therefore, we show the top five commands used under 'builders' category as per Table 4.5. We use only the *Optional* relationship among all features as no phase or command in a `.travis.yml` file is *Mandatory*. Also, there are no conditional constraints that one phase/command should appear if another does or does not, therefore, we do not have *Alternative*, *Or*, *Requires*, or *Excludes* relationships.

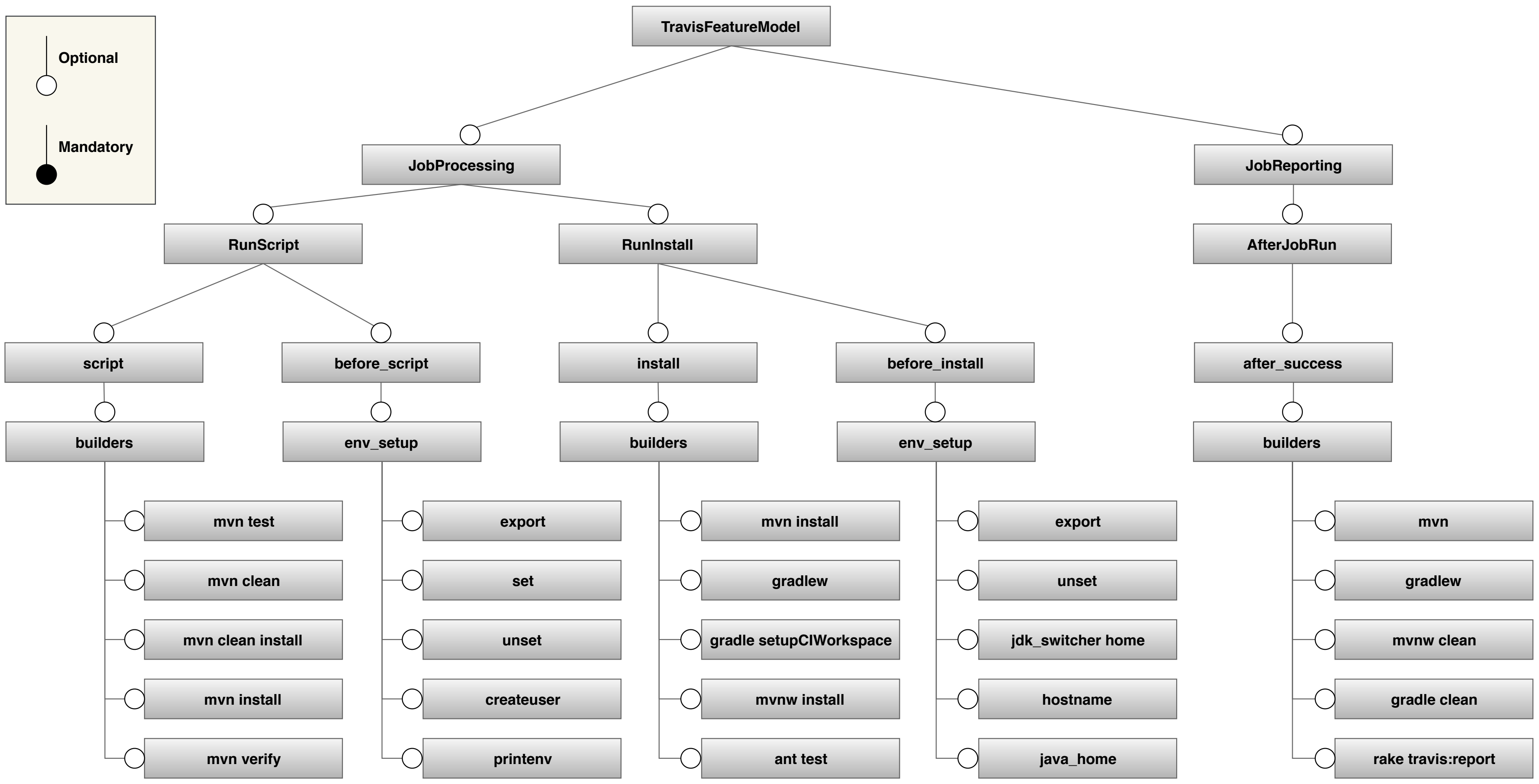


Figure 5.2: Feature Model of Travis CI

We follow a similar approach for all the five phases and create a feature model for Travis CI as shown in Figure 5.2. Thus, our feature model shows all the dominant phases and commands used in Travis CI specification files. This feature model forms the basis for our proof-of-concept as the release engineer will have to select the desired features from this model to automatically create a `.travis.yml` file. To build a textual specification file, we need to define how each feature from the feature model is realized in it, thus specifying the realization models for the features of Travis CI. The features of Travis CI in its feature model are simply the phases and commands defined by the release engineer. For example, the feature 'mvn test' in the script phase of the feature model in Figure 5.2 is realized by the specification file "script: mvn test |a" as shown in Figure 5.3. Since file header and arguments of Travis CI commands are completely application-specific, they are specified as partial elements in the realization models and hence have to be defined by the release engineer. The file header and arguments are identified by a '|' symbol and are hence part of the *Customization Interface* of the reusable artifact. For example, the feature 'mvn test' in the 'script' phase uses the partial element |a for its arguments. Note that the '|' symbol is chosen, because it is the symbol typically used in the CORE reuse process to denote partiality and a check of the 913 project used for the empirical study in Chapter 4 reveals that the '|' symbol does not appear in any of the .travis.yml files. After the definition of features and realization models for Travis CI features, we use the composition mechanism to create a `.travis.yml` file. Once the composition mechanism has created the specification file, it can be executed, i.e., the ability to execute the specification file represents the *Usage Interface* for our Travis CI concern.

In general, CORE allows many concerns to be reused to create a larger reusable artifact. The Travis CI concern is just another one of those reusable concerns. For example, if Continuous Integration is needed by a concern, then the following example demonstrates the CORE reuse process as it is applied to Travis CI specification files.

1. The first step in the CORE reuse process is to select features from the feature model. For demonstration, we select a subset of the Travis CI features shown in Figure 5.2. The selected features are shown in Figure 5.3, including the order in which the features are selected (denoted by a number in a black circle next to the selected feature).

2. We use the composition mechanism to create the `.travis.yml` specification file for the fea-

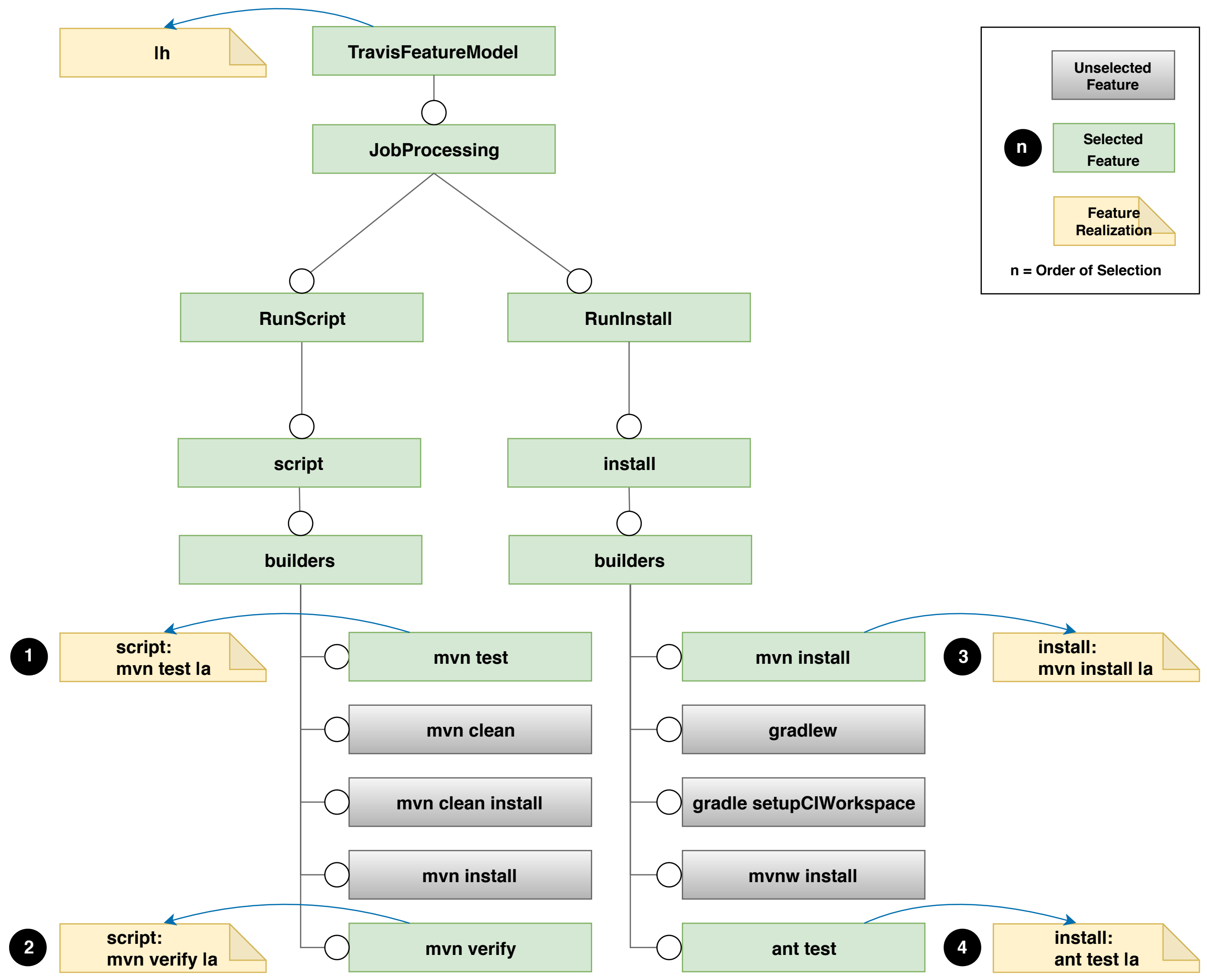
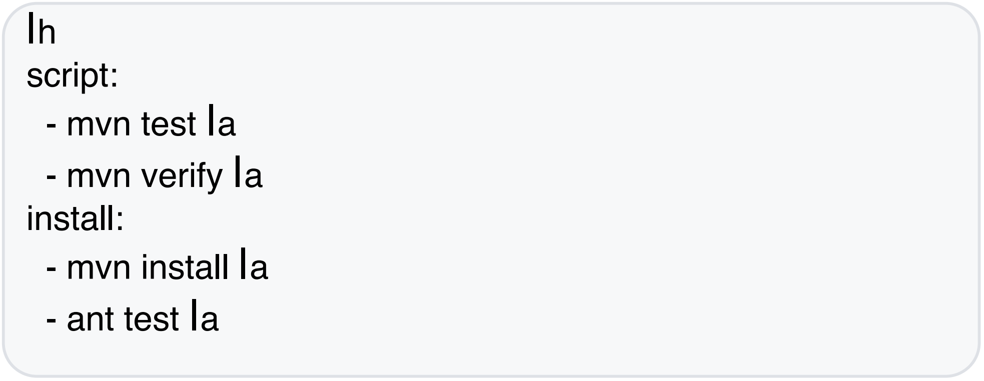


Figure 5.3: Feature Selection from Travis CI concern

tures selected in Step 1. The result is shown in Figure 5.4 and contains the merged realization models of the selected features but still with the '|' symbol showing the partial elements (i.e., headers and arguments). This step hence creates a generic reusable artifact that still needs to be customized by the release engineer.

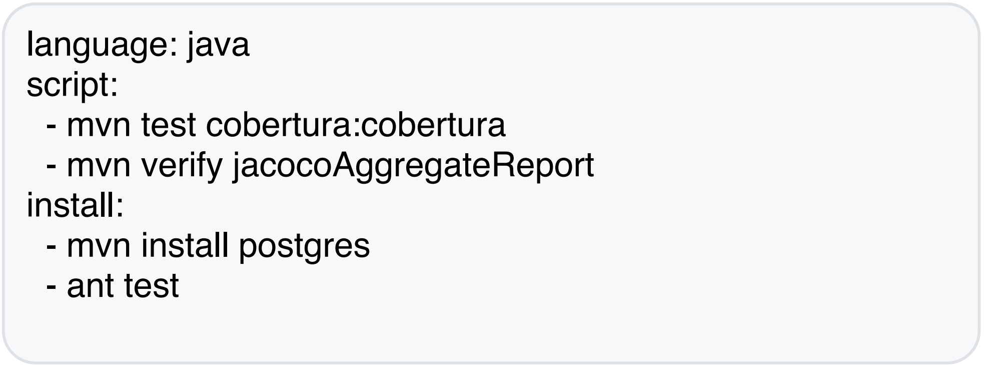
3. After the generic specification file has been created, the release engineer now customizes the generic artifact by specifying the partial elements (i.e., the headers and arguments). The resulting file is shown in Figure 5.5.

4. The reusable artifact created in Step 3 can now be used by the concern in need of Continuous Integration. Let us assume that this concern is the Banking Application concern. Many other concerns may also use the Travis CI concern to create a reusable specification file, e.g.,



```
lh
script:
  - mvn test la
  - mvn verify la
install:
  - mvn install la
  - ant test la
```

Figure 5.4: `.travis.yml` specification file created after execution of the composition mechanism

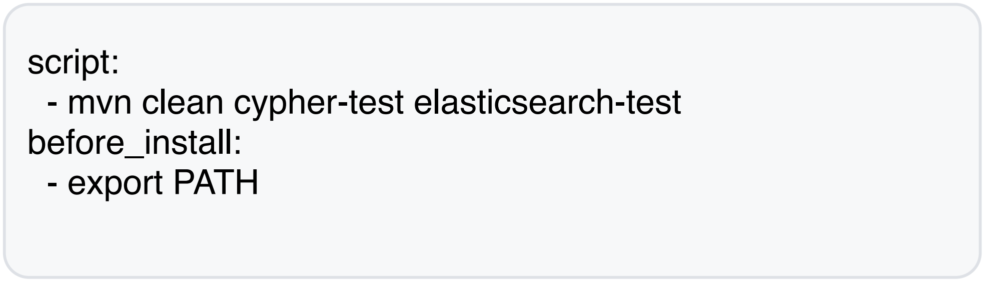


```
language: java
script:
  - mvn test cobertura:cobertura
  - mvn verify jacocoAggregateReport
install:
  - mvn install postgres
  - ant test
```

Figure 5.5: Customized `.travis.yml` specification file

the *Authentication* CORE concern discussed in Chapter 2. When the Banking Application concern reuses the Authentication concern, the realization models of these two concerns need to be combined. Therefore, in this case, we need to combine the specification files of the concerns together to form one Travis CI specification file. This is again done by the composition mechanism. It weaves both the files together by combining their headers, phases, commands, and arguments. Figure 5.6 shows an example of the reusable Travis CI specification file for the Authentication concern. This specification file was created for the Authentication concern by going through Steps 1-3 just like the specification file for the Banking Application concern. We take the file shown in Figure 5.5 to be the specification file for the Banking Application for demonstration purposes.

