The Relation between Bug Fix Change Patterns and Change Impact Analysis

Ekincan Ufuktepe
University of Missouri - Columbia
Columbia, MO, USA
euh46@missouri.edu

Tugkan Tuglular
Izmir Institute of Technology
Izmir, Turkey
tugkantuglular@iyte.edu.tr

Kannappan Palaniappan
University of Missouri - Columbia
Columbia, MO, USA
pal@missouri.edu

***Abstract*—Change impact analysis analyzes the changes that are made in the software and finds the ripple effects, in other words, finds the affected software components. In this study, we analyze the bug fix change patterns to have a better understanding of what types of changes are common in fixing bugs. To achieve this, we implemented a tool that compares two versions of codes and detects the changes that are made. Then, we investigated how these changes are related to change impact analysis. In our case study, we used 13 of the projects and 621 bugs from Defects4J to identify the common change types in bug fixed. Then, to find the change types related to cause an impact in the software, we performed an impact analysis on a subset of projects and bugs of Defects4J. The results have shown that, on average, 90% of the bug fix change types are adding a new method declaration and changing the method body. Then, we investigated if these changes cause an impact or a ripple effect in the software by performing a Markov chain-based change impact analysis. The results show that the bug fix changes had only impact rates within a range of 0.4%–5%. Furthermore, we performed a statistical correlation analysis to find if any of the bug fixes have a significant correlation on the impact of change. The results have shown that there is a negative correlation between caused impact with the change types adding new method declaration and changing method body. On the other hand, we found that there is a positive correlation between caused impact and changing the field type.**

***Index Terms*—change impact analysis, bug fix, change detection**

I. Introduction

Change is a continual and integral part of the process of the software evolution process. A source code change can be performed for enhancing software or fixing a bug. However, source code changes can also introduce bugs into the system. These bugs can be originated due to the ripple effects caused by small changes. In other words, the bugs introduced by a change can be related due to a dependency within the source code. When a bug (issue) is reported in a repository, it is not easy to localize when or how the bug is introduced through which commit. However, bug fixes are relatively easier to localize, if the version control system is used effectively. For instance, Just et. al [1], prepared a collection of real bugs for researchers, which contains the commit hash values when the bug report was entered and the commit hash value when the bug was fixed.

The type of changes made in the software has an important role in the likelihood to introduce an error [2]. For instance, changes that are made in the software that are mostly related to dependency-based changes are more likely to cause a ripple effect. These types of changes could be changes that are made in superclasses, methods that rely on call relationships, deletion/addition of classes, and methods. These types of changes were classified as changes to cause a ripple effect in software [3]. Therefore, knowing the types of changes in the software can have a critical part in detecting changes that might cause an impact on other software components. Furthermore, popular software version control systems like Github are able to provide the changes that are made in the software, however, they do not provide any information on what type of changes are made in the software, which could reduce the time for code reviews.

In addition, the ideal way of using version control systems is to push small commits, rather than pushing large commits. Since large commits contain too much information of changes, this could be a very long task for the code reviewer to analyze the changes that are made, and find possible impacts or ripple effects that might cause in the software. Sometimes, commits contain only code relocations. An example (https://github.com/apache/commons-csv/commit/c203896177b295c2f5319e8c34b9d8bb9f58564e) we found is that some commits in repositories show code changes, however, when code is carefully reviewed there are actually no changes made in the software. In the given example, only some of the methods in the class are just repositioned (method moved on top of the class or moved to the bottom of the class). Even though there is actually no code change in the repository, when the code is being reviewed on Github it could easily mislead the code reviewer to make wrong conclusions such as *(*methods have been added and removed).

Nevertheless, detecting change patterns plays an important role in many other fields of software engineering. For instance, change pattern information has been used in training neural networks [4], [5] for automatic bug fixing, for understanding bug fixes and for automatic program repair [6], detecting readability improvements [7], and in fault localization [8], [9].

In this study, we first share our novel approach and architecture on detecting change types of a given commit that are related to cause an impact on the software. Then, we detect change types for the commits with bug fixes from Defects4J

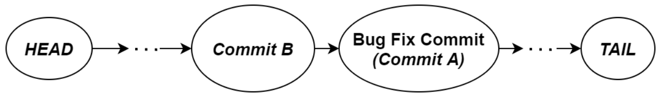


Fig. 1: Experimental design for RQ2. *Commit A* is the bug fix commit, where *Commit B* is the parent commit of *Commit A*.

and investigate the relation between change types that are likely to cause an impact on other software components. Therefore, we develop the following research questions and answer them in the discussion section.

RQ1: What are the most common bug fix change patterns? To answer this question we analyze all the bugs from Defects4J on 13 projects and overall 621 bugs. We identify the change types that are made in the bug fix commits and find the commonly performed change action in these bug fix commits.

RQ2: Is there a relationship between the bug fix change patterns and the impact caused by change? To investigate if there is a relation between bug fix changes and impact causing changes, we perform a change impact analysis using a Markov chain and program slicing-based approach. We simply perform a change impact analysis between two commits, which are the bug fix commit, and the parent commits of a bug fix that is demonstrated in Fig. 1. Once we obtained the change impact analysis results, we divide the impacted methods by the total methods (impact rate). Then we analyze the overall impact for each project from our case study and we perform a statistical analysis to find a correlation between the caused impact and the change type.

This paper makes the following main contributions:

(i) Method: We present an empirical study that analyzes the relation between the bug fix change patterns and changes that causes an impact on other software components. We try to provide a better understanding of bug fix preference, due to its low impact. To evaluate the impact of the bug fixes, we use a Markov chain and program slicing based impact analysis tool called CODE CHANGE SNIFFER [10][1].

Furthermore, we provide a novel approach to detecting change types between two commits related to cause an impact on software components, while other studies [11]–[13] focus on statement-level (fine-grained) changes. Our novel approach is based on the ANTLR parser, which generates a parse tree and compares two trees for detecting the changes. Using parse trees can be more effective than abstract syntax trees, due to its capability of containing keywords, which makes it easy to detect changes like type changes, modifier changes, etc.

(ii) Tool: We developed a tool called CHANGE INSPECTOR JAVA (CIJ)[2] and made it publicly available for detecting code changes. The tool uses PyDriller [14] for mining Github repositories, and CIJ detects the changes that are made implements the end-to-end pipeline for predicting future code changes with the Markov chain.

