This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

Tracking Code Bug Fix Ripple Effects Based on Change Patterns Using Markov Chain Models

Ekincan Ufuktepe, Tugkan Tuglular, *Member, IEEE*, and Kannappan Palaniappan

***Abstract*—Change impact analysis evaluates the changes that are made in the software and finds the ripple effects, in other words, finds the affected software components. Code changes and bug fixes can have a high impact on code quality by introducing new vulnerabilities or increasing their severity. A recent high-visibility example of this is the code changes in the log4j web software CVE-2021-45105 to fix known vulnerabilities by removing and adding method called change types. This bug fix process exposed further code security concerns. In this article, we analyze the most common set of bug fix change patterns to have a better understanding of the distribution of software changes and their impact on code quality. To achieve this, we implemented a tool that compares two versions of the code and extracts the changes that have been made. Then, we investigated how these changes are related to change impact analysis. In our case study, we identified the change types for bug-inducing and bug fix changes using the Quixbugs dataset. Furthermore, we used 13 of the projects and 621 bugs from Defects4J to identify the common change types in bug fixes. Then, to find the change types that cause an impact on 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 with 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*—Bug fix, change detection, change impact analysis.**

I. Introduction

CHANGE is a continual and integral part 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 originate 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 the commit where the bug is introduced. 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, which 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, and deletion/addition of classes and methods. These types of changes were classified as changes that cause a ripple effect in the 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, such as Github, can provide the changes that are made in the software; however, they do not give 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 about 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 we found at[1] is partially given in Fig. 1. This example shows that some commits in repositories present code changes; however, when code is carefully reviewed there are actually no changes made in the software. In this example, there are 67 additions and 67 deletions in one class however, 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 actual code change in the software, when the code is being reviewed on Github it could easily mislead code reviewers 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,

Manuscript received February 7, 2022; accepted March 21, 2022. Associate Editor: Z. Zheng. *(Corresponding author: Ekincan Ufuktepe.)*

Ekincan Ufuktepe and Kannappan Palaniappan are with the University of Missouri - Columbia, Columbia, MO 65211 USA (e-mail: euh46@missouri.edu; pal@missouri.edu).

Tugkan Tuglular is with the Izmir Institute of Technology, 35430 Izmir, Turkey (e-mail: tugkantuglular@iyte.edu.tr).

Color versions of one or more figures in this article are available at https://doi.org/10.1109/TR.2022.3167943.

Digital Object Identifier 10.1109/TR.2022.3167943

[1][Online]. Available: https://github.com/apache/commons-csv/commit/c203896177b295c2f5319e8c34b9d8bb9f58564e

0018-9529 © 2022 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See https://www.ieee.org/publications/rights/index.html for more information.

Authorized licensed use limited to: University of Missouri Libraries. Downloaded on May 11,2022 at 17:56:11 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.


```
66  +     private void assertNotEquals(String name, String type, Object left, Object right) {
67  +         if (left.equals(right) || right.equals(left)) {
68  +             fail("Objects must not compare equal for " + name + "(" + type + ")");
69  +         }
70  +         if (left.hashCode() == right.hashCode()) {
71  +             fail("Hash code should not be equal for " + name + "(" + type + ")");
72  +         }
73  +     }
```

```
1114 -     private void assertNotEquals(String name, String type, Object left, Object right) {
1115 -         if (left.equals(right) || right.equals(left)) {
1116 -             fail("Objects must not compare equal for " + name + "(" + type + ")");
1117 -         }
1118 -         if (left.hashCode() == right.hashCode()) {
1119 -             fail("Hash code should not be equal for " + name + "(" + type + ")");
1120 -         }
1121 -     }
```

Fig. 1. Example commit from Apache commons-csv, in which the diff shows that a new method is implemented (left-hand side) and an existing method is deleted (right-hand side). However, the added and the deleted methods are exactly the same. The only change is that a method is repositioned, but there is no change that would cause a side effect or an impact on the software.

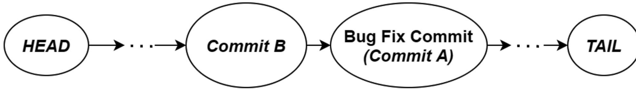

Fig. 2. Experimental design for RQ2. *Commit A* is the bug fix commit, where *Commit B* is the parent commit of *Commit A*.

change pattern information has been used in training neural networks [4], [5] for automatic bug fixing, understanding bug fixes and for automatic program repair [6], detecting readability improvements [7], and in fault localization [8], [9].

In this article, we first share our novel approach and architecture for detecting change types of a given commit that cause an impact on the software. Then, we detect change types for the commits with bug fixes from Defects4J and investigate the relation between change types, which are likely to cause an impact on other software components. Moreover, we utilized the QuixBugs[2] [10] benchmark and dataset. The dataset includes 40 Java programs. For each program, there is a defective version and a corrected version. This gives us the flexibility to analyze change types bidirectionally as follows:

1) First, analyzing the change types based on the fixed defects (defect version changed to corrected version).
2) Second, analyzing the change types based on bug-inducing changes (corrected version changed to defect version).

We develop the following research questions and answer them in the discussion section.

1) *RQ1: What are the Most Common Bug Fix Change Patterns?:* To answer this question, we analyze all the bug fixes from 13 projects in Defects4J, and also all the bug fixes from 40 Java programs in the Quixbugs dataset, for a total of 661 bug fixes. We identify the change types that are performed in the bug fix commits and find the commonly performed change actions in these bug fix commits.
2) *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 perform a change impact analysis between two commits: a) the bug fix commit; and b) the parent commits of a bug, which is demonstrated in Fig. 2. 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 article makes the following main contributions.

1) *Method:* We present an empirical study that analyzes the relation between the bug fix change patterns and changes that cause an impact on other software components. We aim 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 [11].[3] Furthermore, we have extended our previous work [12], and provided a novel approach to detect change types between two commits that cause an impact on software components, while other studies [13]–[15] 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 (ASTs), due to its capability of containing keywords, which makes it easy to detect changes, such as type changes, modifier changes, etc. Our change-type detection works on the following four different levels.
   1) Class-level changes.
   2) Field-level changes.
   3) Method-level changes.
   4) Statement-level (method body) changes.

   We have also investigated change types for the bug fixes that are made for the Log4j CVE-2021-45105 vulnerability.
2) *Tool:* We developed and extended our tool called change inspector Java (CIJ)[4] [12] and made it publicly available for detecting code changes. The tool uses PyDriller [16] for mining Github repositories. CIJ detects the changes that are made and implements the end-to-end pipeline for predicting future code changes with the Markov chain.
3) *Dataset:* We supply our detected change-type dataset to other researchers and practitioners to provide a better reproducibility of our study. Our dataset contains the following.

[2][Online]. Available: https://github.com/jkoppel/QuixBugs

[3][Online]. Available: https://github.com/ekincanufuktepe/code-change-sniffer

[4][Online]. Available: https://github.com/ekincanufuktepe/change-instepector-java

Authorized licensed use limited to: University of Missouri Libraries. Downloaded on May 11,2022 at 17:56:11 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

1) The change types detected by CIJ for the corresponding bug fix commits from Defects4J.
2) The change types detected by CIJ for the corresponding bug-inducing and bug fix changes from Quixbugs.
3) The probabilities of methods being impacted by the bug fixes.

The rest of this article is organized as follows. Section II provides the background on the 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 and gives its 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. Finally, Section VII concludes this article.

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 [11], which we used in our study.

A. Change Impact Analysis

In the 1980s, Lehman [17] and Schneidewind [18] mentioned the difficulties that software evolution has brought to software maintenance. One of these difficulties is the ripple effects, which are caused by changes in the source code. On the other hand, evolution in software development had been considered to be inevitable, and change should be accepted as an intrinsic part of the software development life cycle [17]. Nowadays, this is still true. However, changes are more rapid due to advancements in the technology and user expectations. Lehman and Belady [19] 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 [20]–[22] 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 [23]. An example of fast-evolving software was introduced by Kumar [24], 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 of finding the code sections that need to be changed, or highlight codes that require a change to fix the possible errors, which are introduced by modifications.