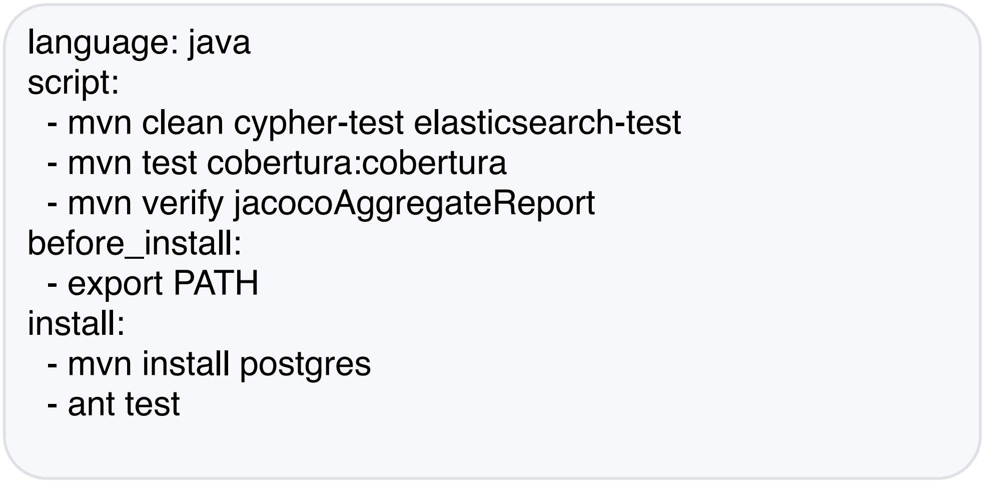
Now, we use the composition mechanism to combine these two files and the result is shown in Figure 5.7. If necessary, the release engineer may still modify this specification file, e.g., to remove duplicate lines from the header. Such optimization of the composition mechanism



```
script:
  - mvn clean cypher-test elasticsearch-test
before_install:
  - export PATH
```

Figure 5.6: Example `.travis.yml` specification file for Authentication concern

is out of scope for this thesis and left for future work.



```
language: java
script:
  - mvn clean cypher-test elasticsearch-test
  - mvn test cobertura:cobertura
  - mvn verify jacocoAggregateReport
before_install:
  - export PATH
install:
  - mvn install postgres
  - ant test
```

Figure 5.7: Woven Travis CI specification file for Authentication and Banking Application concerns

5.2 Extension of CORE's Metamodel

The metamodel shown in Figure 5.8 is an excerpt of the CORE metamodel, focusing on relevant parts for the definition and composition of CI specification files. The **COREModel** class is the main class and the super class of other types of models such as class diagrams, goal models, state machines, and, in our case, **COREFeatureModel** and **SpecFile**. The **COREFeatureModel** (**COREModel**) can have multiple **COREFeature**(s) and a **COREFeature** may be associated with many realization models (**COREModel**). For simplicity, we state in our metamodel of CORE that a **COREFeature** is realized by one **COREModel** as this is the case for the Travis CI Feature Model. In the general case, though, a **COREFeature** may be realized by many **CORE-Model**(s). The class **CORECompositionSpecification** captures details of all the reuses of the

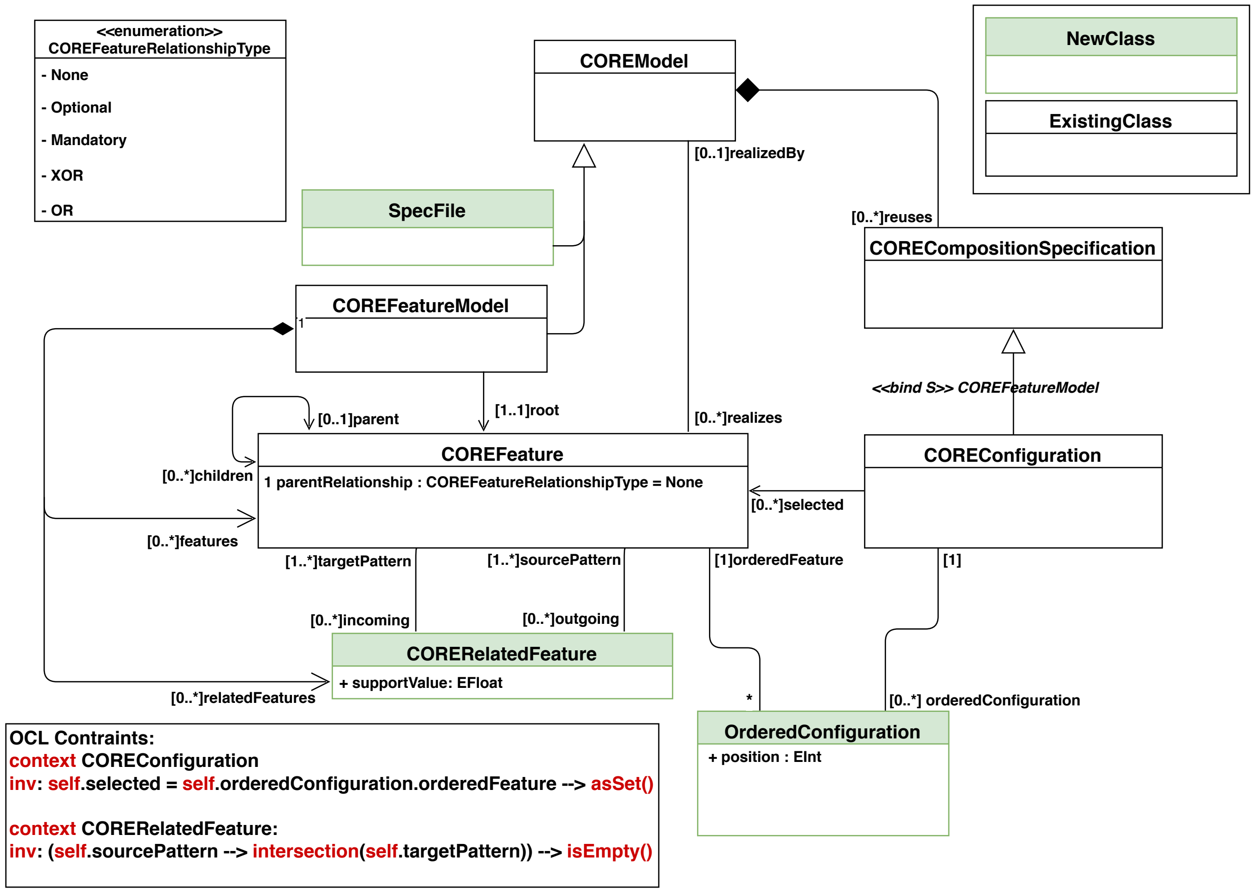
COREModel class. The subclass of **CORECompositionSpecification** that is relevant for the composition of CI specification files is the **COREConfiguration** class, which allows the selection of features in the **COREFeatureModel** (**COREModel**). We explain below our extension to the CORE metamodel as highlighted in Figure 5.8.

Once we have modeled the features of Travis CI, we strive to extend the metamodel of CORE as per our findings. First, it must be possible to choose the same feature (command) within a phase more than once from Program 5.1. Thus, we first inject a class between the **COREFeature** and **COREConfiguration** classes. The **COREFeature** class represents a feature of a feature model and the **COREConfiguration** class keeps a record of all selected features in the current selection. We call this class **OrderedConfiguration** with a *position* attribute, which captures the order in which the features are selected in its **COREConfiguration**. Hence, every selected **COREFeature** has multiple **OrderedConfiguration**(s), which specify the feature's ordered occurrences in the specification file. Only a selected feature of a **COREConfiguration** can be an *OrderedFeature* (see OCL constraint). Therefore, we maintain the order in which phases and commands are to be used by the release engineer. Although the order of phases is not semantically relevant in a `.travis.yml` file, we still need to reflect the user's discretion of ordering.

```
language: java
jdk:
  - oraclejdk8
sudo: false
env:
  - NODE_VERSION=4.4.7
before_install:
  - nvm install $NODE_VERSION
  - npm install -g npm
  - npm install -g bower grunt-cli
install: npm install
```

Program 5.1: An example `.travis.yml` specification from the ServiceCutter project

Our second aim for the tool was to provide suggestions to the release engineer to select features that occur together as per the empirical analysis. Although we did not find significant patterns in the usage of phases or commands used in Travis CI, we still incorporate this functionality in the metamodel of CORE, because further empirical studies may discover such patterns. Hence, we have a class **CORERelatedFeature** that is associated with **COREFeature**. This class signifies that a **COREFeature** can have multiple related features that may be chosen based on a *supportValue*. If all features in the *sourcePattern* of a **CORERelatedFeature** are selected, then the features in the *targetPattern* and the *supportValue* are presented to the release engineer to indicate potential candidates. An OCL constraint ensures that the features in the sourcePattern and targetPattern are distinct.



```
OCL Contraints:
context COREConfiguration
inv: self.selected = self.orderedConfiguration.orderedFeature --> asSet()

context CORERelatedFeature:
inv: (self.sourcePattern --> intersection(self.targetPattern)) --> isEmpty()
```

Figure 5.8: Extended metamodel of CORE to support the concept of specifications files

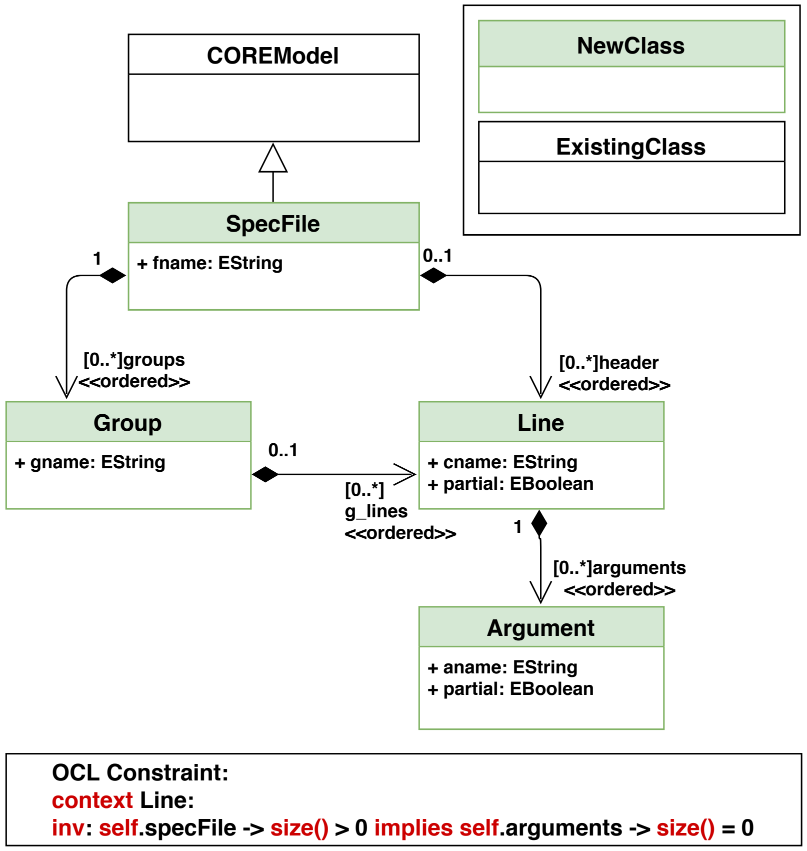
Applying the metamodel of CORE shown in Figure 5.8 to the Travis CI concern, we have a **COREFeatureModel** (Feature Model of Travis - Figure 5.2) which is a type of **COREModel**. This feature model has one root **COREFeature** from where the feature model starts and related features. Furthermore, it is possible to select the features in the Travis CI feature model in an ordered fashion.

Finally, we add the **SpecFile** class as a sub-class of **COREModel** to define a CI specification file as a kind of model in CORE. We explain the metamodel of the CI specification file in the next section.

5.3 Specification File Metamodel

A `.travis.yml` file contains phases, commands, and arguments of commands. The metamodel of a CI specification file is shown in Figure 5.9. We define a **SpecFile** to encapsulate a complete `.travis.yml` file, including all of its phases, commands, and their arguments. We define a phase as a **Group**, which is comprised of **Line**(s) as in commands in the context of Travis. We also show the **Argument** class, which is a part of the **Line** class as we know commands may or may not have arguments. All of the compositions are ordered, as a textual specification file implies a strict sequence of elements. Now, there are certain lines in a `.travis.yml` file which are not represented by any group. For example, the first line in the `.travis.yml` file shown in Program 5.1 is 'language: java'. It is not a part of any phase thus meaning that it applies to the whole file. We support such statements by having a composition association directly between the **SpecFile** class and the **Line** class and we call those lines headers. We know from the examples stated in Section 5.1 that partial file headers and arguments are needed. We represent them with the Boolean attributes **partial** in the **Line** and **Argument** classes of the specification file metamodel, respectively, which allows us to identify partial elements with a '|' symbol in the textual representation of a Travis CI specification file. Hence, from our metamodel in Figure 5.9, we understand that a specification file may have more than one group (phase). A specification file also contains headers, which are not a part of any group. Groups may contain more than one line (command). These lines may consist of arguments and if an argument is present in the specification file then it must have a line containing it.

An OCL constraint ensures that a line in a header does not have any arguments. This is a useful restriction to simplify the composition mechanism, because the header is always application-specific (as we have seen in the examples in Figures 5.4 and 5.5) and hence the header is always a partial element. Since a partial element may be replaced by anything by the release engineer, it is not necessary to model lines and arguments of header lines.



```
OCL Constraint:
context Line:
inv: self.specFile -> size() > 0 implies self.arguments -> size() = 0
```

Figure 5.9: Metamodel of CI specification file

5.4 Composition Mechanism to Create `.travis.yml` Files

In this section, we illustrate how we use the realizations of the selected features from **COREFeatureModel** (**COREModel**) and its **COREConfiguration** to create a `.travis.yml` file. First of all, we generate code from our extended metamodel of CORE and the metamodel of the Specification File. The generated code contains all the classes with their methods and attributes and inter-class method calls to allow consistent manipulation of instances of the two metamodels. We use this code as the base of our weaving algorithm.

From the metamodel of CORE shown in Figure 5.8, we know that we need a **COREFeatureModel** with a root **COREFeature** and other features. The proposed composition mechanism does not consider related features because we did not find any patterns in the usage of Travis CI. Our **COREFeatureModel** has a **COREConfiguration**, which comprises of all the selected features of the feature model. We also need **OrderedConfiguration**(s) as discussed in Section 5.2. After creating a feature, we select it as necessary with the help of a COREConfiguration, and then assign it the positions as required by the multiple selection of the same feature. To do this, we add the created features to the **OrderedConfiguration** list. A **COREFeature** is realized by a **SpecFile**. As we know, **SpecFile** contains **Group**(s), **Line**(s), and **Argument**(s), so our motivation for this choice is to enable weaving of all the different **SpecFile**(s) of the selected features in an ordered fashion. The detailed reuse process and the creation of the Travis CI specification file is explained in Section 5.1.