[1]https://github.com/ekincanufuktepe/code-change-sniffer
[2]https://github.com/ekincanufuktepe/change-instepector-java

(iii) Dataset: We provide our detected change type dataset to other researchers and practitioners to provide a better reproducibility of our study. Our dataset contains:

- The change types detected by CIJ for the corresponding bug fix commits from Defects4J.
- The probabilities of methods of being impacted by the bug fixes.

The manuscript is organized as follows. Section II provides the background on change impact analysis. In Section III, the change detection architecture is presented with the change types that are used. Section IV explains the case study of our and evaluation. Section V outlines threats to validity in our study and answers the research questions introduced in Section I. In Section VI, related work on change type prediction is presented. Section VII concludes the paper.

II. BACKGROUND

In this section, we provide the essential background to provide a better understanding of our study. In the following section, we provide a running example of the change impact analysis technique and tool called CODE CHANGE SNIFFER [10], which we used in our study.

A. Change Impact Analysis

In the 1980s, studies [15], [16] mentioned the difficulties that software evolution has brought into software maintenance. One of these difficulties is the ripple effects that are caused by changes in the source code. On the other hand, evolution in software development had been considered to be inevitable, and should be accepted that change is an intrinsic part of software development lifecycle [15]. Nowadays, it is still true however changes are more rapid due to advancements in technology and user expectations. Lehman and Belady [17] supported this fact in their five laws on software evolution, where they stated the first law as “change is continual”. In such a rapidly evolving software environment, developers need mechanisms and tools to keep up the pace with better resource utilization.

Previous studies have mentioned [18]–[20] that software maintenance consumes the majority of resources in many software organizations. Nevertheless, a rapidly evolving software, due to the changes that are made in the software, is more likely to introduce faults and errors [21]. An example of fast-evolving software was introduced by Kumar [22], which stated that Google was committing 20 code changes per minute, and approximately 50% of their code was changing monthly. In a rapidly evolving software development environment, predicting future code changes related to the current changes could reduce the effort spent on software maintenance. For instance, predicting code changes can reduce the time on finding the code sections that need to be changed, or highlight codes that require a change to fix the possible errors that are introduced by modifications.

B. Running Example of Code Change Sniffer

CODE CHANGE SNIFFER is a change impact analysis tool that uses the Markov chain to calculate the probabilities of impacted methods in the software. The probabilistic information is computed by analyzing the software with static analysis. The *diff* information between two commits (or versions) is used as an initial vector, while the transition matrix is filled with probabilistic information acquired from forward slicing. In the following paragraphs of this section, we provide a small running example of how CODE CHANGE SNIFFER works.

The probabilistic information obtained from forward slicing is encoded into the Markov chain's edges along with the change information based on the type of the model, namely, call graph (CG) and effect graph (EG). However, it is important to mention that we used call graphs in this study to analyze the impacts. On the other hand, the initial vector is encoded with change information, which applies to both models. Starting with encoding the edges, we construct a transition matrix, which is similar to an adjacency matrix.

Another property of the Markov chain is that summation of the outgoing edge probabilities of a node should be equal to 1. Therefore, the probability summation of each row in the transition matrix should be equal to 1. However, a row summation could be less than or greater than 1 depending on the probabilities obtained from forward slicing. For instance, on the left side of Fig. 2, let us assume that we encode the Markov chain model with probabilistic information with forward slicing and change information. We can see that some of the nodes' summation of outgoing edges are less than 1 or greater than 1. To satisfy the properties of the Markov chain, we weight each node's outgoing edges, by dividing the summation of outgoing edges to each outgoing edge of that node. On the right-hand side of Fig. 2, we obtain the updated Markov chain after weighting the edges. Furthermore, assume that the filled methods (m_0, m_1, m_2, m_3, m_4) are the changed methods.

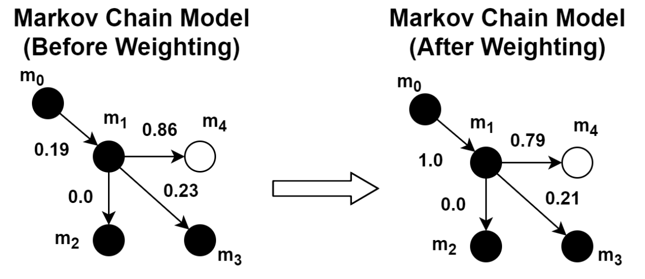

Markov Chain Model (Before Weighting)
m0
m1
m4
m2
m3
0.19
0.86
0.0
0.23
Markov Chain Model (After Weighting)
m0
m1
m4
m2
m3
1.0
0.79
0.0
0.21

Fig. 2: Markov chain model construction with weighted edges

After the weighting process is completed, we construct the transition matrix of the Markov chain model below. According to the graph model in Fig. 2, there is no outgoing edge from methods m_2, m_3, and m_4. Therefore, in the transition matrix, we would expect to have the entire row filled with 0s. However, we have a single 1 that are placed to itself such as $m_2 \rightarrow m_2$, $m_3 \rightarrow m_3$, $m_4 \rightarrow m_4$. According to the Markov chain's properties, the summation of the columns for each row should be equal to 1. Thereby, for a row where the sum of column values is equal to 0, we set the $m_i \rightarrow m_i$ edge probability to 1. If the method m_i is not changed, setting the probability will not affect the overall impact calculation, since that it will be multiplied with 0.

$$T = \begin{array}{c} \\ m_0 \\ m_1 \\ m_2 \\ m_3 \\ m_4 \end{array} \begin{array}{c} \begin{array}{ccccc} m_0 & m_1 & m_2 & m_3 & m_4 \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.21 & 0.79 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \end{array}$$

