Predicting Co-Changes between Functionality Specifications and Source Code in Behavior Driven Development

Aidan Z.H. Yang
Electrical and Computer Engineering
Queen's University
Kingston, Ontario, Canada
a.yang@queensu.ca

Daniel Alencar da Costa
Department of Information Science
University of Otago
Dunedin, Otago, New Zealand
danielcalencar@otago.ac.nz

Ying Zou
Electrical and Computer Engineering
Queen's University
Kingston, Ontario, Canada
ying.zou@queensu.ca

***Abstract*—Behavior Driven Development (BDD) is an agile approach that uses *.feature* files to describe the functionalities of a software system using natural language constructs (English-like phrases). Because of the English-like structure of *.feature* files, BDD specifications become an evolving documentation that helps all (even non-technical) stakeholders to understand and contribute to a software project. After specifying a *.feature* files, developers can use a BDD tool (e.g., Cucumber) to automatically generate test cases and implement the code of the specified functionality. However, maintaining traceability between *.feature* files and source code requires human efforts. Therefore, *.feature* files can be out-of-date, reducing the advantages of using BDD. Furthermore, existing research do not attempt to improve the traceability between *.feature* files and source code files. In this paper, we study the co-changes between *.feature* files and source code files to improve the traceability between *.feature* files and source code files. Due to the English-like syntax of *.feature* files, we use natural language processing to identify co-changes, with an accuracy of 79%. We study the characteristics of BDD co-changes and build random forest models to predict when a *.feature* files should be modified before committing a code change. The random forest model obtains an AUC of 0.77. The model can assist developers in identifying when a *.feature* files should be modified in code commits. Once the traceability is up-to-date, BDD developers can write test code more efficiently and keep the software documentation up-to-date.**

***Index Terms*—Behavior Driven Development, Traceability, Co-Changes, Empirical Software Engineering**

I. Introduction

To reduce the communication barriers between development teams and customers, a new form of testing strategy, *Behavior Driven Development* (BDD), became more prevalent in recent years [1]. BDD aims to combine business and technical interests using a domain-specific language (DSL) that is similar to the English language. In BDD, the stakeholders specify *scenarios* in *.feature* files to describe functionalities of software. The scenarios can be described by all stakeholders of a project due to the simplistic English-like language format of BDD. The stakeholders can be customers, who have little to no programming experience, but can still define software requirements (or understand the logic of an implementation) using *.feature* files. Because scenarios serve as a description of the functionality of a software, it is important to keep scenarios up-to-date as the functionalities of the software evolves. For example, when a new developer arrives in a project, having the *.feature* files up-to-date would ease the learning curve of the developer, i.e., the *.feature* files would capture the most current functionalities in a friendly and precise manner.

The *BDD co-changes* occur when a *.feature* file and its corresponding source code files are modified in the same commit, or over different commits but in a close time span. As a BDD project scales up, the traceability of these co-changes may become harder to grasp because the links between *.feature* files and their corresponding source code files naturally increase. An *out-of-sync* BDD co-change occurs when a *.feature* file and its corresponding source code files are modified in separate commits. The problem of maintaining BDD co-changes becomes more challenging when new developers enter into the project because they would have to identify the existing links between BDD files and source code files.

Existing research establishes the characteristics of BDD by studying large scale BDD projects in detail. Solís *et al.* [1] found that *ubiquitous language* and *Iterative Decomposition Process* are key characteristics of BDD. Other research work discussed the advantages of BDD in circuit design and verification [2]. Regarding testing strategies, Bhat *et al.* [3] evaluated the efficiency of Test Driven Development (TDD), and Zaidman *et al.* [4] explored the traceability of source and test code. With regards to co-changes, Mcintosh *et al.* [5] studied large scale software projects to predict co-changes between build system changes and source code changes. Although the characteristics of BDD and the traceability between test code and source code have been studied, maintaining the traceability between *.feature* files and source code remains unexplored. Maintaining such traceability is important because it keeps the documentation (i.e., software scenarios) up-to-date, help developers learn the project faster, and better enforce the adopted testing strategy. We use an illustrative example in *Section 2* to demonstrate the difficulty in maintaining *.feature* files and source code file co-changes.

Our work aims to help developers identify when *.feature*

files should be modified before committing code. After mining and analyzing 133 GitHub projects that use BDD, we address the following three research questions:

- **(RQ1) Can we accurately identify co-changes between *.feature* files and source code files?**
  We link *.feature* files and source code files by measuring the similarities between English phrases in *.feature* and source code. After performing a manual analysis to evaluate the quality of the established links, we conclude that we can automatically identify co-changes between *.feature* files and source code files with an accuracy of 79%.
- **(RQ2) Can we accurately predict when co-changes between *.feature* files and source code files are necessary?**
  To help developers determine the necessity of changing *.feature* files before committing source code files, we identify the characteristics that can predict the modification of *.feature* files within a commit. We use three classification techniques (random forest, Naive Bayes, and logistic regression) to predict corresponding *.feature* file co-changes when changes in source code are made. Our random forest, Naive Bayes and logistic regression classifiers can predict *.feature* file co-changes with a relatively high AUC of 0.77, 0.74, and 0.70, respectively.
- **(RQ3) What are the most significant characteristics for predicting co-changes between *.feature* files and source code files?**
  To identify the characteristics that are most important to predict co-changes between *.feature* files and source code files, we analyze the classification technique with the highest AUC (i.e., our random forest technique). We use a mean decrease AUC approach on the random forest classifier to identify the most important characteristics to predict co-changes between *.feature* file and source code files. We observe that *test files added* and *source LOC deleted* are the most influential characteristics.

The remainder of the paper is organized as follows. In *Section 2*, we provide a motivating example to our work. *Section 3* outlines the experiment setup. *Section 4* presents the results with respect to our three research questions, and *Section 5* discusses threats to the validity of our results. In *Section 6*, we examine related work and *Section 7* draws conclusions to the paper.

