Automatic Recovery of Root Causes from Bug-Fixing Changes

Ferdian Thung, David Lo, and Lingxiao Jiang
School of Information Systems
Singapore Management University, Singapore
{ferdiant.2013,davidlo,lxjiang}@smu.edu.sg

***Abstract*—What is the root cause of this failure? This question is often among the first few asked by software debuggers when they try to address issues raised by a bug report. Root cause is the erroneous lines of code that cause a chain of erroneous program states eventually leading to the failure. Bug tracking and source control systems only record the symptoms (e.g., bug reports) and treatments of a bug (e.g., committed changes that fix the bug), but *not its root cause*. Many treatments contain non-essential changes, which are intermingled with root causes. Reverse engineering the root cause of a bug can help to understand why the bug is introduced and help to detect and prevent other bugs of similar causes. The recovered root causes are also better ground truth for bug detection and localization studies.**

In this work, we propose a combination of machine learning and code analysis techniques to identify root causes from the changes made to fix bugs. We evaluate the effectiveness of our approach based on a golden set (i.e., ground truth data) of manually recovered root causes of 200 bug reports from three open source projects. Our approach is able to achieve a precision, recall, and F-measure (i.e., the harmonic mean of precision and recall) of 76.42%, 71.88%, and 74.08% respectively. Compared with the work by Kawrykow and Robillard, our approach achieves a 60.83% improvement in F-measure.

I. Introduction

There is abundant information available about many bugs reported, recorded, and managed in bug tracking systems like Bugzilla or JIRA. Many of these bugs could also be linked to the corresponding bug-fixing revisions in the corresponding source control systems [9], [51]. These two kinds of systems provide two different views of a bug: a bug tracking system may provide textual descriptions of the *symptoms* of the bug, how to reproduce the bug, and possible failures caused by the bug; a source control system provides the changes made to a software system that aim to fix the bug (i.e., the *treatments* for the bug). However, they do not directly answer the question: What program elements are the *root cause* of the bug? In other words, what are the erroneous program elements responsible for the bug (i.e., their execution collectively causes a chain of erroneous program states eventually leading to the program failure or error behavior)?

Answers to this root cause question is important as it highlights mistakes that developers make and can help to perform various kinds of postmortem analyses and motivate the design and development of better mechanisms (e.g., better language constructs and features [17], [24], automated bug predictors and detectors [39], [46], [52], [53]) to reduce bugs of similar causes to happen in future.

Our work in this paper aims to *automatically* identify *root causes* of a bug based on information available from bug treatments. Our long-term goal is to build a curated database containing root causes of a large number of bugs that can be used by fellow researchers to study bug patterns and build better bug prediction, localization, detection, and prevention mechanisms.

There are both conceptual and technical challenges for identifying root causes. Firstly, symptoms of a bug may not reveal a root cause. For example, when a program fails with a "division by zero" exception, it does not tell where the zero comes from; a debugger may need to back track the data flow to find the source of the zero. Secondly, treatments of a bug (i.e., changes made to fix the bug) may often contain not only treatments for bugs but also non-essential changes (e.g., code reformatting, code refactoring, etc.) [25], new feature additions, many of which are intermingled. The treatments may add extra code which does not reveal the location of root causes. For example, "division by zero" exceptions can be fixed by adding an exception handler spatially far away from either the failure location or the root cause.

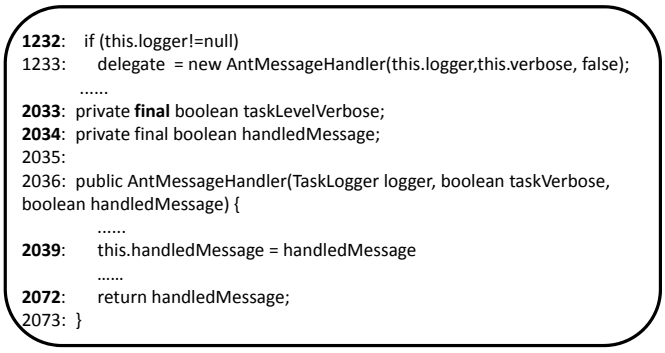
There are various techniques (e.g., fault localization, bug prediction, etc. [2], [23], [28], [33], [40], [49]) have been developed to locate root causes from various failure symptoms with various degrees of success. Different from these techniques, we perform a retrospective analysis to recover root causes *when a bug fix is already there*. Some work has been done [27], [41], [42] to identify bug origins, i.e., versions that induce bug fixes. Their studies mostly focus on faulty versions and all lines changed or deleted (ignoring comment, blank line, and format changes) are implicitly assume to be root causes, which may not be always true. To the best of our knowledge, DiffCat proposed by Kawrykow and Robillard [25], which flags non-essential changes, may be the work having the closest goal as ours. As defined by Kawrykow and Robillard, non-essential changes are those "cosmetic in nature", "generally behavior preserving", and "unlikely to yield future insights into the roles of or relationships between the program entities they modify".

Our work automatically flags likely root causes in code from bug-fixing (file) changes, which are the set of files that get modified between the version of code containing a bug and a new version where the bug has been fixed. Such an automation is feasible because we have previously *manually*

978-1-4799-2931-3/13 © 2013 IEEE

WCRE 2013, Koblenz, Germany

(a) Sample Bug Before Fixes



(b) Sample Bug With Fixes

Fig. 1. A Sample Bug and Its Treatments. Lines 1233 and 2070 in Fig. 1(a) are the root causes, while the treatments in Fig. 1(b) are lines 1232, 2033, 2034, 2039, and 2072.

studied several hundreds of bug fixes and observed that many fixes themselves actually hint at possible locations of the root causes (e.g., the fixed program elements are located close to the root causes). After excluding a small percentage of ambiguous cases, we have obtained a golden set of root causes of 200 bugs from three medium to large open source programs [34], [46], and we believe based on patterns of the fixes, we can *automate* the process of identifying root causes.

Our solution works in mainly two phases. Firstly, lines of code that are part of essential changes are identified using a combination of DiffCat [25] and Unix *diff*. Secondly, we combines machine learning and code analysis techniques to identify root causes. We employ a machine learning solution, in particular a classifier, to predict if a line of code is a root cause and filter out those lines that are unlikely to be root causes. To train such a classifier, we extract relevant features from code changes that could discriminate root causes from others. Further, we perform a light-weight case-based code analysis to recover root causes that do not appear in the treatments but may still be responsible for the bug.

