Automatic Identification of Bug-Introducing Changes

Sunghun Kim[1], Thomas Zimmermann[2], Kai Pan[1], E. James Whitehead, Jr.[1]

[1]*University of California, Santa Cruz, CA, USA*
{hunkim, pankai, ejw}@cs.ucsc.edu

[2]*Saarland University, Saarbrücken, Germany*
tz@acm.org

Abstract

Bug-fixes are widely used for predicting bugs or finding risky parts of software. However, a bug-fix does not contain information about the change that initially introduced a bug. Such bug-introducing changes can help identify important properties of software bugs such as correlated factors or causalities. For example, they reveal which developers or what kinds of source code changes introduce more bugs. In contrast to bug-fixes that are relatively easy to obtain, the extraction of bug-introducing changes is challenging.

In this paper, we present algorithms to automatically and accurately identify bug-introducing changes. We remove false positives and false negatives by using annotation graphs, by ignoring non-semantic source code changes, and outlier fixes. Additionally, we validated that the fixes we used are true fixes by a manual inspection. Altogether, our algorithms can remove about 38%~51% of false positives and 14%~15% of false negatives compared to the previous algorithm. Finally, we show applications of bug-introducing changes that demonstrate their value for research.

1. Introduction

Today, software bugs remain a constant and costly fixture of industrial and open source software development. To manage the flow of bugs, software projects carefully control their changes using software configuration management (SCM) systems, capture bug reports using bug tracking software (such as Bugzilla), and then record which change in the SCM system fixes a specific bug in the change tracking system.

The progression of a single bug is as follows. A programmer makes a change to a software system, either to add new functionality, restructure the code, or to repair an existing bug. In the process of making this change, they inadvertently introduce a bug into the software. We call this a *bug-introducing change*, the modification in which a bug was injected into the software. At some later time, this bug manifests itself in some undesired external behavior, which is recorded in a bug tracking system. Subsequently, a developer modifies the project's source code, possibly changing multiple files, and repairs the bug. They commit this change to the SCM system, permanently recording the change. As part of the commit, developers commonly (but not always) record in the SCM system change log the identifier of the bug report that was just fixed. We call this modification a *bug-fix change*.

Software evolution research leverages the history of changes and bug reports that accretes over time in SCM systems and bug tracking systems to improve our understanding of how a project has grown. It offers the possibility that by examining the history of changes made to a software project, we might better understand patterns of bug introduction, and raise developer awareness that they are working on risky—that is, bug-prone—sections of a project. For example, if we can find rules that associate bug-introducing changes with certain source code change patterns (such as signature changes that involve parameter addition [11]), it may be possible to identify source code change patterns that are bug-prone.

Due to the widespread use of bug tracking and SCM systems, the most readily available data concerning bugs are the bug-fix changes. It is easy to mine an SCM repository to find those changes that have repaired a bug. To do so, one examines change log messages in two ways: searching for keywords such as "Fixed" or "Bug" [12] and searching for references to bug reports like "#42233" [2, 4, 16]. With bug-fix information, researchers can determine the *location* of a bug. This permits useful analysis, such as determining per-file bug counts, predicting bugs, finding risky parts of software [7, 13, 14], or visually revealing the relationship between bugs and software evolution [3].

The major problem with bug-fix data is that it sheds no light on *when* a bug was injected into the code and *who* injected it. The person fixing a bug is often not the person who first made the bug, and the bug-fix must, by definition, occur after the bug was first injected. Bug-fix data also provides imprecise data on *where* a bug occurred. Since functions and methods change their names over time, the fact that a fix was made to function "foo" does not mean the function still had that name when the bug was injected; it could have been named "bar" then. In order to deeply understand the phenomena surrounding the introduction of bugs into code, such as correlated factors and causalities, we need access to the actual moment and point the bug was introduced. This is tricky, and the focus of our paper.

Revision 1 (by kim, bug-introducing)

```
1 kim  1: public void bar() {
1 kim  2:    // print report
1 kim  3:    if (report == null) {
1 kim  4:       println(report);
1 kim  5:
1 kim  6:    }
```

Revision 2 (by ejw)

```
2 ejw  1: public void foo() {
1 kim  2:    // print report
2 ejw  3:    if (report == null)
2 ejw  4:    {
1 kim  5:       println(report);
1 kim  6:
1 kim  7:    }
```

Revision 3 (by kai, bug-fix)

```
2 ejw  1: public void foo() {
3 kai  2:    // print out report
3 kai  3:    if (report != null)
1 kim  4:    {
1 kim  5:       println(report);
1 kim  6:    }
```

Figure 1. Example bug-fix and source code changes. A null-value checking bug is injected in revision 1, and fixed in revision 3.

2. Background

Previous work by the second author developed what was, prior to the current paper, the only approach for identifying bug-introducing changes from bug-fix changes [16]. For convenience, we call this previous approach the SZZ algorithm, after the first letters of the authors' last names. To identify bug-introducing changes, SZZ first finds bug-fix changes by locating bug identifiers or relevant keywords in change log text, or following an explicitly recorded linkage between a bug tracking system and a specific SCM commit. SZZ then runs a *diff* tool to determine what changed in the bug-fixes. The diff tool returns a list of regions that differ in the two files; each region is called a hunk. It observes each hunk in the bug-fix and assumes that the deleted or modified source code in each hunk is the location of a bug. Finally, SZZ tracks down the origins of the deleted or modified source code in the hunks using the built-in *annotate* feature of SCM systems. The annotate feature computes, for each line in the source code, the most recent revision in which the line was changed, and the developer who made the change. The discovered origins are identified as bug-introducing changes.

Figure 1 shows an example of the history of development of a single function over three revisions.