To calculate the impact vector, in other words, the vector that contains the probabilities of predicted methods that will change, an initial vector should be multiplied with the transition matrix. We encode the initial vector with change information we have collected from diff calculations. The change information represents the likelihood of a method that could affect itself by the changes that are made for the current method. Therefore, as the amount of change increases the probability of being affected by changes will be higher. In Fig. 2, let's assume the filled nodes (methods) m_0, m_1, m_2 and m_3 in the Markov chain are the changed methods with given probabilities; $m_0 = 0.5$, $m_1 = 0.71$, $m_2 = 0.78$ and $m_3 = 0.33$. The four given change probabilities are encoded into the initial vector below. Previously, to satisfy the properties of the Markov chain in the transition matrix, we weight the edges of each node's outgoing edges. Similarly, we also need to weight the initial vector values as well. According to the Markov chain's properties, the summation of the probabilities in the initial vector should be equal to 1, where the sum of the probabilities in our initial vector is greater than 1.

$$I = [\ 0.5\ 0.71\ 0.78\ 0.33\ 0\ ]$$

We weight the initial vector by dividing each value in the vector by the summation of the probabilities in the vector. Thereby, we have updated our initial vector I to I_w, which is given below.

$$I_w = [\ 0.216\ 0.306\ 0.336\ 0.142\ 0\ ]$$

Finally, we obtain the final forms of our initial vector and transition matrix, and by using the final forms of the initial vector and transition matrix, we calculate the impact vector in Equation 1, which is predicted to be changed methods. Since our initial vector and transition matrix is weighted, we expect to calculate the impact vector, where the summation of its probabilities is equal to 1.

$$I_w\ x\ T = [\ 0.156\ 0.181\ 0.336\ 0.181\ 0.146\ ] \quad (1)$$

Based on the Markov chain model in Fig. 2 and calculation in Equation 1, the probabilities of the methods being affected by the changes are calculated as $m_0 = 0.156$, $m_1 = 0.181$,

$m_2 = 0.336$, $m_3 = 0.181$, and $m_4 = 0.146$. With respect to the probabilities, m_2 is the method that has the highest likelihood of being affected by the changes.

III. Change Detection Architecture

In this section we introduce the details of our change detection tool and architecture Change Inspector Java given in Fig. 3.

A. Change Types

In the context of change impact analysis, the changes in the source for Java programming language can occur in three categories: class type changes, method type changes, and field type changes. These types of changes were introduced by Ren et al. [2], and Sun et al. [3], in this study we adopted these change types for our automatic change type detection tool with slight modifications.

While we classify the change types on source codes, we focused on the type of changes that occur at a programming language level rather than the developer behavior level changes. For instance, a developer may just change the name of the class, without modifying the class body. On the developer side, this change will be interpreted as a *Changing the class name* type of change. However, in programming language level change this will be interpreted as *Delete Class and Add new Class* because changing the name of the class affects the signature of the class. This similarly applies to methods as well, when a method name is changed, the change reflects as a *Method Deleted, and new Method Added.*

The change types for classes, methods, and fields are given respectively in Tables I, II, III.

TABLE I: Types of Class changes

Type	Description
AC	Add a new class (new class declaration)
DC	Delete a class with all its members
IAC	Increase "***accessibility***" of the class ("***private***" modifier changed to "***public***")
DAC	Decrease "***accessibility***" of the class ("***public***" modifier changed to "***private***")
AFC	Add a "***final***" modifier to the class
DFC	Delete the "***final***" modifier from the class
ASC	Add a "***static***" modifier to the class
DSC	Delete the "***static***" modifier from the class
AAbC	Add a "***abstract***" modifier to the class
DAbC	Delete the "***abstract***" modifier from the class
APC	Add parent class
DPC	Delete parent class

B. Automatic Change Type Detection Architecture

In this section, we present our automatic change type detection architecture. Our change type detection process follows the order of preprocessing, parse tree generation, extracting key change features, and detecting change type, which is also provided in Fig. 3.

TABLE II: Types of Method changes

Type	Description
AM	Add a new method (new method declaration)
DM	Delete a method
CM	Change method body
IAM	Increase "***accessibility***" of the method ("***private***" modifier changed to "***public***")
DAM	Decrease "***accessibility***" of the method ("***public***" modifier changed to "***private***")
AFM	Add a "***final***" modifier to the method
DFM	Delete the "***final***" modifier from the method
ASM	Add a "***static***" modifier to the method
DSM	Delete the "***static***" modifier from the method
AAbM	Add a "***abstract***" modifier to the class
DAbM	Delete the "***abstract***" modifier from the method
CRM	Change return type of the method
CNPM	Change name of parameters of the method
CPM	Change parameters of the method except for the change of the names of the parameters

TABLE III: Types of Field changes

Type	Description
AF	Add a new field (new field declaration)
DF	Delete a field
IAF	Increase "***accessibility***" of the field ("***private***" modifier changed to "***public***")
DAF	Decrease "***accessibility***" of the field ("***public***" modifier changed to "***private***")
AFF	Add a "***final***" modifier to the field
DFF	Delete the "***final***" modifier from the field
ASF	Add a "***static***" modifier to the field
DSF	Delete the "***static***" modifier from the field
CTF	Change type of field

1) Preprocessing: The pre-processing phase is where we first retrieve the changed source codes. Retrieving the changed source codes is achieved by PyDriller [14]. We simply provide the bug fix commit hash to PyDriller and extract the changes with *diff*. Using *diff* also provides which source files are changed and allows us to download the source files as well. The pre-processing phase allows us to eliminate any redundant process of parse tree generation or computation, and only focus on the changed source codes.

2) Parse Tree Generation: For parse tree generation we used ANTLRv4[3] [23], and we have used the grammar designed for Java 1.8. In a previous work [13], abstract syntax trees (AST) were used for change type detection, due to their compactness and easiness of process. We acknowledge and justify that parse trees are complex trees compared to AST, however, they contain details that an AST does not contain. For instance, AST is also known as a logical description of parse trees. Therefore, it does not contain any syntactical constructs, such as braces, parenthesis, white spaces, and keywords. However, based on the change types we have defined and used in the context of change impact analysis, we need changes that are made on the keyword information, such as modifiers, data types, etc.