The composition mechanism runs in two different situations, (i) when we have two different specification files required to be woven together as discussed in step 4 of Section 5.1 and (ii) when we have a **COREFeatureModel** and we need to weave all the **COREFeature**(s) given a **COREConfiguration** and **OrderConfiguration**(s) as discussed in steps 1-3 in Section 5.1. We explain these two scenarios below:

(i) The release engineer inputs the two SpecFiles to the method *weave(SpecFile higherSpecFile, SpecFile lowerSpecFile)* of the composition mechanism. This method duplicates the headers of both the input files and creates a new file (say, *sfNew*) with the headers of *higherSpecFile* followed by that of *lowerSpecFile*. After this, the method duplicates the groups of the *higherSpecFile* and adds them to *sfNew* along with their duplicated lines and arguments using the method *merge(Group higherGroup, Group lowerGroup)* followed by composing the groups of the *lowerSpecFile* with *sfNew* using *weave(SpecFile higherSpecFile, Group lowerGroup)*. This method first checks if the name of the *lowerGroup* is same as the name of a group in the *higherSpecFile*. If yes, it merges the contents of these two groups, else, it duplicates the *lowerGroup* and adds it to the *higherSpecFile* as a new group. Finally, the method, *weave(SpecFile higherSpecFile, SpecFile lowerSpecFile)*, returns the new **SpecFile**, *sfNew*. For all details, see the first test case in Appendix C (i.e., Program C.3).

(ii) The release engineer inputs the Travis CI concern's feature model (**COREFeatureModel**) along with a **COREConfiguration** to the method *weaveAll(COREFeatureModel cFM, COREConfiguration conf)* of the composition mechanism. This method first identifies the realization model of the root feature (**SpecFile**) of the feature model, *cFM*, and proceeds with iteratively weaving the selected features pairwise according to the **OrderedConfiguration**(s) using the *weave(SpecFile higherSpecFile, SpecFile lowerSpecFile)* method. After this, the same procedure as mentioned in the first scenario is followed. For all details, see the second test case in Appendix C (i.e., Program C.4).

The specifications of the two metamodels, i.e., the extension to CORE and the CI specification file, are presented in Program C.1 in Appendix C. The proof-of-concept implementation of the composition mechanism is shown in Program C.2 in Appendix C. The composition mechanism is further explained in the flowchart in Figure 5.10 below. Starting at the start point, the first scenario takes the bottom path through the flowchart, while the second scenario takes the top path through the flowchart.

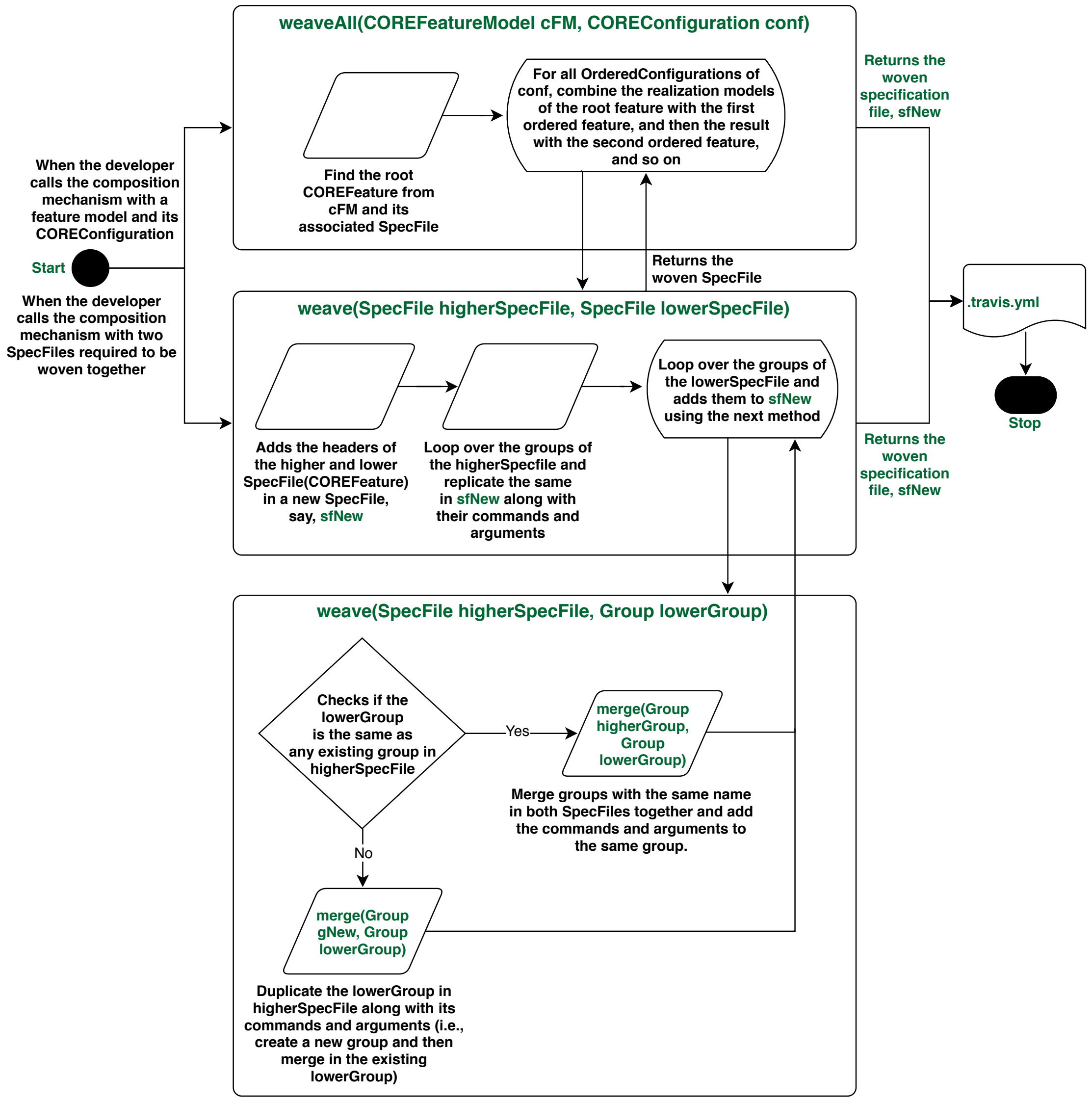

weaveAll(COREFeatureModel cFM, COREConfiguration conf)
Find the root COREFeature from cFM and its associated SpecFile
For all OrderedConfigurations of conf, combine the realization models of the root feature with the first ordered feature, and then the result with the second ordered feature, and so on
Returns the woven specification file, sfNew
When the developer calls the composition mechanism with a feature model and its COREConfiguration
Start
When the developer calls the composition mechanism with two SpecFiles required to be woven together
Returns the woven SpecFile
weave(SpecFile higherSpecFile, SpecFile lowerSpecFile)
Adds the headers of the higher and lower SpecFile(COREFeature) in a new SpecFile, say, sfNew
Loop over the groups of the higherSpecfile and replicate the same in sfNew along with their commands and arguments
Loop over the groups of the lowerSpecFile and adds them to sfNew using the next method
Returns the woven specification file, sfNew
.travis.yml
Stop
weave(SpecFile higherSpecFile, Group lowerGroup)
Checks if the lowerGroup is the same as any existing group in higherSpecFile
Yes
merge(Group higherGroup, Group lowerGroup)
Merge groups with the same name in both SpecFiles together and add the commands and arguments to the same group.
No
merge(Group gNew, Group lowerGroup)
Duplicate the lowerGroup in higherSpecFile along with its commands and arguments (i.e., create a new group and then merge in the existing lowerGroup)

Figure 5.10: Flowchart explaining the composition algorithm

Thus, using the above algorithm, we are able to create CI specification files to which the release engineer can add any additional information for both scenarios described earlier in this section.

5.5 Summary

This chapter presents our proof-of-concept required to implement CORE. We create a feature model for Travis CI specification files and explain the four-step reuse process in Section 5.1. We discuss the changes we implement to extend the metamodel of CORE in Section 5.2 to support a new type of COREModel and other configurational enhancements. We also discuss the metamodel of the SpecFile containing groups, lines, and arguments in Section 5.3. At the end, in Section 5.4, we use all the above content to create a composition mechanism that creates Travis CI specification files based on the features selected by a release engineer.

In the next chapter, we discuss the related work in terms of CI and CORE used for the empirical analysis and reuse process implemented in this thesis.

6
Related Work

In this section, we situate our work with respect to the past work on Continuous Integration (CI) along adoption and outcome dimensions as well as with respect to Concern-Oriented Reuse (CORE).

6.1 Adoption of CI

An important trend of CI research has focused on adoption trends in software organizations. For example, Beller *et al.* [6] analyzed and shared a large, curated sample CI data from thousands of open source projects. Hilton *et al.* [20] studied the characteristics of projects that choose to adopt CI (or not), and barriers that developers face when adopting CI [18].

Moreover, the adoption of CI has been associated with project properties. Vasilescu *et al.* [45] found that CI adoption is often accompanied with a boost in team productivity. Hilton *et al.* [20]

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 [18]. Indeed, Widder *et al.* [49] observed that projects using language tool-chains 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 thesis, 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.

6.2 Outcome of CI

The plethora of available CI data has enabled large-scale analyses of build processes. For example, Zampetti *et al.* [50] analyzed the usage of static analysis tools from within the CI process. Beller *et al.* [6] 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.* [36] studied common types of build breakages in 20 Java open source systems. Vassallo *et al.* [47] compared build failures in the CI processes of open source organizations to those of a large financial institution. Gallaba *et al.* [16] 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 [17] formulated, quantified, and fixed patterns of misuse of CI features (i.e., anti-patterns) in a large sample of Travis CI specifications. In their study on CI anti-patterns, Vassallo *et al.* [46] argue that process-related CI anti-patterns tend to accrue because CI tends to be overlooked when practicalities take precidence, e.g., the pressure to release. Labuschagne *et* *al.* [30] 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.

6.3 Concern-Oriented Reuse

In the era of innovation and automation, reuse is a core concept in the context of SPL engineering [11, 35]. Krueger [29] defines software reuse as the process of creating software systems from existing systems rather than creating them from scratch. He also says that abstraction plays a vital role in software reuse and without abstraction developers will have to juggle through a collection of reusable artifacts trying to figure out the functionality of each artifact and when and how they can be reused. This fact is also supported by various other researchers [1, 8, 38].

Concern-Oriented Reuse (CORE) [39] is a new reuse paradigm for general-purpose software development that combines concepts of Model Driven Engineering (MDE), Component-Based Software Engineering (CBSE), Software Product Lines (SPL), advanced Separation of Concerns (SoC) (including feature-oriented and aspect-oriented software development), and goal modeling [23].

Jézéquel *et al.* [23] say that the main premise of CORE is that recurring development concerns are made available in a concern library, which eventually should cover most recurring software development needs. Similar to class libraries in modern programming languages, this library should grow as new development concerns emerge, and existing concerns should continuously evolve as alternative architectural, algorithmic, and technological solutions become available. Applications are built by reusing existing concerns from the library whenever possible, following a well-defined reuse process supported by clear interfaces.

The TouchCORE tool [40] supports the CORE reuse process and comes with its own concern library containing several often-used concerns. An example of such a concern is the *Associations* concern [8], which captures all variations to associate two classes that might exchange or share data with each other. This concern has been extensively reused in many different models. In this thesis, we add the Travis CI concern to the existing concern library and demonstrate how it can be used to create CI specification files.

The CORE approach is a next-generation reuse technique that supports the encapsulation of different structural and behavioral models and their impacts within one reusable model [1, 2]. CORE is supported by class diagrams, sequence diagrams, and state machines, and more recently also Use Case Maps [43]. In this thesis, we add Travis CI specification files to the models available in CORE.

Jézéquel *et al.* [23] apply the concepts of CORE to the engineering of software languages in an approach called Concern-Oriented Language Development (COLD) that promotes modularity and reusability of language concerns. A language concern is a reusable piece of language that consists of usual language artifacts (e.g., abstract syntax, concrete syntax, semantics) and exhibits the use of the Variation, Customization, and Usage (VCU) interfaces. In future work, the SpecFile language could be defined using the COLD approach instead of the standard MOF metamodel-based approach used in this thesis.

In this thesis, we collect the dominant features of CI so that they can be reused by the developers. Feature Models were first introduced by Kang *et al.* [24] to capture the problem domain of an SPL [34]. A feature model captures the potential features of aspects of an SPL in a tree structure, containing those features that are commonly used across the domain and differ in functionality. A particular subset of the feature model is defined by selecting the desired features from the feature model, resulting in a feature model configuration [1]. We extend the feature configuration provided in CORE's metamodel to be able to select the same feature multiple times in an ordered fashion.

At the more detailed level called the abstraction realization [29], we state the realization models of all the features. Mohagheghi *et al.* [31] state that, similar to knowledge reuse, software reuse partly reflects the reuse of architectures, templates, and processes. This concept is consolidated by MDA, which is based on widely-used industry standards for visualizing, storing, and exchanging software design and models [28]. In this thesis, we use templates of CI specification files as realization models for the features in the Travis CI feature model. Furthermore, we follow the reuse process defined by the CORE approach to support the reuse of specification files.

UMPLE (UML Programming Language) is a textual design modeling tool supporting class diagrams and state diagrams [5]. It has a powerful code generator capabilities that handle multiplicity constraints and referential integrity for associations and compositions. It is able to generate code from class diagrams and state machines in more than 20 languages.[15] In this thesis, we create

[15]`https://cruise.eecs.uottawa.ca/umple/GettingStarted.html`

a proof-of-concept implementation of the composition mechanism for Travis CI specification files based on the code generated by UMPLE using metamodels of CORE and Travis CI. However, this is a choice of convenience due to our familiarity with UMPLE, and we could have used any other code generation environment.

6.4 Summary

In this chapter, we situate our work with respect to the literature on CI and CORE. We state the trends in adoption of CI and its current status and summarize the landscape of reuse related to CORE.

7 Conclusions

Thesis Statement. Despite the lack of tool support, reuse is a common activity when preparing and maintaining CI configuration files. The swaths of available CI configuration files can be mined to guide the extension of reuse frameworks to the CI configuration use case.

7.1 Contributions and Findings

In this thesis, we set out to enhance the experience of Continuous Integration (CI) using Concern-Oriented Reuse (CORE) by letting the release engineers automatically create Travis CI specification files for their projects instead of creating them from scratch. Travis CI is a commonly used CI service for open-source projects.