We have evaluated the effectiveness of our approach on the golden set of root causes [34], [46] manually extracted from the bug tracking and source control systems of AspectJ [5], Lucene [3], and Rhino [1]. In total, we have analyzed 152, 28, and 20 bug reports and their fixing changes from AspectJ, Lucene, and Rhino respectively. We compare our approach with DiffCat. Our results show that our approach achieves a precision, recall, and F-measure of 76.42%, 71.88%, and 74.08%. In comparison, DiffCat achieves a much lower precision (33.30%) but a little higher recall (74.65%). In terms of F-measure that quantifies if a gain in precision (or recall) outweighs a loss in recall (or precision), the results of our approach are 60.83% better than those of DiffCat.

Our contributions are as follows:

1) We propose a novel technique that automatically recover root causes from bug fixing changes: it combines machine learning with code analysis to recover root causes.
2) We have evaluated our approach on a golden set of 200 root causes and find that our approach achieves much better F-measure than DiffCat.

The structure of this paper is as follows. In Section II, we present a motivating example. In Section III, we describe some preliminary materials on non-essential changes, and further elaborate the difference of root causes and their treatments. We present an overview of our approach in Section IV. We explain the major parts of our approach in Sections V, and VI. We present the results of our experiments in Section VII. We discuss related studies in Section VIII. We conclude and mention future work in Section IX.

II. Motivating Example

Figure 1 shows an example illustrating the need of root cause identification. It is adapted, for the purpose of presentation in this paper, from a real bug in AspectJ from iBugs repository [13]. The bug in Figure 1(a) is fixed in Figure 1(a) by changing the lines with bold line numbers, and the treatments for the bug are Lines 1232, 2033, 2034, 2039, and 2072 in Figure 1(b).

One of the root causes is Line 1233 in Figure 1(a) where there may be a null-pointer dereference. The fix in Figure 1(b) adds a null-check at Line 1232, while the line of the root cause is *not* changed. This illustrates that treatments may not be root causes.

Another root cause in Figure 1(a) is at Line 2070 where a wrong return value is used. The fix in Figure 1(b) changes the root cause line, and adds additional code at Lines 2034 and 2039 to record one of the function parameters, which is needed for another part of the program but irrelevant for fixing the root cause. Also, the fix adds the `final` qualifier to the private variable `taskLevelVerbose`, which does not affect the functionality of the code and thus is a non-essential change. Thus, it is a challenge to precisely identify the root causes (e.g., Lines 1233 and 2070 in this example) based on the treatments for bugs.

III. Preliminaries

In this section, we briefly present an introduction to DiffCat, proposed by Kawrykow and Robillard, which detects and removes non-essential changes [25]. Next, we highlight the difference between essential changes, which are the *treatments* of a bug, and the root causes of the bug.

A. DiffCat: Detecting Non-Essential Changes

Many changes committed to source control systems are not essential. Many changes are cosmetic in nature; for example, developers could decide to add new comments or fix the indentations to various source code files in a software system. Other changes are meant to refactor a software system to make it more manageable. These changes do not alter the behavior of the system. These changes is referred to by Kawyrkow and Robillard as non-essential changes [25].

Kawyrkow and Robillard proposed DiffCat which is the state-of-the-art technique that detects non-essential changes from revision histories [25]. DiffCat first generates ASTs from changed files. These ASTs are then enriched with type information that is inferred using partial program analysis (PPA) [12]. The resulting pairs of ASTs (for each pair, one AST is for a file before the change, another is for the file after the change) are then input to ChangeDistiller [14] which performs AST differencing. The output of ChangeDistiller is used to identify *name refactorings* (i.e., cosmetic changes in variable names). These name refactorings are then *roll backed* and the resulting AST pairs are again processed by ChangeDistiller. The output of ChangeDistiller is structural differences between the AST pairs. Each of these differences is compared to a manually constructed catalog of non-essential differences (e.g., trivial type updates, trivial keyword modifications, etc.) and non-essential changes are then identified. DiffCat works on AST node level and it returns AST node changes that are marked as essential or non-essential. It also reports the type of the changes. There are many types that could be reported, such as `ASSIGNMENT_UPDATE`, `IF_STATEMENT_UPDATE`, `WHILE_STATEMENT_MOVE`, `ATTRIBUTE_DELETE`, `FUNCTIONALITY_DELETE`, etc.

In this work, we leverage DiffCat to detect root causes from bug-fixing file changes.

B. Essential Changes versus Root Causes

A non-essential change cannot be a root cause. On the other hand, an essential change is not necessarily a root cause. The relationship between essential changes and root causes is illustrated in Figure 2. The figure contains two circles which represent essential changes (the left circle) and root causes (the right circle).

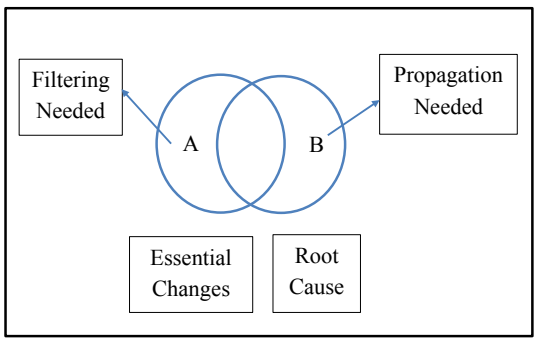


Fig. 2. Essential Changes versus Root Causes

Some essential changes are root causes, as indicated by the overlap of the two circles. However, many essential changes are not root causes (e.g., Lines 1232, 2034 and 2039 in Figure 1(b)). Some lines of code that are part of essential changes are the treatments of a bug but not the root causes of the bug (e.g., Line 1232 in Figure 1(b)). Other changes, although essential (i.e., may change the behavior of a program), may just coincidentally happen to tangle with the changes that are really necessary for fixing the bug [18] (e.g., Lines 2034 and 2039 in Figure 1(b)). These lines of code need to be *filtered*. Conceptually, such code corresponds to the crescent region marked by A in Figure 2.

On the other hand, not all root causes may appear in essential changes. A root cause might remain unmodified from the buggy version to the bug-free version (e.g., Line 1233 in Figure 1(b)). Some bugs can be treated or fixed by adding code (e.g., adding checks for null, adding a resource deallocation statement, adding the missing `unlock` for the `lock`, etc.); the root causes of the bugs are not changed during the bug-fixing process. These cases correspond to the crescent region marked by B in Figure 2. For detecting such root causes, we need to *propagate* our code analysis from essential changes in the treatments to their relevant surrounding code and contexts.

