SZZ Revisited: Verifying When Changes Induce Fixes

Chadd Williams
Pacific University
2043 College Way
Forest Grove, OR 97116
chadd@pacificu.edu

Jaime Spacco
Colgate University
13 Oak Dr
Hamilton, NY, 13346
jspacco@mail.colgate.edu

ABSTRACT

Automatically identifying commits that induce fixes is an important task, as it enables researchers to quickly and efficiently validate many types of software engineering analyses, such as software metrics or models for predicting faulty components. Previous work on SZZ, an algorithm designed by Sliwerski et al and improved upon by Kim et al, provides a process for automatically identifying the fix-inducing predecessor lines to lines that are changed in a bug-fixing commit. However, as of yet no one has verified that the fix-inducing lines identified by SZZ are in fact responsible for introducing the fixed bug. Also, the SZZ algorithm relies on annotation graphs, which are imprecise in the face of large blocks of modified code, for back-tracking through previous revisions to the fix-inducing change.

In this work we outline several improvements to the SZZ algorithm: First, we replace annotation graphs with line-number maps that track unique source lines as they change over the lifetime of the software; and second, we use DiffJ, a Java syntax-aware diff tool, to ignore comments and formatting changes in the source. Finally, we begin verifying how often a fix-inducing change identified by SZZ is the true source of a bug.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

1. INTRODUCTION

Automatically identifying commits that induce fixes is an important task, as it enables researchers to quickly and efficiently validate many types of software engineering analyses. Previous work on SZZ, an algorithm designed by Sliwerski et al [4] and improved upon by Kim et al [2], provides a process for automatically identifying the fix-inducing predecessor lines to lines that are changed in a bug-fixing commit.

DEFECTS'08, July 20, 2008, Seattle, Washington, USA.

SZZ is currently the best available algorithm for automatically identifying fix-inducing commits. The goal of the SZZ algorithm is first to identify the lines modified in a bug-fixing commit, and then to identify the *fix-inducing* change immediately prior to each line of the bug-fixing commit. A major remaining open question regarding the SZZ algorithm is whether the lines identified as fix-inducing by SZZ are actually the source of defects. It's possible that we need to trace the ancestry of the identified lines farther back to find the true source of bugs, or that the source of bugs are lines changed in other revisions that are control-dependent on the lines SZZ identifies as fix-inducing. Whatever the case, an empirical assessment of the accuracy of the SZZ algorithm will help improve the state of the art.

This paper makes three contributions.

- SZZ uses annotation graphs, which are imprecise at tracking lines across large hunks of modified lines, [6] to trace lines back through previous revisions of files. We use a line-number mapping approach described by Williams and Spacco in [5] (which is in turn based on work by Canfora et al [1]) to track unique lines as they evolve across multiple revisions. The added precision of line-number maps will help for cases where annotation graphs are unable to identify the true fix-inducing line.
- SZZ employs several heuristics to disregard certain types of cosmetic changes, such as changes to whitespace, indentation, comments, and some changes that split and merge source lines. However, it is not clear that their techniques can ignore all cosmetic changes in general, and they are unable to identify changes that are clearly not cosmetic but have no effect on the outcome of the program, such as `import` statements in Java or the re-naming of method parameters. We apply DiffJ[3], a Java syntax-aware diff tool, to ignore all non-executable modifications, such as changes to whitespace or comments, as well as to identify other semantic-preserving changes, such as modifications to `import` statements as well as re-ordering of method parameters.
- Finally, we begin the arduous process of verifying which lines identified by SZZ are true fix-inducing lines and which are false positives, and report on our results.

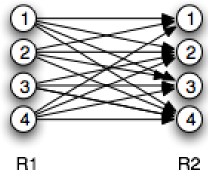

Figure 1: Annotation graph for a large modification. Note that the ancestry of each line in R2 could be any line in R1.