More specifically, we analyze the phases and commands mentioned in the Travis CI files. We start by analyzing 913 open source projects that use Travis CI and are implemented in Java. We observe that the “script”, “before_install”, “install”, “after_success”, and “before_script” phases most frequently appear in those Travis CI specifications. Furthermore, these phases contain 245 unique commands that can be categorized based on the similarity of their functionality. We discover that although the source sample was large, 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 analyze 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 sample of projects using Association Rule Mining (ARM). However, techniques other than ARM may find different results.

We transform these frequent phases and commands categorized by usage in a CORE feature model. Since we do not find patterns in the usage of Travis CI specifications, our intention to provide suggestions to release engineers to consider choosing related phases and commands in Java projects while making selections from the Travis feature model cannot be realized. Nonetheless, we lay the groundwork to enable release engineers to choose features from a feature model and create a corresponding CI specification file. To do this, we extend the metamodel of CORE to allow selection of the same feature multiple times in a specific order as the same command can appear multiple times within the same phase. We also propose an enhancement to the metamodel of CORE to support the selection of related features. In addition, we create a metamodel for Travis CI specification files, capturing groups, lines, and arguments. Based on these two metamodels, we introduce a composition mechanism that automates the creation of the CI specification file given an ordered feature selection.

Our contribution towards software reuse with CORE is the Travis CI concern created on the basis of the Travis CI feature model, the metamodel of Travis CI specification files, and enhancements to the CORE metamodel. The composition mechanism built on top of these metamodels lets the release engineers create the Travis CI specification files by choosing the required Travis CI features from its feature model in an ordered fashion.

7.2 Future Research

7.2.1 Empirical Study

We plan to take the technological 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 CI reuse patterns. Additional empirical studies could be performed (e.g., for other languages than Java, using other techniques than ARM, for a specific application domain), which could further enhance the results of the usage of Travis CI phases and commands. While our search for reuse patterns in this context did not yield useful results, 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.

7.2.2 Reuse

We want to provide suggestions during feature selection, which will be possible once we detect more useful reuse patterns. We want to investigate the impact of Travis CI features on system qualities and non-functional requirements and then formalize the results in a goal model for the Variation Interface of the Travis CI concern. We plan to optimize the results of our composition mechanism by incorporating the partial elements like prefixes and options for Travis CI commands. We also plan to determine a feature selection for Travis CI based on the feature selection for the actual concern (e.g., Authentication), i.e., if a specific Authentication feature is chosen, then a specific CI commands needs to be added.

While the proposed SpecFile metamodel focuses on Travis CI specification files, the metamodel could be extended more generally to textual configuration files not limited to CI. This would allow additional information about the software product and the employed development process to be incorporated into the CORE approach.

While we demonstrate the feasibility of a Travis CI concern and its composition in a reuse hierarchy, an empirical analysis of the usage of our Travis CI concern would allow us to determine how efficient, easy to use, and useful the concern is to release engineers.

Bibliography

[1] Omar Alam, Jörg Kienzle, and Gunter Mussbacher. Concern-oriented software design. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems*, pages 604–621. Springer, 2013.

[2] Omar Alam, Jörg Kienzle, and Gunter Mussbacher. Modelling a family of systems for crisis management with concern-oriented reuse. *Software: Practice and Experience*, pages 985–999, 2017.

[3] Daniel Amyot, Sepideh Ghanavati, Jennifer Horkoff, Gunter Mussbacher, Liam Peyton, and Eric Yu. Evaluating goal models within the goal-oriented requirement language. *International Journal of Intelligent Systems*, pages 841–877, 2010.

[4] Daniel Amyot and Gunter Mussbacher. User requirements notation: the first ten years, the next ten years. *Journal of Software*, pages 747–768, 2011.

[5] Omar Badreddin, Andrew Forward, and Timothy C. Lethbridge. Improving code generation for associations: enforcing multiplicity constraints and ensuring referential integrity. In *Software Engineering Research, Management and Applications*, pages 129–149. Springer, 2014.

[6] Moritz Beller, Georgios Gousios, and Andy Zaidman. Oops, my tests broke the build: An explorative analysis of travis ci with github. In *Proceedings of the IEEE/ACM International Conference on Mining Software Repositories*, pages 356–367. IEEE, 2017.

[7] David Benavides, Alexander Felfernig, José A. Galindo, and Florian Reinfrank. Automated analysis in feature modelling and product configuration. In *Proceedings of the International Conference on Software Reuse*, pages 160–175. Springer, 2013.

[8] Céline Bensoussan, Matthias Schöttle, and Jörg Kienzle. Associations in mde: A concern-oriented, reusable solution. In *Proceedings of the European Conference on Modelling Foundations and Applications*, pages 121–137. Springer, 2016.

[9] Grady Booch. *Object Oriented Design with Applications.* Benjamin-Cummings Publishing Co., Inc., 1991.

[10] Marko Bošković, Gunter Mussbacher, Ebrahim Bagheri, Daniel Amyot, Dragan Gašević, and Marek Hatala. Aspect-oriented feature models. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems*, pages 110–124. Springer, 2010.

[11] Lianping Chen and Muhammad Ali Babar. A systematic review of evaluation of variability management approaches in software product lines. *Information and Software Technology*, pages 344–362, 2011.

[12] Peter Pin-Shan Chen. The entity-relationship model—toward a unified view of data. *ACM Transactions on Database Systems*, pages 9–36, 1976.

[13] Michael A. Cusumano and Richard W. Selby. *Microsoft secrets: how the world's most powerful software company creates technology, shapes markets, and manages people.* Simon and Schuster, 1998.

[14] Krzysztof Czarnecki and Ulrich W. Eisenecker. Components and generative programming. In *ACM SIGSOFT Software Engineering Notes*, pages 2–19. Springer-Verlag, 1999.

[15] Martin Fowler and Matthew Foemmel. Continuous integration. *https://www.thoughtworks.com/continuous-integration*, page 14, 2006.

[16] Keheliya Gallaba, Christian Macho, Martin Pinzger, and Shane McIntosh. Noise and heterogeneity in historical build data: an empirical study of travis ci. In *Proceedings of the ACM/IEEE International Conference on Automated Software Engineering*, pages 87–97. ACM, 2018.

[17] Keheliya Gallaba and Shane McIntosh. Use and misuse of continuous integration features: An empirical study of projects that (mis) use travis ci. *IEEE Transactions on Software Engineering*, pages 1–1, 2018.

[18] Michael Hilton, Nicholas Nelson, Danny Dig, Timothy Tunnell, Darko Marinov, et al. Continuous integration (ci) needs and wishes for developers of proprietary code. Technical report, Corvallis, OR: Oregon State University, Dept. of Computer Science, 2016.

[19] Michael Hilton, Nicholas Nelson, Timothy Tunnell, Darko Marinov, and Danny 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*, pages 197–207. ACM, 2017.

[20] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. Usage, costs, and benefits of continuous integration in open-source projects. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, pages 426–437. ACM, 2016.

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

[22] ITU-T. Recommendation z.151 (10/18): User requirements notation (urn) - language definition. In *SERIES Z: Languages and General Software Aspects for Telecommunication Systems*, Geneva, Switzerland, 2018.

[23] Jean-Marc Jézéquel, Manuel Leduc, Olivier Barais, Tanja Mayerhofer, Erwan Bousse, Walter Cazzola, Philippe Collet, Sébastien Mosser, Benoit Combemale, Thomas Degueule, et al. Concern-oriented language development (cold): Fostering reuse in language engineering. *Computer Languages, Systems & Structures*, pages 139–155, 2018.

[24] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst, 1990.

[25] Noureddine Kerzazi, Foutse Khomh, and Bram Adams. Why do Automated Builds Break? An Empirical Study. In *Proceedings of the International Conference on Software Maintenance and Evolution*, pages 41–50. IEEE, 2014.

[26] Jörg Kienzle, Wisam Al Abed, Franck Fleurey, Jean-Marc Jézéquel, and Jacques Klein. Aspect-oriented design with reusable aspect models. In *Transactions on aspect-oriented software development VII*, pages 272–320. Springer, 2010.

[27] Jörg Kienzle, Gunter Mussbacher, Omar Alam, Matthias Schöttle, Nicolas Belloir, Philippe Collet, Benoit Combemale, Julien Deantoni, Jacques Klein, and Bernhard Rumpe. Vcu: the three dimensions of reuse. In *Proceedings of the International Conference on Software Reuse*, pages 122–137. Springer, 2016.

[28] Anneke G. Kleppe, Jos Warmer, Jos B. Warmer, and Wim Bast. *MDA explained: the model driven architecture: practice and promise.* Addison-Wesley Professional, 2003.

[29] Charles W. Krueger. Software reuse. *ACM Computing Surveys*, pages 131–183, 1992.

[30] Adriaan Labuschagne, Laura Inozemtseva, and Reid Holmes. Measuring the cost of regression testing in practice: a study of Java projects using continuous integration. In *Proceedings of the International Symposium on the Foundations of Software Engineering*, pages 821–830. ACM, 2017.

[31] Parastoo Mohagheghi and Reidar Conradi. Quality, productivity and economic benefits of software reuse: a review of industrial studies. *Empirical Software Engineering*, pages 471–516, 2007.

[32] Object Management Group (OMG). Unified modeling language.

[33] Harold Ossher and Peri Tarr. *Multi-Dimensional Separation of Concerns and the Hyperspace Approach*, pages 293–323. Springer US, 2002.

[34] Klaus Pohl, Günter Böckle, and Frank J. van Der Linden. *Software product line engineering: foundations, principles and techniques.* Springer Science & Business Media, 2005.

[35] Klaus Pohl and Andreas Metzger. Variability management in software product line engineering. In *Proceedings of the International Conference on Software Engineering*, pages 1049–1050. ACM, 2006.

[36] Thomas Rausch, Waldemar Hummer, Philipp Leitner, and Stefan Schulte. How Open Source Projects Use Static Code Analysis Tools in Continuous Integration Pipelines. In *Proceedings of the International Conference on Mining Software Repositories*, pages 345–355. IEEE, 2017.

[37] Douglas C. Schmidt. Model-driven engineering. *IEEE Computer*, pages 25–31, 2006.

[38] Matthias Schöttle. Model-based reuse of apis using concern-orientation. In *Proceedings of the ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, pages 41–45. ACM, 2015.

[39] Matthias Schöttle, Omar Alam, Jörg Kienzle, and Gunter Mussbacher. On the modularization provided by concern-oriented reuse. In *Proceedings of the International Conference companion on Modularity*, pages 184–189. ACM, 2016.

[40] Matthias Schöttle, Nishanth Thimmegowda, Omar Alam, Jörg Kienzle, and Gunter Mussbacher. Feature modelling and traceability for concern-driven software development with touchcore. In *Proceedings of the International Conference companion on Modularity*, pages 11–14. ACM, 2015.

[41] Hyunmin Seo, Caitlin Sadowski, Sebastian Elbaum, Edward Aftandilian, and Robert Bowdidge. Programmers' Build Errors: A Case Study (at Google). In *Proceedings of the International Conference on Software Engineering*, pages 724–734. ACM, 2014.

[42] Puneet Kaur Sidhu, Gunter Mussbacher, and Shane McIntosh. Reuse (or lack thereof) in travis ci specifications: An empirical study of ci phases and commands. In *Proceedings of the IEEE International Conference on Software Analysis, Evolution and Reengineering*, pages 524–533. IEEE, 2019.

[43] Cheuk Chuen Siow. Concern-oriented use case maps. Master's thesis, School of Computer Science, McGill University, Canada, 2018.

[44] Nishanth Thimmegowda, Omar Alam, Matthias Schöttle, Wisam Al Abed, Thomas Di'Meco, Laura Martellotto, Gunter Mussbacher, and Jörg Kienzle. Concern-driven software development with jucmnav and touchram. In *Proceedings of the ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, page 5. Citeseer, 2014.

[45] Bogdan Vasilescu, Yue Yu, Huaimin Wang, Premkumar Devanbu, and Vladimir Filkov. Quality and productivity outcomes relating to continuous integration in github. In *Proceedings of the Joint Meeting on Foundations of Software Engineering*, pages 805–816. ACM, 2015.

[46] Carmine Vassallo, Sebastian Proksch, Harald C. Gall, and Massimiliano Di Penta. Automated reporting of anti-patterns and decay in continuous integration. In *Proceedings of the International Conference on Software Engineering*, pages 105–115. IEEE, 2019.

[47] Carmine Vassallo, Gerald Schermann, Fiorella Zampetti, Daniele Romano, Philipp Leitner, Andy Zaidman, Massimiliano Di Penta, and Sebastiano 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*, pages 183–193. IEEE, 2017.

[48] Stephen A. White. *BPMN modeling and reference guide: understanding and using BPMN*. Future Strategies Inc., 2008.

[49] David Widder, Bogdan Vasilescu, Michael Hilton, and Christian Kästner. I'm leaving you, travis: a continuous integration breakup story. In *Proceedings of the IEEE/ACM International Conference on Mining Software Repositories*, pages 165–169. IEEE, 2018.

[50] Fiorella Zampetti, Simone Scalabrino, Rocco Oliveto, Gerardo Canfora, and Massimiliano Di Penta. How Open Source Projects Use Static Code Analysis Tools in Continuous Integration Pipelines. In *Proceedings of the International Conference on Mining Software Repositories*, pages 334–344. IEEE, 2017.

[51] Yangyang Zhao, Alexander Serebrenik, Yuming Zhou, Vladimir Filkov, and Bogdan Vasilescu. The impact of continuous integration on other software development practices: a large-scale empirical study. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, pages 60–71. IEEE, 2017.

A
Python Code for Parser

The code below shows the algorithm which we used to parse the `.travis.yml` files and find data of their phases and commands.