IV. Overall Approach

The overall structure of our approach is shown in Figure 3. The approach processes bug-fixing file changes one at a time; each of them is a file that is modified between the version before the change and the one after the change. There are three main blocks in our root cause extraction approach: essential change extraction, filtering component, and propagation component. We describe each of these three components in the following.

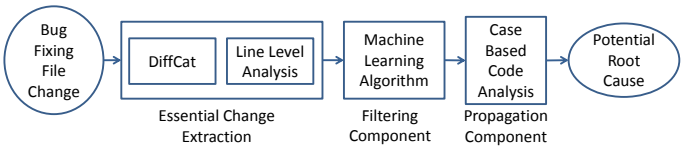


Fig. 3. Overall Structure of Our Root Cause Extraction Approach

A. Essential Change Extraction

In this block, we extract essential changes from a bug-fixing file change. Non-essential changes correspond to cosmetic changes or refactoring, and need to be removed as they cannot be the root causes of the bug. There are two sub-blocks in this block: DiffCat and Line Level Analysis.

DiffCat works at the level of AST nodes – it returns AST node changes and label them as essential or non-essential. Each AST node change contains AST nodes from the version of a program before the change and that after the change. Root causes of a bug must exist in the program before the change. Thus, after employing DiffCat, for every AST node change that is marked as essential, we only retain AST nodes from the version before the change.

At times the returned AST node (e.g., method declaration) could be quite big (i.e., it spans many lines of code) and not every line in a modified AST node is modified. Thus, we perform a line level analysis to remove lines that are not modified. We want to extract the essential lines of code, in the version before the change, that get modified. To do this, we perform the following steps:

1) Get the lines of code covered by the AST nodes reported by DiffCat in the program version before the change.
2) Perform a standard UNIX diff, between the versions before and after the change, to get the lines of code in the version before the change which get modified.
3) Intersect the two sets of lines produced by the above two steps.
4) Return the resulting set of lines.

B. Filtering

In the filtering component, we remove source code lines that are part of an essential change, but unlikely part of a root cause of a bug. The filtering component takes as input the lines that are produced by the essential change extraction component, and utilizes a machine learning algorithm to predict whether a line of code is likely to be part of a root cause. The main idea is to build a classification model that would extract features from a source code line and differentiate root-cause lines from non-root-cause lines based on a set of training data. We describe this machine learning block in detail in Section V.

C. Propagation

The propagation component looks beyond the lines that are parts of essential changes. It propagates the effects of each essential change to relevant neighboring lines of code (e.g., other statements in the same block scope as the essential change), and performs a case-based code analysis based on data dependencies to detect more possible root causes of the bug. We describe the propagation component in more details in Section VI.

V. Filtering Component

Treatments for a bug can take various forms, and the challenge for the filter component is to decide whether lines of code in the treatments are unlikely to be root causes. We employ a machine learning-based solution to address the challenge. The solution learns from known examples (i.e., source code lines for which we know whether they are part of a root cause or not) and extrapolates to unknown examples.

The structure of our machine learning block is shown in Figure 4. It works in two phases: training and deployment. In the training phase, we train the classifier to produce a discriminative model that could differentiate lines that are part of a root cause of a bug from those that are not part of a root cause. In the deployment phase, we take the model and use it to classify if any given line is part of a root cause or not.

The training phase takes as input a set of training data which are lines of code with their labels (i.e., root cause or non-root-cause). This set of training data is then subject to

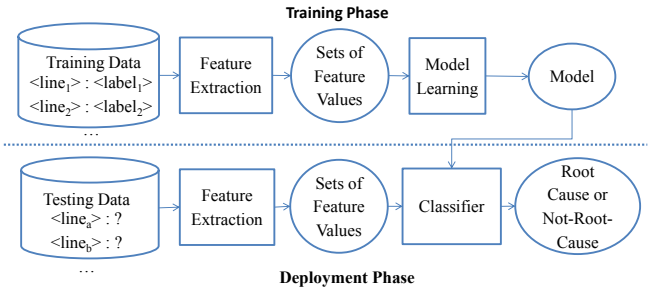


Fig. 4. Structure of Machine Learning Block

a feature extraction process. The feature extraction process would reduce each data point in the training data to some important facets or factors that characterize the data point. At the end of the feature extraction process we have a set of features along with their values that characterizes each data point. For example, if a feature is whether the line contains an if statement or not, at the end of the process, the line could be represented as either $\langle if, 0\rangle$ or $\langle if, 1\rangle$ depending if the line contains or does not contain an if statement respectively. We describe in more details the features that we extract from these lines in Section V-A. The set of features and their corresponding feature values for each of the training data is then fed to a model learning component. We describe the model learning component in more details in Section V-B. This model is then output to the deployment phase.

The deployment phase takes as input the testing data which comprises of a set of program lines with unknown labels (i.e., root cause or non-root-cause). Again we reduce each data point (i.e., each line) in the training data into a set of important factors by the feature extraction process. This process is the same process that we perform for each data point in the training data. The resultant set of features along with their values are fed to the model application step (cf. Section V-B). The output after this step is a label for each data point to identify if it is a root cause or not.

A. Feature Extraction

Many features could be extracted from a line of code. We could extract features from the context of a line. We refer to the former as line features, and the latter as context features.

Features extracted from a line itself include:

1) The type of essential changes it belongs to, which is outputted by DiffCat in our essential change extraction component.
2) The types of program elements contained in the line.

Features extracted from the context of a line include:

1) The type of the line immediately before the current line.
2) The type of the line immediately after the current line.

The full set of features used to characterize a line is shown in Table I. 140 of the features ($F_1 - F_{84}$ and $F_{253} - F_{308}$) are line features. 168 of the features ($F_{85} - F_{252}$) are context features. We find DiffCat change type features carry more useful semantics for root cause identification than AST node types. Thus we give more weight to DiffCat change type

features than to AST node type features. For AST node type features we assign it a binary value, either 0 (a node of that type does not exist in the line) or 1 (a node of that type exists in the line). For DiffCat change type features, we assign a value of either 0 (a line is not part of an AST node that is assigned that change type) or 10 (a line is part of an AST node is assigned that change type).