B. Running Example of Code Change Sniffer

Code Change Sniffer [11] 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 example code, which we will be running, is a small program that finds the prime numbers, which is shown in Fig. 3(a) and will be referred to as program *SIEVE*.

Code Change Sniffer first extracts the static call graph (CG) of the program, which represents the call relationships between methods. For program *SIEVE* from Fig. 3(a), we extract the static CG, which is shown in Fig. 4. Then, we perform the forward slicing on each method in the program based on its parameters. The program slicing example and results for the program *SIEVE* are shown in Fig. 3(b), together with the original code in Fig. 3.

Based on the results of program slicing, we calculate the probability of how much the method will be affected by changed arguments/parameters. To calculate the probability of how much the program will be affected, we divide the remaining lines after slicing by the total lines before slicing. In the following, we show probabilities for each method, which will be used as a part of the transition matrix.

1) *main:* $0/1 = 0$.
2) *sieve:* $4/5 = 0.8$.
3) *all:* $2/3 = 0.67$.
4) *list_comp:* $3/4 = 0.75$.

After the probabilistic information is obtained from forward slicing, we encode them into the Markov chain's edges along with the change information based on the type of the model, namely CG and the effect graph. However, it is important to mention that we used CGs 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 T, such as given in the following, for example Fig 5, which is similar to an adjacency matrix:

1) m_0**:** *main*;
2) m_1**:** *sieve*;
3) m_2**:** *all*;
4) m_3**:** *list_comp*;

$$T = \begin{array}{c} \\ m_0 \\ m_1 \\ m_2 \\ m_3 \end{array} \begin{array}{c} \begin{array}{cccc} m_0 & m_1 & m_2 & m_3 \end{array} \\ \left[\begin{array}{cccc} 0 & 0.8 & 0 & 0 \\ 0 & 0 & 0.67 & 0.75 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{array}\right] \end{array}.$$

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-hand side of Fig. 5, 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'

Authorized licensed use limited to: University of Missouri Libraries. Downloaded on May 11,2022 at 17:56:11 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.


```java
public class SIEVE {

    public static void main(String args[]) {
        sieve(100);
    }

    public static boolean all(ArrayList<Boolean> arr) {
        for (boolean value : arr) {
            if (!value) { return false; }
        }
        return true;
    }

    public static ArrayList<Boolean> list_comp(int n, ArrayList<Integer> primes) {
        ArrayList<Boolean> built_comprehension = new ArrayList<Boolean>();
        for (Integer p : primes) {
            built_comprehension.add(n % p > 0);
        }
        return built_comprehension;
    }

    public static ArrayList<Integer> sieve(Integer max) {
        ArrayList<Integer> primes = new ArrayList<Integer>();
        for (int n=2; n<max+1; n++) {
            if (all(list_comp(n, primes))) {
                primes.add(n);
            }
        }
        return primes;
    }
}
```

(a)

```java
public class SIEVE {

    public static void main(String args[]) {

    }

    public static boolean all(ArrayList<Boolean> arr) {
        for (boolean value : arr) {
            if (!value) { return false; }
        }
    }

    public static ArrayList<Boolean> list_comp(int n, ArrayList<Integer> primes) {
        for (Integer p : primes) {
            built_comprehension.add(n % p > 0);
        }
        return built_comprehension;
    }

    public static ArrayList<Integer> sieve(Integer max) {
        for (int n=2; n<max+1; n++) {
            if (all(list_comp(n, primes))) {
                primes.add(n);
            }
        }
        return primes;
    }
}
```

(b)

Fig. 3. (a) Original code is given. (b) Sliced code is given, which is a subset of the original code, where the unaffected lines of codes are removed. The remaining code represents the affected lines after a change are made in the parameters for every method in the code.

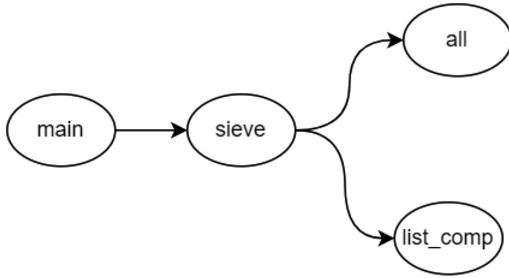

Fig. 4. Static CG of the program *SIEVE*.

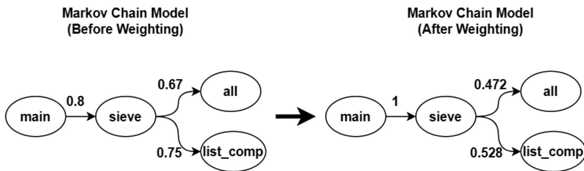

Fig. 5. Markov chain model construction with weighted edges.

summation of outgoing edges is 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 by each outgoing edge of that node. On the right-hand side of Fig. 5, we obtain the updated Markov chain after weighting the edges.

After the weighting process is completed, we construct the transition weighted matrix T_w of the Markov chain model as follows. According to the graph model in Fig. 5, there is no outgoing edge from methods m_2 (*all*) and m_3 (*list_com*). Therefore, in the transition matrix, we would expect to have the entire row filled with zeroes. However, we have a single 1, which is placed to itself, such as $m_2 \to m_2$ and $m_3 \to m_3$. 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 $m_i \to m_i$ edge probability to 1. If the method m_i is not changed, setting the probability will not affect the overall impact calculation, since it will be multiplied by 0.

$$T_w = \begin{array}{c} \\ m_0 \\ m_1 \\ m_2 \\ m_3 \end{array} \begin{array}{c} \begin{array}{cccc} m_0 & m_1 & m_2 & m_3 \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0.472 & 0.528 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \end{array}.$$

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 by 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, which are made to the current method. Therefore, as the amount of change increases the probability of being affected by changes will be higher.

Fig. 6 shows the original code and the changed code example for SIEVE. In Fig. 6(b), we see that only the *sieve* (m_1) method is changed, where only the FOR-*statement* is modified. To measure the number of changes made can be quantified differently. For example, the change can be calculated based on the lines of source codes, or for Java it could be the number of statements in bytecode. Even though the tool [11] we used is for quantifying the percentage of change is based on bytecode; to simplify the example, we will use lines of Java source code. In the running example given in Fig. 6(b), the percentage of change made in *sieve* method is $1/5 = 0.2$. Thereby, in the Markov chain, the changed methods with given probabilities are $m_0 = 0.0$, $m_1 = 0.2$, $m_2 = 0.0$, and $m_3 = 0.0$. The four given change-probabilities are encoded into the initial vector as follows. Previously, to satisfy the properties of the Markov chain in the

Authorized licensed use limited to: University of Missouri Libraries. Downloaded on May 11,2022 at 17:56:11 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.


```
public class SIEVE {

    public static void main(String args[]) {
        sieve(100);
    }

    public static boolean all(ArrayList<Boolean> arr) {
        for (boolean value : arr) {
            if (!value) { return false; }
        }
        return true;
    }

    public static ArrayList<Boolean> list_comp(int n, ArrayList<Integer> primes) {
        ArrayList<Boolean> built_comprehension = new ArrayList<Boolean>();
        for (Integer p : primes) {
            built_comprehension.add(n % p > 0);
        }
        return built_comprehension;
    }

    public static ArrayList<Integer> sieve(Integer max) {
        ArrayList<Integer> primes = new ArrayList<Integer>();
        for (int n=2; n<max+1; n++) {
            if (all(list_comp(n, primes))) {
                primes.add(n);
            }
        }
        return primes;
    }
}
```

(a)

```
public class SIEVE {

    public static void main(String args[]) {
        sieve(100);
    }

    public static boolean all(ArrayList<Boolean> arr) {
        for (boolean value : arr) {
            if (!value) { return false; }
        }
        return true;
    }

    public static ArrayList<Boolean> list_comp(int n, ArrayList<Integer> primes) {
        ArrayList<Boolean> built_comprehension = new ArrayList<Boolean>();
        for (Integer p : primes) {
            built_comprehension.add(n % p > 0);
        }
        return built_comprehension;
    }

    public static ArrayList<Integer> sieve(Integer max) {
        ArrayList<Integer> primes = new ArrayList<Integer>();
        for (int n=1; n<=max; n++) {        change in for loop
            if (all(list_comp(n, primes))) {
                primes.add(n);
            }
        }
        return primes;
    }
}
```

(b)

Fig. 6. (a) Original code is given. (b) Changed code is given, which is in the method *sieve*, where the FOR loop is modified.