Program A.1: Parser.py

```python
import os, sys, yaml, codecs, csv, pprint, argparse
import sqlite3 as lite
from collections import Iterable, defaultdict
from bashlex import parser, ast, errors
from utils import get_category, valid_top_level_keys
from importPhase import getPhase
from insertTravisAnalysisJavaDB import insertRowsTA

con = lite.connect('test.db')

# List of phases that we are analysing
phases = ['before_install', 'install', 'after_install',
          'before_script', 'script', 'after_script',
          'after_deploy','before_deploy',
          'after_failure', 'after_success']

# Listing commands which appear in pairs. For example: pip can be accompanied with install or
    update
pairCommands = ['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', 'python']
results = {}
everything = set()
travis_keys = set()
travis_keys_count = defaultdict(int)
results_section_line_count = {}
global_category_count = defaultdict(int)
results_phase_type = {}
results_line_count = {}
```

```python
global_phase_line_count = defaultdict(int)
global_phase_count = defaultdict(int)
global_phase_project_list = defaultdict(list)
global_lang_count = defaultdict(int)
y = ""
x = ""
z = ""
option = ""
param = ""
setFlag = ""


class nodevisitor(ast.nodevisitor):
    def __init__(self, phase, phase_by_type, commands_in_phase):
        self.phase = phase
        self.phase_by_type = phase_by_type
        self.commands_in_phase = commands_in_phase

    # command pairs, remove if(), if command numeric
    def visitcommand(self, n, parts):
        global x
        x = next((x.word.split('/')[-1] for x in parts if (
            x.kind == 'word' and '=' not in x.word.split('/')[-1]
            and x.word.split('/')[-1] not in
            ('sudo', '-c', '-e', '=', ']', '[', '!', 'bash)', '!',
            ')', '(', '.', ':'))), None)
        # collected all x from parts
        if x in pairCommands:
            # checked if any x exists in pair commands
            global lenOfParts
            lenOfParts = len(parts)
            for index, item in enumerate(parts):
                global setFlag
                xMatch = item.word.split('/')[-1]
                # xMatch is same as x as it is the part here
                if (xMatch == x and index < (lenOfParts - 1)):
                    # checking if the xMatch(part extracted) is same as x i.e. if it is in pair
                        commands
                    try:
                        nextItem = parts[index + 1]
                        nextWord = nextItem.word.split('/')[-1]
                        paramItem = parts[index + 2]
                        paramWord = paramItem.word.split('/')[-1]
                        secNextItem = parts[index + 2]
                        secNextWord = secNextItem.word.split('/')[-1]
                        param_secNextItem = parts[index + 3]
                        param_secNextWord = param_secNextItem.word.split('/')[-1]
                    except IndexError:
                        secNextWord = ''
                        param_secNextWord = ''
                        paramWord = ''
                    global z
                    z = nextWord
                    global param
                    param = paramWord
                    if z in ('-y', '-q', '-v', '-qq',
                        '-yq', '-p', '-B', '-e', '-U',
                        '-u', '-C', '-N', '-X',
                        '-c', '-s', '--force-yes'):
```

```
                    global option
                    option = z
                    z = secNextWord
                    param = param_secNextWord
                    if param in ('-y', '-q', '-v',
                        '-qq', '-yq', '-p', '-B',
                        '-e', '-U', '-u', '-C',
                        '-N','-X', '-c', '-s',
                        '--force-yes'):
                        setFlag = 'yes'
                        param = ''
                if z in ('-c', '-s'):
                    return False
                if z == 'clean':
                    z = z + ' ' + param_secNextWord
                    # param = paramWord4
                    param = ''
        else:
            global lenOfParts1
            lenOfParts1 = len(parts)
            for index, item in enumerate(parts):
                xMatch = item.word.split('/')[-1]
                # xMatch is same as x as it is the part here
                if (xMatch == x and index < (lenOfParts1 - 1)):
                    # checking if the xMatch(part extracted) is same as x i.e. if it is in pair
                        commands

                    try:
                        paramItem = parts[index + 1]
                        paramItem2 = parts[index + 2]
                        param = paramItem.word.split('/')[-1]
                        paramWord2 = paramItem2.word.split('/')[-1]
                        if param in ('-y', '-q', '-v',
                            '-qq', '-yq', '-p', '-B',
                        '-e', '-U', '-u', '-C', '-N',
                        '-X', '-c', '-s', '--force-yes'):
                            option = param
                            param = paramWord2
                            if param in ('-y', '-q', '-v', '-qq',
                                '-yq', '-p', '-B', '-e', '-U',
                                '-u', '-C', '-N', '-X', '-c', '-s',
                                '--force-yes'):
                                param = ''
                                setFlag = 'yes'
                    except IndexError:
                        param = ''
                        paramItem2 = ''
                        paramWord2 = ''
        if x.startswith("$", 0, len(x)):
            return False
        if x.startswith("'", 0, len(x)):
            return False
        if x.startswith("`", 0, len(x)):
            return False
        if x.startswith("-", 0, len(x)):
            return False
        if x.startswith("_", 0, len(x)):
            return False
        if (x and x[0].isdigit()):
            return False
        if "$" in x:
            return False
        if "." in x:
            return False
        if " fi" in x:
            return False
        if ";" in x:
            return False
        if "&&" in x:
            return False
        if ":" in x:
            return False
        if x in phases:
            return False
        global y
        y = next((y.word.split('/')[-1] for y in parts if
                 (y.kind == 'word' and '=' not in
                  y.word.split('/')[-1] and
                  y.word.split('/')[-1] in ('sudo'))), None)
        if x and x not in self.commands_in_phase:
            self.phase[x] += 1
            self.phase_by_type[get_category(x)] += 1
            print("Phase command: ", x, "--> Category :", get_category(x))
            global_category_count[x] += 1
            print ("command is ---> ", x)
        self.commands_in_phase.add(x)
        # TravisAnalysisJava (PROJECT_NAME,SUB_PHASE,COMMAND_PREFIX,COMMAND,COMMAND_ARGUMENT)
        print("final values prefix is:", y, " command is: ", x, " option is: ", option, " param is:
            ", param)
        if (x):
            fileData = [project_name, testPhase, y, x, z, option, param, setFlag]
            finalFile = open('superDS.csv', 'a')
            outputWriter = csv.writer(finalFile)
            z = ""
            setFlag = ""
            option = ""
            param = ""
            try:
                outputWriter.writerow(fileData)
            except UnicodeEncodeError as err:
                pass

        return True


def parse_line(line, results_phase, results_phase_type_phase, commands_in_phase):
    try:
        trees = parser.parse(line)
        for tree in trees:
            visitor = nodevisitor(results_phase, results_phase_type_phase, commands_in_phase)
            visitor.visit(tree)
    except errors.ParsingError:
        print("Bash Parsing Error...", yaml_path)
    except NotImplementedError:
        print("Not implemented command...", yaml_path)
    except AttributeError:
        print("Bashlex bug...", yaml_path)

```

```

def traverse_node(root_node_name, content, results, current_node_name, r, p):
    if isinstance(content, dict):
        for k, v in content.items():
            traverse_node(root_node_name, v, results, k, r, p)
    elif isinstance(content, list) or isinstance(content, tuple):
        for item in content:
            traverse_node(root_node_name, item, results, current_node_name, r, p)
    else:
        if root_node_name in phases:
            global sub_phase
            sub_phase = root_node_name
            r[root_node_name] += 1


def analyze_phases(language, doc, project_name):
    global global_phase_count
    global global_phase_line_count
    global global_phase_project_list
    global global_lang_count
    r = defaultdict(int)
    global_lang_count[language] += 1
    temp_res = defaultdict(int)
    for k, v in doc.items():
        traverse_node(k, v, temp_res, k, r, project_name)
    if True:
        results_line_count[project_name] = defaultdict(int)
        for phase in phases:
            if phase in doc and doc[phase]:
                commands_in_phase = set()
                if not phase in results:
                    results[phase] = defaultdict(int)
                    results_phase_type[phase] = defaultdict(int)
                phase_arr = doc[phase]
                if isinstance(phase_arr, str):
                    phase_arr = [phase_arr]
                if not isinstance(phase_arr, Iterable):
                    continue
                global_phase_line_count[phase] += len(phase_arr)
                global_phase_count[phase] += 1
                global_phase_project_list[phase].append(project_name)
                results_line_count[project_name][phase] = len(phase_arr)
                for line in phase_arr:
                    if not isinstance(line, str):
                        continue
                    global testPhase
                    testPhase = phase
                    parse_line(line, results[phase], results_phase_type[phase], commands_in_phase)
    else:
        pass
if __name__ == '__main__':
    argparser = argparse.ArgumentParser(description='-d Directory -p path_to_project_list')
    argparser.add_argument('-d', dest='directory', default='travis_yml_only')
    argparser.add_argument('-p', dest='projects_path', default='javaProjectList.txt')
    args = argparser.parse_args()
    directory = 'travis_yml_only'
    if args.directory:
        directory = args.directory

    projects_path = 'javaProjectList.txt'
    if args.projects_path:
        projects_path = args.projects_path

    with open(projects_path, 'r') as f:
        for line in f:
            project_name = line.strip()
            yaml_path = os.path.join(directory, project_name, '.travis.yml')
            print('projectname', project_name)
            print('parsing %s' % yaml_path)
            try:
                num_lines = sum(1 for line in codecs.open(yaml_path, 'r', 'utf-8'))
            except UnicodeDecodeError as err:
                print(err)
                continue
            with codecs.open(yaml_path, 'r', 'utf-8') as f:
                try:
                    doc = yaml.load(f)
                    if doc:
                        if 'language' in doc and doc['language']:
                            lang = doc['language']
                            if not isinstance(lang, str):
                                for language in lang:
                                    language = language.strip('",').lower()
                                    analyze_phases(language, doc, project_name)
                                    break
                            else:
                                lang = lang.strip('",').lower()
                                analyze_phases(lang, doc, project_name)
                except yaml.scanner.ScannerError as err:
                    print("Scanner Error...", yaml_path)
                    print (err)
                except yaml.composer.ComposerError as err:
                    print("Composer Error...", yaml_path)
                    print (err)
                except yaml.parser.ParserError:
                    print("Malformed YAML...", yaml_path)
                except yaml.reader.ReaderError:
                    print("Reader Error...", yaml_path)
    print('.........................')
```

B

R Code for Empirical Study and Association Rule Mining

The program listings below state the R code used to analyze the data generated after parsing Travis CI specification files.

Program B.1: FrequencyAnalysis.R

```r
#reading input dataset
library(readr)
superDS <- read_csv("superDS.csv")

#Analysis


totalProjects<-sqldf::sqldf("select count(distinct project) from [superDS]")
totalPhases<-sqldf::sqldf("select distinct phase from [superDS]")
totalDistinctCommands <- sqldf::sqldf("select distinct command from [superDS]")
totalDistinctFullCommands <- sqldf::sqldf("select distinct fullCommand from [superDS]")

#Gives the frequency of the phases
phaseForAllProjects <- sqldf::sqldf("select count(distinct project) as p, phase from [superDS]
                                group by phase order by p desc")

#Gives the usage of top categories per phase
projectCount_buildersFrequency_scriptPhase<-sqldf::sqldf("select count(category) as buildersFreq,
    project from [superDS]
                                                where category = 'builders' and phase = 'script'
                                                group by project order by buildersFreq desc")
projectCount_env_setupFrequency_before_installPhase<-sqldf::sqldf("select count(category) as c,
    project from [superDS]
                                                where category = 'env_setup' and phase = 'before_
                                                    install'
                                                group by project order by c desc")
projectCount_buildersFrequency_installPhase<-sqldf::sqldf("select count(category) as c, project
    from [superDS]
                                                where category = 'builders' and phase = 'install'
                                                            group by project order by c desc")
```

```r
projectCount_buildersFrequency_after_successPhase<-sqldf::sqldf("select count(category) as c, project from [superDS]
                                                  where category = 'builders' and phase = 'after_success'
                                                  group by project order by c desc")
projectCount_env_setupFrequency_before_scriptPhase<-sqldf::sqldf("select count(category) as c, project from [superDS]
                                                  where category = 'env_setup' and phase = 'before_script'
                                                               group by project order by c desc")
projectCount_categoryUsage_allPhases<-sqldf::sqldf("select count(distinct project) as count, category from [superDS]
                                                  where category != 'other'
                                                  group by category order by count desc")

#Finding usage of commands per phase and category
commandUsage <- sqldf::sqldf("select command, count(distinct project) as c from [superDS]
                          group by command order by c desc")
commandUsage_script_builders <- sqldf::sqldf("select command, count(distinct project) as c from [superDS]
                         where phase = 'script' and category = 'builders'
                         group by command order by c desc")
commandUsage_before_install_env_setup <- sqldf::sqldf("select command, count(distinct project) as c from [superDS]
                         where phase = 'before_install' and category = 'env_setup'
                                        group by command order by c desc")
commandUsage_install_builders <- sqldf::sqldf("select command, count(distinct project) as c from [superDS]
                         where phase = 'install' and category = 'builders'
                                        group by command order by c desc")
commandUsage_after_success_builders <- sqldf::sqldf("select command, count(distinct project) as c from [superDS]
                         where phase = 'after_success' and category = 'builders'
                         group by command order by c desc")
commandUsage_before_script_env_setup <- sqldf::sqldf("select command, count(distinct project) as c from [superDS]
                         where phase = 'before_script' and category = 'env_setup'
                                        group by command order by c desc")
```

Program B.2: ARM_Phases.R

```r
#1. Which phases occur together?

library(arules)
library(arulesViz)
library(ggplot2)
arm_phase<-sqldf::sqldf("select distinct phase, project from [superDS]")
View(arm_phase)
write.csv(arm_phase, "arm_phase.csv")
#I run numbers, delete 1st column and 1 row of the above csv.
#Then run arules.py on the resultant after giving proper names to files.
#Created arule_arm_phase with arule source data. Now will apply apriori:
arule_arm_phase <- read.transactions("arule_arm_phase.csv", sep = ",", rm.duplicates= FALSE)
rules_arm_phase <- apriori(arule_arm_phase,parameter = list(supp=0.001,conf=0.7))
rules_arm_phase.sorted <- sort(rules_arm_phase,by="count")
inspect(rules_arm_phase.sorted)
write(rules_arm_phase.sorted, file = "rules_arm_phase.sorted1.csv", sep = ",")
#=12 rules

```

```
plot(rules_arm_phase.sorted)
plot(rules_arm_phase.sorted, method="graph", control=list(type="items"))
```