II. Motivating Example

We study the *Trivial-Graph* [6] GitHub project to demonstrate the importance of maintaining the traceability between *.feature* files and source code files. *Trivial-Graph* is a web-based trivia game application in which teams compete to answer questions in multiple rounds. *Trivial-Graph* uses the *Cucumber* BDD framework [6], [7] to write *.feature* files and *step definition* files. The *.feature* files describe the scenarios (i.e., functionalities) of the project. The *step definition* files are automatically generated by *Cucumber* based on the *.feature*

***.feature* file code**:

```
@players
Feature: Trivialt Players and Teams
Scenario: Register as a new player
When you register "Tobias" with handle
    "@thobe"
Then trivialt knows "@thobe" is "Tobias"
And "@thobe" should be the current player
```

Step definition file code:

```
@players
@When("^you register \"([^\"]*)\" with
    handle \"([^\"]*)\"$")
public void youRegisterUser(String name,
    String handle){
  currentPlayer =
      trivialtWorld.register(handle, name);
  assertThat(currentPlayer,
      is(not(nullValue())));
}
@Then("^trivialt knows \"([^\"]*)\" is
    \"([^\"]*)\"$")
public void trivialtKnowsPlayer(String
    handle, String name){
      Player foundPlayer =
          trivialtWorld.findPlayer(handle);
     assertThat(foundPlayer,
         is(not(nullValue())));
     assertThat(foundPlayer.getName(),
         is(name));
}
 @Then("^\"([^\"]*)\" should be the current
     player$")
public void assertTheCurrentPlayerIs(String
    handle){
  assertThat(currentPlayer.getHandle(),
      is(handle));
}
```

Fig. 1: Code example from BDD files

files. The *step definition* files define the tests for the functionalities specified in the *.feature* files. Figure 1 shows an example of a *.feature* file and its corresponding *step definition* file. A *step definition* file has annotations (e.g., "@When" and "@Then") to indicate which step of a *.feature* file is tested by a given test method. The developer then runs the tests on the source code using the *step definition* files, until every tests passes, satisfying the functions of each scenario

We observe co-changes between *.feature* files and source code files in the *Trivial-Graph* by studying six commits in a one month time frame during 2011 (shown in Figure 2). Figure 2 shows an out-of-sync co-change in the dotted lines surrounding commits 4 and 5. In commit 1, the author creates the project. A week later, the author adds the *app.feature* file in commit 2 as well as the corresponding source code files. Similarly, in commit 3, the author adds the *team_players.feature* file and the corresponding source code files (i.e., Add *Teams.java* and *Players.java*). The author