- Revision 1 shows the initial creation of function *bar*, and the injection of a bug into the software, the line *'if (report == null) {'* which should be *'!='* instead. The leftmost column of each revision shows the output of the SCM annotate command, identifying the most recent revision and the developer who made the revision. Since this is the first revision, all lines were first modified at revision 1 by the initial developer '*kim.'* The second column of numbers in revision 1 lists line numbers within that revision.
- In the second revision, two changes were made. The function *bar* was renamed to *foo*, and a cosmetic change was made where the angle bracket at the end of line 3 in revision 1 was moved down to its own line (4) in revision 2. As a result, the annotate output shows lines 1, 3, and 4 as having been most recently modified in revision 2 by *'ejw.'*
- Revision 3 shows three changes, a modification to the comment in line 2, deleting the blank line after the println, and the actual bug-fix, changing line 3 from *'=='* to *'!='*.

Let us consider what happens when the SZZ algorithm tries to identify the fix-inducing change associated with the bug-fix in revision 3. SZZ starts by computing the delta between revisions 3 and 2, yielding the lines 2, 3, and 6 (these are highlighted in the figure). SZZ then uses SCM annotate data to determine the initial origin of these three lines. The first problem we encounter is that SZZ seeks the origin of the comment line (2) and the blank line (6); clearly neither contains the injected bug, since these lines are not executable. The next problem comes when SZZ tries to find the origin of line 3. Since revision 2 modified this line to make a cosmetic change (moving the angle bracket), the SCM annotate data indicates that this line was most recently modified at revision 2. SZZ stops there, claiming that revision 2 is the bug-introducing change. This is incorrect, since revision 1 was the point at which the bug was initially entered into the code. The cosmetic change threw off the algorithm.

A final problem is that, using just SCM annotate information, it is impossible to determine that the name of the function containing the bug changed its name from *bar* to *foo*. The annotate information only contains triples of (current revision line #, most recent modification revision, developer who made modification). There is no information here that states that a given line in one revision maps to a specific line in a previous (or following) revision. It is certainly possible to compute this information—indeed, we do so in the approach we outline in this paper—but to do so requires more information than is provided solely by SCM annotate capabilities.

We can now summarize the main two limitations of the SZZ algorithm:

SCM annotation information is insufficient: there is not enough information to identify bug-introducing changes. The previous example demonstrates how a simple formatting change (moving the bracket) modifies SCM annotate data so an incorrect bug-introducing revision is chosen. It also highlights the need to trace the evolution of individual lines across revisions, so function/method containment can be determined.

Not all modifications are fixes: Even if a file change is defined as a bug-fix by developers, not all hunks in the change are bug-fixes. As we saw above, changes to

comments, blank lines, and formatting are not bug-fixes, yet are flagged as such.

These two limitations result in the SZZ algorithm inaccurately identifying bug-introducing changes. To address these issues, in this paper we present an improved approach for achieving accurate bug-introducing change identification by extending SZZ. In the new approach, we employ *annotation graphs*, which contain information on the cross-revision mappings of individual lines. This is an improvement over SCM annotate data, and permits a bug to be associated with its containing function or method. We additionally remove false bug-fixes caused by comments, blank lines, and format changes.

An important aspect of this new approach is that it is automated. Since revision histories for large projects can contain thousands of revisions and thousands of files, automated approaches are the only ones that scale to this size. As an automated approach, the bug-introducing identification algorithm we describe can be employed in a wide range of software evolution analyses as an initial clean-up step to obtain high quality data sets for further analysis on the causes and patterns of bug formation.

To determine the accuracy of the automatic approach, we use a manual approach as well. Two human judges manually verified all hunks in a series of bug-fix changes to ensure the corresponding hunks are real bug-fixes.

We applied our automatic and manual approach to identify bug-introducing changes at the method level for two Java open source projects, Columba and Eclipse (jdt.core). We propose the following steps, as shown in Figure 2, to remove false positive and false negatives in identifying bug-introducing changes.

1. Use annotation graphs to provide more detailed annotation information
2. Ignore comment and blank line changes
3. Ignore format changes
4. Ignore outlier bug-fix revisions in which too many files were changed
5. Manually verify all hunks in the bug-fix changes

Figure 2. Summary of approach

In overview, applying this new approach (steps 1-5) removes 38%~51% of false positives and 14%~15% of false negatives as compared to the original SZZ algorithm. Using only the automated algorithms (steps 1-4), we can remove 36~48% false positives and 14% of false negatives. The manual fix verification does not scale, but highlights the low residual error remaining at the end of the automated steps, since it removes only 2~3% of false positives and 1% of false negatives.

In the remainder of the paper, we begin by describing our experimental setup (Section 3). Following are results from our experiments (Section 4), along with discussion of the results (Section 5). Rounding off the paper, we end with some existing applications of bug-introducing changes (Section 6) and conclusions (Section 7).

3. Experimental Setup

In this section, we describe how we extract the change history from an SCM system for our two projects of interest. We also explain the accuracy measures we use for assessing the performance of each stage in our improved algorithm for identifying bug-introducing changes.

3.1. History Extraction

Kenyon is a system that extracts source code change histories from SCM systems such as CVS and Subversion [1]. Kenyon automatically checks out the source code for each revision and extracts change information such as the change log, author, change date, source code, change delta, and change metadata. We used Kenyon to extract the histories of two open source projects, as shown in Table 1.

3.2. Accuracy Measures

A *bug-introducing change set* is all of the changes within a specific range of project revisions that have been identified as bug-introducing. Suppose we identify a bug-introducing change set, ***P***, using a bug-introducing identification algorithm such as SZZ [16]. We then apply the algorithm described in this paper, and derive another bug-introducing change set, ***R,*** as shown in Figure 3. The common elements of the two sets are $P \cap R$.