3) Extracting key change features: Once we have obtained the parse tree, we extract the information in three categories:

[3]https://github.com/antlr/grammars-v4

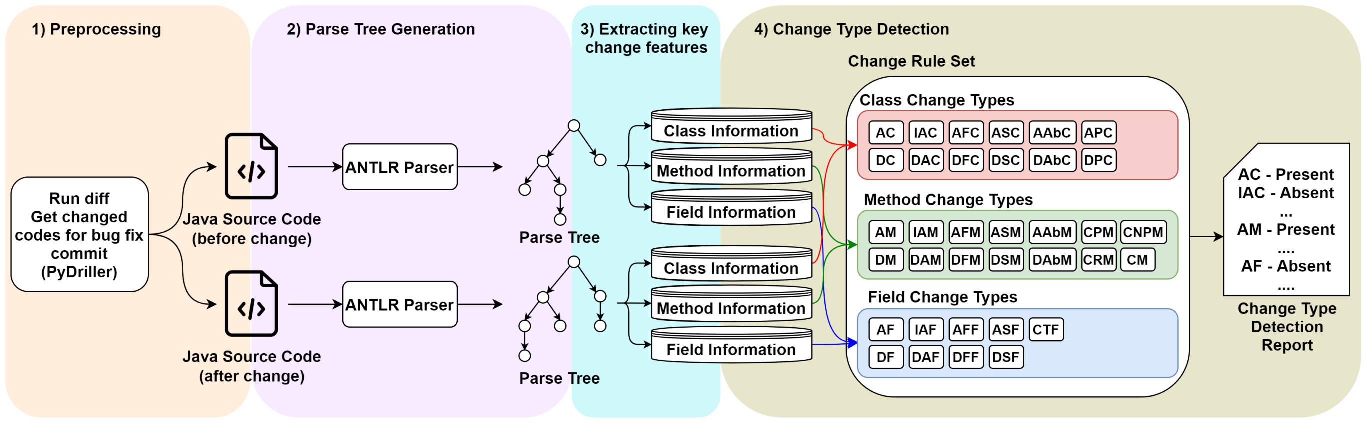


Fig. 3: Change Inspector Java Architecture, with multi-threaded change type detection. The Change detection starts with preprocessing by only extracting changed files, then follows a parse tree generation for each changed file. From parse trees, key change features are extracted to be used in change type detection. For each change type and its detection, a thread is generated and reported.

class, method, and field. For each category we have separate abstract data types defined, and since that we are interested in particular data in each category we have different information extracted. For a class we extract: Class name, parent classes, and modifiers. For a field, we extract field names, modifiers, and types. Finally, for a method, we extract method name, modifiers, return type, parameter names, parameter modifiers, parameter types, and method body.

4) Change Type Detection: The change type detection is performed based on comparing two abstract data types. Each change rule is defined as a sub-class of *ChangeRule*, where every change rule overrides two methods *getCategory* and *isChangeCategory*. This design allows developers to easily define new change rules. During the change type detection phase, if the related change is found, the *getCategory* method returns the change type, which is triggered and determined by the method *isChangeCategory*. Each change type has its own unique implementation and definition of change rule. This type of design allows us to implement our architecture in multi-threaded to achieve better performance.

Each change rule class receives two parse trees as inputs, one derived from bug fix changes, and the other is derived from the parent commit of the bug fix changes. For each change type, the parse trees are parsed separately and only target the interested key features. For instance, as demonstrated in Fig. 4, to check if a method's accessibility is increased (IAM), we first search for the method. We search for the method based on its signature which are: method name, return type, and parameter types (order sensitive). Thereby, we are not interested in the method body or the name of the parameters, thus this information is not extracted from the parse tree. Once the method is found, we check if the method had a *private* access modifier before the bug fix, and if the modifier has been changed to a *public* access modifier in the bug fix, then an *IAM* change is detected.

IV. Case Study

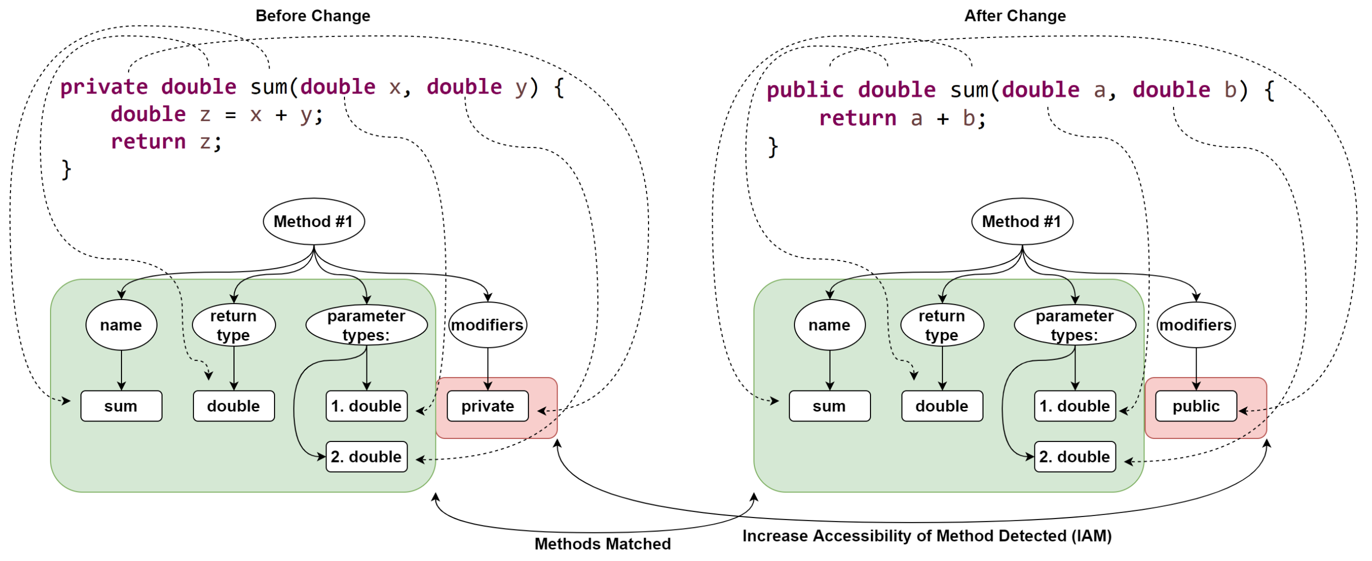
Our case study is composed of two phases. The first phase is identifying the common change types among 13 Java projects and 621 bug fixes from Defects4J. The second phase is performing a change impact analysis on a subset of projects and bug fixed from Defects4J. Some of the commits had missing source files, which prevented us to compile and perform change impact analysis. These projects and commits were discarded from our change impact analysis. Therefore, we performed change impact analysis only on 8 Java projects and 232 bug fixes from Defects4J, which is a subset of our phase one analysis.