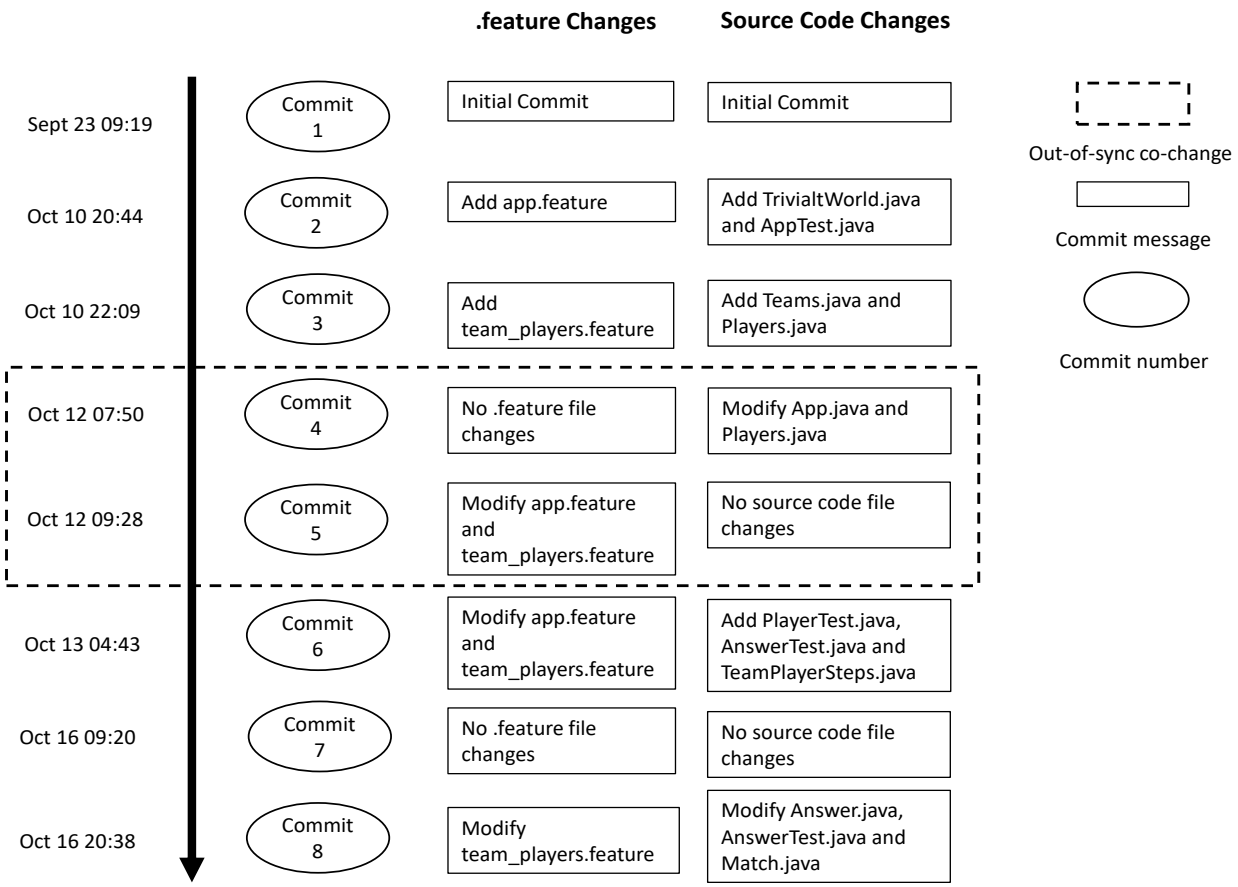

Fig. 2: Time line of an example BDD project

then follows the requirements described by *app.feature* and *team_players.feature* to add LOC in App.java and *Players.java* in commit 4. However, in commit 4, the author does not modify *.feature* files to describe the newly added source code functionalities. Roughly two hours later, in commit 5, the author adds LOC to *app.feature* and *team_players.feature* and writes the following commit message "Added missing steps". In each of these commits, we observe co-changing *.feature* files and source code files (i.e., *app.feature* co-changing with *App.java*, and team_players.feature co-changing with *Players.java*). However, the co-changing files are not modified simultaneously. We identify *.feature* file co-changes across different commits, i.e., source code files in commit 4 co-changing with *.feature* files in commit 5. We observe that, although *app.feature* and *AppTest.java* are co-changing files, the author does not include *app.feature* changes in commit 4, showing a temporary *out-of-sync* co-change. As a result of this out-of-sync co-change, the author needs to make another commit roughly two hours after commit 4. After step 6, we observe that *app.feature* is modified with *AppTest.java* in the same commit frequently, showing a continuing co-change link between the two files.

As the project scales up, it becomes harder to understand which *.feature* file corresponds with which source file or methods, and out-of-sync co-changes become more prevalent. In commit 5 of Figure 2, we observe that *team_players.feature* correspond to three different source files, and the *.feature* file is still modified 18 days after its first co-changing source file was added. As the project grows larger, the *.feature* file will need to be modified to describe functionalities of other source files, causing the *.feature* file maintenance time even longer. For a developer joining such a project, it is time-consuming and challenging to identify every co-change between *.feature* files and source code files. We explore approaches that can ease the traceability of co-changes between *.feature* files and source code files and reduce the out-of-sync co-changes.

III. Experiment Setup

In this section, we explain how we collect the data used for answering our research questions.

Figure 3 shows an overview of our entire experiment setup process. In **Step 1**, we use the GitHub search API to find project that mainly uses Java. We choose Java because the BDD framework *Cucumber* for Java is currently the most popular BDD tool [7]. Filtering Java projects is done by the search API of GitHub. Using the API, we can identify the major language of a project. GitHub uses the *Linguist* library to identify the major language in each project. Our search

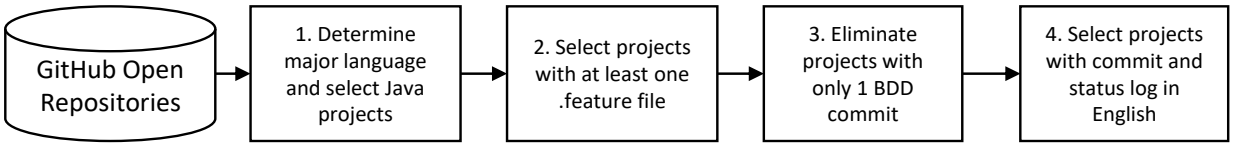

Fig. 3: An overview of the experiment design

retrieves 1,005,247 projects in total and we extract 59,933 Java projects amongst all the retrieved projects. In **Step 2**, we use the tree API in GitHub to find projects that include at least one *.feature* file, which we deem as a BDD project. We find that 927 out of 59,933 projects actually contain *.feature* files. In **Step 3**, we filter out the projects that contain only one commit manipulating a *.feature* file to avoid projects that do not actually use BDD. After this filtering, 890 projects survive. In **Step 4**, we eliminate BDD projects that do not have both *commit log* and *status* data in English. Commit logs provide information regarding every modification to the project (e.g., why certain files are modified). The status data stores the status of every commit, i.e., author, date and modified files. For our research, it is important for the information to be in English to understand the purpose of each commit and to make our research more reproducible. Also, we perform text processing with regards to English words in RQ1, so projects with information in a foreign language (e.g., Arabic) would hinder our analyses. In the end, 133 BDD projects remain. We highlight some key characteristics of our studied projects by selecting 50 of our 133 BDD projects with the highest total LOC. Table 1 shows the year created, total commits, total pull requests, total forks and total stars for each of our studied projects.

IV. Results

In this section, we provide the motivation, approach and results for each of our research questions.

(RQ1) Can we accurately identify co-changes between .feature files and source code files?

Motivation: The traceability between *.feature* files and source code becomes harder to maintain when a project scales up. Maintaining the traceability between *.feature* files and source code files is important to keep the *.feature* files up-to-date. When *.feature* files are up-to-date in a software project, the stakeholders can better discuss the functionalities of the project. Additionally, up-to-date *.feature* files ease the learning curve of new developers when they join the project. Therefore, in this RQ, we investigate whether we can accurately identify existing co-changes between *.feature* files and source code files in a BDD project.

Approach: To identify co-changes between *.feature* files and source files, we identify common keywords in both files. We use the Stanford CoreNLP toolkit to categorize all words in *.feature* files into word type brackets. Stanford CoreNLP is an extensible pipeline that provides core natural language analysis, and we use its parse function to analyze words syntactically. The parse function is based on a probabilistic parser [8]. We identify keywords as either nouns or verbs. Since we intend to link *.feature* files with Java files, words that are neither nouns nor verbs are eliminated and not used for comparison (i.e., adjectives, adverbs, Gherkhin language keywords) because words other than nouns and verbs are rarely used in Java source code [9].

To find keywords for source code, we use a Java parser to identify and eliminate Java language specific words (e.g., import, public, class). The remaining words are user specific and kept as keywords for comparison.