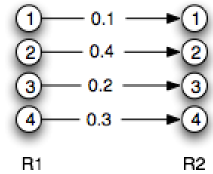

Figure 2: Line Number Map for a large modification. We are able to match each line in R2 with the corresponding line in R1 that is most likely to be its ancestor, allowing us to trace lines beyond large modifications.

2. FIX INDUCING CHANGES

Fix inducing changes, edits that are later changed during a bug fix, are found using a slight variant of the SZZ algorithm outlined in [4]. The SZZ algorithm first identifies *bug fixing* commits and the source lines changed in those commits. *Fix inducing* commits are commits previous to a bug fix commit that modify the same lines as the bug fixing commit.

2.1 SZZ Algorithm

The SZZ algorithm identifies commits that fix bugs by matching bug numbers listed in commit messages with bugs in the bug database that have been marked as FIXED. The SZZ algorithm specifies regular expressions for identifing probable bug numbers in commit messages and for identifying keywords that are likely to indicate a bug fix has occurred. Each commit that contains a probable bug number is given a score to reflect how likely the commit actually is to contain a bug fix. There are a number of analyses that are applied that can raise this score. Once such analysis uses the name of the committer and the name of the person who has been assigned the bug in the bug database. If these names match, the score for the commit is raised. The bug database we used for Eclipse did not contain the names of the committers so we skipped this analysis when determining which commits were likely to contain bug fixes.

To identify fix inducing commits, the original implementation of SZZ used the CVS *annotate* command to determine where a line changed in a bug fix commit was previously changed. Rather than use CVS annotate, we use the line mapping algorithm described by Williams and Spacco [5] to trace a changed line back through the revision history to the point of its previous change. Revisions to this line mapping algorithm are described in the following section.

The original implementation of SZZ was done against the Eclipse and Mozilla projects. This raises concerns that the regular expressions used to identify bug numbers and keywords in commit messages may be tuned to a particular set of developers. In this work we study the Eclipse project, so any concerns of this nature, while valid in general, should not affect this work.

3. LINE MAPPING ALGORITHM

The primary advantage of line number maps over annotation graphs is that annotation graphs cannot track individual lines across large modifications. For example, in Figure 1, lines 1-4 are all changed betwen revision R1 and R2. An annotation graph representation is unable to determine the precise ancestry of each line and therefore must conservatively assume that any line in R1 could have spawned any other line in R2. Figure 2 demonstrates how, if the edit distance betwen lines is not too large, line number maps could reconstruct the correspondances between these lines and therefore track the sources of changes farther back into the past. This enables us to peer beyond large modifications to find additional fix-inducing commits.

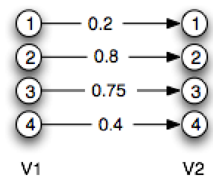

Figure 3: Normally, only lines 1 and 4 would match (as they are below the 0.4 normalized edit distance threshold). We can improve the line-mapping results by allowing lines 2 and 3 to match because they are bookended by matching lines.

We used the line mapping algorithm described by Williams and Spacco [5], with slight modifications, to trace the history of a particular line of code across multiple revisions of the file to determine when fix inducing changes happened.

As in the original algorithm, the mapping was done on a per method and per class basis. The DiffJ tool used to generate syntactic diffs is also used by the line mapper to identify renamed methods and classes. The normalized Levenstein edit distance was used to compare lines in adjacent revisions as in Figure 4. These values were used to weight edges in a bipartite graph connecting the lines in the two revision. A minimum weight bipartite matching is found to determine the best matching between the two version of the code. Pairs of lines in this graph with a normalized edit distance of less than or equal to 0.4 were deemed to be the same line (the value 0.4 was found experimentally and agrees with [1]). These lines are said to be a *valid mapping*.