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 less than 1.

$$I = [\ 0.0\ 0.2\ 0.0\ 0.0\ ].$$

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 as follows:

$$I_w = [\ 0\ 1\ 0\ 0\ ].$$

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 (1), which is predicted to be changed methods. Since our initial vector and transition matrix are weighted, we expect to calculate the impact vector, where the summation of its probabilities is equal to 1.

$$I_w x T_w = [\ 0\ 0\ 0.472\ 0.528\ ]. \tag{1}$$

Based on the Markov chain model in Fig. 5 and calculation in (1), the probabilities of the methods being affected by the changes are calculated as $m_0 = 0$, $m_1 = 0.0$, $m_2 = 0.472$, and $m_3 = 0.528$. With respect to the change made in method *sieve* (m_1), two methods are affected: 1) m_2; and 2) m_3. However, the results indicate that method *list_comp* (m_3) has the highest likelihood of being affected by the changes.

In Fig. 7, we show the CG of one of the small-scale projects from our case study. The CG is used for constructing the transition matrix, where the edge weights are calculated with program slicing.

III. Change Detection Architecture

In this section, we introduce the details of our change detection tool and architecture CIJ, as shown in Fig. 8.

A. Change Types

In the context of change impact analysis, the changes in the source for Java programming language can occur in four categories: 1) class-type changes; 2) method-type changes; 3) method body (statement)-type changes; and 4) field-type changes. These types of changes were introduced by Ren *et al.* [2], Sun *et al.* [3], Duraes and Madeira [25], and Martinez *et al.* [26]. In this article, 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 *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 delete, and a new method added.*

The change types for classes, methods, fields, and statements are given in Tables I–IV, respectively.

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

Authorized licensed use limited to: University of Missouri Libraries. Downloaded on May 11,2022 at 17:56:11 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

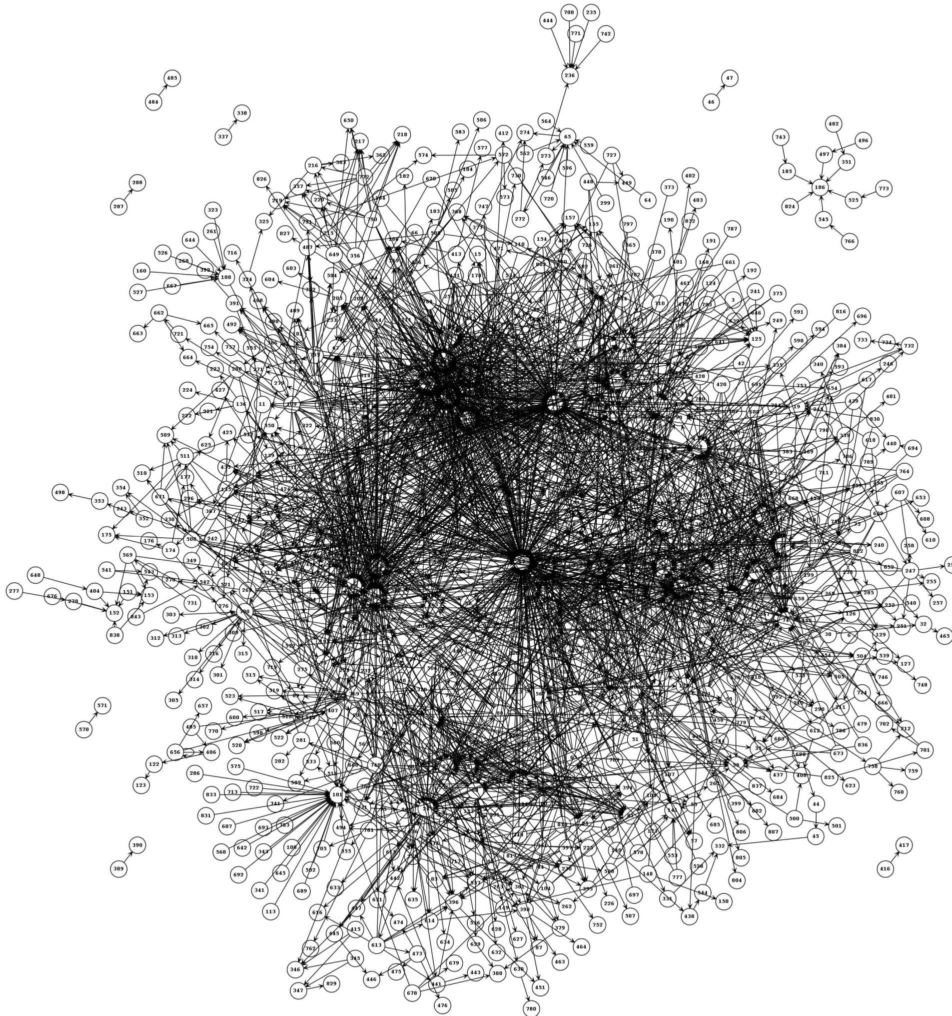

Fig. 7. CG example from Apache *commons-csv* project, which is used for constructing the transition matrix for the Markov chain model.

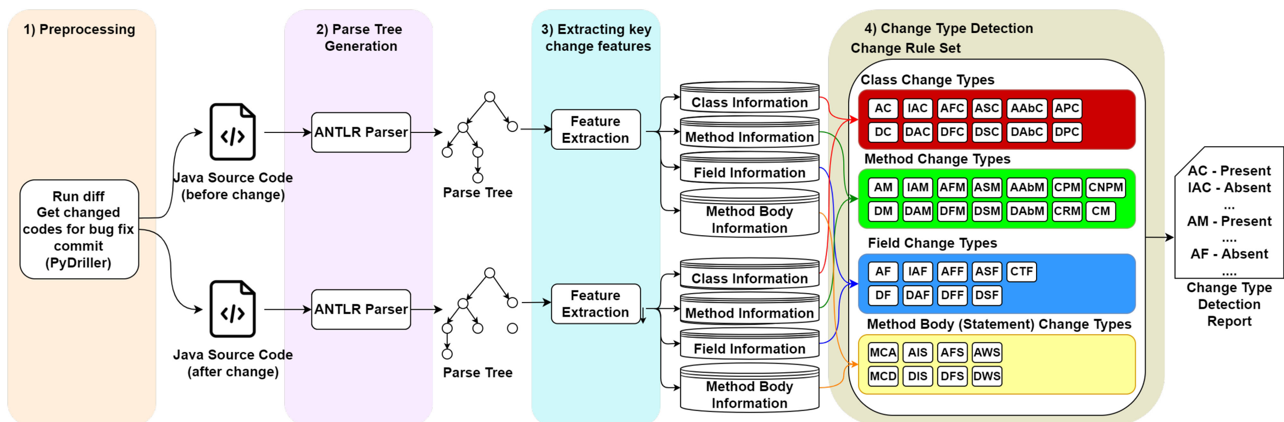

Fig. 8. CIJ architecture with multithreaded change-type detection. The change detection starts with preprocessing by only extracting changed files, then follows a parse tree generation for each changed file. From parse trees, key change features are extracted to be used in change-type detection. For each change type and its detection, a thread is generated and reported.

Authorized licensed use limited to: University of Missouri Libraries. Downloaded on May 11,2022 at 17:56:11 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

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

key change features, and detecting change type, which is also provided in Fig. 8.