We then represent each file by a String consisting of the desired keywords, and use the cosine similarity algorithm to observe the similarities between the Strings. The cosine similarity converts two Strings into vectors, and calculates the $cosin(\theta)$ between them, where θ is the angle between the vectors. $cosin(\theta)$ is a number between 0 and 1, where 0 means the two vectors are perpendicular (completely different), and 1 means that the two vectors are parallel (identical). We take the upper 25% of all cosine similarities as our similarity threshold. This threshold is $cosin(\theta) = 0.95$, and determine that two files are co-changing only if their cosine similarity is above 0.95.

Using the Stanford CoreNLP toolkit, Java parser, and the cosine similarity algorithm, we observe that all *.feature* files within a commit are linked to at least one source code file in the same commit. There are also numerous cross commit links (i.e., a *.feature* file and the corresponding source code file in separate commits). When multiple commits are linked together by the identified links, we aggregate those commits together as *work items*. In some cases, commits within a work item could be weeks, or even months apart. However, when files are committed months apart they are likely modified for completely unrelated tasks. Therefore, we stipulate that one work week (or 2400 minutes) is a reasonable threshold to determine cross-commit co-changes between *.feature* files and source code files.

Findings: **Our analysis yields an accuracy of 79.8%.** After the identification of all possible *.feature* file and source code links, we manually examine the Git commit logs to verify the accuracy of our method. We use a confidence level of 95% and confidence interval of 5% on all of our identified co-changes to find a sample size for manual analysis. Our analysis obtains 60,203 links within the same commit and 1,815 cross commit links. We determine a sample size of 451. We analyze the commit logs and modified code for all 451 sampled links. Two of the three authors perform the manual

TABLE I: Characteristics of 50 studied BDD projects

Project (Username/Project-Name)	**Year Created**	**Total commits**	**Pull requests**	**Forks**	**Stars**
tyen/cs320tests	2009	207	0	0	9
caspian311/Scripturelookup	2009	109	0	0	2
epabst/expressive	2009	72	0	1	3
epabst/expressiveBDD	2009	46	0	2	2
mkristian/slf4r	2009	41	0	0	6
Vaysman/jvote	2009	29	0	0	0
Serabe/javascreepy	2009	22	0	0	1
rapidftr/RapidFTR-Android	2010	1214	31	83	38
davidbkemp/nate	2010	259	0	0	0
jtigger/kanban-simulator	2010	196	0	0	4
yujunliang/lambda	2010	83	0	0	8
jacek99/maven-python-mojos	2010	73	1	12	18
rapaul/cuke4ninja	2010	70	0	0	1
trevershick/jook	2010	69	2	2	1
sveinung/pritest-server	2010	57	1	0	6
openengsb-labs/labs-yaste	2010	32	0	1	3
runeflobakk/poker	2010	27	0	1	1
bugsnag/bugsnag-android	2011	645	29	154	872
AndreasWilhelm/neo4j-spatial	2011	360	0	0	2
resthub/springmvc-router	2011	214	29	59	170
rlogiacco/SmartUnit	2011	178	5	5	15
vitormcruz/payroll_cs	2011	173	0	0	2
mkristian/rails-resty-gwt	2011	164	0	2	12
akollegger/trivial-graph	2011	85	0	0	1
jfinkhaeuser/androdyne	2011	64	0	0	5
Drin/spam	2011	50	0	1	1
dokipen/embedly-java	2011	39	1	9	1
jescov/jescov	2011	39	0	4	10
jkransen/treemarks	2011	38	0	0	0
Jennifer-fu/practices	2011	36	0	0	1
lukasz-kaniowski/cucumber-selenium-rc	2011	33	0	1	2
ZsoltFabok/cucumber-jvm-post	2011	22	0	10	17
Chorus-bdd/Chorus	2012	1124	17	9	35
bartbaas/spatial	2012	565	0	0	2
daniel-andersen/Q-Cumberless-Testing	2012	348	0	0	4
leviwilson/oasis-android	2012	244	0	0	1
tlauchenauer/gaia-pdb	2012	237	0	0	1
bclozel/springmvc-router	2012	214	2	7	42
rabid-fish/JavaTechExamples	2012	178	0	0	1
iantmoore/old-substeps-webdriver	2012	163	0	1	1
ericlemerdy/one-kata-per-day	2012	118	0	3	7
mikael-wilhelm/LoadPlannerAmaz	2012	109	0	0	1
marky-mark/MTT	2012	77	0	0	1
leefaus/soa-petstore	2012	56	0	4	4
suggitpe/java-web	2012	40	0	1	0
japonophile/jescov	2012	39	0	0	1
sandromancuso/tww-java	2012	33	0	0	2
ilanpillemer/gherkin-eclipse-plugin	2012	26	0	2	18
talios/cucumber-testng-factory	2012	26	3	9	13
hayatoshimizuBSKYB/apollo	2012	23	3	1	5

analysis independently to verify the accuracy of our method. When merging the manual analyses performed by each author, we obtained a first consensus of 82% of 100 samples. For the remaining 18%, the authors discussed together in order to reach further consensus.

We determine that a *.feature* file co-change is present if the commit logs or modified code describe how *.feature* files and source code files are written together. Links including identical *.feature* file and source code file names also validate our analysis (i.e., our motivating example has *teamPlayer.feature* and *teamPlayer.java* within the same link). Of 451 cross commit work items across 133 BDD projects, we conclude that 360 work items are actually linked together, and the rest are unable to be fully identified. This yields an accuracy of 79.8%. The commit links that we are unable to fully identify as *.feature* and source code links contain some shared keywords between commits. After manually inspecting the commit logs and modified code, however, we conclude that these commits may address different functionalities within the project. The 79.8% accuracy is a lower bound for our approach because commit logs and modified code do not always explain the functionality of a commit accurately.

Our analysis obtains 60,203 links within the same commit and 1,815 cross commit links. After manual analysis, we observe that we can identify co-changes between *.feature* files and source code files with an accuracy of 79.8%.

(RQ2) Can we accurately predict when co-changes between .feature files and source code files are necessary?