TABLE I
FEATURES USED TO REPRESENT EACH LINE OF CODE

ID	Type	Description
F_1–F_{84}	ASTNode	Each of the features denotes the existence or absence of an ASTNode type (e.g., assignment, for statement, method declaration, etc.) in the data point (i.e., current line).
F_{85}–F_{168}	ASTNode	Similar to $F_1 - F_{84}$ for the line immediately preceding the data point
F_{169}–F_{252}	ASTNode	Similar to $F_1 - F_{84}$ for the line immediately after the data point
F_{253}–F_{308}	DiffCatChange	Type of change assigned by DiffCat [25] to the AST node containing this data point (i.e., current line)

B. Model Learning and Application

The purpose of a model learning step is to build a model that could discriminate data points belonging to one label from data points belonging to another label. In our setting, we would like to differentiate root cause from non-root-cause lines of code. We use Support Vector Machine (SVM) [16] and one of its implementation (i.e., svm-perf [22]) to build our model. SVM has been shown to be effective in many past studies in software engineering, e.g., [19], [44], [45]. Each data point could be viewed as a point in a multi-dimensional space with each feature representing a dimension. A machine learning solution could then try to draw a line (or a hyperplane) that separates as many root cause data points from non-root-cause data points. A representation of this hyperplane is the model that is generated by SVM.

This model is then applied to classify whether a line with an unknown label is a root cause or not. In order to do this, SVM simply investigates if the feature representation of the line falls on one side of the hyperplane or the other side. SVM also outputs the probability of a particular line being a root cause or not based on the distance of the feature representation of the line to the hyperplane. If the feature representation of the line is far from the hyperplane, then the probability of it belonging to one class is much higher than that of it belonging to the other class.

VI. PROPAGATION COMPONENT

In the propagation component, the challenge is to discover possible root causes that do not appear in the treatments of a bug. Our goal is to propagate the effects of the lines flagged as essential changes to the lines surrounding these essential changes and locate lines that may affect or be affected by the essential changes as possible additional root causes. For this purpose, we perform a case-based code analysis based on data dependencies. We have formulated several cases for which we can identify root causes with high confidence based on empirical studies of known essential changes.

To identify these several cases, we follow a grounded theory approach [35]: We analyze the data and find cases that require propagation. We then group these cases together. Past studies in software engineering have also followed a similar approach, e.g., [6], [20], [37]. FindBugs [6], [7], JLint [4], PMD [11], and similar bug finding tools, analyze commonly found bugs and create a repository of rules to capture various kinds of bugs. Following a similar approach, we create several propagation rules. In this paper, we create three rules to capture three different cases. In the future, more rules could be added, just like new rules could be added to FindBugs and other similar bug finding tools.

A. Case 1: Null Check Not Performed

"Null check not performed" means a portion of the code is executed without checking whether or not a variable is null. An example of this case is shown in Figure 5. We see that the new version of the code adds a null check for `probs` at line 82 and covers a portion of the code at line 83-92 that also appears in the old version by indenting them further. This portion of the code, which should be not run when `probs` is null, is the root causes of the bug, and we label the line 82-91 in the old version as the root causes.

```
org.aspectj/modules/org.aspectj.ajdt.core/src/org/aspectj
/ajdt/internal/core/builder/EclipseSourceContext.java
81: IProblem[] probs = result.getProblems();
82: for (int i = 0; i < probs.length; i++) {
83:     IProblem problem = probs[i];
84:     if (problem == null) continue;
85:     if (problem.getID() == IProblem.UnusedMethod...
86:             || problem.getID() == IProblem.Unused...
87:         if (problem.getSourceLineNumber() == ...
88:             UnusedDeclaredThrownExceptionFilter filter =
89:                 new UnusedDeclaredThrownException...
90:             result.removeProblems(filter);
AFTER
81: IProblem[] probs = result.getProblems();
82: if (probs!=null) {
83:     for (int i = 0; i < probs.length; i++) {
84:         IProblem problem = probs[i];
85:         if (problem == null) continue;
86:         if (problem.getID() == IProblem.UnusedMethod...
87:                 || problem.getID() == IProblem.Unused...
88:             if (problem.getSourceLineNumber() == ...
89:                 UnusedDeclaredThrownExceptionFilter filter =
90:                     new UnusedDeclaredThrownException...
91:                 result.removeProblems(filter);
```

Fig. 5. Example Case 1 in AspectJ Bug #151845

We pinpoint the location where this case happens by first picking the changes that add a null checking condition in the code after fixes. For each of those changes, we check for the existence of the entire original portion of the code in its revised portion. If it exists, then it is indeed the case that a null check is not performed in the original code. We then label all lines in the original portion of code guarded by this condition as the root causes. We notice that sometimes it could be more accurate if we only identify lines which have data dependencies on the variables checked in the additional code (e.g., Lines 83 and 84 in Figure 5 directly use the variable `probs` which is checked at Line 82). We leave it as future extension to improve our current propagation rule. The pseudocode for this algorithm is in Figure 6.

```
1: Input:
2:    ChangedFiles = list of files that is changed to fix the bug
3: Output:
4:    Lines that are root cause of the bug
5: Method:
6: Let RootLines =
7: for all File in ChangedFiles do
8:     Let DiffRanges = diff of new and old version of File
9:     for all DiffRange in DiffRanges do
10:        Let OldLines = DiffRange lines in old file
11:        Let NewLines = DiffRange lines in new file
12:        If NewLines contains OldLines
13:           Let SubLines = Lines in NewLines and not in OldLines
14:           If SubLines contains null-check
15:              Add OldLines to RootLines
16:    end for
17: end for
18: Output RootLines
```

Fig. 6. Algorithm for Case 1

B. Case 2: Exception Not Thrown

"Exception not thrown" means a case where an exception should be thrown. An example of this case is shown in Figure 7. In the new version of the code, there is an addition of null check for *input* variable that will throw an exception if it is null. Thus, the line of the code at line 69 in the old version that uses *input* variable without checking is the root cause.

```
src/java/org/apache/lucene/util/AttributeSource.java
BEFORE
68:    public AttributeSource(AttributeSource input) {
69:    this.attributes = input.attributes;
AFTER
68:    public AttributeSource(AttributeSource input) {
69:    if (input == null) {
70:      throw new IllegalArgumentException("input ...
71:    }
72:    this.attributes = input.attributes;
```

Fig. 7. Example Case 2 in Lucene Bug #12425426

We locate this kind of root causes by taking the changes that add an if statement and a throw statement inside the if statement. We take the variable(s) checked in the if conditional expression. We then blame all lines in the original code that satisfy the following conditions: after the location where the if statement is added, in the same block scope as the if statement, and containing a use of at least one of the variable(s) (i.e., sharing data with the if statement). The pseudocode for this algorithm is in Figure 8.