1) Preprocessing: The preprocessing phase is where we first retrieve the changed source codes. Retrieving the changed source codes is achieved by PyDriller [16]. We simply provide the bug fix commit hash to PyDriller and extract the changes with the *diff* tool. Using the *diff* tool also provides which source files are changed and allows us to download the source files as well. The preprocessing 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[5] [27], and we have used the grammar designed for Java 1.8. In a previous work [15], ASTs were used for change-type detection, due to their compactness and ease of the 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 to 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 four categories: a) class; b) method; c) field; and d) method body (statements). For each category, we have separate abstract data types defined, and since we are interested in particular data in each category, we have different information extracted. For a class, we extract the class name, parent classes, and modifiers. For a field, we extract field names, modifiers, and types. For a method, we extract method name, modifiers, return type, parameter names, parameter modifiers, parameter types, and method body. For method body information (statements), we extract method call information, IF-statements, WHILE-statements, and FOR-statements.

4) Change-Type Detection: The change-type detection is performed based on comparing two abstract data types. Each change rule is defined as a subclass of *ChangeRule*, where every change rule overrides two methods: a) *getCategory;* and b) *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 the change rule. This type of design allows us to implement our architecture in multithreaded to achieve better performance.

Each change rule class receives two parse trees as inputs: 1) one derived from bug fix changes; and 2) the other derived from the parent commit of the bug fix changes. For each change type, the parse trees are parsed separately and only target the key features that we are interested in. For instance, as demonstrated in Fig. 9, 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 consists of the 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 we label the change as an *IAM* change.

Currently, CIJ contains 43 change rules in total. New rules can be implemented or existing rules can be extended. We have also built and designed CIJ, which allows users to select that which change types they are interested in finding. Even though by default, all the change rules are set active, some can be deactivated, which could increase the speed.

IV. CASE STUDY

Our case study is composed of the following two phases.

1) The first phase is identifying the common change types among 13 Java projects from Defects4J and 40 Java programs from Quixbugs.
2) The second phase is performing a change impact analysis on a subset of projects and bug fixes from Defects4J.

Some of the commits had missing source files, which prevented us to compile and perform change impact analysis on them. These projects and commits were discarded from our change impact analysis. Therefore, we performed change impact analysis only on eight Java projects and 232 bug fixes from Defects4J, which is a subset of our phase 1 analysis.

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], as well as 40 bug fixes from 40 Java programs in Quixbugs [10]. The selected projects and bugs are given in Table V. The bugs are collected from the *active bugs*, which contain the following two commit hash information.

1) The first commit hash value corresponds to the commit when the bug was first reported.
2) The second commit hash value corresponds to the commit when the bug fix was made.

[5][Online]. Available: https://github.com/antlr/grammars-v4

Authorized licensed use limited to: University of Missouri Libraries. Downloaded on May 11,2022 at 17:56:11 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

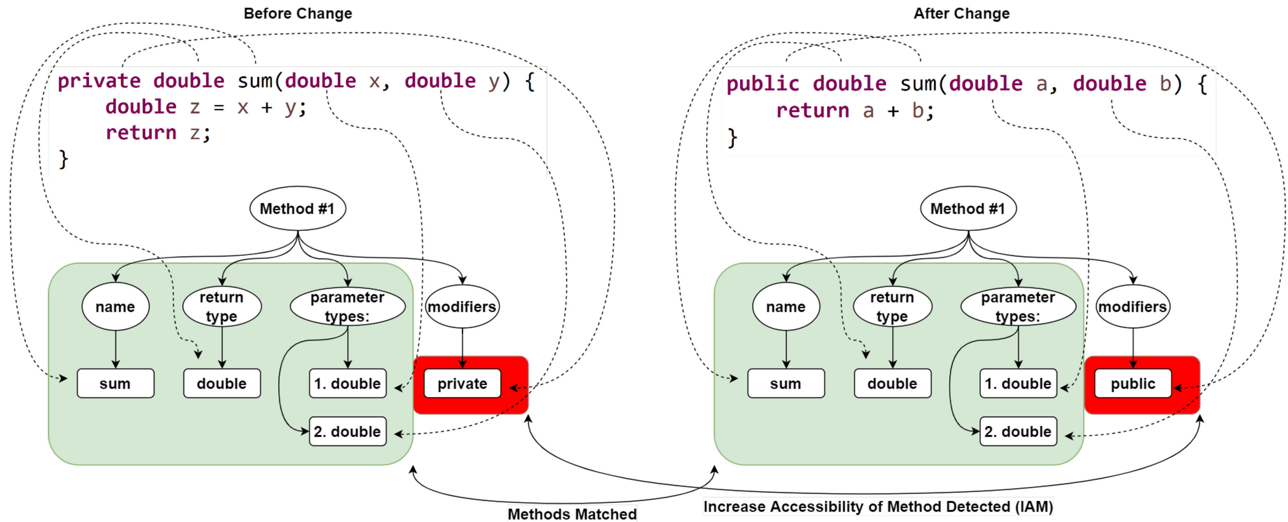

Fig. 9. Example of *increase accessibility of method* (IAM) change detection. The first method signatures (green sections) are matched. Once signatures are matched, modifiers are compared for change-type detection (red).

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

TABLE IV
TYPES OF STATEMENT CHANGES

Type	Description
MCA	Method call added (adding a method invocation)
MCD	Method call deleted (deleting a method invocation)
AIS	Adding *if-statement*
DIS	Deleting *if-statement*
AWS	Adding *while-statement*
DWS	Deleting *while-statement*
AFS	Adding *for-statement*
DFS	Deleting *for-statement*

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 and 40 Java programs are shown in Fig. 10. Except for QuixBugs in Fig. 10(h), the results in Fig. 10 indicate the type of changes involved per commit. They do not represent the number of changes made per change type in a commit. However, these results can also be obtained in our repository. For instance, in Fig. 10(e) and (l), 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 Tables I–IV, which are missing in Fig. 10, indicate that those change types did not exist in any bug fix commits for the corresponding project. According to the results, there are four types of changes that are commonly and consistently made while fixing bugs: 1) changes made in the method body (*CM*); 2) adding a new method declaration (*AM*); 3) adding a method invocation (*MCA*); and 4) deleting a method invocation (*MCD*). We also would like to highlight that although there are change types defined at class-level and field-level, four common change types, which were found in bug fixes, are at method-level and statement-level (in method body) 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 fixes and failure-inducing changes.

Authorized licensed use limited to: University of Missouri Libraries. Downloaded on May 11,2022 at 17:56:11 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

Fig. 10. Change types per commit for each project [except for QuixBugs 10(h)]. The change types are ordered alphabetically in a clockwise direction. For each project, change types *CM*, *AM*, *MCA*, and *MCD* are found common change types in bug fixes. The labels of changes type are given in Tables I–IV. (a) Jsoup. (b) Gson. (c) Joda-time. (d) Closure-compiler. (e) Jackson-core. (f) Jackson-databind. (g) Mockito. (h) QuixBugs. (i) Commons-cli. (j) Commons-codec. (k) Commons-collections. (l) Commons-jxpath. (m) Commons-csv. (n) Commons-compress.

We have also evaluated the run-time performance of our automatic change-type analysis, as given in Table VI and Fig. 11. For the Defects4J projects, our run-time is evaluated based on the average time spent on the change-type analysis per commit, while for QuixBugs programs the time represents the change-type analysis spent per file. With all the change rules activated, the results have shown that per each commits/file our change-type analysis ranges between ~1–7 s, which is a reasonable time for aiding code reviewers.