Motivation: It would be beneficial for developers to determine the necessity of co-changes between *.feature* files and source code files before committing source code files. We aim to predict the modification of *.feature* files within the same commit. If we find that code characteristics can accurately predict BDD co-changes, we can warn developers that they should probably change or create a *.feature* file before committing the code.

Approach: Both language agnostic and Java specific characteristics are used as independent variables in predicting whether a new commit should include a *.feature* file co-change (i.e., a BDD co-change). We call these independent variables *predictors*. We approach the prediction of BDD co-changes as a binary classification problem. If a work item (as defined in RQ2) has both *.feature* files and .java files, our response class is *true*. Otherwise, our response class is *false*. We use three classification techniques: random forest, Naive Bayes, and logistic regression.

To ensure low correlation between the predictors, we perform a correlation test on our predictors. It is harder to assess the importance of a specific predictor if it is correlated with other predictors. Since all predictors are numeric, we use the Pearson correlation coefficient to evaluate the correlation between predictors. We use a *squared Pearson product moment correlation coefficient (SPPMCC)* threshold of $r^2 = 0.7$. SPPMCC is the squared Pearson correlation coefficient (SPCC) when applied to a sample. SPCC is defined as $\rho^2(a,b) = \frac{E^2(a,b)}{\sigma_a \sigma_b}$, where a and b are two zero-mean real-valued random variables, and $E(ab)$ is the cross-correlation between a and b, and σ_a and σ_b are the variances of the variables a and b, respectively. One of the most important properties of the SPCC is that $0 \leq \rho^2(a,b) \leq 1$. If $\rho^2(a,b) = 0$, then a and b are completely uncorrelated. The closer the value of $\rho^2(a,b)$ is to 1, the stronger the correlation between the two variables [10]. Based on prior research [5], we compute 24 predictors that can potentially predict BDD co-changes. Table 2 describes each predictor in detail and provides a rationale for the potential to predict BDD co-changes. Figure 4 shows the Pearson correlation coefficient for all predictors. The horizontal line shows the PPMCC threshold (0.7) from which the variables are highly correlated with each other. We can see that *files deleted* and *other LOC deleted* are highly correlated with each other. Interestingly, we also observe that *source files modified* and *test files modified* yield a SPPMCC of $r^2 = 0.75$ in BDD projects, showing a high degree of correlation between these two predictors. We also eliminate predictors that we intuitively know as directly related

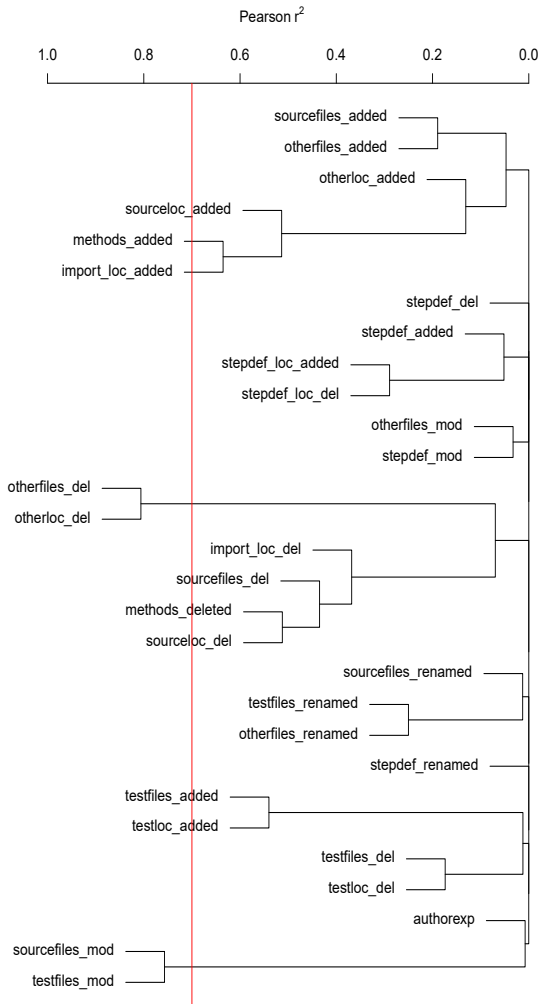

Fig. 4: Hierarchical clustering of predictors

TABLE II: Taxonomy of the studied co-change predictors

Attribute Name	**Type**	**Definition**	**Rationale**
Source file added	Numeric	Number of source files added in a commit	Changing a source file may add new functionalities, which requires new .feature file scenarios
Test file added	Numeric	Number of test files added in a commit	Developers often add test files and .feature files together to test new functionalities
Other file added	Numeric	Number of other files added in a commit	All other files changed could also add new functionalities, which requires new .feature file scenarios
Source file modified	Numeric	Number of source files modified in a commit	Same as source file added
Test file modified	Numeric	Number of test files modified in a commit	Same as test file added
Other file modified	Numeric	Number of other files modified in a commit	Same as other file added
Source file deleted	Numeric	Number of source files deleted in a commit	Deleted functionalities might require deletion of the scenarios in .feature files that describe the deleted functionalities
Test file deleted	Numeric	Number of test files deleted in a commit	Deleted tests might require deletion of the scenarios in .feature files corresponding with the deleted tests
Other file deleted	Numeric	Number of other files deleted in a commit	All other files deleted might cause deletion of scenarios, which requires .feature file changes
Source file renamed	Numeric	Number of source files renamed in a commit	Renaming source files might require renaming .feature files
Test file renamed	Numeric	Number of test files renamed in a commit	Renaming test files might require renaming .feature files
Other file renamed	Numeric	Number of other files renamed in a commit	Renaming all other files might require renaming .feature files
Source LOC added	Numeric	Source LOC added in a commit	A large LOC change may signify changes in functionalities, which requires changes in .feature file scenarios
Test LOC added	Numeric	Test LOC added in a commit	Same as source LOC added
Other LOC added	Numeric	Other LOC added in a commit	Same as source LOC added
Added dependencies	Numeric	LOC including “import” in a commit	New dependencies introduced could mean additions of new functionalities, which requires .feature file changes
Author experience	Numeric	Number of commits the author has written prior to current commit	High author experience could signify more comfortability with using .feature files, causing more .feature file changes
methods added	Numeric	Number of methods added in a commit	Methods may add or change functionalities, which requires new .feature file scenarios
methods deleted	Numeric	Number of methods deleted in a commit	Deleting methods might delete functionalities, which requires accompanying deletion of .feature scenarios

to *.feature* files, such as *step definition file changes*. Since *step definition* files are automatically generated based on their respective *.feature* files, the link between *step definition* files and *.feature* files are obvious. Hence, to assess the explanatory power of less obvious predictors, we remove the changes in *step definition* files from our predictors.