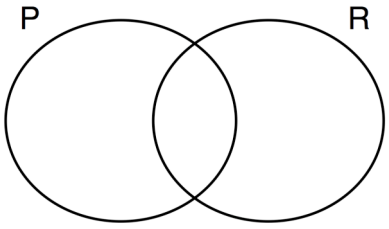

Figure 3. Bug-introducing change sets identified using SZZ (P) and with the new algorithm (R)

Assuming ***R*** is the more accurate bug-introducing change set, we compute false positives and false negatives for the set ***P*** as follows:

$$\text{False positive (FP)} = \frac{|P - R|}{|P|}$$

$$\text{False negative (FN)} = \frac{|R - P|}{|R|}$$

4. Algorithms and Experiments

In this section, we explain our approach in detail and present our results from using the improved algorithm to identify bug-introducing changes.

Table 1. Analyzed projects. # of revisions indicates the number of revisions we analyzed. # of fix revisions indicates the number of revisions that were identified as bug-fix revisions. Average LOC indicates the average lines of code of the projects in given periods.

Project	Software type	Period	# of revision	# of fix revision	% of fix revision	Average LOC
Columba	Email Client	11/2002 ~ 06/2003	500	143	29%	48,135
Eclipse (jdt.core)	IDE	06/2001 ~ 03/2002	1000	158	16%	111,059

4.1. Using Annotation Graph

The SZZ algorithm for the identification of bug-introducing changes for fine-grained entities such as functions or methods uses SCM annotation data. In this section, we show that this information is insufficient, and may introduce false positives and negatives.

Assume a bug-fix change occurs at revision 20, and involves the deletion of three lines (see Figure 4). Since they were deleted, the three lines are likely to contain a bug. In the SZZ approach, SCM *annotate* data is used to obtain the revisions in which these lines were initially added. The first two lines were added at revision 3, and the third line was added at revision 9. Thus, we identify the changes between revisions 2 and 3 and between revisions 8 and 9 as bug-introducing changes at the file level.

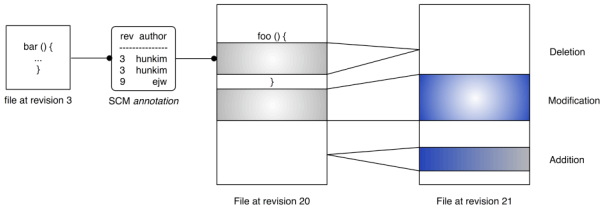

Figure 4. Finding bug-Introduction changes in the function level.

A problem occurs when we try to locate bug-introducing changes for entities such as functions or methods. Suppose the deleted source code at revision 20 was part of the '*foo()*' function (see Figure 4). Note that SCM annotation data for CVS or Subversion includes only revision and author information. This means we only know that the first two lines in Figure 4 were added at revision 3 by '*hunkim*', but we do not know the actual line numbers of the deleted code at revision 3. In past research, it was assumed that the lines at revision 3 are part of the '*foo()*' function, which is marked as a bug-introducing change, even though there is no guarantee that the function '*foo()*' existed at revision 3.

Suppose at revision 3 that '*foo()*' does not exist and the '*bar()*' function does exist, as shown in Figure 4. One explanation for how this could occur is the '*bar()*' function changes its name to '*foo()*' at some later revision. One consequence is the above assumption is wrong and the '*foo()*' function at revision 3 does not contain the bug-introducing change (false positive). We also miss a real bug-introducing change, '*bar()*' at revision 3 (false negative). Since SCM annotations do not provide the line numbers for the annotated lines at revision 3, it is not possible to identify the function where the bug-introducing lines were inserted.

We address this problem by using annotation graphs [18], a representation for origin analysis [6, 10] at the line level, as shown in Figure 5. In an annotation graph, every line of a revision is represented as a node; edges connect lines (nodes) that evolved from each other: either by modification of the line itself or by moving the line in the file. In Figure 5 two regions were changed between revisions r_1 and r_2: lines 10 to 12 were inserted and lines 19 to 23 were modified. The annotation graph captures these changes as follows: line 1 in r_2 corresponds to line 1 in r_1 and was not changed (the edge is not marked in bold), the same holds for lines 2 to 9. Lines 10 to 12 were inserted in r_2, thus they have no origin in r_1. Line 13 in r_2 was unchanged but has a different line number (10) in r_1, this is indicated by the edge (same for 14 to 18 in r_2). Lines 19 to 23 were modified in r_2 and originated from lines 16 to 20 (edges are marked in bold). Note that we approximate origin conservatively, i.e., for modifications we need to connect all lines affected in r_1 (lines 16 to 20) with every line affected in r_2 (lines 19 to 23).

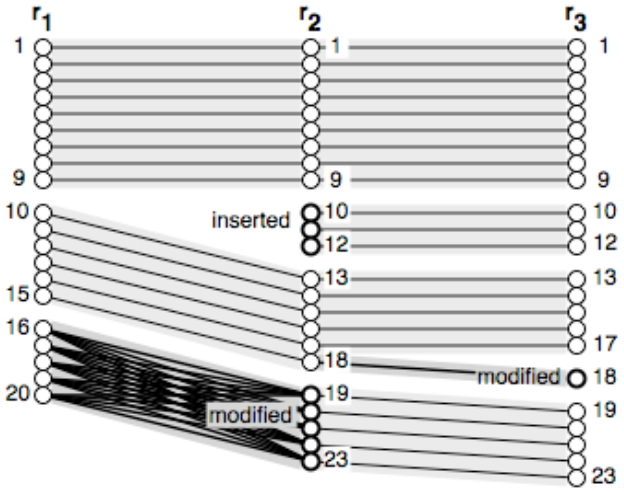

Figure 5. An annotation graph shows line changes of a file for three revisions [18]. A single node represents each line in a revision; edges between nodes indicate that one line originates from another, either by modification or by movement.

The annotation graph improves identification of bug-introducing code by providing for each line in the bug-fix change the line number in the bug-introducing revision. This is computed by performing a backward directed depth-first search. The resulting line number is then used to identify the correct function name in the bug-fix revision. For the above example, the annotation graph

would annotate the deleted lines with the line numbers in revision 3, which are then used to identify function '*bar*'.