B. Change Impact Analysis Results

For the change impact analysis, we utilized a recently proposed Markov chain-based tool to perform a change impact

Authorized licensed use limited to: University of Missouri Libraries. Downloaded on May 11,2022 at 17:56:11 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

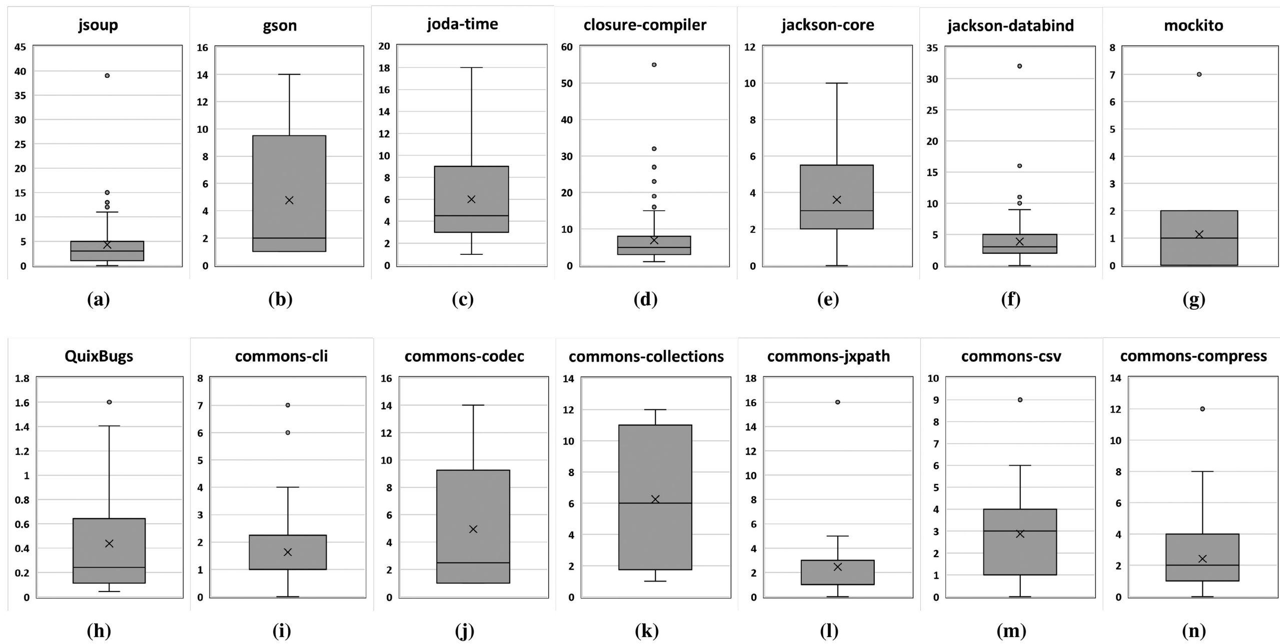

Fig. 11. Timing in seconds for the change-type analysis, using box plots per commit for each project [except for (h)]. The change-type analysis time execution for [11(h)] QuixBugs, which is calculated for each class. The "x" in each plot is the mean value. (a) Jsoup. (b) Gson. (c) Joda-time. (d) Closure-compiler. (e) Jackson-core. (f) Jackson-databind. (g) Mockito. (h) QuixBugs. (i) Commons-cli. (j) Commons-codec. (k) Commons-collections. (l) Commons-jxpath. (m) Commons-csv. (n) Commons-compress.

TABLE V
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	-
quixbugs	40	-

TABLE VI
RUN-TIME EVALUATION OF THE CHANGE-TYPE ANALYSIS

Project	Number of commits	Average run-time per commit/file (sec)
closure-compiler	174	6.931
commons-cli	39	1.632
commons-codec	18	4.944
commons-collections	4	6.250
commons-compress	47	2.426
commons-csv	16	2.875
commons-jxpath	22	2.455
gson	18	4.778
jackson-core	26	3.600
jackson-databind	108	3.843
joda-time	26	6.000
jsoup	93	4.301
mockito	30	1.133
quixbugs	30	0.350

TABLE VII
CHANGE IMPACT ANALYSIS RESULTS

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

analysis [11]. The change impact analysis approach uses forward slicing for data dependency to find affected statements after a change and uses CG information to find the dependency relationship between methods. However, as mentioned in Section IV and given in Table V, 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 from Defects4J. We have not performed a change impact analysis on the QuixBugs dataset, due to the Java programs only consisting of a single Java class file with very few methods, which is insignificant to perform a change impact analysis on.

Authorized licensed use limited to: University of Missouri Libraries. Downloaded on May 11,2022 at 17:56:11 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

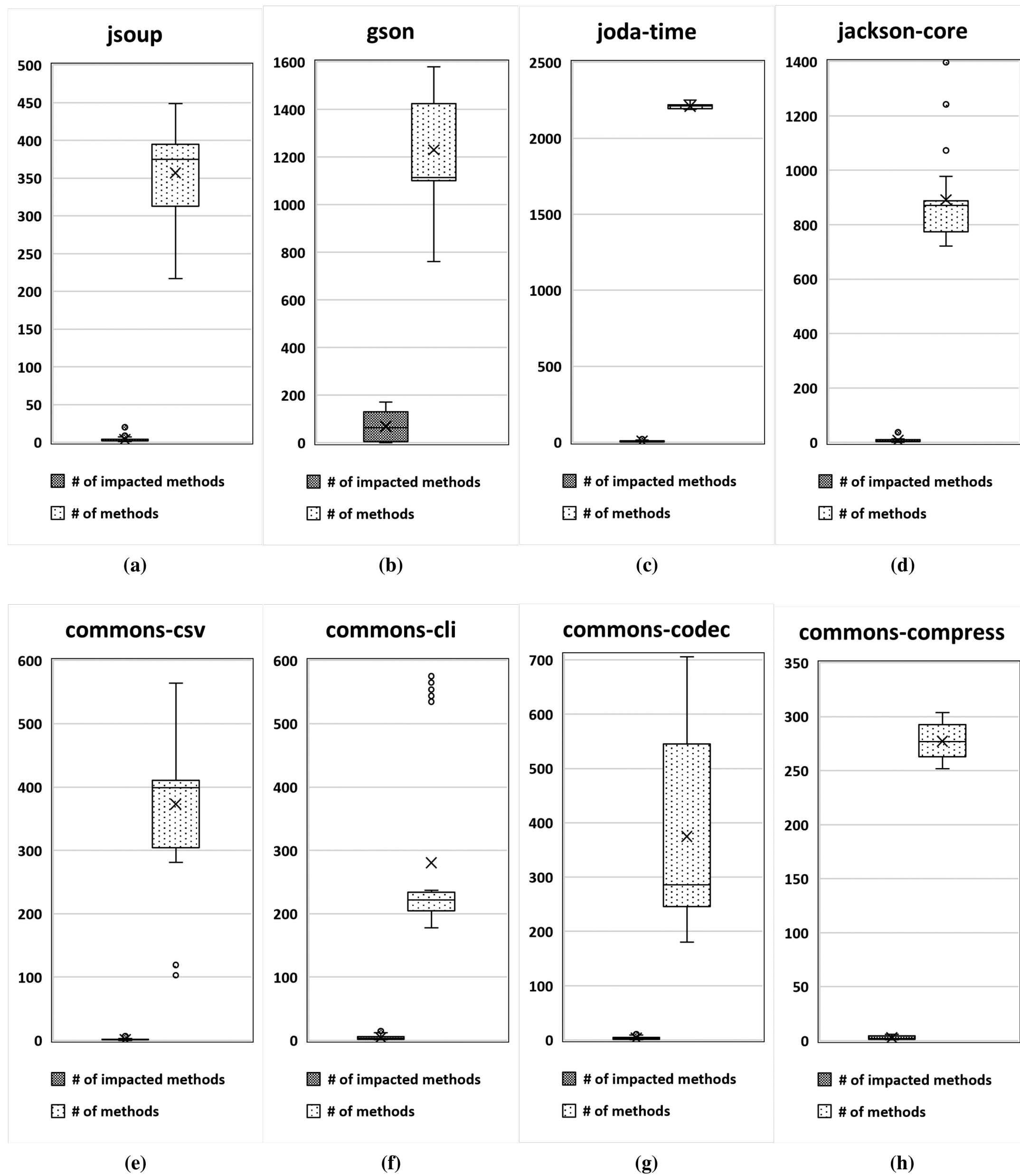

Fig. 12. Box plots of change impact analysis results are represented in the number of impacted methods and the total existing methods in the analyzed project. (a) Jsoup. (b) Gson. (c) Joda-time. (d) Jackson-core. (e) Commons-csv. (f) Commons-cli. (g) Commons-codec. (h) Commons-compress.