3.1 Improvements to the Line Mapping Algorithm

This approach works well when the total change to a line between revisions is small, relative to the total size of the line. Large edits to a line prevent a valid mapping to be made where one should be (false negative). To map these

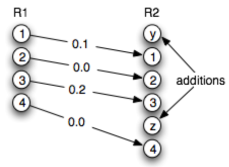

Figure 4: Lines 1-4 of R1 are matches with 1-4 or R2 based on the normalized edit distance between the lines. Lines y and z are new lines added in R2.

heavily edited lines, their neighbors are consulted. For each highly edited line its immediate neighbors above and below are insptected. If the neighbors have a valid mapping (weight <= 0.4), and the line's connection to the next revision does not cross another connection, the line is marked as a *probable mapping*. If either neighbor does not exist, i.e. the unmapped line is the first or last line in a method, but the other neighbor has a valid mapping, then line is marked as a *possible mapping*. For the work outlined in this paper, only lines marked Valid or Probable were considered to have successfully matched.

3.2 Generalization

There are many cases where a hunk of lines are highly edited and do not produce a valid mapping using the weight <= 0.4 rule. Clearly, in this case the immediate neigbhors will not have a valid mapping. We generalize the above modification to not just look at the immediate neighbors but rather to start with the immediate neighbors and move further out line by line until either a valid mapping is found or a line is found whose connection participates in a crossing. In the latter case the mapping fails and the line under consideration is left as an invalid mapping. If a valid mapping is found above and below the line under consideration with not intervening crossings, the line is marked as a *probable mapping*. The *possible mapping* works in likewise manner.

4. IDENTIFYING CHANGE TYPES

We see 23,322 changes to a source line that are fix inducing instances. The break down of change types in the fix inducing commit are shown in Table 3. The changes that have a change type of NULL indicate lines that have changed in the file but for which we were unable to find a corresponding diffj change type due to minor inconsistencies in how our line number mapping algorithm and DiffJ match source lines across files. In general, our line number mapping algorithm does a better job matching lines and DiffJ overmatches. For example, in Figure 5 a change is shown where a large number of lines are changed around a single line that remains unchanged. In the change that creates revision 24897, lines 651 and 653-656 are removed. DiffJ lists the changes as codeRemoved 648-652 and code removed 653-656. No distinction is made for line 652 (or line 655 which is equivalent). DiffJ markes this entire range as a removal of code. Our line mapping algorithm correclty maps line 646 in revision 24897 to line 652 in revision 24863. Line 645 in revision 24897 is mapped to line 647 in the previous revision.

By inspecting the DiffJ change types of each line we can remove cosmetic changes that cannot be part of a bug fix,

Table 1: Breaking across multiple lines

`int x = foo() + bar(3);`

`int x =`
`  foo() +`
`  bar(3);`

Table 2: Change Types in Bug Fix Lines

Count	Change Type
55827	methodAdded
29606	codeChanged
25103	codeAdded
21892	methodRemoved
12638	codeRemoved
6600	importAdded
3936	importRemoved
3802	innerClassAdded
2732	fieldAdded
2488	innerClassRemoved
1143	typeDeclarationAdded
1067	NULL
1044	fieldRemoved
687	parameterAdded
520	constructorAdded
445	codeChanged
431	accessChanged
244	returnTypeChanged
235	parameterTypeChanged
233	throwsAdded

Table 3: Change Types in Fix Inducing Lines

Count	Change Type
9203	methodAdded
6633	codeChanged
4388	codeAdded
696	typeDeclarationAdded
611	innerClassAdded
463	fieldAdded
423	importAdded
292	NULL
196	parameterAdded
81	returnTypeChanged
67	accessChanged
57	constructorAdded
50	throwsAdded
38	parameterTypeChanged
29	variableChanged
24	importSectionAdded
23	parameterNameChanged
13	parameterReordered
12	methodBlockAdded
8	accessAdded