To demonstrate the usefulness of annotation graphs for locating bug-introducing changes, we identify bug-introducing changes at the method level for our two projects *with* and *without* the use of annotation graphs. The left circle in Figure 6 (a) shows the count of bug-introducing changes at method level identified without using the annotation graph; the right circle shows the count when using the annotation graphs. Without the annotation graph we have about 2% false positives and 1~4% false negatives (total 3~6% errors) in identifying bug-introducing changes. Thus, annotation graphs provide information for more accurate bug-introducing change identification at the method level.

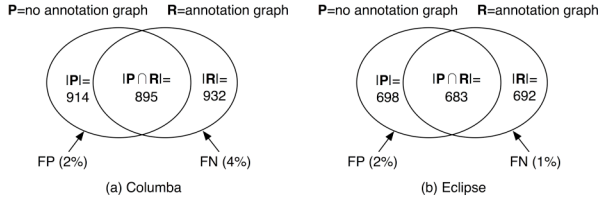

Figure 6. Bug-introducing change sets with and without annotation graph.

4.2. Non Behavior Changes

Software bugs involve incorrect behavior of the software [8], and hence are not located in the formatting of the source code, or in comments. Changes to source code format or comments, or the addition/removal of blank lines, do not affect software's behavior. For example, Figure 7 shows a change in which one blank line was deleted and an '*if condition*' was added to fix a bug. If we just apply SZZ, we identify the blank line as a problematic line and search for the origin of the blank line. We identify the revision and corresponding method of the blank line as a bug-introducing change, which is a false positive.

To remove such false positives, we ignore blank lines and comment changes in the bug-fix hunks.

```
  public void  notifySourceElementRequestor()
{
  -
  +    if (reportReferenceInfo) {
  +        notifyAllUnknownReferences();
  +    }
      // collect the top level ast nodes
    int length = 0;
```

Figure 7. Blank line deletion example in Eclipse (compiler/org/eclipse/jdt/internal/compiler/SourceElementParser.java)

Figure 8 shows the difference in identified bug-introducing change sets by ignoring comment and blank line changes. This approach removes 14%~20% of false positives.

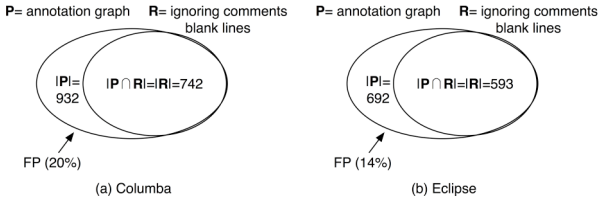

Figure 8. Identified bug-introducing change sets by ignoring comment and blank line changes.

4.3. Format Changes

Similar to the comment and blank line changes, source code format changes do not affect software behavior. So if the source code's format was changed during a bug-fix, as is shown in Figure 9, the source code format change should be ignored when we identify bug-introducing changes.

```
  - if ( folder == null ) return;
  + if (folder == null)
  +        return;
```

Figure 9. Format change example in Columba (mail/core/org/columba/mail/gui/table/FilterToolbar.java)

Unlike the comment and blank line changes, format changes affect the SCM annotation information. For example, consider the '*foo*' function changes shown in Figure 10. Revision 10 is a bug-fix change, involving repair to a faulty '*if*'. To identify the corresponding bug-introducing changes, we need to find the origin of the '*if*' at revision 10. Revision 5 only involves a formatting change to the code. If we do not ignore source code format changes, when we examine the SCM annotation information, we identify that '*foo*' at revision 5 is a bug-introducing change (a false positive). In fact, the problematic line was originally created at revision 3 (this was missed, hence a false negative). Due to inaccurate annotation information, source code format changes lead to significant amounts of false positives and false negatives. Ignoring software format changes is an important process in the accurate identification of bug-introducing changes.

```
  Revision 3
 if ( a == true ) return;

  Revision 5
   if (a == true)
          return;
Revision 10 (bug-fix)
   if (a == false)
          return;
```

Figure 10. False positive and false negative example caused by format changes.

Figure 11 compares the results of the SZZ approach with the improved approach that identifies bug-introducing changes by ignoring format changes in bug-fix hunks. Overall, ignoring source code format changes

removes 18%~25% of false positives and 13%~14% of false negatives.

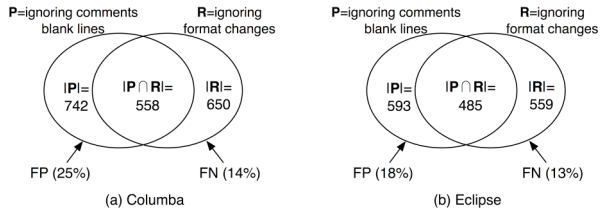

Figure 11. Bug-introducing change sets identified by ignoring source code format changes.

4.4. Remove Fix Revision Outliers

It is questionable if all the file changes in a bug-fix revision are bug-fixes, especially if a bug-fix revision contains large numbers of file changes. It seems very improbable that in a bug-fix change containing *hundreds* of file changes every one would have some bearing on the fixed bug. We observed the number of files changed in each bug-fix revision for our two projects, as shown in Figure 12. Most bug-fix revisions contain changes to just one or two files. All 50% of file change numbers per revision (between 25% and 75% quartiles) are about 1-3. A typical approach for removing outliers from data is if a data item is 1.5 times greater than the 50% quartile, it is assumed to be an outlier. In our experiment, we adopt a very conservative approach, and use as our definition of outlier file change counts that are greater than 5 times the 50% quartile. This ensures that any changes we note as outliers truly have a large number of file changes. Changes identified as outliers for our two projects are shown as '+' in Figure 12.