Our change impact results are given in Table VII and Fig. 12. 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 [11], [28] has shown that using the threshold 0.1 provides higher f-measure and recall results. To calculate the impact rate, we divide the impacted methods by the total methods. According to our change impact analysis results, the project *gson* has the maximum impact rate with 5%, and the project *joda-time* has the lowest impact rate with 0.4%. The results show that the bug fix changes do not seem to have a high impact on other software components. However, to have a better understanding of the relationship between change types and impact, in Section IV-C, we performed a statistical correlation analysis.

TABLE VIII
RUN-TIME ANALYSIS FOR THE CHANGE IMPACT ANALYSIS PER EACH BUG FIX

Project Name	Average run-time per bug fix (sec)
commons-cli	32.983
commons-codec	25.746
commons-compress	24.758
commons-csv	15.440
gson	30.812
jackson-core	28.675
joda-time	185.320
jsoup	35.145

In Table VIII, we present the run-time evaluation for the change impact analysis. The change impact analysis run-time

Authorized licensed use limited to: University of Missouri Libraries. Downloaded on May 11,2022 at 17:56:11 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

TABLE IX
CORRELATION RESULTS BETWEEN CAUSED IMPACT AND CHANGE TYPES

	Imp.	CM	AM	AF	AC	DM	APC	CPM	DPC	CNPM	CRM	CTF	DC	DF	IAM	AIS	AFS	AWS	MCA	MCD	DIS	DFS	DWS
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								
AIS	-0.035	0.266	0.080	0.172	0.132	0.204	-0.047	0.138	-0.047	-1.130	0.039	0.079	0.092	0.132	-0.067	1							
AFS	0.050	0.074	0.090	0.369	-0.070	0.152	-0.013	0.191	-0.013	-0.019	0.118	0.175	-0.013	0.442	-0.019	0.185	1						
AWS	-0.035	0.065	0.079	0.264	0.182	0.023	-0.012	0.226	-0.012	0.256	0.079	-0.020	-0.012	0.137	-0.016	0.139	0.226	1					
MCA	-0.704	0.602	0.496	0.121	0.078	0.076	0.015	0.045	0.015	0.021	0.088	0.026	0.015	0.065	0.021	0.160	0.045	0.039	1				
MCD	-0.704	0.602	0.496	0.121	0.078	0.076	0.015	0.045	0.015	0.021	0.088	0.026	0.015	0.065	0.021	0.160	0.045	0.039	1	1			
DIS	0.012	0.172	-0.003	0.098	-0.052	0.177	-0.030	0.024	-0.030	0.079	-0.016	0.147	-0.030	0.119	0.079	0.526	0.082	-0.016	0.103	0.103	1		
DFS	0.061	0.070	0.085	0.347	-0.066	0.091	-0.012	0.207	-0.012	-0.018	0.134	0.187	-0.012	0.652	-0.018	0.113	0.696	0.243	0.042	0.042	0.036	1	
DWS	-0.013	0.070	0.085	0.291	0.011	-0.064	-0.012	0.084	-0.012	0.238	0.065	-0.022	-0.012	0.033	-0.018	-0.036	0.084	0.519	0.042	0.042	0.098	0.094	1

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

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 on 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 (2), $\text{cov}(X, Y)$ is the covariance of random variable pairs (X, Y), while σ_x and σ_y are the standard deviation for X and Y, respectively. Based on the value obtained from (2), the value of 1 represents a perfect positive relationship, -1 is a perfect negative relationship, and 0 indicates the absence of a linear relationship between variables.

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

We calculate the *sample Pearson correlation coefficient* (r_{xy}), using (3), which uses the estimates of the covariances and variances based on a sample size of n in (2). It is important to mention that the *Pearson correlation coefficient* works on numerical samples, and since we only had 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 and absent-0). Finally, the calculated correlation coefficients are given in Table IX, and in this table, we only included the change types that were only detected in bug fixes.

$$r_{xy} = \frac{\sum_{i=1}^{n}(x_i - \overline{x})(y_i - \overline{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \overline{x})^2}\sqrt{\sum_{i=1}^{n}(y_i - \overline{y})^2}}. \quad (3)$$

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

$$\text{if } |r| \geq \frac{2}{\sqrt{n}}, \text{ then relationship exists.} \quad (4)$$

According to the results in Table IX, we only see that there are two change types that have a significant relationship with the caused impact on 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 caused impact on 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 IX, 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* and *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. Our correlation analysis has also shown that in the change types, there is a significant positive correlation between *CM* and the change types *MCA* and *MCD*. This correlation confirms that adding and deleting method calls occur with the changes made in the method body (*CM*). We can also mention that the method invocations mostly occur within the method body. On the other hand, we can see that there is no significant correlation between *CM* and the change types *AWS*, *DWS*, *AFS*,

Authorized licensed use limited to: University of Missouri Libraries. Downloaded on May 11,2022 at 17:56:11 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

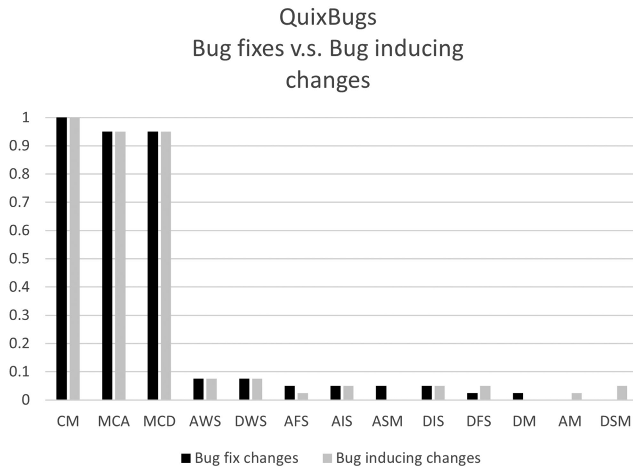

Fig. 13. Comparison of bug fixes and bug-inducing changes.

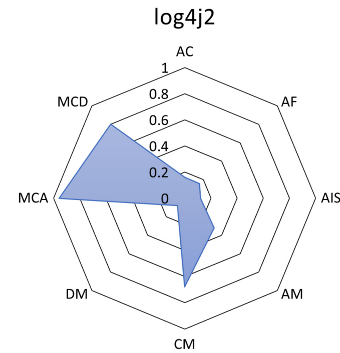

Fig. 14. Change types per file for the Log4J2 project CVE-2021-45105 vulnerability fix.

DFS, and *DIS*, which are expected to occur within the method body. However, it is important to recall that our case study has emphasized change types related to bug fixes, which is a special case of changes. For instance, we do see a significant correlation between *CM* and *adding an* IF*-statement (AIS)*, which is not surprising. Defects generally occur when there is an unhandled exception, or caused by an unexpected event. Once these defects are identified, the codes are modified to handle the unexpected exception with an IF-statement that covers the condition for handling the unexpected event.

D. Analyzing Bug Fix and Bug-Inducing Changes Using the QuixBugs Dataset

Our change-type analysis was based on bug fix changes, due to the difficulty of finding the bug-inducing change source and commits. Therefore, we used the QuixBugs [10] benchmark and dataset, which are commonly used for automatic program repair [29]–[32]. The dataset includes 40 Java programs and 40 Python programs. For each program, there are a defective version and a corrected version. This gives us the flexibility to analyze change types bidirectionally, i.e., analyzing the change types based on the fixed defects (defect version changed to corrected version), and analyzing the change types based on bug-inducing changes (corrected version changed to defect version).