We eliminate *other LOC deleted*, *test files modified*, and all *step definition changes* for the classification process, and are left with 19 total predictors.

We use the random forest, Naive Bayes and logistic regression algorithms due to our dichotomous response class (Presence or absence of *.feature* file modification in a commit). We also choose these three techniques because they are widely used in previous research that used classification models in software engineering [5], [11], [12], [13].

To evaluate the performance of our predictions, we construct classifiers using a testing corpus and compare its deduction against the training corpus. We use tenfold cross-validation to obtain the testing corpus, which splits the data into ten equal parts, and takes one part at random as the testing corpus while the other nine as the training corpus. The process is repeated ten times, using a different part as the training corpus each time. We use the Area Under the Curve (AUC) metric to evaluate the performance of our three models. AUC is the area under the plot of true positive rate against false positive rate. AUC is a number between 0 and 1 [12]. A higher AUC value means that our predictors have high discriminatory power (i.e., it better distinguishes whether a *.feature* file change or creation is needed or not).

Findings: **Our random forest model obtains an AUC of 0.77. Our naive Bayes and Logistic Regression models obtain AUC values of 0.74 and 0.70, respectfully.** We observe that both language agnostic and language specific predictors can potentially predict *.feature* file co-changes. This result can be helpful for BDD developers. For example, code-change predictors can warn them of the necessity to modify or create *.feature* files before committing the changes, so they can keep the *.feature* file and source code file links up-to-date. Once our model predicts that a commit needs to change a *.feature* file, our NLP technique presented in RQ1 could be used to show suggestions for developers.

Our best performing classification technique (random forest model) obtains an AUC of 0.77, which can help BDD developers maintain traceability more efficiently.

(RQ3) What are the most significant code change characteristics for predicting co-changes between .feature files and source code files?

Motivation: In RQ1, we determine that we can detect BDD co-changes with relatively high accuracy. In RQ2, we observe that code-change characteristics, independent of each other, can predict those co-changes. To further study BDD traceability, we highlight the code change predictors with the highest importance for co-changes between *.feature* files and source code files. Knowing the most important predictors of co-changes between *.feature* files and source code files can further help developers to understand why *.feature* files should be modified before committing source code file.

Approach: To analyze the predictors with the highest explanatory power, we use our random forest model because it obtains the highest AUC value (AUC of 0.77). We compute the mean decrease accuracy with respect to the model's AUC of each predictor to evaluate their importance. This computation is performed as follows. We first obtain the AUC of our random forest classification algorithm without a particular predictor and observe the decrease in AUC due to the elimination of that predictor. A large AUC decrease, after eliminating a predictor, indicates a higher predictive power associated with that predictor. We perform the same steps for all predictors to assess their predictive power in terms of AUC.

Findings: **Test files added, other files modified, test files re-named, and source LOC deleted are the most important predictors for co-changes between *.feature* files and source code files.** We observe from Figure 5 that eliminating *test files added*, *other files modified*, *test files renamed*, and *source LOC deleted* yield AUC decreases of 0.17, 0.11, 0.09 and 0.08, respectively. We can use these findings to recommend changing a *.feature* file when adding or renaming a test file, modifying non-Java files, and deleting LOC from Java source files. By considering *.feature* file changes before committing source file changes, we can potentially help developers save *.feature* file maintenance time. Furthermore, a new developer can identify *.feature* file co-change links by observing the presence of these predictors in previous commits.

Our results suggest that developers should consider modifying or adding *.feature* files in cases of *test files added*, *other files modified*, *test files re-named* and *source LOC deleted*.

V. Threats to Validity

In this section, we discuss threats to the validity of our case studies.

A. *Construct validity.*

Construct threats to validity are errors caused by the methodology used in collecting data. For RQ2, we make an assumption that one work week is the maximum time for co-changes between *.feature* files and source code files across different commits to occur. Changing the time-window might have produced more identified co-changes, but a longer time window would mean a higher risk of linking unrelated changes. Moreover, it is not unlikely that commits performed within a work week regarding the same functionalities are still related. Therefore, one work week seemed a safe assumption. We use Git logs for the manual analysis performed in RQ2 (i.e., to find the accuracy of our identifying co-changes method), which may contain bias. We combat this bias by analyzing LOC changes when a Git log does not describe a commit adequately.