Program B.3: ARM_Commands.R

```
#2(a) Which commands occur together in script phase?
arm_command_allCategories<-sqldf::sqldf("select distinct command, project from [superDS] where
    phase = 'script'")
View(arm_command_allCategories)
write.csv(arm_command_allCategories, "arm_command_allCategories.csv")
#I run numbers, delete 1st column and 1 row of the above csv.
#Then run arules.py on the resultant after giving proper names to files.
#Created arm_command_allCategories with arule source data. Now will apply apriori:
arule_arm_command_allCategories <- read.transactions("arule_arm_command_allCategories.csv", sep = "
    ,", rm.duplicates= FALSE)
rules_arm_command_allCategories <- apriori(arule_arm_command_allCategories,parameter = list(supp
    =0.002,conf=0.7))
#rules_arm_command_allCategories <- apriori(arule_arm_command_allCategories,parameter = list(supp
    =0.003,conf=1))
rules_arm_command_allCategories_script.sorted <- sort(rules_arm_command_allCategories,by="count")
inspect(rules_arm_command_allCategories_script.sorted)
plot(rules_arm_command_allCategories_script.sorted, method = "scatterplot", engine = "htmlwidget",
    control = list(max = 10))
84
#2(b) Which commands occur together in before_install phase?
arm_command_allCategories_before_install<-sqldf::sqldf("select distinct command, project from [
    superDS] where phase = 'before_install'")
View(arm_command_allCategories_before_install)
write.csv(arm_command_allCategories_before_install, "arm_command_allCategories_before_install.csv")
#I run numbers, delete 1st column and 1 row of the above csv.
#Then run arules.py on the resultant after giving proper names to files.
#Created arm_command_allCategories with arule source data. Now will apply apriori:
arule_arm_command_allCategories_before_install <- read.transactions("arule_arm_command_
    allCategories_before_install.csv", sep = ",", rm.duplicates= FALSE)
rules_arm_command_allCategories_before_install <- apriori(arule_arm_command_allCategories_before_
    install,parameter = list(supp=0.04,conf=0.8))
#rules_arm_command_allCategories <- apriori(arule_arm_command_allCategories,parameter = list(supp
    =0.003,conf=1))
rules_arm_command_allCategories_before_install.sorted <- sort(rules_arm_command_allCategories_
    before_install,by="count")
inspect(rules_arm_command_allCategories_before_install.sorted)
write(rules_arm_command_allCategories_before_install.sorted, file = "rules_arm_command_
    allCategories.csv", sep = ",")
plot(rules_arm_command_allCategories_before_install.sorted, method = "scatterplot", engine = "
    htmlwidget", control = list(max = 10))
41
#2(c) Which commands occur together in install phase?
arm_command_allCategories_install<-sqldf::sqldf("select distinct command, project from [superDS]
    where phase = 'install'")
View(arm_command_allCategories_install)
write.csv(arm_command_allCategories_install, "arm_command_allCategories_install.csv")
#I run numbers, delete 1st column and 1 row of the above csv.
#Then run arules.py on the resultant after giving proper names to files.
#Created arm_command_allCategories with arule source data. Now will apply apriori:
arule_arm_command_allCategories_install <- read.transactions("arule_arm_command_allCategories_
    install.csv", sep = ",", rm.duplicates= FALSE)
rules_arm_command_allCategories_install <- apriori(arule_arm_command_allCategories_install,
    parameter = list(supp=0.001,conf=0.7))
```

```
#rules_arm_command_allCategories <- apriori(arule_arm_command_allCategories,parameter = list(supp
    =0.003,conf=1))
rules_arm_command_allCategories_install.sorted <- sort(rules_arm_command_allCategories_install,by="
    count")
inspect(rules_arm_command_allCategories_install.sorted)
plot(rules_arm_command_allCategories_install.sorted, method = "scatterplot", engine = "htmlwidget",
     control = list(max = 10))
7
#2(d) Which commands occur together in after_success phase?
arm_command_allCategories_after_success<-sqldf::sqldf("select distinct command, project from [
    superDS] where phase = 'after_success'")
View(arm_command_allCategories_after_success)
write.csv(arm_command_allCategories_after_success, "arm_command_allCategories_after_success.csv")
#I run numbers, delete 1st column and 1 row of the above csv.
#Then run arules.py on the resultant after giving proper names to files.
#Created arm_command_allCategories with arule source data. Now will apply apriori:
arule_arm_command_allCategories_after_success <- read.transactions("arule_arm_command_allCategories
    _after_success.csv", sep = ",", rm.duplicates= FALSE)
rules_arm_command_allCategories_after_success <- apriori(arule_arm_command_allCategories_after_
    success,parameter = list(supp=0.001,conf=0.7))
rules_arm_command_allCategories_after_success.sorted <- sort(rules_arm_command_allCategories_after_
    success,by="count")
inspect(rules_arm_command_allCategories_after_success.sorted)
plot(rules_arm_command_allCategories_after_success.sorted, method = "scatterplot", engine = "
    htmlwidget", control = list(max = 10))
13

#2(e) Which commands occur together in before_script phase?
arm_command_allCategories_before_script<-sqldf::sqldf("select distinct command, project from [
    superDS] where phase = 'before_script'")
View(arm_command_allCategories_before_script)
write.csv(arm_command_allCategories_before_script, "arm_command_allCategories_before_script.csv")
#I run numbers, delete 1st column and 1 row of the above csv.
#Then run arules.py on the resultant after giving proper names to files.
#Created arm_command_allCategories with arule source data. Now will apply apriori:
arule_arm_command_allCategories_before_script <- read.transactions("arule_arm_command_allCategories
    _before_script.csv", sep = ",", rm.duplicates= FALSE)
rules_arm_command_allCategories_before_script <- apriori(arule_arm_command_allCategories_before_
    script,parameter = list(supp=0.001,conf=0.7))
#rules_arm_command_allCategories <- apriori(arule_arm_command_allCategories,parameter = list(supp
    =0.003,conf=1))
rules_arm_command_allCategories_before_script.sorted <- sort(rules_arm_command_allCategories_before
    _script,by="count")
inspect(rules_arm_command_allCategories_before_script.sorted)
plot(rules_arm_command_allCategories_before_script.sorted, method = "scatterplot", engine = "
    htmlwidget", control = list(max = 10))
```

Program B.4: ARM_Categories.R

```
library(readr)
library(arules)
library(arulesViz)
superDS <- read_csv("superDS.csv")
#2(a). Which categories of commands occur together under script phase?

arm_category_script<-sqldf::sqldf("select distinct category, project from [superDS] where phase = '
    script' and category !='other'")
View(arm_category_script)
write.csv(arm_category_script, "arm_category_script.csv")
```

```r
#I run numbers, delete 1st column and 1 row of the above csv.
#Then run arules.py on the resultant after giving proper names to files.
#Created arule_arm_category with arule source data. Now will apply apriori:
arule_arm_category_script <- read.transactions("arule_arm_category_script.csv", sep = ",", rm.
    duplicates= FALSE)
rules_arm_category_script <- apriori(arule_arm_category_script,parameter = list(supp=0.001,conf
    =0.7))
#rules_arm_category <- apriori(arule_arm_category,parameter = list(supp=0.002,conf=1))
rules_arm_category_script.sorted <- sort(rules_arm_category_script,by="count")
inspect(rules_arm_category_script.sorted)
plot(rules_arm_category_script.sorted, method = "scatterplot", engine = "htmlwidget", control =
    list(max = 10))

#2(b). Which categories of commands occur together under before_install phase?

arm_category_before_install<-sqldf::sqldf("select distinct category, project from [superDS] where
    phase = 'before_install' and category !='other'")
View(arm_category_before_install)
write.csv(arm_category_before_install, "arm_category_before_install.csv")
#I run numbers, delete 1st column and 1 row of the above csv.
#Then run arules.py on the resultant after giving proper names to files.
#Created arule_arm_category with arule source data. Now will apply apriori:
arule_arm_category_before_install <- read.transactions("arule_arm_category_before_install.csv", sep
     = ",", rm.duplicates= FALSE)
#rules_arm_category_before_install <- apriori(arule_arm_category_before_install,parameter = list(
    supp=0.001,conf=0.8))
rules_arm_category_before_install <- apriori(arule_arm_category_before_install,parameter = list(
    supp=0.001,conf=0.7))
rules_arm_category_before_install.sorted <- sort(rules_arm_category_before_install,by="count")
inspect(rules_arm_category_before_install.sorted)
plot(rules_arm_category_before_install.sorted, method = "scatterplot", engine = "htmlwidget",
    control = list(max = 10))

#2(c). Which categories of commands occur together under install phase?

arm_category_install<-sqldf::sqldf("select distinct category, project from [superDS] where phase =
    'install' and category !='other'")
#View(arm_category_install)
write.csv(arm_category_install, "arm_category_install.csv")
#I run numbers, delete 1st column and 1 row of the above csv.
#Then run arules.py on the resultant after giving proper names to files.
#Created arule_arm_category with arule source data. Now will apply apriori:
arule_arm_category_install <- read.transactions("arule_arm_category_install.csv", sep = ",", rm.
    duplicates= FALSE)
rules_arm_category_install <- apriori(arule_arm_category_install,parameter = list(supp=0.001,conf
    =0.7))
#rules_arm_category_install <- apriori(arule_arm_category_install,parameter = list(supp=0.002,conf
    =1))
rules_arm_category_install.sorted <- sort(rules_arm_category_install,by="count")
inspect(rules_arm_category_install.sorted)
plot(rules_arm_category_install.sorted, method = "scatterplot", engine = "htmlwidget", control =
    list(max = 10))

#2(d). Which categories of commands occur together under after_success phase?

arm_category_after_success<-sqldf::sqldf("select distinct category, project from [superDS] where
    phase = 'after_success' and category !='other'")
#View(arm_category_after_success)
write.csv(arm_category_after_success, "arm_category_after_success.csv")
```

```
#I run numbers, delete 1st column and 1 row of the above csv.
#Then run arules.py on the resultant after giving proper names to files.
#Created arule_arm_category with arule source data. Now will apply apriori:
arule_arm_category_after_success <- read.transactions("arule_arm_category_after_success.csv", sep =
      ",", rm.duplicates= FALSE)
rules_arm_category_after_success <- apriori(arule_arm_category_after_success,parameter = list(supp
    =0.001,conf=0.7))
#rules_arm_category_after_success <- apriori(arule_arm_category_after_success,parameter = list(supp
    =0.002,conf=1))
rules_arm_category_after_success.sorted <- sort(rules_arm_category_after_success,by="count")
inspect(rules_arm_category_after_success.sorted)
plot(rules_arm_category_after_success.sorted, method = "scatterplot", engine = "htmlwidget",
    control = list(max = 10))

#2(e). Which categories of commands occur together under before_script phase?

arm_category_before_script<-sqldf::sqldf("select distinct category, project from [superDS] where
    phase = 'before_script' and category !='other'")
#View(arm_category_before_script)
write.csv(arm_category_before_script, "arm_category_before_script.csv")
#I run numbers, delete 1st column and 1 row of the above csv.
#Then run arules.py on the resultant after giving proper names to files.
#Created arule_arm_category with arule source data. Now will apply apriori:
arule_arm_category_before_script <- read.transactions("arule_arm_category_before_script.csv", sep =
      ",", rm.duplicates= FALSE)
rules_arm_category_before_script <- apriori(arule_arm_category_before_script,parameter = list(supp
    =0.001,conf=0.7))
#rules_arm_category_before_script <- apriori(arule_arm_category_before_script,parameter = list(supp
    =0.002,conf=1))
rules_arm_category_before_script.sorted <- sort(rules_arm_category_before_script,by="count")
inspect(rules_arm_category_before_script.sorted)
plot(rules_arm_category_before_script.sorted, method = "scatterplot", engine = "htmlwidget",
    control = list(max = 10))
```

Program B.5: ARM_CommandsAcrossPhases.R

```
#Which commands occur together across phases?
library(arules)
library(arulesViz)
arm_command_allPhases<-sqldf::sqldf("select distinct command, project from [superDS]")
View(arm_command_allPhases)
write.csv(arm_command_allPhases, "arm_command_allPhases.csv")
#I run numbers, delete 1st column and 1 row of the above csv.
#Then run arules.py on the resultant after giving proper names to files.
#Created arm_command_allPhases with arule source data. Now will apply apriori:
arule_arm_command_allPhases <- read.transactions("arule_arm_command_allPhases.csv", sep = ",", rm.
    duplicates= FALSE)
rules_arm_command_allPhases <- apriori(arule_arm_command_allPhases,parameter = list(supp=0.02,conf
    =0.8))
rules_arm_command_allPhases.sorted <- sort(rules_arm_command_allPhases,by="count")
inspect(rules_arm_command_allPhases.sorted)
#write(rules_arm_command_allPhases.sorted, file = "rules_arm_command_allPhasesFile2.csv", sep =
    ",")
plot(rules_arm_command_allPhases.sorted, method = "scatterplot", engine = "htmlwidget", control =
    list(max = 10))
```

Program B.6: ARM_CategoriesAcrossPhases.R

```r
#Which categories of commands occur together under script phase?

arm_categoryAllPhases<-sqldf::sqldf("select distinct category, project from [superDS] where category !='other'")
View(arm_categoryAllPhases)
write.csv(arm_categoryAllPhases, "arm_categoryAllPhases.csv")
#I run numbers, delete 1st column and 1 row of the above csv.
#Then run arules.py on the resultant after giving proper names to files.
#Created arule_arm_categoryAllPhases with arule source data. Now will apply apriori:
arule_arm_categoryAllPhases <- read.transactions("arule_arm_categoryAllPhases.csv", sep = ",", rm.duplicates= FALSE)
rules_arm_categoryAllPhases <- apriori(arule_arm_categoryAllPhases,parameter = list(supp=0.009,conf=0.7))
rules_arm_categoryAllPhases.sorted <- sort(rules_arm_categoryAllPhases,by="count")
inspect(rules_arm_categoryAllPhases.sorted)
#write(rules_arm_categoryAllPhases.sorted, file = "rules_arm_categoryAllPhasesFile3.csv", sep = ",")
plot(rules_arm_categoryAllPhases.sorted, method = "scatterplot", engine = "htmlwidget", control = list(max = 10))
```