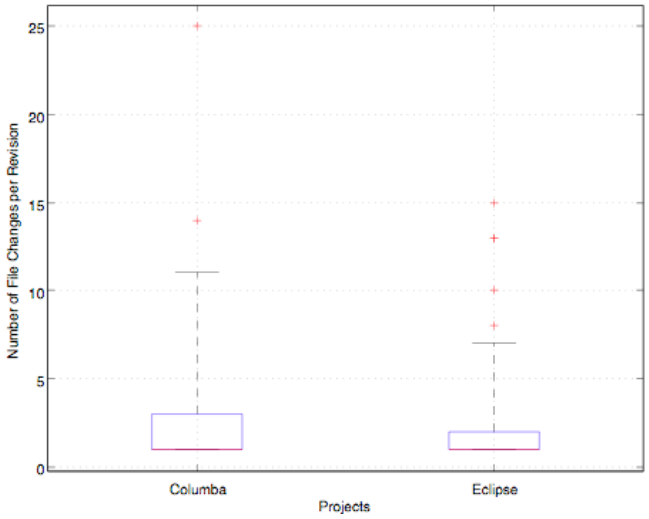

Figure 12. Box plots for the number of file changes per revision.

To ensure we were not incorrectly labeling these changes as outliers, we manually inspected each file change in the outlier revisions. We observed that most of the changes are method name and parameter name changes. For example, one parameter type changed from 'TypeDeclaration' to 'LocalTypeDeclaration', and hence the revision contains 7 file changes related to this change, as shown Figure 13.

```
- public boolean visit(TypeDeclaration
-   typeDeclaration, BlockScope scope){
+ public boolean visit(LocalTypeDeclaration
+  typeDeclaration, BlockScope scope){
```

Figure 13. Object type change example in Eclipse (search/org/eclipse/jdt/internal/core/search/matching/MatchSet.java)

As shown in Figure 14, ignoring outlier revisions removes 7%~16% of false positives. Even though most changes in the outlier revisions contain method name changes or parameter changes, it is possible that these changes are real bug-fixes. A determination of whether they are truly ignorable outliers will depend on the individual project. As a result, ignoring outlier revisions is an optional aspect of our approach for identifying bug-introducing changes.

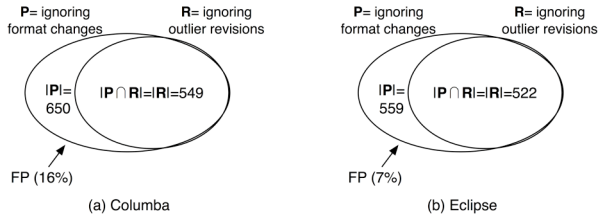

Figure 14. Bug-introducing change sets identified by ignoring outlier revisions.

4.5. Manual Fix Hunk Verification

We identify bug-fix revisions by mining change logs, and bug-fix revision data is used to identify bug-introducing changes. If a change log indicates the revision is a bug-fix, we assume the revision is a bug-fix and all hunks in the revision are bug-fixes. Then how many of them are true bug-fixes? It depends on the quality of the change log and understanding the degree of the bug-fixes. One developer may think a change is a bug-fix, while others think it is only a source code cleanup or a new feature addition. To check how many bug-fix hunks are true bug-fixes, we manually verified all bug-fix hunks and marked them as bug-fix or non-bug-fix. Two human judges, graduate students who have multiple years of Java development experience, performed the manual verification. A judge marks each bug-fix hunk of two projects (see Table 1) and another judge reviews the marks. Judges use a GUI-based bug-fix hunk verification tool. The tool shows individual hunks in the bug-fix revision. Judges read the change logs and source code carefully and decide if the hunk is a bug-fix. The total time spent is shown in Table 2.

Table 2. Manual fix hunk validation time of two human judges.

Judges	Columba	Eclipse
Judge 1	3.5 hours	4 hours
Judge 2	4.5 hours	5 hours

The most common kind of non-bug-fix hunks in the bug-fix revision involves variable renaming, as shown in Figure 15. This kind of variable renaming does not affect software behavior, but it is not easy to automatically detect this kind of change without performing deep static or dynamic analysis.

```
  deleteResources(actualNonJavaResources,fForce);
- IResource[] remaingFiles;
+ IResource[] remainingFiles;
  try {
-       remaingFiles=((IFolder)res).members();
+       remainingFiles=((IFolder)res).members();
}
```

Figure 15. Variable Renaming example in Eclipse (model/org/eclipse/jdt/internal/core/DeleteResourceElements Operation)

We identify bug-introducing changes after the manual fix hunk validation, as shown in Figure 16. Manual verification removes 4~5% false positives. Unfortunately, the manual validation requires domain knowledge and does not scale. However, the amount of false positives removed by manual verification was not substantial. We believe it is possible to skip the manual validation for bug-introducing change identification. We compare the overall false positives and false negatives using the automatic algorithms with manual validation in next section.

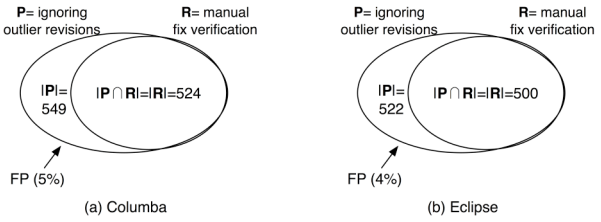

Figure 16. Bug-introducing change sets after manual fix hunk validation.

4.6. Summary

We applied the steps described in Figure 2 to remove false positive and false negative bug-introducing changes. In this section we compare the identified bug-introducing change sets gathered using the original SZZ algorithm [16] and those from our new algorithm (steps 1-5 in Figure 2). Overall, Figure 17 shows that applying our algorithms removes about 38%~51% of false positives and 14~15% of false negatives—a substantial error reduction.

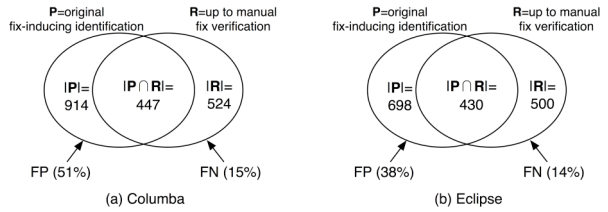

Figure 17. Bug-introducing changes identified by the original SZZ algorithm [16] (P) and by the approach (steps 1-5) proposed in this paper (R).