C. Case 3: Method Not Called

"Method not called" means the code does not invoke a prerequisite method required to use a variable correctly. An example of this case is shown in Figure 9. In this case, the new version of the code adds a `checkMutableReference` method invocation in line 1242 that makes use of a `ref` variable. This variable is used in the next line as an input for an `Node` object instantiation. Since the change is related to the bug fix, the method addition is likely intended to check whether the object pointed to by the `ref` variable is in a valid state. Thus, the line of code at line 1242 in the old version that uses `ref` variable without calling `checkMutableReference` is the root cause.

```
1: Input:
2:    ChangedFiles = list of files that is changed to fix the bug
3: Output:
4:    Lines that are root cause of the bug
5: Method:
6: Let RootLines =
7: for all File in ChangedFiles do
8:     Let DiffRanges = diff of new and old version of File
9:     for all DiffRange in DiffRanges do
10:        Let OldLines = DiffRange lines in old file
11:        Let NewLines = DiffRange lines in new file
12:        If NewLines add if statement containing throw statement
13:           Let checkVar = variable checked in if condition
14:           Let l = line number in OldLines where if statement is added
15:           l = l + 1
16:           While l in the same block scope as if statement in old file
17:              Let Var = variable used in line l of the old file
18:              if Var contains checkVar
19:                 Add l to RootLines
20:              l = l + 1
21:    end for
22: end for
23: Output RootLines
```

Fig. 8. Algorithm for Case 2

```
mozilla/js/rhino/src/org/mozilla/javascript/IRFactory.java
BEFORE
1240:    case Token.GET_REF: {
1241:        Node ref = left.getFirstChild();
1242:        return new Node(Token.SET_REF, ref, right);
AFTER
1240:    case Token.GET_REF: {
1241:        Node ref = left.getFirstChild();
1242:        checkMutableReference(ref);
1243:        return new Node(Token.SET_REF, ref, right);
```

Fig. 9. Example Case 3 in Rhino Bug #277935

We strictly pick only the changes that add a single method invocation that contains a use of variable(s). We take those variable(s) and perform the similar data sharing analysis as described in Case 2. We blame all lines in the original code that satisfy these conditions: after the location in which the method invocation will be added, in the same block scope as the method invocation, and contains a use to at least one of the variable(s). The pseudocode for this algorithm is in Figure 10.

```
1: Input:
2:    ChangedFiles = list of files that is changed to fix the bug
3: Output:
4:    Lines that are root cause of the bug
5: Method:
6: Let RootLines = {}
7: for all File in ChangedFiles do
8:     Let DiffRanges = diff of new and old version of File
9:     for all DiffRange in DiffRanges do
10:        Let OldLines = DiffRange lines in old file
11:        Let NewLines = DiffRange lines in new file
12:        If NewLines add method invocation And length of OldLines is 0 And length
           of NewLines is 1
13:           Let usedVar = variable used in method invocation
14:           Let l = line number in OldLines where method invocation is added
15:           l = l + 1
16:           While l in the same block scope as method invocation in old file
17:              Let Var = variable used in line l of the old file
18:              if Var contains usedVar
19:                 Add l to RootLines
20:              l = l + 1
21:    end for
22: end for
23: Output RootLines
```

Fig. 10. Algorithm for Case 3

VII. Experiments & Analysis

In this section, we first present our experimental setting which describes the environment where we run our algorithm,

and how we construct the golden set of root causes from bug-fixing file changes. Next, we present our evaluation criteria along with a baseline solution that we compare with our approach. We then present our experimental results answering a number of research questions. We end the section by presenting several threats to validity.

Our experimental setting follows that of *Turing test* [48]. We would like to build a machine learning solution that can make a judgment close to that made by a human being. This is the grand challenge that many artificial intelligence studies are trying to achieve, e.g., [38]. Many past software engineering papers also follow similar experimental setting, e.g., [8], where the goal is to produce a machine learning solution that can perform as good as human beings to automate the analysis of large amount of data.

A. Experiment Setting

We analyze bug-fixing file changes for AspectJ, Lucene, and Rhino. We retrieve the changes for AspectJ and Rhino from iBugs [13] while Lucene's is retrieved from its JIRA issue tracking system [3]. iBugs provides us with pre-fix and post-fix source code for a selected set of bugs. JIRA is a commercial issue tracking system used in many Apache projects. It enables a subversion-commit plugin that provides links between a bug and its related subversion revisions. Bug-fixing file changes can be retrieved from such links.

We manually identify the root causes of the bugs for iBugs and Lucene version 2.9. We define the root causes of a bug as the lines of code in the buggy version that is responsible for the bug. Two persons manually identify the root causes of each bug and if there are uncertainties, they meet to discuss until an agreement is met. In an effort to ensure the quality of the data, we select only unambiguous bugs to be the golden set of the root causes. Unambiguous bugs mean that we have high confidence that the lines labeled are the root causes of the bugs. In the end, we have 152 labeled bugs from AspectJ, 28 from Lucene, and 20 from Rhino.

B. Evaluation Criteria

Here we first present the metrics we use to evaluate how good a solution is in extracting root causes. Next, we present the baselines that we would compare with our approach.

1) Metrics: To evaluate the performance of our approach, we use three metrics commonly used in data mining and software mining, namely precision, recall, and F-measure. Each of those metrics is defined as follows.

$$Precision = \frac{\#IntersectedLine}{\#PotentialRootCauseLine}$$

$$Recall = \frac{\#IntersectedLine}{\#RootCauseLine}$$

$$F\text{-}measure = \frac{2 * Precision * Recall}{Precision + Recall}$$

#IntersectedLine is the number of lines that appear in the set of *RootCauseLine* and *PotentialRootCauseLine*. ***#RootCauseLine*** is the number of lines labeled as a root cause of a bug in the golden set. ***#PotentialRootCauseLine*** is the number of lines labeled as a root cause by our approach.

Following commonly used evaluation in data mining, we also employ k-fold cross validation [16] that split the whole golden set of root causes into 10 subsets (i.e., k=10), and repeat the experiments 10 times to calculate the average precision, recall, and F-measure; each time it uses 9 of the subsets as the training data and the other as the testing data.

2) Baselines: To the best of our knowledge, there is no existing technique that recovers root causes from bug-fixing file changes. The closest to our work is DiffCat which returns essential changes. Thus, as a baseline we consider the lines flagged by DiffCat and compare them with lines flagged by our approach. We run DiffCat, and take the lines of code identified by DiffCat as essential changes from the version before the bug-fixing change. We treat these lines as the root causes identified by DiffCat and use them to compute the precision, recall, and F-measure for DiffCat.

We also use standard Unix *diff* as another baseline. We run *diff* on the two versions of code before and after a bug-fixing change, and take the lines from the old version in the diff as the root causes identified by diff to compute the precision, recall, and F-measure for diff.