Program B.7: CohenKappa.R

```r
install.packages('psych')
library(psych)

# Puneet
profOrig = c("builders","builders","interpreter","internet","pkg_mgr","pkg_mgr","pkg_mgr","pkg_mgr","other_unix","vcs","travis_command","compress","env_setup","compress","fs","fs","not_mutate","mobile_framework","not_mutate","builders","execute_script","execute_script","execute_script","pkg_mgr","fs","not_mutate","env_setup","builders","other_unix","not_mutate","not_mutate","other_unix","not_mutate","interpreter","text_manipulate","not_mutate","text_manipulate","execute_script","builders","other_unix","other_unix","other_unix","execute_script","pkg_installers","pkg_mgr","not_mutate","other","interpreter","fs","security","code_coverage","execute_script","execute_script","fs","env_setup","database","execute_script","execute_script","","env_setup","not_mutate","interpreter","database","compress","interpreter","database","execute_script","fs","execute_script","execute_script","not_mutate","compiler","compiler","interpreter","env_setup","internet","not_mutate","builders","execute_script","execute_script","process_mgmt","security","interpreter","security","security","fs","fs","compress","internet","other","security","interpreter","builders","execute_script","database","execute_script","execute_script","execute_script","execute_script","???","execute_script","other","database","database","travis_command","not_mutate","execute_script","execute_script","env_setup","compiler","fs","other_unix","text_manipulate","interpreter","interpreter","fs","pkg_installers","pkg_mgr","pkg_mgr","interpreter","builders","execute_script","execute_script","execute_script","execute_script","other_unix","execute_script","fs","env_setup","execute_script","sca","pkg_mgr","execute_script","other_unix","execute_script","pkg_installers","compress","not_mutate","pkg_mgr","interpreter","builders","fs","travis_command","interpreter","interpreter","execute_script","execute_script","travis_command","execute_script","execute_script","execute_script","execute_script","text_manipulate","env_setup","not_mutate","browser_env","code_coverage","env_setup","pkg_installers","interpreter","interpreter","other","other","security","not_mutate","execute_script","execute_script","execute_script","other_unix","execute_script","execute_script","execute_script","other","other","other","other","interpreter","builders","other_unix","builders","process_mgmt","process_mgmt","other","sca","vcs","other_unix","env_setup","execute_script","compiler","pkg_mgr","execute_script","execute_script","compiler","execute_script","execute_script","execute_script","interpreter","internet","execute_script","process_mgmt","sca","other_unix","execute_script","code_coverage","env_setup","database","execute_script","compiler","other","other","interpreter","text_manipulate","text_manipulate","fs","env_setup","env_setup","interpreter","compiler","other","browser_env","security","builders","pkg_mgr","builders","builders","execute_
script","sca","pkg_installers","pkg_mgr","env_setup","other","anti-virus","not_mutate","execute_script","other_unix","builders","builders","interpreter","compress","internet","not_mutate","database","interpreter","","builders")
puneetOrig = c("builders","builders","interpreter","internet","pkg_installers","pkg_installers","pkg_installers","pkg_mgr","other","vcs","travis_command","compress","other_unix","compress","fs","fs","not_mutate","mobile_framework","not_mutate","compiler","execute_script","execute_script","execute_script","pkg_mgr","fs","not_mutate","not_mutate","builders","other","not_mutate","other_unix","other","not_mutate","interpreter","text_manipulate","text_manipulate","text_manipulate","execute_script","builders","not_mutate","builders","other","execute_script","version_mgr","pkg_mgr","other","env_setup","interpreter","fs","security","code_coverage","execute_script","execute_script","security","not_mutate","storage","execute_script","execute_script","execute_script","daemon_runner","daemon_runner","interpreter","storage","compress","env_setup","storage","execute_script","fs","execute_script","execute_script","not_mutate","compiler","compiler","interpreter","other","internet","execute_script","builders","execute_script","execute_script","other","security","other_unix","security","security","not_mutate","not_mutate","compress","internet","other","other_unix","other","other","execute_script","storage","execute_script","execute_script","execute_script","execute_script","other","execute_script","other","storage","storage","travis_command","text_manipulate","execute_script","env_setup","fs","not_mutate","security","other","other","pkg_installers","execute_script","fs","version_mgr","pkg_mgr","pkg_installers","web_framework","builders","execute_script","execute_script","execute_script","execute_script","other","execute_script","fs","execute_script","execute_script","other","pkg_mgr","execute_script","browser_env","execute_script","env_setup","fs","fs","pkg_installers","interpreter","pkg_mgr","fs","travis_command","other","pkg_installers","execute_script","execute_script","travis_command","execute_script","execute_script","execute_script","execute_script","text_manipulate","web_framework","not_mutate","pkg_mgr","other","security","pkg_mgr","pkg_mgr","not_mutate","other","other","security","other","execute_script","execute_script","execute_script","other","execute_script","execute_script","execute_script","other","env_setup","other","other","other","pkg_mgr","other_unix","builders","process_kill","other","other","other","vcs","other","other","execute_script","other","other","other","execute_script","other","execute_script","execute_script","execute_script","other","other","execute_script","other_node_modules","other","browser_env","other","code_coverage","security","storage","other","sca","other","other","pkg_installers","not_mutate","security","fs","env_setup","pkg_installers","other","other","other","web_framework","security","fs","pkg_installers","builders","execute_script","daemon_runner","other","env_setup","interpreter","other","other","security","not_mutate","pkg_installers","other","execute_script","execute_script","pkg_mgr","other","security","other","storage","other","not_mutate","other")
cohen.kappa(x=cbind(profOrig, puneetOrig))
catAgree1.df <- data.frame(profOrig, puneetOrig)
ck1 <- cohen.kappa(catAgree1.df)
ck1
ck1$kappa
#= 0.4617242
#---------------------
# With consenses of all authors
profLatest = c("builders","builders","interpreter","internet","pkg_mgr","pkg_mgr","pkg_mgr","pkg_mgr","other","vcs","travis_command","compress","env_setup","compress","fs","fs","not_mutate","mobile_framework","not_mutate","builders","execute_script","execute_script","execute_script","pkg_mgr","fs","not_mutate","env_setup","builders","other","not_mutate","not_mutate","other","not_mutate","interpreter","text_manipulate","text_manipulate","text_manipulate","execute_script","builders","not_mutate","env_setup","other","execute_script","pkg_mgr","pkg_mgr","other","execute_script","interpreter","fs","security","sca","execute_script","execute_script","security","env_setup","database","execute_script","execute_script","execute_script","process_mgmt","not_mutate","interpreter","database","compress","interpreter","database","execute_script","fs","execute_script","execute_script","not_mutate","compiler","compiler","interpreter","env_setup","internet","execute_script","builders","execute_script","execute_script","process_mgmt","security","execute_script","security","security","not_mutate","not_mutate","compress","internet","other","security","interpreter","builders","execute_script","database","execute_script","execute_script","execute_script","execute_script","other","execute_script","other","database","database","travis_command","text_manipulate","execute_script","execute_script","env_setup","
    compiler","security","other","text_manipulate","interpreter","execute_script","fs","pkg_mgr","
    pkg_mgr","pkg_mgr","other","builders","execute_script","execute_script","execute_script","
    execute_script","other","execute_script","fs","not_mutate","execute_script","sca","pkg_mgr","
    execute_script","browser_env","execute_script","pkg_mgr","compress","not_mutate","pkg_mgr","
    interpreter","builders","fs","travis_command","env_setup","interpreter","execute_script","
    execute_script","travis_command","execute_script","execute_script","execute_script","execute_
    script","text_manipulate","env_setup","not_mutate","pkg_mgr","sca","env_setup","pkg_mgr","pkg_
    mgr","builders","other","other","security","not_mutate","execute_script","execute_script","
    execute_script","other","execute_script","execute_script","execute_script","other","execute_
    script","other","other","compiler","pkg_mgr","other","builders","process_mgmt","process_mgmt","
    other","sca","vcs","text_manipulate","env_setup","execute_script","compiler","pkg_mgr","execute
    _script","execute_script","compiler","execute_script","execute_script","execute_script","
    interpreter","internet","execute_script","process_mgmt","sca","execute_script","execute_script"
    ,"sca","env_setup","database","execute_script","vcs","other","other","interpreter","text_
    manipulate","not_mutate","fs","env_setup","env_setup","interpreter","compiler","other","browser
    _env","security","builders","pkg_mgr","builders","builders","process_mgmt","other","pkg_mgr","
    builders","env_setup","other","security","not_mutate","execute_script","other","builders","
    builders","other","compress","internet","not_mutate","database","interpreter","builders")
puneetLatest = c("builders","builders","interpreter","internet","pkg_mgr","pkg_mgr","pkg_mgr","pkg_
    mgr","other","vcs","travis_command","compress","env_setup","compress","fs","fs","not_mutate","
    mobile_framework","not_mutate","builders","execute_script","execute_script","execute_script","
    pkg_mgr","fs","not_mutate","env_setup","builders","other","not_mutate","not_mutate","other","
    not_mutate","interpreter","text_manipulate","text_manipulate","text_manipulate","execute_script
    ","builders","not_mutate","env_setup","other","execute_script","pkg_mgr","pkg_mgr","other","
    execute_script","interpreter","fs","security","sca","execute_script","execute_script","security
    ","env_setup","database","execute_script","execute_script","execute_script","process_mgmt","not
    _mutate","interpreter","database","compress","interpreter","database","execute_script","fs","
    execute_script","execute_script","not_mutate","compiler","compiler","interpreter","env_setup","
    internet","execute_script","builders","execute_script","execute_script","process_mgmt","
    security","execute_script","security","security","not_mutate","not_mutate","compress","internet
    ","other","security","interpreter","builders","execute_script","database","execute_script","
    execute_script","execute_script","execute_script","other","execute_script","other","database","
    database","travis_command","text_manipulate","execute_script","execute_script","env_setup","
    compiler","security","other","text_manipulate","interpreter","execute_script","fs","pkg_mgr","
    pkg_mgr","pkg_mgr","other","builders","execute_script","execute_script","execute_script","
    execute_script","other","execute_script","fs","not_mutate","execute_script","sca","pkg_mgr","
    execute_script","browser_env","execute_script","pkg_mgr","compress","not_mutate","pkg_mgr","
    interpreter","builders","fs","travis_command","env_setup","interpreter","execute_script","
    execute_script","travis_command","execute_script","execute_script","execute_script","execute_
    script","text_manipulate","env_setup","not_mutate","pkg_mgr","sca","env_setup","pkg_mgr","pkg_
    mgr","builders","other","other","security","not_mutate","execute_script","execute_script","
    execute_script","other","execute_script","execute_script","execute_script","other","execute_
    script","other","other","compiler","pkg_mgr","other","builders","process_mgmt","process_mgmt","
    other","sca","vcs","text_manipulate","env_setup","execute_script","compiler","pkg_mgr","execute
    _script","execute_script","compiler","execute_script","execute_script","execute_script","
    interpreter","internet","execute_script","process_mgmt","sca","execute_script","execute_script"
    ,"sca","env_setup","database","execute_script","vcs","other","other","interpreter","text_
    manipulate","not_mutate","fs","env_setup","env_setup","interpreter","compiler","other","browser
    _env","security","builders","pkg_mgr","builders","builders","process_mgmt","other","pkg_mgr","
    builders","env_setup","other","security","not_mutate","execute_script","other","builders","
    builders","other","compress","internet","not_mutate","database","interpreter","builders")

cohen.kappa(x=cbind(profLatest, puneetLatest))
catAgree2.df <- data.frame(profLatest, puneetLatest)
ck2 <- cohen.kappa(catAgree2.df)
ck2
ck2$kappa
#=1
```

C
UMPLE and Java Code for the Composition Mechanism

The UMPLE code below in Program C.1 defines the metamodel of CORE and SpecFile together. We generate Java code from it which we use to create our composition (weaving) algorithm in Listing C.2. In Listing C.3, we show a test case where we take two sample SpecFiles shown in Figure C.1 and weave them to form a complete `.travis.yml` file shown in Figure C.2. In Listing C.4, we show the test case for a real project - ServiceCutter also shown in Program 5.1 - followed by the desired output in Figure C.3.

Program C.1: Metamodel.ump

```
namespace ca.mcgill.core.travisconcern;

class COREModel{
  1 coremodel <@>- * CORECompositionSpecification reuses;
}

class COREFeatureModel{
  isA COREModel;
  1 model <@>- * COREFeature features;
  1 model <@>- * CORERelatedFeature relatedFeatures;
  1 -> 1 COREFeature root;
  COREFeatureModel(){};
}

class COREFeature{
  enum COREFeatureRelationshipType { None, Optional, XOR, OR, Mandatory }
  1..* targetPattern -- * CORERelatedFeature incoming;
  1..* sourcePattern -- * CORERelatedFeature outgoing;
  0..1 parent -- * COREFeature children;
  * realizes -- 0..1 COREModel realizedBy; // The 0..1 association is a simplification of the
      actual CORE metamodel
  COREFeatureRelationshipType parentRelationship;
  1 -- * OrderedConfiguration;
}

class CORERelatedFeature{
  Float supportValue;
}

class COREConfiguration{
  isA CORECompositionSpecification;
  * -> * COREFeature selected;
  1 -- * OrderedConfiguration orderedConf;
}

class CORECompositionSpecification{
}

class SpecFile{
  isA COREModel;
  fname;
  1 sFile <@>- * Group groups;
  0..1 <@>- * Line headers; // header lines are not associated with arguments
}

class Group{
  gname;
  0..1 <@>- * Line g_lines;
}

class Line{
  cname;
  Boolean partial;
  1 line <@>- * Argument arguments;
}

class Argument{
  aname;
  Boolean partial;
}

class OrderedConfiguration{
  Integer position;
}
```

Program C.2: Weaver.java

```
package ca.mcgill.core.travisconcern;

import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;

/**
 * This class creates .travis.yml file based on feature selection
 *
 * @author Puneet Kaur Sidhu
 */
public class Weaver {

    /**
     * Weave all the configurations into the model
     * @param cFM Core Feature Model
     * @param conf Core Feature Configuration
     */
    public void weaveAll(COREFeatureModel cFM, COREConfiguration conf) {
        //Collecting all the selected features and the root
        COREFeature root = cFM.getRoot();
        SpecFile first = (SpecFile) root.getRealizedBy();
        for(OrderedConfiguration oC : conf.getOrderedConf()) {
            COREFeature f = oC.getCOREFeature();
            if(f != root) {
                SpecFile second = (SpecFile) f.getRealizedBy();
                first = weave(first, second);
            }
        }
        saveFile(first);
        printFile(first);
    }

    /**
     * Method to weave the lower spec file into the higher spec file
     * @param higherSpecFile File to merge the lower spec file to
     * @param lowerSpecFile Spec File to be merged
     * @return sfNew result of the merge of two spec files
     */
    public SpecFile weave(SpecFile higherSpecFile, SpecFile lowerSpecFile) {
        //Replicating the root SpecFile
        SpecFile sfNew = new SpecFile(higherSpecFile.getFname());
        /* Combining headers of higher and lower
         * spec files together in higher spec file */
        addHeadersToNewSpecFile(higherSpecFile, sfNew);
        addHeadersToNewSpecFile(lowerSpecFile, sfNew);

        //Replicating groups of higherSpecFile to the new root file object
        for (Group g : higherSpecFile.getGroups()) {
            Group gNew = new Group(g.getGname(), sfNew);
            merge(gNew, g);
        }


        /* Getting groups of lower spec file
         * and adding them to root file one by one */
        for(Group g : lowerSpecFile.getGroups()) {
            weave(sfNew, g);
        }
        return sfNew;
    }

    /**
     * Method to add lines of the original spec file to the new spec file
     * @param origFile original spec file
     * @param newFile new spec file
     */
    private void addHeadersToNewSpecFile(SpecFile origFile, SpecFile newFile) {
        /* Since headers do not have arguments,
         * thus, they have not been considered here */
        for (Line l : origFile.getHeaders()) {
            Line hNew = new Line(l.getCname(), l.getPartial());
            newFile.addHeader(hNew);
        }
    }

    /**
```

```
     * Merge the group in the spec file
     * @param higherSpecFile the spec file
     * @param lowerGroup group to be weaved in
     */
    private void weave(SpecFile higherSpecFile, Group lowerGroup) {
        int flag = 0;
        for(Group higherGroup : higherSpecFile.getGroups()) {

            /*Checking if the group names in root file and lower file match
             * If yes, then we merge the contents of those groups
             * Else, we create a new group in the root file */
            if(higherGroup.getGname().equals(lowerGroup.getGname())) {
                merge(higherGroup, lowerGroup);
                flag = 1;
                break;
            }
        }
        if (flag == 0) {
            Group gNew = new Group(lowerGroup.getGname(), higherSpecFile);
            merge(gNew, lowerGroup);
        }
    }

    /**
     * Merge two groups
     * @param higherGroup new group
     * @param lowerGroup group to be merged
     */
    private void merge(Group higherGroup, Group lowerGroup) {
        //Merging contents of two groups
        for(Line l : lowerGroup.getG_lines()) {
            Line lNew = new Line(l.getCname(), l.getPartial());
            lNew.setGroup(higherGroup);
            for(Argument a : l.getArguments()) {
                Argument aNew = new Argument(a.getAname(), a.getPartial(), lNew);
            }
        }
    }

    /**
     * Save the .travis.yml specification file in the project directory
     * @param sf new spec file
     */
    public void saveFile(SpecFile sf) {
        try(FileWriter fw = new FileWriter(sf.getFname(), false);
                BufferedWriter bw = new BufferedWriter(fw);
                PrintWriter out = new PrintWriter(bw)) {
            for (Line h : sf.getHeaders()) {
                if (h.isPartial() == true) {
                    out.println("|" + h.getCname());
                }
                else{
                    out.println(h.getCname());
                }
            }
            for(Group g : sf.getGroups()) {
                out.println(g.getGname() + ":");
                for(Line l : g.getG_lines()) {
                    if (l.getArguments().size() > 0) {
```

```java
                        String wholeLine = null;
                        wholeLine = " - " + l.getCname() + " ";
                        for(Argument a : l.getArguments()) {
                            if(a.isPartial() == true) {
                                wholeLine = wholeLine + "|" + a.getAname() + " ";
                            }
                            else {
                                wholeLine = wholeLine + a.getAname() + " ";
                            }
                        }
                        out.println(wholeLine);
                    }
                    else {
                        out.println(" - " + l.getCname());
                    }
                }
            }
        }
        catch (IOException e) {
            System.out.println(e.getMessage());
        }
    }

    /**
     * Prints the .travis.yml specification file on the console
     * @param sf new spec file
     */
    public void printFile(SpecFile sf) {
        for (Line h : sf.getHeaders()) {
            if (h.isPartial() == true) {
                System.out.println("|" + h.getCname());
            }
            else{
                System.out.println(h.getCname());
            }
        }
        for(Group g : sf.getGroups()) {
            System.out.println(g.getGname() + ":");
            for(Line l : g.getG_lines()) {
                if (l.getArguments().size() > 0) {
                    String wholeLine = null;
                    wholeLine = " - " + l.getCname() + " ";
                    for(Argument a : l.getArguments()) {
                        if(a.isPartial() == true) {
                            wholeLine = wholeLine + "|" + a.getAname() + " ";
                        }
                        else {
                            wholeLine = wholeLine + a.getAname() + " ";
                        }
                    }
                    System.out.println(wholeLine);
                }
                else {
                    System.out.println(" - " + l.getCname());
                }
            }
        }
    }
}
```