The manual bug-fix hunk verification gives us a good sense of how many hunks in bug-fix revisions are true bug-fixes. There is no doubt that manual bug-fix hunk verification leads to more accurate bug-introducing changes. Unfortunately, manual fix hunk verification does not scale. The reason that we examined only the first 500~1000 revisions (Table 1) is the high cost of the manual verification. Figure 18 shows the false positives and false negatives removed by applying only automatic algorithms (steps 1-4 in Figure 2). Automatic algorithms remove about 36~48% of false positives and 14% of false negatives, yielding only 1~3% difference as compared to applying all algorithms (steps 1-5 in Figure 2). Since the errors removed by manual verification are not significant, manual fix hunk verification can be skipped when identifying bug-introducing changes.

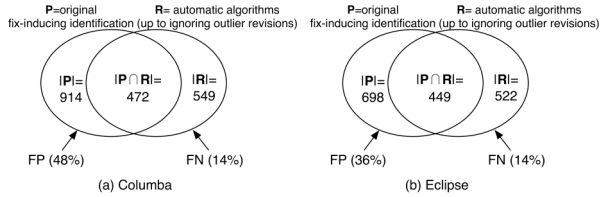

Figure 18. Bug-introducing changes identified by the original SZZ algorithm [16] (P) and by the automatable steps (1-4) described in this paper (R).

5. Discussion

In this section, we discuss the relationship between identified bug-fixes and true bug-fixes. We also discuss the relationship between identified bug-introducing changes and true bugs.

5.1. Are All Identified Fixes True Fixes?

We used two approaches to identify bug-fixes: searching for keywords such as "Fixed" or "Bug" [12] and searching for references to bug reports like "#42233" [2, 4, 16]. The accuracy of bug-fix identification depends on the quality of change logs and linkages between SCM and bug tracking systems. The two open source projects we examined have, to the best of our knowledge, the highest quality change log and linkage information of any open source project. In addition, two human judges manually validated all bug-fix hunks. We believe the identified bug-fix hunks are, in almost all cases, real fixes. Still there might be false negatives. For example, even though

a change log does not indicate a given change is a fix, it is possible that the change includes a fix. To measure false negative fix changes, we need to manually inspect all hunks in all revisions, a daunting task. This remains future work.

5.2. Are Bug-Introducing Changes True Bugs?

Are all identified bug-introducing changes real bugs? It may depend on the definition of 'bug'. IEEE defines anomaly, which is a synonym of fault, bug, or error, as: *"any condition that departs from the expected* [8]." Verifying whether all identified bug-introducing changes meet a given definition of bug remains future work.

More importantly, we propose algorithms to remove false positives and false negatives in the identified bugs. As shown in Figure 19, even though we do not know the exact set of real bugs, our algorithms can identify a set that is closer to the real bug set than the set identified by the original SZZ algorithm [16]. Even if not perfect, our approach is better than the current state of the art.

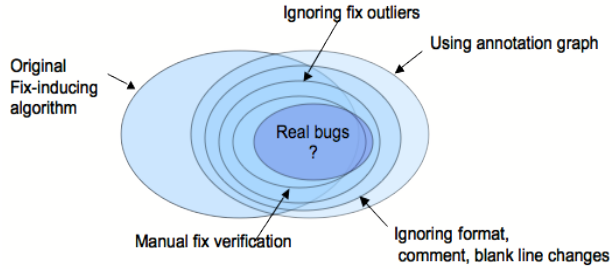

Figure 19. False positives and false negatives of each bug-introducing identification process.

5.3. Threat to Validity

There are four major threats to the validity of this work.

Systems examined might not be representative. We examined 2 systems, so it is possible that we accidentally chose systems that have better (or worse) than average false positive and negative bug-introducing changes. Since we intentionally only chose systems that had some degree of linkage between bug tracking systems and the change log (so we could determine bug-fixes), we have a project selection bias. It certainly would be nice to have a larger dataset.

Systems are all open source. The systems examined in this paper all use an open source development methodology, and hence might not be representative of all development contexts. It is possible that the stronger deadline pressure of commercial development could lead to different results.

Bug-fix data is incomplete. Even though we selected projects that have change logs with good quality, we still are only able to extract a subset of the total number of bug-fixes. For projects with a poor change log quality, the false negatives of bug-introducing change identification will be higher.

Manual fix hunk verification may include errors. Even though we selected two human judges who have multiple years of Java programming experience, their manual fix hunk validation may contain errors.

6. Applications

In the first part of this paper, we presented an approach for identifying bug-introducing changes more accurately than SZZ. In this section, we discuss possible applications for these bug-introducing changes.

6.1. Bug-Introduction Statistics

Information about bug-introducing changes can be used to help understand software bugs. Unlike bug-fix information, bug-introducing changes provide the exact time a bug occurs. For example, it is possible to determine the day in which bugs are most introduced. We can also now determine the most bug-prone authors. When combined with bug-fix information, we can determine how long it took to fix a bug after it was introduced.

Sliwerski et al. performed an experiment to find out the most bug-prone day by computing bug-introducing change rates over all changes [16]. They found that Friday is the most bug-prone day in the projects examined.

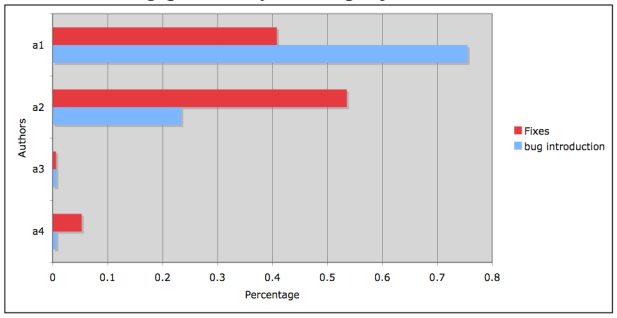

Figure 20. Eclipse author bug-fix and bug-introducing change contributions.