In Fig. 13, we show the change types for bug fix changes (left-hand side columns) and bug-inducing changes (right-hand side columns) that we found. Our results show that the change types that exist for both are almost the same. However, we also observed opposite types of change actions between the bug-fixing and bug-inducing changes. For instance, we see two *add static modifier* (ASM to a method) change types while fixing bugs, and we also see two *delete static modifier* (DSM from a method) change types while inducing bugs. Similarly, other opposite changes coexist as well, such as DM and AM, AFS and DFS, AIS and DIS, and AWS and DWS. On the other hand, we see three change types that are dominant in both bug fixing and bug inducing: 1) *CM*; 2) *MCA*; and 3) *MCD*. The change-type *CM* existed in every change, while MCA and MCD occurred in 38 out of 40 programs. However, this gives us the motivation to focus on the details of *CM* types of changes.

Even though we are aware that the defects and fixes are synthetic in QuixBugs, it still is a good indicator in which granularity-level defects could be introduced and the bugs could be fixed. Furthermore, it still supports previous studies [2] that defects are introduced in the method body, which was found in all 40 Java programs in every file.

E. Change-Type Analysis of Log4J2 Vulnerability Fixes

In December 2021, the vulnerabilities (*CVE-2021-44228*,[6] *CVE-2021-45046*,[7] and *CVE-2021-45105*[8]) that were found in Log4J2 have affected a lot of software, and the severity of these vulnerabilities were classified between *medium* and *critical*. Therefore, we wanted to investigate that what types of changes are made to fix these vulnerabilities. We were able to find the commit [33] where the CVE-2021-45105 vulnerability was fixed and analyzed the changes that were made.

According to the common vulnerabilities and exposures (CVEs) [34], the *CVE-2021-45105* is a vulnerability that did not protect from uncontrolled recursion from self-referential look-ups, which allowed the attackers to control the thread context map data to perform a denial-of-service attack when the forged string input is interpreted. This vulnerability was enumerated as an *improper input validation* (CWE-20[9]) and an *uncontrolled recursion* (CWE-674[10]) weakness.

The fix for the *CVE-2021-45105* vulnerability included 35 file changes, of which 25 of them were Java source files. Among our defined change rules, we only found eight distinct types of changes, which are shown in Fig. 14. Our findings show that the most common changes are *MCA* (adding a method invocation), *MCD* (deleting a method invocation), and *CM* (changing method body). The changes *MCD* and *MCA* are actions of replacing a vulnerable method with a secure method. We also

[6][Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2021-44228
[7][Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2021-45046
[8][Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2021-45105
[9][Online]. Available: https://cwe.mitre.org/data/definitions/20.html
[10][Online]. Available: https://cwe.mitre.org/data/definitions/674.html

Authorized licensed use limited to: University of Missouri Libraries. Downloaded on May 11,2022 at 17:56:11 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

see a significant number of *AM* (adding a method declaration) changes, which are newly implemented unit tests. In addition, against input validation vulnerabilities (CWE-20), it is very common to see change actions, such as *AIS*, to mitigate input vulnerabilities [35], which we have also found in fixing the *CVE-2021-45105* vulnerability for Log4J2.

V. Discussion

A. Answering Research Questions

1) RQ1: What are the Most Common Bug Fix Change Patterns?: To answer *RQ1*, we performed a change-type analysis on 40 Java programs with versions of defects and bug fixes, 13 Java projects, and 621 bug fix commits. We have observed that there are four commonly performed changes: a) *MCA - adding a method invocation*; b) *MCD - deleting a method invocation*; c) *CM - change in the method body*; and d) *AM - adding a new method declaration*. Our observations indicated that the change-type *MCA* was present in every bug fix commit. For the change-type *MCD*, the maximum change-type presence per commit was found 1 and the minimum was found 0.99. For change-type *CM*, the maximum change-type per commit is 1, while the minimum is 0.5. On the other hand, the maximum change-type per commit for *AM* is also 1, while its minimum is 0.73.

We have also observed that among our 43 defined change types, only 22 of them were found in the case study. The changes that were not found were mostly related to class- and field-based changes. Most of the changes are mostly made in the method-level and statement-level granularities. It is important to mention that the results represented in this article only refer to the bug fix patterns we have defined. Therefore, there could be other change patterns that are not addressed in this article.

Nevertheless, while answering research question *RQ1*, it is critical to highlight that we are able to label change types, and our tool can be extended with new or additional change types. Labeling changes can be used for training deep neural networks for detecting code updates, bug fixes, and malicious codes.

2) RQ2: Is There a Relationship Between the Bug Fix Change Patterns and the Impact Caused by Change?: To answer *RQ2*, we have followed the following two steps.

a) The first step was to find the impacted software components and calculate the impact rate.
b) 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*, *MCA*, *MCD*, 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 *MCA*, *MCD*, *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.

We included the QuixBugs dataset to detect the types of changes introduced while inducing a bug, or fixing a bug. The QuixBugs dataset does not contain large scale programs; therefore, they may not be realistic bug fix changes. However, QuixBugs has been used extensively in automatic program repair [10], [29], [30], [36] to learn the characteristics of bug fix changes. With the QuixBugs dataset and benchmark, these previous studies were able to achieve successful automatic repairing results. This indicates that, even though the bug fix changes are synthetic, they still do represent actual bug fix changes.

For the 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 than it is supposed to be. However, the change impact analysis we have used [11] 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 article are slightly high. We can conclude that the impact caused by bug fixes is actually very low.

For generating a parse tree, we have used ANTLR, and the grammar that we used for the change-type analysis is designed for the 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.

The statement-level change rules are currently limited. We have statement-level change rules for adding/deleting IF-statements, FOR-statements, and WHILE-statements. However, we do not have change rules for try-catch blocks or switch cases, which are not included in our analysis and might miss some valuable information.

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 [37]. Later, a taxonomy of source code changes was built, which defined source code changes related to tree edit operations in AST and classified each change type [13]. Then, Fluri *et al.* [14] 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.

Authorized licensed use limited to: University of Missouri Libraries. Downloaded on May 11,2022 at 17:56:11 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

Lin *et al.* [38] implemented an automatic tool called PyCT, which is based on ChangeDistiller to reduce the effort for change extraction and classification. They have also introduced 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 at a method-level on JavaScript programs, by using the change types definitions from [39]. 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 condition-related bugs. Roy *et al.* [7] proposed a model for detecting readability improvements and used static analysis and change-type analysis tools (e.g., Coming [15] and ChangeDistiller [14]) 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 article, we investigated the relationship between bug fix changes and the impacts of these changes caused in the software. To investigate this relationship, we have proposed and publicly shared an automatic change detection tool called CIJ. We analyzed the bug fix change types from Defects4J with CIJ. We found that there are four common changes that are made in bug fixes: 1) changing the method body; 2) adding a new method declaration; 3) deleting; and 4) adding a method call. Then, we performed a change impact analysis on the bug fix changes to analyze their impact on the software. Among the projects we analyzed, the caused impact in the software ranged between 0.4%–5%, which indicated a very low impact. However, we wanted to investigate deeper to find if any of the changes have a higher or lower impact. Therefore, we performed a statistical analysis using the Pearson correlation coefficient to find any correlation between change types and the caused impact. We have found that among 22 change types, there were only five change types that had a significant correlation between the caused impacts. The change types, adding a new method declaration, changing the method body, adding a method call, and deleting a method call 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 was able to distinguish that there was no change performed, which could reduce the code reviewing process when insufficient commit messages are provided.

Furthermore, based on our case study results and observations, the project commits we analyzed did not include any changes that would affect the design, but progressed and evolved with local changes. Therefore, we have not found high impacts in the software. Most of the changes we found were at the method level and statement level, which are less likely to cause an impact. Accordingly, we may categorize commits into three types: 1) bug-fix; 2) change request; and 3) refactoring. If the changes in a commit are made with refactoring intentions, the change types are also expected to be small structural and local changes, which are again less likely to cause an impact. However, to distinguish the difference between refactoring and a bug fix commit, we require information on *change intentions* and code quality measurements. Overall, to facilitate the practical usage of our work and results, further analysis is required, which is caused by inadequately and insufficiently written unit tests, and nonexplanatory commit messages.