C. Experiment Results

We consider the following research questions:

- RQ1 How effective is our approach as compared to standard *diff* and DiffCat [25] in identifying root causes?
- RQ2 How effective is our essential change extraction component?
- RQ3 How effective is our filtering component?
- RQ4 How effective is our propagation component?
- RQ5 How effective is our approach when less training data is used?
- RQ6 How efficient is our solution? How much time does our approach need to extract root causes from bug-fixing file changes?

1) RQ1: Effectiveness of Our Approach: Table II shows the precisions, recalls, and F-measures of our approach, standard Unix *diff*, and DiffCat [25].

TABLE II
EFFECTIVENESS OF OUR APPROACH AS COMPARED TO *diff* AND DIFFCAT [25] IN IDENTIFYING ROOT CAUSES

Approach	Precision	Recall	F-Measure
diff	22.74%	96.24%	36.78%
DiffCat [25]	33.30%	74.65%	46.06%
Our Approach	76.42%	71.88%	74.08%

We note that diff has a high recall of root causes because it removes fewer lines of code. DiffCat has a higher recall but a much lower precision than our approach. F-measure evaluates if the gain in precision (or recall) outweighs the loss in recall (or precision) and provides a balance view on precision and recall. Our approach achieves a F-measure of 74.08%, and

outperforms the two baseline approaches, namely standard *diff* and DiffCat, by 101.41% and 60.83% respectively.

2) RQ2: Effectiveness of the Essential Change Extraction Component: Essential change extraction component consists of DiffCat and Line Level Analysis. To evaluate if this component is effective, we only turn on this component and turn off all other components in our approach (which is referred to as as *ECE*), and compute the resultant precision, recall and F-measure. The results of our experiments are shown in Table III.

TABLE III
EFFECTIVENESS OF THE ESSENTIAL CHANGE EXTRACTION COMPONENT

Approach	Precision	Recall	F-Measure
diff	22.74%	96.24%	36.78%
DiffCat [25]	33.30%	74.65%	46.06%
ECE	56.24%	74.06%	63.93%

Our approach with only essential change extraction component turned on (*ECE*) could achieve a precision, recall, and F-measure of 56.24%, 74.06%, and 63.93% respectively. Thus *ECE* outperforms the two baseline approaches namely standard *diff* and DiffCat. In terms of F-measure, we could improve *diff* and DiffCat by 73.82% and 38.80% respectively.

3) RQ3: Effectiveness of Filtering Component: Our filtering component performs machine learning. To evaluate if this component is effective we turn on the essential change extraction component and this component (which is referred to as *ECE+F*), and compute the resultant precision, recall and F-measure. The results of our experiments are shown in Table IV.

TABLE IV
EFFECTIVENESS OF ESSENTIAL FILTERING COMPONENT

Approach	Precision	Recall	F-Measure
ECE	56.24%	74.06%	63.93%
ECE+F	75.38%	67.92%	71.46%

Our approach with only essential change extraction component turned on (*ECE+F*) could achieve a precision, recall, and F-measure of 75.38%, 67.92%, and 71.46% respectively. Thus *ECE+F* outperforms *ECE*. The filtering component increases precision by a large amount with a small loss of recall. In terms of F-measure, it improves ECE by 11.78%. F-measure measures if the gain in precision outweighs the loss in recall.

4) RQ4: Effectiveness of the Propagation Component: Comparing Table IV and Table II, we note that our overall approach that includes the propagation component outperforms *ECE+F* by an increase in F-measure by 3.67% percent.

5) RQ5: Effectiveness when Less Training Data is Used: To evaluate the effects of less training data, we reduce the value of k during the k-fold cross validation. The lower the value of k, the less is the number of data that is used for training. The results for different k are shown in Table V.

From Table V, we notice that in terms of F-measure, precision, and recall, the differences among different k are minimal, around 1-2%.

TABLE V
EFFECTIVENESS OF k-FOLD CROSS VALIDATION FOR DIFFERENT k

K	Precision	Recall	F-Measure
10	76.42%	71.88%	74.08%
9	76.39%	72.38%	74.33%
8	74.74%	72.08%	73.39%
7	76.26%	71.88%	74.01%
6	76.34%	72.18%	74.2%
5	76.22%	71.39%	73.72%
4	75.81%	71.68%	73.69%
3	76.31%	72.08%	74.13%
2	75.31%	71.58%	73.4%

6) RQ6: Efficiency of Our Approach: The machine we use to run the experiments runs Windows Server 2008 SP2 with a 2.53GHz Intel Xeon CPU and 24GB of memory. We find that our approach extracts root causes from a bug-fixing change within 2–678 seconds. On average, we could extract root causes from a bug-fixing change in 41 seconds. Thus, we believe our approach is practically efficient and usable.

D. Threats To Validity

There are several threats to validity that we consider in this study: threats to internal validity, threats to external validity, and threats to construct validity.

Threats to internal validity includes experimenter bias. The golden set of root causes is extracted manually. There might be subjectivity in choosing which lines are the root cause. Many past studies that require manual labor to produce ground truth data, e.g., [8], [15], [21], also suffer from similar threat to validity. Still, we have tried to reduce this threat to internal validity by checking our answers a number of times and by asking more than one person to label the root causes and resolve discrepancies through discussions. Also, we exclude ambiguous bugs; decision whether a bug is ambiguous or not is subjective.

Threats to external validity refers to the generalizability of our findings. In this study, we have only investigated root causes of 200 bugs from 3 software systems. Although this does not represent all software systems, the scale that we consider is on par with past studies that involve manual effort in labeling data which takes much time (it takes weeks in our case). In the future, we plan to reduce this threat further by labeling more root causes from more bug-fixing file changes.

Threats to construct validity refers to the suitability of our evaluation metrics. We make use of standard metrics of data mining namely precision, recall, and their harmonic mean (i.e., F-measures). These metrics have also been used in many other software engineering studies in the past, e.g., [36], [47], [51]. Although it is important to compare to DiffCat, we need to clearly mention that DiffCat is designed for a different purpose. Our comparison with DiffCat is meant to demonstrate the value of the extensions that we build on top of DiffCat to detect root causes.

VIII. RELATED WORK

We highlight some related studies on software changes, root causes, and empirical studies on bugs. This section is by no

means of a complete list of all related work.

A. Software Changes

Kawrykow and Robillard propose a method named DiffCat that detects and removes non-essential changes [25]. In this work, we address a different problem namely on the identification of root causes from bug-fixing file changes. We use DiffCat as a building block in our approach.