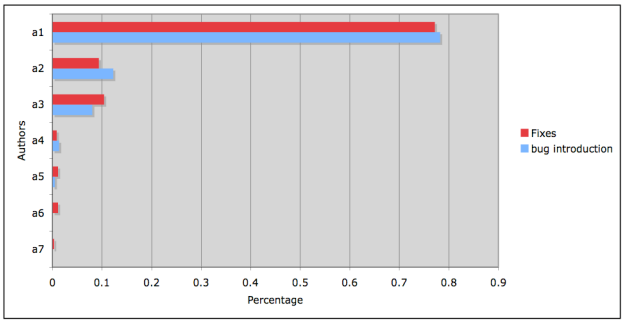

Figure 21. Columba author bug-fix and bug-introducing change contributions.

In the two projects we examined, we computed the bug-introducing change rates and bug-fix change rates per author, shown in Figure 20 and Figure 21. The figures show that rates of bug-introduction and bug-fixing are different. For example, in Eclipse, author a1 makes about 40% of all fixes, but introduces about 75% of all bugs. In

contrast, author a2 fixes far more bugs than they introduce. These numbers do not allow conclusions on the performance of individual developers: in many projects the most skillful developers are assigned to the most difficult parts; thus they are likely to introduce more bugs. Using the bug-introducing change information, we can determine the exact bug residency time, the elapsed time between initial injection of a bug and its eventual fix. The bug residency time provides a good understanding of the entire life cycle of a bug, starting with the injection of the bug in a bug-introducing change, appearance of the bug in a bug report, and the end of the bug in a bug-fix change. Previous research tries to measure the time it takes to fix a bug after a bug report has been entered, but without the bug-introducing changes, it is not possible to determine the entire life cycle of a bug. Figure 22 shows the average bug residency time using box-plots for Columba and Eclipse. For example, the box-plot for Columba shows that the average bug residency time is around 40 days, the 25% quartile is around 10 days and 75% quartile is around 100 days.

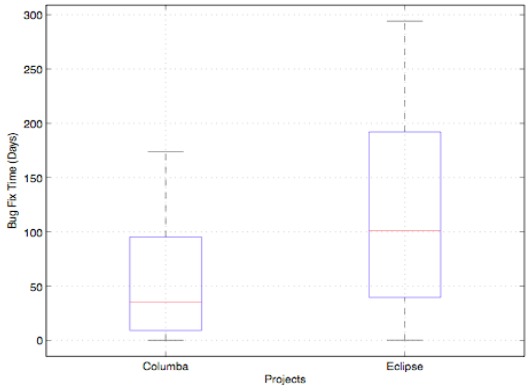

Figure 22. Average bug residency time of Columba and Eclipse.

6.2. Bug Prone Change Patterns

Since we can determine bug-introducing changes, it is possible to analyze the source code for any patterns that might exist in bug prone code. Signature changes [11] and micro pattern changes [5] are examples of source code change patterns. Suppose we identify bug-introducing changes and function signature changes as shown in Figure 23. We can then try to find correlations between signature and bug-introducing changes [11].

We analyzed micro pattern changes in Java source code using bug-introducing changes to determine what kinds of micro pattern changes introduce more/less bugs [9]. Micro patterns capture non-trivial idioms of Java programming languages [5]. This work did identify some bug prone micro patterns such as Box, CompoundBox, Sampler, Pool, Outline, and CommonState [9].

The bug prone change pattern analysis depends on having access to bug-introducing changes, since otherwise we do not know when a bug was introduced.

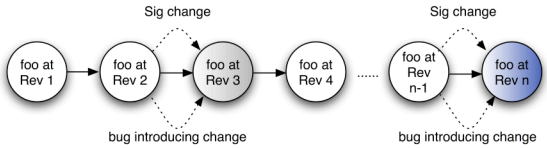

Figure 23. Bug-introducing changes and signature changes.

6.3. Change Classification

In the previous section, we provided one example of finding bug-prone source code change patterns. If a source code change pattern is consistent with bug-introducing changes, then we can use such factors to predict unknown changes as buggy or clean. Suppose we observe various change factors between 1 to n revisions as shown in Figure 24. We know which changes are bug-introducing changes and which changes are not. This permits us to train a model using labeled change factors, where the changes are labeled as being bug-introducing or clean. Using the trained model, we can predict whether future unknown changes are buggy or clean.

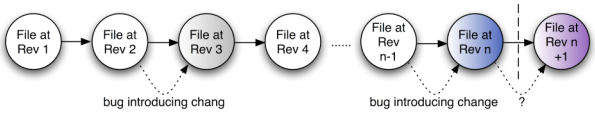

Figure 24. Predicting future changes using identified bug-introducing changes.

There are many machine learning algorithms [17] that take pre-labeled instances, train a model, and predict unknown instances using the model. Finding consistent bug-prone factors might be challenging, but it is possible to label changes and make a training data set using bug-introducing changes. Such change classification is not possible without the bug-introducing change data. Hence, one key benefit of ready access to bug-introducing changes is the ability to apply machine learning techniques to bug prediction.

6.4. Awareness Tool: HATARI

Every programmer knows that there are locations in the code where it is difficult to get things right. The HATARI tool [15] *identifies the individual risk* for all code locations by examining, for each location, whether earlier changes caused problems. To identify such changes HATARI mines bug-introducing changes automatically from version archives and bug databases. The risk of a location L is then estimated as the percentage of "bad" changes at that location:

$$\text{risk}(L) = \frac{\text{number of bug introducing changes at } L}{\text{number of changes at } L}$$

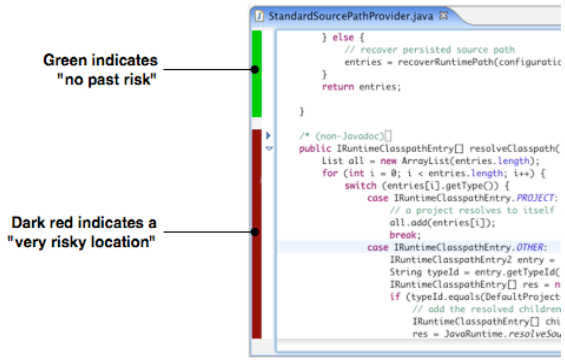

Figure 25. Source code highlights of HATARI.