TABLE IV: Selected project and bug fix information

Project	Number of bugs fixed	
	Change Type Analysis	Change Impact Analysis
closure-compiler	174	-
commons-cli	39	31
commons-codec	18	18
commons-collections	4	-
commons-compress	47	8
commons-csv	16	16
commons-jxpath	22	-
gson	18	18
jackson-core	26	23
jackson-databind	108	-
joda-time	26	25
jsoup	93	93
mockito	30	-

A. Change Type Analysis Results

In our case study of the change type analysis, we used 13 Java projects and 621 bug fixes from Defects4J [1]. The selected projects and bugs are given in Table IV. The bugs are collected from the *active bugs*, which contain the two commit hash information. The first commit hash value corresponds to the commit when the bug was first reported, and the second commit hash value corresponds to the commit when the bug



```java
private double sum(double x, double y) {
    double z = x + y;
    return z;
}
```

```java
public double sum(double a, double b) {
    return a + b;
}
```

Fig. 4: An example of *Increase Accessibility of Method*(IAM) change detection, where first method signatures (green sections) are matched. Once signatures are matched modifiers are compared for change type detection (red).

fix was made. In this study, we only focused on the bug fix commits, since we cannot locate when the bug was first introduced among the past commits.

Our change type analysis results for 13 projects are given in Figure 5. The results in Figure 5 indicate the type of changes involved per commit. They do not represent the number of changes that are made per change type in a commit. However, these results can also be obtained in our repository as well. For instance, in Figures 5k and 5h, both have a value of 1 for change type CM, which indicates that in every bug fix commit there was at least one CM type of change. The change types defined in Table I, II, and III, but do not exist in Figure 5 indicate that those change types did not exist in any of the bug fix commits for the corresponding project. According to the results, there are two types of changes that are commonly and consistently made while fixing bugs: changes made in the method body (*CM*), and adding a new method declaration (*AM*). We also would like to highlight that, although there are change types defined in class-level and field-level, both common change types that are found in bug fixes are method-level changes.

Ren et al. [2] reported that, in Java programs, *CM* and *AM* change types are found failure-inducing changes. By checking the bug fix changes we found that the types of changes for failure-inducing and bug fixing are exactly the same. Therefore, there might be a strong and positive correlation between bug fix and failure-inducing changes.

We have also evaluated the run-time performance of our automatic change type analysis that is given in Table V. Our run-time is evaluated based on the average time spent change type analysis per commit. The results have shown that per each commits our change type analysis ranges between ∼1–6 seconds, which is a reasonable time for aiding code reviewers.

TABLE V: Run-time evaluation of change type analysis

Project	Number of commits	Average run-time per commit (sec.)
closure-compiler	174	5.5287
commons-cli	39	1.0789
commons-codec	18	3.2222
commons-collections	4	4.5
commons-compress	47	1.5319
commons-csv	16	2.125
commons-jxpath	22	1.3182
gson	18	3
jackson-core	26	2.24
jackson-databind	108	2.1574
joda-time	26	3.3462
jsoup	93	3.5591
mockito	30	0.7

B. Change Impact Analysis Results

For change impact analysis, we utilized a recently proposed Markov chain-based tool to perform a change impact analysis [10]. The change impact analysis approach uses forward slicing for data dependency to find affected statements after a change and uses call graph information to find the dependency relationship between methods. However, as mentioned in Section IV and given in Table IV, we were only able to perform the change impact analysis on a subset of the change type analysis due to incomplete commits. Some of the analyzed projects were not compilable due to missing source files. Therefore, we have performed our change impact analysis on 232 of the 621 bug fix commits.

Our change impact results are given in Table VI. We measure the impact based on the impacted methods, where their probability is over 0.1. We have selected the 0.1 threshold due to the change impact analysis tool we used and its related study [10], [24] having shown that using the threshold 0.1 provides higher f-measure and recall results. To calculate the