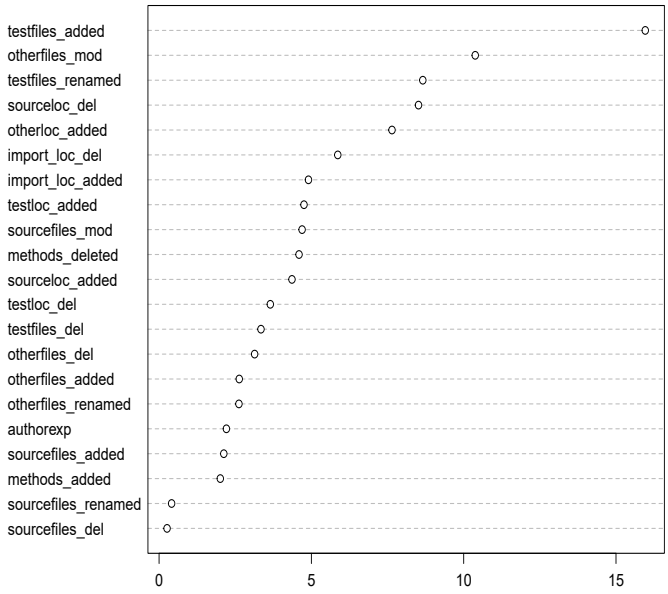

Fig. 5: Explanatory power of co-change predictors by mean decrease accuracy (% decrease)

B. *Internal validity.*

Internal threats are concerned with the relationships between our dependent variable (*.feature* file co-changes) and independent variables (code characteristics). We select predictors that cover a wide range of characteristics that would likely cause *.feature* file changes. However, our selection is not exhaustive and some predictors that we have overlooked may have improved the performance of classifiers. For example, the amount of project stakeholders involved in a functionality or the amount of concurrent tasks the developers were performing during a commit (i.e., causing the developers to have less time to implement *.feature* files) may have impacted the results of our classifiers. However, GitHub does not record commit information in such detail.

C. *External validity.*

External threats are concerned with the extent to which we can generalize our results. We only use 133 BDD projects in our analysis. We reached 133 as the number of projects to analyze to avoid noise in the dataset by including *toy* projects or inactive projects. further studies should investigate more projects to avoid any sample bias. preferably we would use large industry projects.

VI. RELATED WORK

In this section, we discuss the related work with respect to prior work focusing on BDD, testing strategies, co-changes, traceability and requirements engineering.

A. *BDD*

Carvalho *et al*. [14], [15] found that BDD is a specification technique that automatically certifies that all functional requirements are treated properly by source code. Solis *et al*. [1] concluded that there are main characteristics of BDD, such as ubiquitous language, and the iterative decomposition process, in which developers repeatedly test a set of features satisfying customer priorities. The other characteristics of BDD include plain text description, automated acceptance testing, and readable behavior oriented specification code. Soeken *et al*. [16] proposed an assisted flow for BDD using natural language processing. Our work differs from the existing BDD work as it focuses on the co-changing *.feature* files and source code files.

B. *Testing Strategies*

Basili *et al*. [17] compared three different types of testing strategies: code reading, functional testing, and structural testing. Bhat *et al*. [3] evaluated the efficiency of Test Driven Development (TDD), and observed that a significant increase (greater than two times) in quality of code occurs due to the usage of TDD. Differently from prior work, our paper explores the traceability problem in BDD practices.

C. *Co-changes*

With regards to co-changes, Zaidman *et al*. [4] used file change history view, in which production and test files added, changed and deleted are recorded and examined. This work also examines LOC growth view to study two large repositories. Beyer and Hassan visualized software history by displaying sequences of cluster layouts based upon co-change graphs [18]. To identify co-changing lines, Zimmermann *et al*. [19] built an annotation graph based upon the identification of lines across several versions of a file. Kagdi *et al.* [20] applied sequential pattern mining to file commits in software repositories to discover traceability links between software artifacts. Mcintosh *et al.* [5] mined two large scale software repositories, and used code change characteristic to explain build system and source code co-changes. With regards to co-change based recommendations systems, Robillard *et al*. [21] described recommendation systems in assessing and improving evolving software. Robillard *et al*. found that quantity, heterogeneity, context-sensitivity, dynamicity, and partial generation made it difficult to analyze and assess software engineering data. D'Ambros *et al*. [22] attempted to link co-changing software artifacts to software defects, and found that there was a correlation between change coupling and defects. Unlike previous work on co-changes, our study identifies and predicts co-changes to increase the efficiency of BDD developers regarding keeping the *.feature* files up-to-date.

D. *Traceability*

Sundaram *et al*. [23] presented the importance of generating traceability links and discussed two different ways to enhance traceability in software engineering: vocabulary base and secondary measures. Sundaram *et al*. found that high and low level vocabulary artifacts did not perform as well as only using low level artifacts. Gotel and Finkelstein [24] investigated the challenges of the requirements traceability problem by empirically studying code by over 100 practitioners. Gotel and

Finkelstein recommended the increase of awareness, recording and organizing of information to improve traceability. Our work focuses on the traceability problem for BDD *.feature* files by leveraging information found in GitHub repositories.

E. *Requirements Engineering*

Cheng *et al*. [25] found that requirements engineering is difficult because requirements analysts start with ill-defined specifications and cannot predict how the system's environment behaves. Cheng *et al*. recommended that researcher should work with practitioners, requirements engineering researchers should work with other software engineering researchers, and industrial organizations should provide industrial-strength project data to researchers [25]. Paetsch *et al*. [26] presented the characteristics of requirements engineering and agile development. Paetsch *et al*. recommended interviews and Joint Application Development (JAD) sessions with customers to incorporate requirements engineering with agile development. To improve links between source code and requirements, Rahimi *et al*. [27] presented TLE (Trace Link Evolver) to automate the trace links as changes are introduced to source code. Rahimi *et al.* applied their method across 27 releases of the Cassandra Database System. They found a recall of 0.945 and a precision of 0.919 [27]. Our work focused on *.feature* files as requirement files in BDD. We analyzed GitHub commits containing both *.feature* files and source code files. Our work used NLP techniques to identify trace links between *.feature* files and source code in BDD projects, and used classification techniques to predict those trace links.

VII. Conclusions

BDD is a relatively new testing strategy that uses English-like syntax in describing code functionalities. BDD makes it easier for all stakeholders involved in a project to understand the functionalities of a software project. With the use of *.feature* files in describing the functionalities of a software, the co-evolution of *.feature* files and source code files must be kept up-to-date. In our work, we detect the co-changes between *.feature* files and source code files, and find characteristics that can accurately predict the co-changes in order to improve the traceability between *.feature* files and source code files.