In future work, we plan to extend our change detection tool by including more comprehensive statement-level change types, such as assignments, arithmetic operations, etc. So far, we only detect adding/deleting IF-statements, WHILE-statements, FOR-statements, and method calls. Since our results found that changing the method body is common in bug fixes, we found that the adding/deleting method calls are very common. However, the presence of change types, such as assignments and arithmetic operations, is still unknown to us. Therefore, we want to investigate the characteristics of statement-level changes and their impact caused in the software. Finally, we plan to combine and support our change types with commit messages and code quality metrics for extracting the intentions of the changes.

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 *Proc. Int. Symp. Softw. Testing Anal.*, 2014, pp. 437–440.

[2] X. Ren, O. C. Chesley, and B. G. Ryder, "Identifying failure causes in Java programs: An application of change impact analysis," *IEEE Trans. Softw. Eng.*, vol. 32, no. 9, pp. 718–732, Sep. 2006.

[3] X. Sun, B. Li, C. Tao, W. Wen, and S. Zhang, "Change impact analysis based on a taxonomy of change types," in *Proc. IEEE 34th Annu. Comput. Softw. Appl. Conf.*, 2010, pp. 373–382.

[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 *Proc. 33rd ACM/IEEE Int. Conf. Automated Softw. Eng.*, 2018, pp. 832–837.

[5] M. Tufano, J. Pantiuchina, C. Watson, G. Bavota, and D. Poshyvanyk, "On learning meaningful code changes via neural machine translation," in *Proc. 41st Int. Conf. Softw. Eng.*, 2019, pp. 25–36.

[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 *Proc. IEEE 25th Int. Conf. Softw. Anal., Evol. Reengineering*, 2018, pp. 130–140.

[7] D. Roy, S. Fakhoury, J. Lee, and V. Arnaoudova, "A model to detect readability improvements in incremental changes," in *Proc. 28th Int. Conf. Prog. Comprehension*, 2020, pp. 25–36.

[8] A. Szatmári, B. Vancsics, and Á. Beszédes, "Do bug-fix types affect spectrum-based fault localization algorithms' efficiency?," in *Proc. IEEE Workshop Validation, Anal. Evol. Softw. Tests*, 2020, pp. 16–23.

[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 *Proc. ieee 27th Int. Conf. Softw. Anal., Evol. Reengineering*, 2020, pp. 308–319.

Authorized licensed use limited to: University of Missouri Libraries. Downloaded on May 11,2022 at 17:56:11 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

[10] D. Lin, J. Koppel, A. Chen, and A. Solar-Lezama, "QuixBugs: A multi-lingual program repair benchmark set based on the Quixey challenge," in *Proc. ACM SIGPLAN Int. Conf. Syst., Program., Lang., Appl.: Softw. Humanity*, 2017, pp. 55–56.

[11] E. Ufuktepe and T. Tuglular, "Code change sniffer: Predicting future code changes with Markov chain," in *Proc. IEEE 45th Annu. Int. Comput. Softw. Appl. Conf.*, 2021, pp. 1014–1019.

[12] E. Ufuktepe, T. Tuglular, and K. Palaniappan, "The relation between bug fix change patterns and change impact analysis," in *Proc. IEEE 21st Int. Conf. Softw. Qual., Rel. Secur.*, 2021, pp. 1089–1099.

[13] B. Fluri and H. C. Gall, "Classifying change types for qualifying change couplings," in *Proc. 14th IEEE Int. Conf. Prog. Comprehension*, 2006, pp. 35–45.

[14] B. Fluri, M. Wursch, M. Pinzger, and H. Gall, "Change distilling: Tree differencing for fine-grained source code change extraction," *IEEE Trans. Softw. Eng.*, vol. 33, no. 11, pp. 725–743, Nov. 2007.

[15] M. Martinez and M. Monperrus, "Coming: A tool for mining change pattern instances from git commits," in *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng.: Companion*, 2019, pp. 79–82.

[16] D. Spadini, M. Aniche, and A. Bacchelli, "Pydriller: Python framework for mining software repositories," in *Proc. ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2018, pp. 908–911.

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

[18] N. F. Schneidewind, "The state of software maintenance," *IEEE Trans. Softw. Eng.*, vol. SE-13, no. 3, pp. 303–310, Mar. 1987.

[19] M. M. Lehman and L. A. Belady, *Program Evolution: Processes of Software Change*. New York, NY, USA: Academic Press, 1985.

[20] 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 *Proc. IEEE 18th Int. Conf. Softw. Eng.*, 1996, pp. 464–474.

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

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

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

[24] A. Kumar, "Development at the speed and scale of Google," QCon San Francisco, 2010. [Online]. Available: https://qconsf.com/sf2010/dl/qcon-sanfran-2010/slides/AshishKumar_DevelopingProductsattheSpeedandScaleofGoogle.pdf

[25] J. A. Duraes and H. S. Madeira, "Emulation of software faults: A field data study and a practical approach," *IEEE Trans. Softw. Eng.*, vol. 32, no. 11, pp. 849–867, Nov. 2006.

[26] M. Martinez, L. Duchien, and M. Monperrus, "Automatically extracting instances of code change patterns with AST analysis," in *Proc. IEEE Int. Conf. Softw. Maintenance*, 2013, pp. 388–391.

[27] T. Parr, *The Definitive ANTLR 4 Reference*. Raleigh, NC, USA: Pragmatic Bookshelf, 2013.

[28] E. Ufuktepe and T. Tuglular, "A program slicing-based Bayesian network model for change impact analysis," in *Proc. IEEE Int. Conf. Softw. Qual., Rel. Secur.*, 2018, pp. 490–499.

[29] H. Ye, M. Martinez, T. Durieux, and M. Monperrus, "A comprehensive study of automatic program repair on the QuixBugs benchmark," *J. Syst. Softw.*, vol. 171, 2021, Art. no. 110825.

[30] M. Asad, K. K. Ganguly, and K. Sakib, "Impact of similarity on repairing small programs: A case study on QuixBugs benchmark," in *Proc. IEEE/ACM 42nd Int. Conf. Softw. Eng. Workshops*, 2020, pp. 21–22.

[31] Z. Bian, A. Blot, and J. Petke, "Refining fitness functions for search-based program repair," in *Proc. IEEE/ACM Int. Workshop Automated Prog. Repair*, 2021, pp. 1–8.

[32] J. A. Prenner and R. Robbes, "Automatic program repair with openAI's codex: Evaluating quixbugs," 2021, *arXiv:2111.03922*.

[33] Log4J2 CVE-2021-45105 vulnerability fix. [Online]. Available: https://github.com/apache/logging-log4j2/commit/806023265f8c905b2dd1d81fd2458f64b2ea0b5e

[34] Log4J2 CVE-2021-45105 vulnerability description. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-45105

[35] E. Ufuktepe and T. Tuglular, "Estimating software robustness in relation to input validation vulnerabilities using Bayesian networks," *Softw. Qual. J.*, vol. 26, no. 2, pp. 455–489, 2018.

[36] Y. Yang, T. He, Y. Feng, S. Liu, and B. Xu, "Mining Python fix patterns via analyzing fine-grained source code changes," *Empirical Softw. Eng.*, vol. 27, no. 2, pp. 1–37, 2022.

[37] B. Fluri, H. C. Gall, and M. Pinzger, "Fine-grained analysis of change couplings," in *Proc. IEEE Int. Workshop Source Code Anal. Manipulation*, 2005, pp. 66–74.

[38] 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 *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, 2016, pp. 188–199.

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

Authorized licensed use limited to: University of Missouri Libraries. Downloaded on May 11,2022 at 17:56:11 UTC from IEEE Xplore. Restrictions apply.