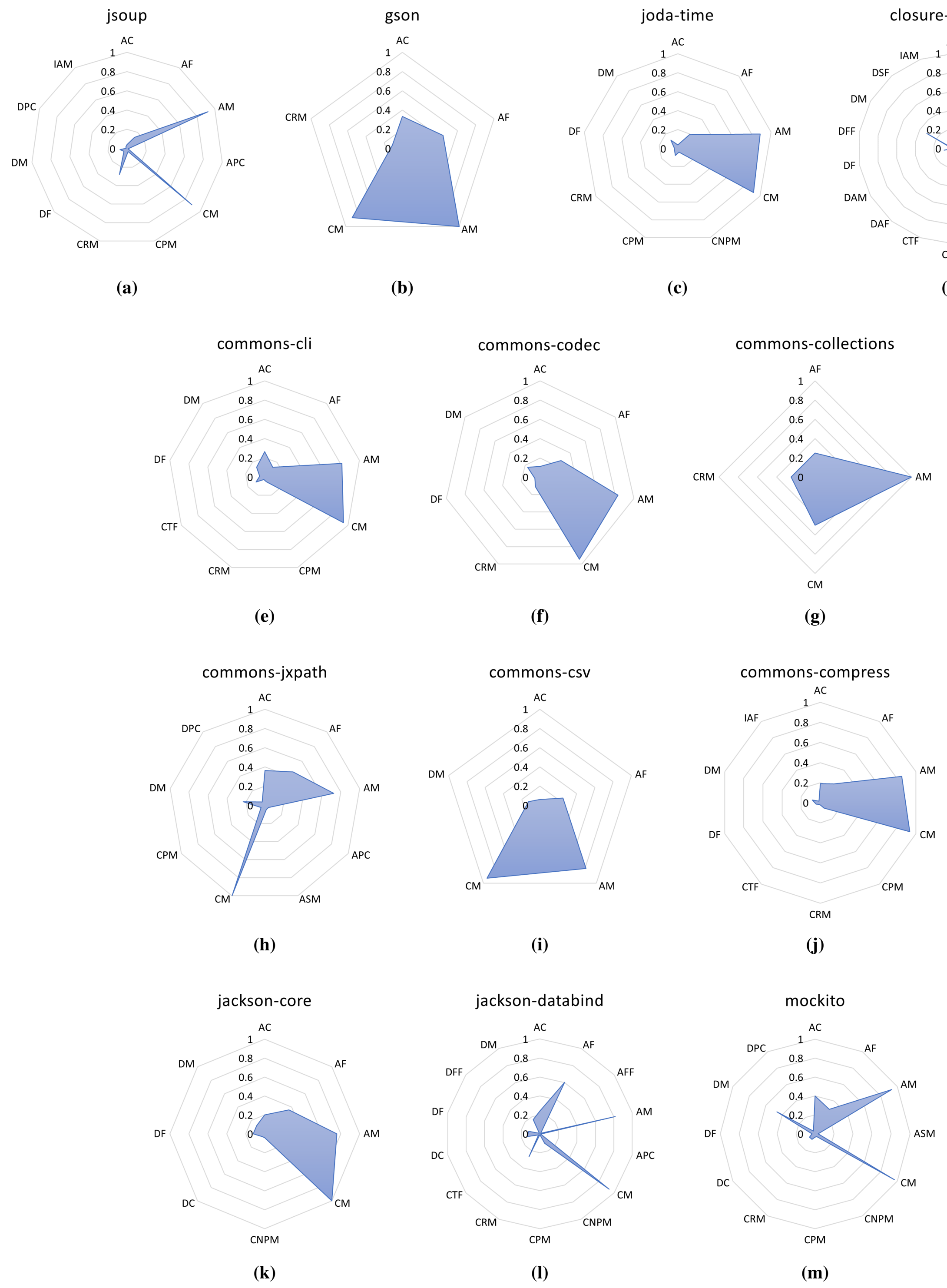


Fig. 5: Change types per commit for each project. The change types are ordered alphabetically in a clockwise direction. For each project, change types *CM* and *AM* are found common change types in bug fixes.

impact rate we divide the impacted methods by the total methods. According to our change impact analysis results, project *gson* has the maximum impact rate with 5%, and project *joda-time* has the lowest impact rate with 0.4%. The results show that the bug fix changes do not seem to have an high impact on other software components. However, to have a better understanding the relationship between change types and impact, in Section IV-C, we performed a statistical correlation analysis.

TABLE VI: Change Impact Analysis Results

Project Name	Avg. Impacted Methods	Avg. Total Methods	Avg. Impact Rate
jsoup	3.82	596.65	0.7%
joda-time	7.84	2210.28	0.4%
gson	68.94	1230.61	5%
commons-csv	1.56	373.06	0.5%
commons-codec	3.72	374.83	1.2%
jackson-core	7.39	891.74	0.8%
commons-compress	2.75	277.38	1%
commons-cli	4.35	278.39	1.7%

In Table VII, we present the run-time evaluation for change impact analysis. The change impact analysis run-time represents the average run-time per bug fix in seconds. The average run-times in our case study ranged between ∼16–186 seconds. However, we remark that this study does not propose a change impact analysis approach, and uses change impact analysis to find any correlation between bug fix changes and the impact that it causes. Therefore, the run-time in Table VII does not reflect any performance related to change type analysis.

TABLE VII: Run-time analysis for Change Impact Analysis per each bug fix

Project Name	Average run-time per bug fix (sec.)
jsoup	35.1445
joda-time	185.32
gson	30.812
commons-csv	15.44
commons-codec	25.746
jackson-core	28.675
commons-compress	24.758
commons-cli	32.9825

C. Correlation between bug fix changes and change impact analysis

To investigate if there is a correlation between any of the bug fix change types causing an impact in the software, we perform a statistical correlation analysis. For statistical correlation, we use *the Pearson correlation coefficient* which is a widely used measure for linear relationships between two normally distributed variables. In Equation (2), $cov(X,Y)$ is the covariance of random variable pairs (X,Y), while σ_x and σ_y are respectfully the standard deviation for X and Y. Based on the value obtained from Equation 2, the value of 1 represents a perfect positive relationship, -1 is a perfect negative relationship, and 0 indicates the absence of a relationship between variables.

$$\rho = \frac{\text{cov}(X,Y)}{\sigma_x \sigma_y} \tag{2}$$

We apply the correlation calculation (r_{xy}) on our sample by using the following Equation (3), which is also known as the *sample Pearson correlation coefficient*. The Equation (3) is simply obtained by substituting estimates of the covariances and variances based on a sample from the Equation (2), where n refers to the sample size. It is important to mention that the *Pearson correlation coefficient* works on numerical samples, and since that we had only information on the presence of change types (categorical data) corresponding to the caused impact rate (numerical data), we represent our change types in numerical format (Present-1, Absent-0). Finally, the calculated correlation coefficients are given in Table VIII. In Table VIII, we only included the change types that were only detected in bug fixes.

$$r = \frac{\sum_{i=1}^{n}(x_i - \overline{x})(y_i - \overline{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \overline{x})^2 (y_i - \overline{y})^2}} \tag{3}$$

To distinguish if there is a significant correlation between two variables (impact and change type), we use Equation 4 to calculate the minimum threshold that a significant relationship exists between the change type and the caused impact. The variable $|r|$ is the absolute value of variable r (correlation coefficient) from Equation 3, and the variable n is the size of the sample. Since that our sample size is *n = 232*, our minimum threshold is calculated as *0.1313*.

$$\text{if } |r| = \frac{2}{\sqrt{n}}, \text{ then relationship exists} \tag{4}$$

According to the results in Table VIII, we only see that there are two change types that have a significant relationship with the impact caused in the software. We observe that there is a negative correlation between the change types *CM* and *AM* with the caused impact in the software, which indicates that whenever a new method is added or whenever a change in the method body occurs, the impact decrease. On the other hand, there is a positive correlation between change type *CTF* with the impact caused in the software. This indicates that whenever a type of field is changed it is likely to cause and increase the impact on the software.

In Table VIII, it is also possible to extract information on co-changes in bug fixes. For instance, we observe that there are strong positive correlations between *DC* and *DF*, *AC* and *AF* which are meaningful. These changes indicate that whenever a new class is declared, a change of adding a new field follows. Similarly, whenever a class declaration is deleted, a change of field deletion follows. We also see that there is an exceptionally high correlation between *APC* and *DPC*, this correlation is a strong indicator that during bug fixes there have been changes in parent classes, by replacing a parent class with another class.