Risky locations are important for maintenance, such as adding extra documentation or restructuring, and for quality assurance, because changes that occur at risky locations should get more attention. In order to support developers during these tasks, HATARI highlights such locations (see Figure 25) and provides views to browse the most risky locations and to analyze the risk history of particular locations. HATARI depends strongly on the quality of bug-introducing changes. By reducing false positives and negatives, its annotations will be improved.

7. Conclusions

Bug-introducing changes are important information for understanding properties of bugs, mining bug prone change patterns, and predicting future bugs. In this paper we describe a new approach for more accurately identifying bug-introducing changes from bug-fix data. The approach in this paper removes many false positives and false negatives as compared to the prior SZZ algorithm. Our experiments show that our approach, including manual validation, can remove 38~51% of false positives and 14% of false negatives as compared to SZZ. Omitting the manual validation and using only automatable processes, we can still remove 36%~48% of false positives and 14% of false negatives. Using our approach, we can identify bug-introducing changes more accurately than the prior SZZ algorithm, which is the current state of the art. We also showed various applications of the bug-introducing changes. We believe that software bug related research should use bug-introducing change information.

8. References

[1] J. Bevan, E. J. Whitehead, Jr., S. Kim, and M. Godfrey, "Facilitating Software Evolution with Kenyon," Proc. of the 2005 European Software Engineering Conference and 2005 Foundations of Software Engineering (ESEC/FSE 2005), Lisbon, Portugal, pp. 177-186, 2005.

[2] D. Cubranic and G. C. Murphy, "Hipikat: Recommending pertinent software development artifacts," Proc. of 25th International Conference on Software Engineering (ICSE 2003), Portland, Oregon, pp. 408-418, 2003.

[3] M. D'Ambros and M. Lanza, "Software Bugs and Evolution: A Visual Approach to Uncover Their Relationships," Proc. of 10th European Conference on Software Maintenance and Reengineering (CSMR 2006), Bari, Italy, pp. 227-236, 2006.

[4] M. Fischer, M. Pinzger, and H. Gall, "Populating a Release History Database from Version Control and Bug Tracking Systems," Proc. of 19th International Conference on Software Maintenance (ICSM 2003), pp. 23-32, 2003.

[5] J. Y. Gil and I. Maman, "Micro Patterns in Java Code," Proc. of the 20th Object Oriented Programming Systems Languages and Applications (OOPSLA '05), San Diego, CA, USA, pp. 97 - 116, 2005.

[6] M. W. Godfrey and L. Zou, "Using Origin Analysis to Detect Merging and Splitting of Source Code Entities," *IEEE Trans. on Software Engineering*, vol. 31, pp. 166-181, 2005.

[7] A. E. Hassan and R. C. Holt, "The Top Ten List: Dynamic Fault Prediction," Proc. of 21st International Conference on Software Maintenance (ICSM 2005), Budapest, Hungary, pp. 263-272, 2005.

[8] IEEE, "IEEE Standard Classification for Software Anomalies," IEEE Std 1044-1993 Dec 1993.

[9] S. Kim, K. Pan, and E. J. Whitehead, Jr., "Micro Pattern Evolution," Proc. of Int'l Workshop on Mining Software Repositories (MSR 2006), Shanghai, China, pp. 40 - 46, 2006.

[10] S. Kim, K. Pan, and E. J. Whitehead, Jr., "When Functions Change Their Names: Automatic Detection of Origin Relationships," Proc. of 12th Working Conference on Reverse Engineering (WCRE 2005), Pennsylvania, USA, pp. 143-152, 2005.

[11] S. Kim, E. J. Whitehead, Jr., and J. Bevan, "Properties of Signature Change Patterns," Proc. of 22nd International Conference on Software Maintenance (ICSM 2006), Philadelphia, Pennsylvania, 2006.

[12] A. Mockus and L. G. Votta, "Identifying Reasons for Software Changes Using Historic Databases," Proc. of 16th International Conference on Software Maintenance (ICSM 2000), San Jose, California, USA, pp. 120-130, 2000.

[13] N. Nagappan and T. Ball, "Use of Relative Code Churn Measures to Predict System Defect Density," Proc. of 27th International Conference on Software Engineering (ICSE 2005), Saint Louis, Missouri, USA, pp. 284-292, 2005.

[14] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Where the Bugs Are," Proc. of 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis, Boston, Massachusetts, USA, pp. 86 - 96, 2004.

[15] J. Śliwerski, T. Zimmermann, and A. Zeller, "HATARI: Raising Risk Awareness. Research Demonstration," Proc. of the 2005 European Software Engineering Conference and 2005 Foundations of Software Engineering (ESEC/FSE 2005), Lisbon, Portugal, pp. 107-110, 2005.

[16] J. Śliwerski, T. Zimmermann, and A. Zeller, "When Do Changes Induce Fixes?" Proc. of Int'l Workshop on Mining Software Repositories (MSR 2005), Saint Louis, Missouri, USA, pp. 24-28, 2005.

[17] I. H. Witten and E. Frank, *Data Mining: Practical Machine Learning Tools and Techniques (Second Edition)*: Morgan Kaufmann, 2005.

[18] T. Zimmermann, S. Kim, A. Zeller, and E. J. Whitehead, Jr., "Mining Version Archives for Co-changed Lines," Proc. of Int'l Workshop on Mining Software Repositories (MSR 2006), Shanghai, China, pp. 72 - 75, 2006.