Sliwerski et al. [42], Kim et al. [27], and Sinha et al. [41] propose approaches that detect bug origins, i.e., versions that induce bug fixes. They find the versions, made before a bug gets reported, that introduce all lines that are changed or deleted (ignoring comment, blank line, and format changes) in the immediate version prior to the bug fix. These approaches implicitly assume that these lines are root causes. However, not every such lines are root causes. Many bug fixes involve touching hundreds or even thousands of lines of code, c.f., [34], many of these lines are changed not because they are root causes but because they need to be modified to facilitate the treatment of the bug. For example, line 2033 in Figure 1(a) is not a root cause although it is touched by the bug fix, however, the previous approaches would find the origins of line 2033 as well. In our work, we consider a different goal from these past approaches, we want to classify which of these changed and deleted lines are root causes. Thus our goal is complementary with theirs, we can apply our technique to find root causes and then apply theirs to find the origins of these root causes.

Kim et al. propose a technique that predicts if a software change is clean or buggy [26]. Features are extracted from revision histories and used for the prediction task with the aid of a classifier. They show that a recall of 65% and an accuracy of 78% could be achieved. In this paper, we have an orthogonal goal to predict lines in a bug-fixing change that are the root cause of the bug. Our approach is a retrospective analysis of bugs while Kim et al's approach is a prospective analysis. We show that we can achieve recall, precision, F-measure, and accuracy of more than 70%.

Wu et al. [51] propose an approach to automatically link code changes with corresponding bug reports together. Our work is complementary to theirs. We recover root causes from such code changes that correspond to fixes to issues mentioned in bug reports, and our results may potentially be used to improve the quality of their results.

B. Root Causes

There have been a number of studies that try to locate root causes given software failures. One family of techniques is spectrum based fault localization which analyzes program traces [2], [23], [33]. Their goal is to find an association between program failures and the execution of some program elements. These program elements that are associated with failures are identified as suspicious program elements (e.g., lines, blocks, methods, etc.) and could then be presented to debuggers.

Many spectrum-based techniques analyze two sets of program traces: a set of correct program executions and a set of program failures [2], [23], [33], [49]. Based on these two sets of program traces, potential root causes are highlighted. Jones and Harrold propose Tarantula that assign suspiciousness scores to program elements based on the number of times they are executed by correct program executions, the number of times they are executed by program failures, the number of correct program executions, and the number of failures [23]. Different suspiciousness formulas are proposed by several other authors [2], [33]. Abreu et al. propose a suspiciousness formula called Ochiai which has been shown to be more accurate than Tarantula [2]. Lucia et al. analyze many association measures and investigate their effectiveness for fault localization [33]. Wang et al. compose many association measures using search based techniques for fault localization [49].

In this work, we are also interested in finding root causes. However, rather than finding root causes based on a set of program failures, we find root causes from bug-fixing file changes. Execution traces (including failures) are often not available in bug reports [43].

Aside from our definition of a root cause (i.e., program elements whose executions triggers a chain of erroneous program states eventually leading to a program failure or error behavior) there have been other definitions. For example, Leszak et al. [29] defines a root cause as a factor in a software development process that causes bugs, e.g., lack of domain knowledge, time pressure, management mistake, incomplete review, etc.

IX. Conclusion and Future Work

Current bug tracking and source control systems only record symptoms and treatments of bugs. One crucial information is missing, namely: which program elements are the root causes of the bugs? This information is important as it is often first in the mind of developers when performing debugging activities. Many past studies often manually recover root causes from information available in bug tracking and source control systems. However, this process is painstaking, slow, and error prone. In this work, we automate this root cause extraction process. We propose a framework which consists of 3 components: essential change extraction, filtering, and propagation. We combine case-based code analysis and machine learning to develop our solution. We have experiment our solution on a dataset of root causes from 200 bugs from 3 systems. These root causes are manually extracted from bug-fixing file changes based on information from bug tracking systems and source control systems; it took weeks for us to get these root causes manually. We show that our automated process could complete on average in 41 seconds, and achieve a precision, recall, and F-measure of 76.42%, 71.88%, and 74.08%. respectively. Comparing with baseline approaches, namely *diff* and DiffCat, we show that we can increase F-measure by 101.41% and 60.83%, respectively.

In the future, we plan to improve our approach further by incorporating more cases in our case-based code analysis solutions. We only include simple rules in this work, we plan to add rules that involve advanced static analysis techniques

in the future. We also plan to investigate with more features and more classifiers to make our machine learning block better in differentiating root causes from non root causes. We have only used bug-fixing file changes to recover root causes; in the future, we want to use bug symptoms (e.g., textual descriptions in bug reports, program spectra produced by test cases (i.e., a log of program elements executed when the test cases are run)) in addition to the bug-fixing file changes, by employing text mining and software analytic solutions, e.g., [10], [30]–[32], [49], [50], to improve the accuracy of root cause recovery. We also plan to increase the number of bugs used as the golden set of root causes. Also, we want to apply our automated approach on a large number of bug-fixing file changes and repeat some past empirical studies that require the identification of root causes, e.g., [34], [46], on a larger number of bugs. Furthermore, we plan to further investigate the effectiveness of our approach on different kinds of bugs. We also would like to perform a user study to solicit developers feedback on the outputs of our tool. It would also be interesting to use our proposed technique to create a large repository of root causes and use the resulting repository as input to a machine learning solution that can identify root causes of not yet solved bugs.

REFERENCES

[1] "Rhino—JavaScript for Java," http://www.mozilla.org/rhino/.

[2] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund, "On the Accuracy of Spectrum-based Fault Localization," in *TAICPART-MUTATION*, 2007.

[3] Apache Software Foundation, "Apache Lucene and Its JIRA," http://lucene.apache.org/core/, https://issues.apache.org/jira/browse/LUCENE.

[4] C. Artho, "Jlint - find bugs in java programs," http://jlint.sourceforge.net/, 2006.

[5] "The AspectJ project," *http://eclipse.org/aspectj*.

[6] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh, "Using static analysis to find bugs," *IEEE Software*, vol. 25, no. 5, pp. 22–29, 2008.

[7] N. Ayewah and W. Pugh, "The google findbugs fixit," in *ISSTA*, 2010, pp. 241–252.

[8] A. Bacchelli, T. D. Sasso, M. D'Ambros, and M. Lanza, "Content classification of development emails," in *ICSE*, 2012, pp. 375–385.

[9] T. F. Bissyandé, F. Thung, S. Wang, D. Lo, L. Jiang, and L. Réveillère, "Empirical evaluation of bug linking," in *CSMR*, 2013, pp. 89–98.