V. Discussion

A. Answering Research Questions

RQ1: What are the most common bug fix change patterns? To answer ***RQ1***, we performed a change type analysis on 13 Java projects and on 621 bug fix commits. We have observed that there are two commonly performed changes: *CM - Change in the method body*, and *AM - Adding a new method declaration*. For change type *CM*, the maximum change type per commit is 1, while the minimum is 0.5. On the other hand,

TABLE VIII: Correlation Results between caused impact and change types.

	Imp.	CM	AM	AF	AC	DM	APC	CPM	DPC	CNPM	CRM	CTF	DC	DF	IAM
Imp.	1														
CM	-0.449	1													
AM	-0.304	0.258	1												
AF	0.017	0.154	0.141	1											
AC	0.021	0.086	0.156	0.413	1										
DM	0.066	0.126	0.153	0.146	0.110	1									
APC	-0.027	0.024	0.030	0.131	0.189	-0.022	1								
CPM	0.018	0.074	0.031	0.065	0.002	0.078	-0.013	1							
DPC	-0.027	0.024	0.030	0.131	0.189	-0.022	1	-0.013	1						
CNPM	-0.033	0.035	0.042	0.069	0.118	-0.032	-0.006	-0.019	-0.006	1					
CRM	0.033	0.068	0.075	0.023	-0.136	0.075	-0.026	0.184	-0.026	-0.037	1				
CTF	0.236	0.042	0.051	-0.058	-0.040	0.086	-0.008	0.175	-0.008	-0.011	0.067	1			
DC	0.096	0.024	0.030	0.131	0.189	0.194	-0.004	-0.013	-0.004	-0.006	-0.026	-0.008	1		
DF	0.023	0.079	-0.018	0.157	-0.005	0.137	-0.014	-0.043	-0.014	-0.020	0.166	-0.024	0.310	1	
IAM	-0.021	0.035	-0.083	-0.047	-0.032	0.121	-0.006	-0.019	-0.006	-0.009	0.100	-0.011	-0.006	0.210	1

the maximum change type per commit for *AM* is also 1, while its minimum is 0.73.

We have also observed that among our 35 defined change types, only 14 of them were found. The changes that were not found were mostly related to class and field-based changes. Most of the changes are mostly made in a method-level granularity.

RQ2: Is there a relationship between the bug fix change patterns and the impact caused by change? To answer ***RQ2***, we have followed two steps. The first step was to find the impacted software components and calculate the impact rate. Then, in the second step, we perform a statistical analysis, where we find the correlation between change types and the caused impact. During our change impact analysis, we have not found high impact rates, in which our impact rates ranged between 0.4%–5%. However, we did realize that there was a significant gap between our lowest and highest impact rates. Therefore, we used *the Pearson correlation coefficient* to find the correlations between change types and impact results. The correlation results have shown that there was a significant correlation between the change types *AM*, *CM*, and *CTF* with the caused impact. However, we found that the correlation between change type *CTF* and caused impact was positive, while it was found negative for change types *AM* and *CM*.

B. Threats to Validity

Even though we have performed our case study on 13 open source projects by using a well-known bug dataset, it is important to mention that our case study is only limited to Java projects. Therefore, the bug fix change type characteristics may vary on different programming languages.

For change impact analysis, we have used one of the most recent techniques for finding impacted methods in the software. However, just as in every proposed change impact analysis technique, the impact analysis results might contain false positives and false negatives. During our impact analysis evaluations, we assumed that all the impacted methods are correct. Therefore, our impact analysis results might be higher or lower then it is supposed to be. However, the change impact analysis we have used [10] has very few false negatives. In other words, the change impact analysis technique we use has shown high recall results, and slightly low precision results, which indicates that the impact analysis results we presented in this study are slightly high. We can conclude that the impact caused by bug fixes is actually very low.

For generating parse tree we have used ANTLR, and the grammar that we used for change type analysis is designed for Java 1.8 syntax. Therefore, our change type analysis might have compatibility issues with older versions or newer versions of Java. However, it is important to remark that the projects we used in our case study, which are from Defects4J are based on Java version 1.8. Furthermore, Java 1.8 is still actively used in the industry.

VI. Related Work

There are successful studies on detecting change types. These studies started by focusing on adding structural change information to existing release history data for CVS [25]. Later, a taxonomy of source code changes was built, which defined source code changes related to tree edit operations in abstract syntax tree (AST) and classified each change type [11]. Then, Fluri et al. [12] have proposed an Eclipse plug-in called ChangeDistiller, where they also proposed a tree differencing algorithm to find the changes. Thereby, they were able to extract fine-grained source code changes, which can get fine-grained change information. Lin et al. [26] implemented an automatic tool called PyCT which is based on ChangeDistiller to reduce the effort for change extraction and classification. They have also introduced a taxonomy of Python source code changes.

Studies have also focused on investigating the change types from different perspectives. For instance, Vansics et al. [9] investigated the bug fix types in a method-level on JavaScript programs, by using the change types definitions from Pal et al. [27]. They investigated the relationship between the effectiveness of popular spectrum-based fault localization techniques and the bug fix changes. They found that some bug fix types were difficult and some were trivial to localize by an algorithm. For instance, changes in operation sequence tended to be difficult, while it was easier in if conditionrelated bugs. Roy et al. [7], proposed a model for detecting readability improvements, and used static analysis and change type analysis tools (COMING [13], CHANGEDISTILLER [12]. e.g.) for extracting change features.

There are also studies that exploited change patterns in training neural networks to automatically reproduce code changes implemented by the developers in pull requests of open source projects [5]. Furthermore, change pattern information has also been used to train neural networks to learn how to automatically fix bugs [4].