Figure 5: Code Removal

/trunk/org.eclipse.jdt.debug.ui/ui/org/eclipse/jdt/internal/debug/ui/JDIModelPresentation.java

revision 24863

```
  } else {
    descriptor = new JDIImageDescriptor(JavaDebugImages.DESC_OBJS_SCOPED_BREAKPOINT, flags);
  }
} else {
  if (exception.getFilters().length == 0) {
    descriptor = new JDIImageDescriptor(JavaDebugImages.DESC_OBJS_ERROR, flags);
  } else {
    //currently do not have scoped icon
    descriptor = new JDIImageDescriptor(JavaDebugImages.DESC_OBJS_ERROR, flags);
  }

}
```

revision 24897

```
} else {
  descriptor = new JDIImageDescriptor(JavaDebugImages.DESC_OBJS_ERROR, flags);
}
```

Table 4: Change Types in Fix Inducing Lines, Deleted Lines in Bug Fixes

Change Type	Count
codeChanged	1043
codeAdded	695
methodAdded	608
fieldAdded	120
importAdded	110
NULL	71
typeDeclarationAdded	36
innerClassAdded	35
parameterAdded	32
returnTypeChanged	20
throwsAdded	15
accessChanged	14
parameterTypeChanged	11
variableChanged	11
constructorAdded	8
parameterNameChanged	5
accessAdded	5
innerInterfaceAdded	2
importSectionAdded	2
methodBlockAdded	2

thus reducing both the *number* of lines of the commit that need to be inspected, as well as reducing the number of outliers thrown out by the modified SZZ algorithm. In particular, the change types parameterReordered and parameterNameChanged are highly unlikely to actually be part of the bug fix. Additionally, formatting changes are already ignored by the DiffJ tool. These include whitespace and comment changes as well as breaking a single statement across multiple lines as shown in Table 1.

The break down of change types in the bug fix commits are shown in Table 2. Only the first 20 most common DiffJ change types are shown. Again, by inspecting the change types various lines can be disregarded as bug inducing. The change types that can be ignored are similar to those listed above.

We see 15,003 unique lines that are deleted in bug fix commits that map back to a line in a fix inducing commit. This set of changes is interesting because it is the last change to a line before it is deleted to fix a bug. The break down of change types in the fix inducing commits are shown in Table 4.

5. MANUAL VERIFICATION

Kim et al [2] describe the results of manually inspecting the commits marked as bug fix commits to confirm that SZZ was finding the correct commits. We did this as well with a small sample of commits. Additionally we inspected the fix-inducing changes that were identified to determine if they contained changes that induced the later fix. We randomly selected 25 bug fix commits mined from the first 37,000 commits applied to the trunk of the Eclipse project. These 25 commits contained a total of 50 changed lines that were mapped back to a fix-inducing commit.

As expected, 43 of the 50 lines changed in the bug fix commits appear to actually fix a bug. This result is not surprising in light of the previous manual verfication discussed above. We also inspected the changes that were marked as fix inducing. Of the 43 lines that are likely bug fixes, 33 of the associated fix inducing lines contained a change that lead to the bug being fixed. Four of the false positive fix inducing lines already contained the bug when the commit was made. The rest of the false positives stem from DiffJ not quite producing an accurate set of change and the source lines being lost by the line mapper. In instances of the former, DiffJ occasionally does not produce accurate line number information around large changes in the file. For the line mapper to lose a line, a few things can occur. The line can change radically from one revision to the next, a similar line can be added near the line, or a large number of lines can be added before the line. These are all weaknesses we need to study further.

6. THREATS TO VALIDITY

We have not fully validated our line number mapping algorithm by measuring its precision and recall. Anecdotally, we have seen that revisions that add a large number of lines around existing lines can cause the algorithm problems. Specifically, it is possible that some of the added lines will match to existing lines and cause the true descendents of those exisiting lines to be marked as new. Since we are only looking at small changes in this work, the line mapping for the revisions we are inspecting are unlikely to exhibit this problem. However, it is possible that previously in the history of the inspected line a large addition of lines has introduced this error, thus confusing the history of the line.