[10] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation," *Journal of Machine Learning Research*, vol. 3, 2003.

[11] T. Copeland, *PMD Applied*. Centennial Books, 2005.

[12] B. Dagenais and L. J. Hendren, "Enabling static analysis for partial java programs," in *OOPSLA*, 2008, pp. 313–328.

[13] V. Dallmeier and T. Zimmermann, "Extraction of bug localization benchmarks from history," in *ASE*, 2007, pp. 433–436.

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

[15] M. Gegick, P. Rotella, and T. Xie, "Identifying security bug reports via text mining: An industrial case study," in *MSR*, 2010, pp. 11–20.

[16] J. Han and M. Kamber, *Data Mining Concepts and Techniques*, 2nd ed. Morgan Kaufmann, 2006.

[17] T. Harris, J. R. Larus, and R. Rajwar, *Transactional Memory*, ser. Synthesis Lectures on Computer Architecture. Morgan & Claypool, 2010, vol. 5, no. 1.

[18] K. Herzig and A. Zeller, "Untangling changes," September 2011, under submission.

[19] A. Hindle, D. M. Germán, M. W. Godfrey, and R. C. Holt, "Automatic classication of large changes into maintenance categories," in *ICPC*, 2009, pp. 30–39.

[20] R. Hoda, J. Noble, and S. Marshall, "Developing a grounded theory to explain the practices of self-organizing agile teams," *Empirical Software Engineering*, vol. 17, no. 6, 2012.

[21] L. Huang, V. Ng, I. Persing, R. Geng, X. Bai, and J. Tian, "Autoodc: Automated generation of orthogonal defect classifications," in *ASE*, 2011.

[22] T. Joachims, "Training linear svms in linear time," in *KDD*, 2006, pp. 217–226.

[23] J. Jones and M. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *ASE*, 2005.

[24] R. Jones, A. Hosking, and E. Moss, *The Garbage Collection Handbook: The Art of Automatic Memory Management*, ser. CRC Applied Algorithms and Data Structures Series. Chapman and Hall/CRC, 2011.

[25] D. Kawrykow and M. P. Robillard, "Non-essential changes in version histories," in *ICSE*, 2011, pp. 351–360.

[26] S. Kim, E. J. W. Jr., and Y. Zhang, "Classifying software changes: Clean or buggy?" *IEEE TSE*, vol. 34, no. 2, pp. 181–196, 2008.

[27] S. Kim, T. Zimmermann, K. Pan, and E. J. J. Whitehead, "Automatic identification of bug-introducing changes," in *ASE*, 2006, pp. 81–90.

[28] T. Lee, J. Nam, D. Han, S. Kim, and H. P. In, "Micro interaction metrics for defect prediction," in *FSE*, 2011, pp. 311–321.

[29] M. Leszak, D. E. Perry, and D. Stoll, "A case study in root cause defect analysis," in *ICSE*, 2000, pp. 428–437.

[30] D. Lo and S.-C. Khoo, "Smartic: towards building an accurate, robust and scalable specification miner," in *SIGSOFT FSE*, 2006, pp. 265–275.

[31] D. Lo and S. Maoz, "Mining hierarchical scenario-based specifications," in *ASE*, 2009.

[32] ——, "Scenario-based and value-based specification mining: better together," in *ASE*, 2010, pp. 387–396.

[33] Lucia, D. Lo, L. Jiang, and A. Budi, "Comprehensive evaluation of association measures for fault localization," in *ICSM*, 2010.

[34] Lucia, F. Thung, D. Lo, and L. Jiang, "Are faults localizable?" in *MSR*, 2012, pp. 74–77.

[35] P. Martin and B. Turner, "Grounded theory and organizational research," *The Journal of Applied Behavioral Science*, vol. 22, no. 2, 1986.

[36] T. Menzies and A. Marcus, "Automated severity assessment of software defect reports," in *ICSM*, 2008, pp. 346–355.

[37] K. Pan, S. Kim, and E. J. W. Jr., "Toward an understanding of bug fix patterns," *Empirical Software Engineering*, 2009.

[38] J. Pearl, "The mechanization of causal inference: A "mini turing test" and beyond," in *Turing Award Lecture*, 2012.

[39] S. Rao and A. C. Kak, "Retrieval from software libraries for bug localization: a comparative study of generic and composite text models," in *MSR*, 2011, pp. 43–52.

[40] S. Shivaji, E. J. W. Jr., R. Akella, and S. Kim, "Reducing features to improve bug prediction," in *ASE*, 2009, pp. 600–604.

[41] V. S. Sinha, S. Sinha, and S. Rao, "Buginnings: identifying the origins of a bug," in *ISEC*, 2010.

[42] J. Sliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" in *MSR*, 2005.

[43] C. Sun, D. Lo, S.-C. Khoo, and J. Jiang, "Towards more accurate retrieval of duplicate bug reports," in *ASE*, 2011.

[44] C. Sun, D. Lo, X. Wang, J. Jiang, and S.-C. Khoo, "A discriminative model approach for accurate duplicate bug report retrieval," in *ICSE (1)*, 2010, pp. 45–54.

[45] F. Thung, D. Lo, and L. Jiang, "Automatic defect categorization," in *WCRE*, 2012.

[46] F. Thung, Lucia, D. Lo, L. Jiang, F. Rahman, and P. T. Devanbu, "To what extent could we detect field defects? an empirical study of false negatives in static bug finding tools," in *ASE*, 2012, pp. 50–59.

[47] Y. Tian, J. L. Lawall, and D. Lo, "Identifying linux bug fixing patches," in *ICSE*, 2012, pp. 386–396.

[48] A. Turing, "Computing machinery and intelligence," *Mind LIX*, vol. 236, pp. 433–460, 1950.

[49] S. Wang, D. Lo, L. Jiang, Lucia, and H. C. Lau, "Search-based fault localization," in *ASE*, 2011.

[50] X. Wang, D. Lo, J. Jiang, L. Zhang, and H. Mei, "Extracting paraphrases of technical terms from noisy parallel software corpora," in *ACL/IJCNLP*, 2009.

[51] R. Wu, H. Zhang, S. Kim, and S.-C. Cheung, "Relink: recovering links between bugs and changes," in *SIGSOFT FSE*, 2011, pp. 15–25.

[52] J. Zhou, H. Zhang, and D. Lo, "Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports," in *ICSE*, 2012, pp. 14–24.

[53] T. Zimmermann, R. Premraj, and A. Zeller, "Predicting defects for eclipse," in *PROMISE*, 2007.