Our approach can link *.feature* files with source code files with an accuracy of 79% and predict co-changes between *.feature* files and source code files with an AUC of 0.77. *Test files added*, *other files modified*, *test files renamed*, and *source LOC deleted* are the best predictors for co-changes between *.feature* files and source code files. Our results demonstrate that co-changes between *.feature* files and source code files can be detected, and that source code change characteristics can predict those co-changes. Our findings can help developers to keep software documentation (i.e., *.feature* files) up-to-date and help projects to adopt the BDD practices in developing software more efficiently. To further assist BDD developers, we plan to explore BDD co-changes with regards to more languages and analyze more projects using BDD.

References

[1] C. Solis and X. Wang, "A study of the characteristics of behaviour driven development," in *Software Engineering and Advanced Applications (SEAA), 2011 37th EUROMICRO Conference on*. IEEE, 2011, pp. 383–387.

[2] M. Diepenbeck, M. Soeken, D. Große, and R. Drechsler, "Behavior driven development for circuit design and verification," in *High Level Design Validation and Test Workshop (HLDVT), 2012 IEEE International*. IEEE, 2012, pp. 9–16.

[3] T. Bhat and N. Nagappan, "Evaluating the efficacy of test-driven development: industrial case studies," in *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*. ACM, 2006, pp. 356–363.

[4] A. Zaidman, B. Van Rompaey, S. Demeyer, and A. Van Deursen, "Mining software repositories to study co-evolution of production & test code," in *Software Testing, Verification, and Validation, 2008 1st International Conference on*. IEEE, 2008, pp. 220–229.

[5] S. Mcintosh, B. Adams, M. Nagappan, and A. E. Hassan, "Mining co-change information to understand when build changes are necessary," in *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*. IEEE, 2014, pp. 241–250.

[6] Akollegger, "akollegger/trivial-graph," Oct 2011. [Online]. Available: https://github.com/akollegger/trivial-graph

[7] "Bdd framework list," https://codoid.com/bdd-framework-list/, accessed: 2018-10-26.

[8] C. Manning, M. Surdeanu, J. Bauer, J. Finkel, S. Bethard, and D. McClosky, "The stanford corenlp natural language processing toolkit," in *Proceedings of 52nd annual meeting of the association for computational linguistics: system demonstrations*, 2014, pp. 55–60.

[9] M. Pawlan, "Essentials of the java programming language." [Online]. Available: https://www.oracle.com/technetwork/java/index-138747.html

[10] J. Benesty, J. Chen, Y. Huang, and I. Cohen, "Pearson correlation coefficient," in *Noise reduction in speech processing*. Springer, 2009, pp. 1–4.

[11] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[12] I. Rish *et al.*, "An empirical study of the naive bayes classifier," in *IJCAI 2001 workshop on empirical methods in artificial intelligence*, vol. 3, no. 22. IBM New York, 2001, pp. 41–46.

[13] J. Friedman, T. Hastie, R. Tibshirani *et al.*, "Additive logistic regression: a statistical view of boosting (with discussion and a rejoinder by the authors)," *The annals of statistics*, vol. 28, no. 2, pp. 337–407, 2000.

[14] R. A. de Carvalho, R. S. Manhães *et al.*, "Filling the gap between business process modeling and behavior driven development," *arXiv preprint arXiv:1005.4975*, 2010.

[15] R. A. de Carvalho, R. S. Manhaes *et al.*, "Mapping business process modeling constructs to behavior driven development ubiquitous language," *arXiv preprint arXiv:1006.4892*, 2010.

[16] M. Soeken, R. Wille, and R. Drechsler, "Assisted behavior driven development using natural language processing," in *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*. Springer, 2012, pp. 269–287.

[17] V. R. Basili and R. W. Selby, "Comparing the effectiveness of software testing strategies," *IEEE transactions on software engineering*, no. 12, pp. 1278–1296, 1987.

[18] D. Beyer and A. E. Hassan, "Animated visualization of software history using evolution storyboards," in *Reverse Engineering, 2006. WCRE'06. 13th Working Conference on*. IEEE, 2006, pp. 199–210.

[19] T. Zimmermann, S. Kim, A. Zeller, and E. J. Whitehead Jr, "Mining version archives for co-changed lines," in *MSR*, vol. 6. Citeseer, 2006, pp. 72–75.

[20] H. Kagdi, J. I. Maletic, and B. Sharif, "Mining software repositories for traceability links," in *Program Comprehension, 2007. ICPC'07. 15th IEEE International Conference on*. IEEE, 2007, pp. 145–154.

[21] M. Robillard, R. Walker, and T. Zimmermann, "Recommendation systems for software engineering," *IEEE software*, vol. 27, no. 4, pp. 80–86, 2010.

[22] M. D'Ambros, M. Lanza, and R. Robbes, "On the relationship between change coupling and software defects," in *2009 16th Working Conference on Reverse Engineering*. IEEE, 2009, pp. 135–144.

[23] S. K. Sundaram, J. H. Hayes, A. Dekhtyar, and E. A. Holbrook, "Assessing traceability of software engineering artifacts," *Requirements engineering*, vol. 15, no. 3, pp. 313–335, 2010.

[24] O. C. Gotel and C. Finkelstein, "An analysis of the requirements traceability problem," in *Requirements Engineering, 1994., Proceedings of the First International Conference on*. IEEE, 1994, pp. 94–101.

[25] B. H. Cheng and J. M. Atlee, "Research directions in requirements engineering," in *2007 Future of Software Engineering*. IEEE Computer Society, 2007, pp. 285–303.

[26] F. Paetsch, A. Eberlein, and F. Maurer, "Requirements engineering and agile software development," in *WET ICE 2003. Proceedings. Twelfth IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises, 2003.* IEEE, 2003, pp. 308–313.

[27] M. Rahimi, W. Goss, and J. Cleland-Huang, "Evolving requirements-to-code trace links across versions of a software system," in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2016, pp. 99–109.