VII. CONCLUSION AND FUTURE WORK

In this study, we investigated the relationship between bug fix changes and it's impacts caused in the software. To investigate this relationship we have proposed and publicly shared an automatic change detection tool called CHANGE INSPECTOR JAVA (CIJ). We analyzed the bug fix change types from Defects4J with CIJ. We found that there are two common changes that are made in bug fixes: changing the method body and adding a new method declaration. Then we performed change impact analysis on the bug fix changes to analyze their impact on the software. Among the projects we analyzed, the impact caused in the software ranged between 0.4%–5%, which indicated a very low impact. However, wanted to investigate deeper to find any of the changes have a higher or lower impact. Therefore, we performed a statistical analysis using Pearson correlation coefficient to find any correlation between change types and the caused impact. We have found that among 14 change types, there were only 3 change types that had a significant correlation between the caused impact. The change types, adding a new method declaration, and changing the method body have shown a negative correlation with the caused impact, while the change type changing field type has shown a positive correlation.

Our study was not only limited to finding change types, and finding any correlation between bug fix changes and impact analysis. During our research, we have found that in commits such as given in https://github.com/apache/commons-csv/commit/c203896177b295c2f5319e8c34b9d8bb9f58564e, our change detection tool is able to distinguish that there is no change performed, which could reduce the code reviewing process when insufficient commit messages are provided.

In future work, we plan to extend our change detection tool by introducing statement-level change types. Since our results found that changing the method body is common in bug fixes, we want to investigate the characteristics of statement-level changes and their impact caused in the software.

REFERENCES

[1] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pp. 437–440, 2014.

[2] X. Ren, O. C. Chesley, and B. G. Ryder, "Identifying failure causes in java programs: An application of change impact analysis," *IEEE transactions on software engineering*, vol. 32, no. 9, pp. 718–732, 2006.

[3] X. Sun, B. Li, C. Tao, W. Wen, and S. Zhang, "Change impact analysis based on a taxonomy of change types," in *2010 IEEE 34th Annual Computer Software and Applications Conference*, pp. 373–382, IEEE, 2010.

[4] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshyvanyk, "An empirical investigation into learning bug-fixing patches in the wild via neural machine translation," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pp. 832–837, 2018.

[5] M. Tufano, J. Pantiuchina, C. Watson, G. Bavota, and D. Poshyvanyk, "On learning meaningful code changes via neural machine translation," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pp. 25–36, IEEE, 2019.

[6] V. Sobreira, T. Durieux, F. Madeiral, M. Monperrus, and M. de Almeida Maia, "Dissection of a bug dataset: Anatomy of 395 patches from defects4j," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 130–140, IEEE, 2018.

[7] D. Roy, S. Fakhoury, J. Lee, and V. Arnaoudova, "A model to detect readability improvements in incremental changes," in *Proceedings of the 28th International Conference on Program Comprehension*, pp. 25–36, 2020.

[8] A. Szatmári, B. Vancsics, and Á. Beszédes, "Do bug-fix types affect spectrum-based fault localization algorithms' efficiency?," in *2020 IEEE Workshop on Validation, Analysis and Evolution of Software Tests (VST)*, pp. 16–23, IEEE, 2020.

[9] B. Vancsics, A. Szatmári, and Á. Beszédes, "Relationship between the effectiveness of spectrum-based fault localization and bug-fix types in javascript programs," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 308–319, IEEE, 2020.

[10] E. Ufuktepe and T. Tuglular, "Code change sniffer: Predicting future code changes with markov chain," in *IEEE Annual International Computer Software and Applications Conference*, pp. 1014–1019, IEEE, 2021.

[11] B. Fluri and H. C. Gall, "Classifying change types for qualifying change couplings," in *14th IEEE International Conference on Program Comprehension (ICPC'06)*, pp. 35–45, IEEE, 2006.

[12] B. Fluri, M. Wursch, M. PInzger, and H. Gall, "Change distilling: Tree differencing for fine-grained source code change extraction," *IEEE Transactions on software engineering*, vol. 33, no. 11, pp. 725–743, 2007.

[13] M. Martinez and M. Monperrus, "Coming: A tool for mining change pattern instances from git commits," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pp. 79–82, IEEE, 2019.

[14] D. Spadini, M. Aniche, and A. Bacchelli, "Pydriller: Python framework for mining software repositories," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 908–911, 2018.

[15] M. M. Lehman, "Programs, life cycles, and laws of software evolution," *Proc. of the IEEE*, vol. 68, no. 9, pp. 1060–1076, 1980.

[16] N. F. Schneidewind, "The state of software maintenance," *IEEE Trans. Softw. Eng.*, no. 3, pp. 303–310, 1987.

[17] M. M. Lehman and L. A. Belady, *Program evolution: processes of software change*. Academic Press Professional, Inc., 1985.

[18] V. Basili, L. Briand, S. Condon, Y.-M. Kim, W. L. Melo, and J. D. Valen, "Understanding and predicting the process of software maintenance releases," in *IEEE Int. Conf. on Softw. Eng.*, pp. 464–474, 1996.

[19] W. Harrison and C. Cook, "Insights on improving the maintenance process through software measurement," in *IEEE Conf. on Softw. Maintenance*, pp. 37–45, 1990.

[20] A. Abran and H. Nguyenkim, "Analysis of maintenance work categories through measurement.," in *IEEE Conf. on Softw. Maintenance*, pp. 104–113, 1991.

[21] A. Mockus and D. M. Weiss, "Predicting risk of software changes," *Bell Labs Technical Journal*, vol. 5, no. 2, pp. 169–180, 2000.

[22] A. Kumar, "Development at the speed and scale of google," *QCon San Francisco*, 2010.

[23] T. Parr, *The definitive ANTLR 4 reference*. Pragmatic Bookshelf, 2013.

[24] E. Ufuktepe and T. Tuglular, "A program slicing-based bayesian network model for change impact analysis," in *IEEE International Conference on Software Quality, Reliability and Security*, pp. 490–499, 2018.

[25] B. Fluri, H. C. Gall, and M. Pinzger, "Fine-grained analysis of change couplings," in *Fifth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'05)*, pp. 66–74, IEEE, 2005.

[26] W. Lin, Z. Chen, W. Ma, L. Chen, L. Xu, and B. Xu, "An empirical study on the characteristics of python fine-grained source code change types," in *2016 IEEE international conference on software maintenance and evolution (ICSME)*, pp. 188–199, IEEE, 2016.

[27] K. Pan, S. Kim, and E. J. Whitehead, "Toward an understanding of bug fix patterns," *Empirical Software Engineering*, vol. 14, no. 3, pp. 286–315, 2009.