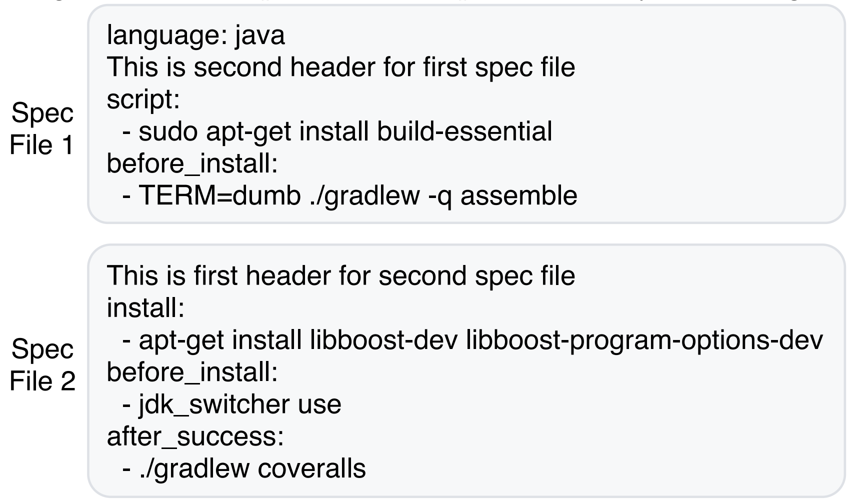
Program C.3: TestCase.java

```java
package ca.mcgill.core.travisconcern;

import ca.mcgill.core.travisconcern.SpecFile;
import ca.mcgill.core.travisconcern.Argument;
import ca.mcgill.core.travisconcern.Line;
import ca.mcgill.core.travisconcern.Group;
import ca.mcgill.core.travisconcern.Weaver;
import java.util.ArrayList;
import java.util.List;

public class SpecsWeaver{

    public static void main(String[] args) {
        try {
            //Spec File 1
            SpecFile f1 = new SpecFile(".travis.yml");
            Line l0 = new Line("language: java", false);
            f1.addHeader(l0);
            Line l00 = new Line("This is second header for first spec file", false);
            f1.addHeader(l00);
            Group g1 = new Group("script", f1);
            Line l1 = new Line("sudo apt-get install", false);
            g1.addG_line(l1);
            Argument a0 = new Argument("build-essential", false, l1);
            Group g12 = new Group("before_install", f1);
            Line l2 = new Line("TERM=dumb ./gradlew -q assemble", false);
            g12.addG_line(l2);
            //Spec File 2
            SpecFile f2 = new SpecFile("");
            Line l01 = new Line("This is first header for second spec file", false);
            f2.addHeader(l01);
            Group g2 = new Group("install", f2);
            Line l3 = new Line("apt-get install", false);
            g2.addG_line(l3);
            Argument a1 = new Argument("libboost-dev", false, l3);
            Argument a2 = new Argument("libboost-program-options-dev", false, l3);
            Group g3 = new Group("before_install", f2);
            Line l4 = new Line("jdk_switcher use", false);
            g3.addG_line(l4);
            Group g4 = new Group("after_success", f2);
            Line l5 = new Line("./gradlew coveralls", false);
            g4.addG_line(l5);

            // Testing the composition mechanism
            Weaver obj = new Weaver();
            SpecFile finalFile = obj.weave(f1, f2);
            obj.saveFile(finalFile);
            obj.printFile(finalFile);
        }
        catch(Exception e) {
            System.out.println(e.getMessage());
        }
    }
}
```

Test `.travis.yml` custom file

This example shows how we combine two SpecFiles, SpecFile 1 and 2 shown in Figure C.1 with different headers, different groups with their lines and arguments, merging of lines and arguments under the same groups. The final file after the execution of our composition algorithm is shown in Figure C.2.

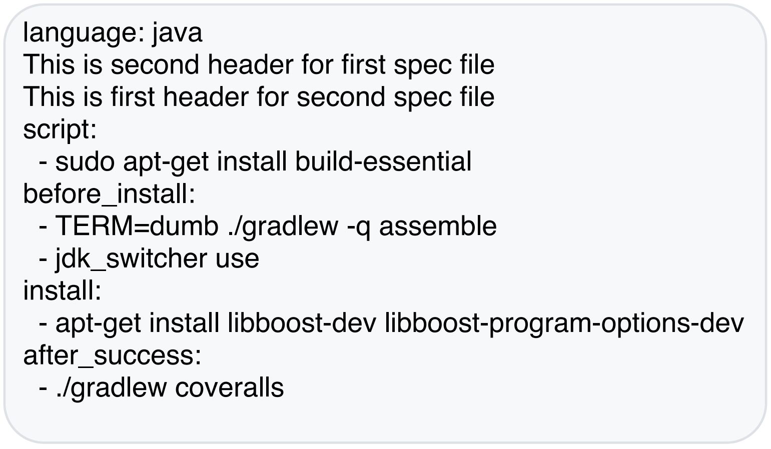
Figure C.1: Test SpecFile 1 and SpecFile 2 ready to be merged


Spec File 1
Spec File 2

```
language: java
This is second header for first spec file
script:
  - sudo apt-get install build-essential
before_install:
  - TERM=dumb ./gradlew -q assemble
```

```
This is first header for second spec file
install:
  - apt-get install libboost-dev libboost-program-options-dev
before_install:
  - jdk_switcher use
after_success:
  - ./gradlew coveralls
```

Figure C.2: Composed SpecFile 1 and 2 to create a final `.travis.yml` specification



```
language: java
This is second header for first spec file
This is first header for second spec file
script:
  - sudo apt-get install build-essential
before_install:
  - TERM=dumb ./gradlew -q assemble
  - jdk_switcher use
install:
  - apt-get install libboost-dev libboost-program-options-dev
after_success:
  - ./gradlew coveralls
```

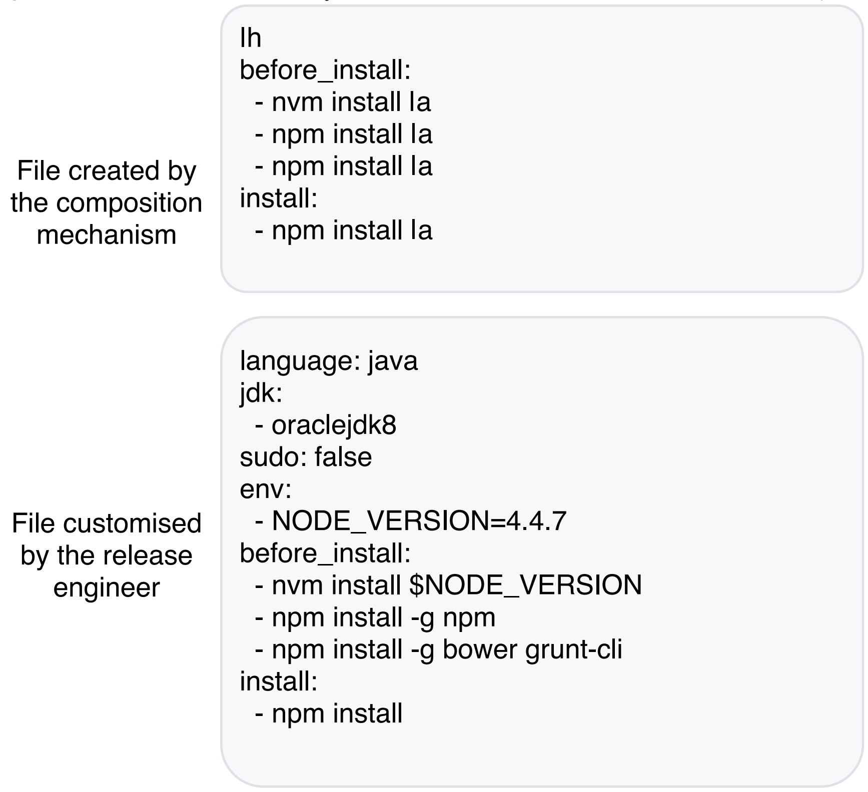
Program C.4: TestCase_ServiceCutter.java

```java
package ca.mcgill.core.travisconcern;

import ca.mcgill.core.travisconcern.SpecFile;
import ca.mcgill.core.travisconcern.Argument;
import ca.mcgill.core.travisconcern.Line;
import ca.mcgill.core.travisconcern.COREConfiguration;
import ca.mcgill.core.travisconcern.COREFeature;
import ca.mcgill.core.travisconcern.COREFeature.COREFeatureRelationshipType;
import ca.mcgill.core.travisconcern.COREFeatureModel;
import ca.mcgill.core.travisconcern.Group;
import ca.mcgill.core.travisconcern.Weaver;
import ca.mcgill.core.travisconcern.OrderedConfiguration;

public class TestCase_ServiceCutter {
    public static void main(String[] args) {
        try {
            SpecFile f0 = new SpecFile(".travis.yml");
            Line l0 = new Line("h", true);
            f0.addHeader(l0);

            SpecFile f1 = new SpecFile("f1");
            Group g1 = new Group("before_install", f1);
            Line l1 = new Line("nvm install", false);
            g1.addG_line(l1);
            Argument a1 = new Argument("a", true, l1);

            SpecFile f2 = new SpecFile("f2");
            Group g2 = new Group("before_install", f2);
            Line l2 = new Line("npm install", false);
            g2.addG_line(l2);
            Argument a2 = new Argument("a", true, l2);

            // install phase
            SpecFile f3 = new SpecFile("f3");
            Group g3 = new Group("install", f3);
            Line l3 = new Line("npm install", false);
            g3.addG_line(l3);
            Argument a3 = new Argument("a", true, l3);

            //Initializing metamodel objects for spec files
            COREFeatureModel cFM = new COREFeatureModel();

            COREFeature c0 = new COREFeature(COREFeatureRelationshipType.Mandatory, cFM);
            cFM.setRoot(c0);
            c0.setRealizedBy(f0);

            COREFeature c1 = new COREFeature(COREFeatureRelationshipType.Mandatory, cFM);
            c1.setRealizedBy(f1);

            COREFeature c2 = new COREFeature(COREFeatureRelationshipType.Mandatory, cFM);
            c2.setRealizedBy(f2);

            COREFeature c3 = new COREFeature(COREFeatureRelationshipType.Mandatory, cFM);
            c3.setRealizedBy(f3);

            COREConfiguration conf = new COREConfiguration(cFM);
            conf.addSelected(c0);
            conf.addSelected(c1);
            conf.addSelected(c2);
            conf.addSelected(c3);

            OrderedConfiguration o1 = new OrderedConfiguration(1, c1, conf);
            OrderedConfiguration o2 = new OrderedConfiguration(2, c2, conf);
            OrderedConfiguration o3 = new OrderedConfiguration(3, c2, conf);
            OrderedConfiguration o4 = new OrderedConfiguration(4, c3, conf);

            Weaver obj = new Weaver();
            obj.weaveAll(cFM, conf);
        }
        catch(Exception e) {
            System.out.println(e.getMessage());
        }
    }
}
```

Test `.travis.yml` file for ServiceCutter project

This example shows the Travis Specification file for the ServiceCutter project shown in Program 5.1. As the first step in the reuse process, the release engineer selects the desired features from the Travis Feature Model. In this case, the same feature is selected twice for the before_install phase. Then, the composition mechanism creates a partially completed file shown in the top part of Figure C.3 in which the release engineer can add headers and arguments as required. The completed file is shown in the bottom part of Figure C.3 as per the ServiceCutter project.

Figure C.3: Test `.travis.yml` specification for ServiceCutter Project


File created by the composition mechanism
File customised by the release engineer

```
lh
before_install:
  - nvm install la
  - npm install la
  - npm install la
install:
  - npm install la
```

```
language: java
jdk:
  - oraclejdk8
sudo: false
env:
  - NODE_VERSION=4.4.7
before_install:
  - nvm install $NODE_VERSION
  - npm install -g npm
  - npm install -g bower grunt-cli
install:
  - npm install
```