7. RELATED WORK

The original SZZ algorithm was defined in [4] and modified in [2]. In the second paper, a number of needs are detailed to provide improvements to the original SZZ algorithm. The first is the need to track individual source lines across revisions. Included with this is the need to identify function renaming. The second is the fact that not all modifications are fixes. Some changes are cosmetic changes such as comment or formatting changes or variable name renaming. To deal with the latter issue changes caused by comments, blank lines, and format changes are ignored. The former issue is dealt with by using annotation graphs [6]. Annotation graphs map lines from one version to the next using results return by *GNU diff*. The weakness in this approach is that *large changes* are not mapped between the two revisions of the file. A large change is defined in terms of the percent of the file affected by the change or the ratio of the length of the left and right side of the change as specified by *diff*. This is a potential weakness because the ancestory of a line cannot be traced back through a large change.

As discussed above, our line number mapping algorithm shares characteristics with the algorithm described in [1]. Their algorithm starts with the output of CVS/SVN diff which produces sets of lines that are added and deleted from the previous revision to create the new revision. The intuition is that parts of the hunks of adds and deletes actually represent modifications to the file. Similar hunks are matched using a weighted vector of tokens extracted from the hunk. Once similar hunks are identified, individual lines within the pairs are mapped using the normalized Levenshtein edit distance. Our line mapping algorithm starts by pairing methods across revisions and using a normalized Levenshtein edit distance to weight the edges in a bipartite graph mapping individual lines across versions. A minimum weight bipartite matching is then found to map each line in the previous reviions to a line in the new revision (if possible).

8. CONCLUSIONS

Automatically identifying fix inducing commits as well as bug fixing commits is an important task as we try to understand software by studying its revision history. In this paper we have discussed our implementation of the SZZ algorithm. Our implementation relies on our line number mapping algorithm rather than annotation graphs to track source code lines back through the revision history. We have shown how with our algorithm more of the lines can be mapped to a previous revision. The weakness of the annotation graphs are large hunks of code that are heavily modified. In the future, we may be able to combine the two methods to track lines through the revision history. The annotation graphs may be applied on a macro scale and use our line number mapping algorithm on a micro scale to fine tune the results, especially in large areas of heavily modified code.

Also we have used a Java-syntax aware diff tool to allow us to ignore a large variety of formatting changes. These include white space changes, changes to comments, and breaking a statement across multiple lines. This also helps to identify renamed methods to allow lines to be mapped back through a method renaming. While the DiffJ tool is not as accurate as we would like now, we are confident that it can be improved in the future.

Finally, we have begun to verify the intuition that the change previous to a bug fix introduces the bug. Our small sample has so far shown positive results, 33 of 43 lines mapped to a bug fix show evidence of the bug entering the code in the previous change. Clearly the sample size is too small to drawn conclusions from but we feel this does show that this is worthy of further work. We expect the results of this manual verification will provide guidance on how to continue to refine this technique.

9. REFERENCES

[1] G. Canfora, L. Cerulo, and M. D. Penta. Identifying changed source code lines from version repositories. In *MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 14, Washington, DC, USA, 2007. IEEE Computer Society.

[2] S. Kim, T. Zimmermann, K. Pan, and E. J. J. Whitehead. Automatic identification of bug-introducing changes. In *ASE '06: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, pages 81–90, Washington, DC, USA, 2006. IEEE Computer Society.

[3] J. Pace. A tool which compares java files based on content. http://www.incava.org/projects/java/diffj, 2007.

[4] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? *SIGSOFT Softw. Eng. Notes*, 30(4):1–5, 2005.

[5] C. Williams and J. Spacco. Branching and merging in the repository. In *MSR '08: Proceedings of the Fifth International Workshop on Mining Software Repositories*, Leipzig, Germany, 2008.

[6] T. Zimmermann, S. Kim, A. Zeller, and J. E. James Whitehead. Mining version archives for co-changed lines. In *MSR '06: Proceedings of the 2006 international workshop on Mining software repositories*, pages 72–75, New York, NY, USA, 2006. ACM.