Fuzzy Fine-grained Code-history Analysis

Francisco Servant
Virginia Tech
fservant@vt.edu

James A. Jones
University of California, Irvine
jajones@uci.edu

***Abstract*—Existing software-history techniques represent source-code evolution as an absolute and unambiguous mapping of lines of code in prior revisions to lines of code in subsequent revisions. However, the true evolutionary lineage of a line of code is often complex, subjective, and ambiguous. As such, existing techniques are predisposed to, both, overestimate and underestimate true evolution lineage. In this paper, we seek to address these issues by providing a more expressive model of code evolution, the *fuzzy history graph*, by representing code lineage as a continuous (*i.e.,* fuzzy) metric rather than a discrete (*i.e.,* absolute) one. Using this more descriptive model, we additionally provide a novel multi-revision code-history analysis — *fuzzy history slicing*. In our experiments over three real-world software systems, we found that the fuzzy history graph provides a tunable balance of precision and recall, and an overall improved accuracy over existing code-evolution models. Furthermore, we found that the use of such a fuzzy model of history provided improved accuracy for code-history analysis tasks.**

***Index Terms*—software engineering; computer aided software engineering; software maintenance; reasoning about programs;**

I. Introduction

The analysis of source-code evolution provides automatic support for a variety of software-engineering tasks, *e.g.,* identifying bug-introducing changes [49], recommending developers to fix bugs (*e.g.,* [42], [48]), identifying changes in third-party APIs [52], and discovering code clones [6]. Furthermore, developers often need to examine and understand code evolution for a wide variety of purposes, requiring high effort [12], [37].

While code-history analysis techniques support developers for numerous tasks, their accuracy is still limited by the accuracy of their underlying code-history models. Multiple techniques have been proposed to model and analyze the evolution of source code at the line-of-code level of granularity (*e.g.,* [10], [11], [44], [47]). Yet, existing techniques present potential limitations, in terms of modeling too many false positives (low precision) or too many false negatives (low recall), respectively, when compared with true code history.

In this paper, we follow the intuition that such limitations in code history models originate in the fact that existing models map lines of code in prior revisions to lines of code in subsequent revisions in an absolute and unambiguous manner. Consider the following output of the `diff` command:

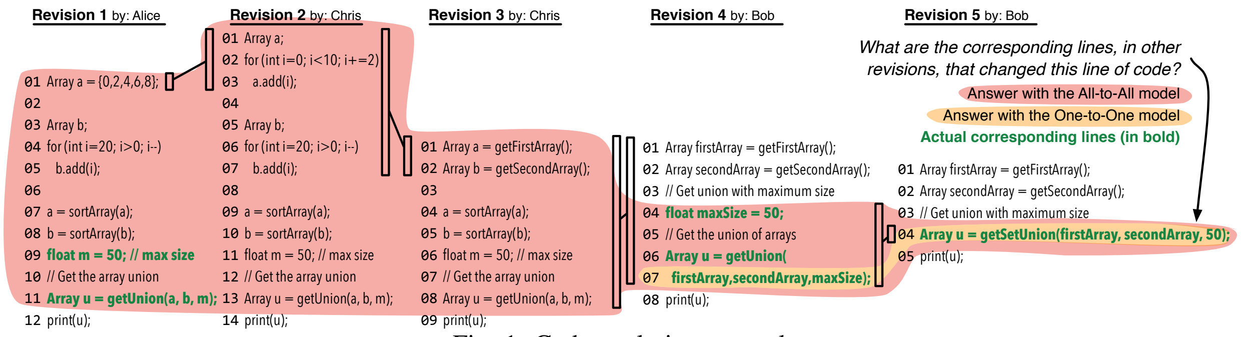
```
5,7c5,7
<  if ((a>0)&&(a<=10)) {
<       // in range
<       // valid
---
>  if ((a>0)&&
>        (a<=10)) {
>       // in valid range
```

Given this textual differencing result, existing techniques follow one of two approaches. Some existing history analyses (*e.g.,* [55]) conservatively represent the lineage of lines of code — mapping each and every candidate line of code in the older revision to each and every candidate line of code in the newer revision. In the example `diff` output, these analyses would assess that every line (i.e., lines 5, 6, and 7) in the prior revision absolutely evolved into every line (i.e., lines 5, 6, and 7) in the subsequent revision. Other techniques (*e.g.,* [10], [44], [47]) further analyze such results to disambiguate each line in the prior revision to each line in the subsequent revision. In the example `diff` output, these analyses may assess that line 5 in the prior revision evolved to line 5 in the subsequent revision, and line 6 in the prior revision evolved to line 7 in the subsequent revision; however the backward history of line 6 in the subsequent revision and the forward history of line 7 in the prior revision are lost.

As a result, existing approaches emphasize either precision or recall, but present limitations in the opposite metric. Moreover, such errors typically are compounded for analyses performed over multiple revisions, which can lead to substantially inaccurate results.

In this paper, we propose the idea that, in truth, lines of code evolve into other lines **to varying degrees** — instead of in an absolute manner. Therefore, we present a novel approach for modeling and analyzing code history in a **fuzzy** manner. We present the *fuzzy history graph* as a model of fine-grained code history that represents the varying degrees to which individual lines of code evolve into others. Additionally, we present *fuzzy history slicing* as an automatic code-history-analysis technique that takes advantage of the fuzzy history graph to improve the accuracy of a fundamental task in code history analysis: identifying all the revisions of a set of lines of code.

Through our experiments in this paper, we found that the fuzzy history graph provided higher accuracy than existing models of fine-grained code history. In addition to the improved overall accuracy, the fuzzy history graph also provided a tunable balance of both precision and recall — as opposed to existing models that emphasize only one of the two. We also found that such accuracy improvement led to higher accuracy in code-history analysis tasks. In our experiments, a common code-history analysis task — identifying bug-introducing changes with the SZZ approach [49] — provided higher accuracy with an underlying fuzzy history graph than with existing, absolute code-history models. In all, the fuzzy history graph and fuzzy history slicing allowed for: (1) multi-revision fine-grained analyses of code history that provided a flexible balance between precision and recall, and (2) higher accuracy in multi-revision code-history analysis tasks.


Revision 1 by: Alice
Revision 2 by: Chris
Revision 3 by: Chris
Revision 4 by: Bob
Revision 5 by: Bob
What are the corresponding lines, in other revisions, that changed this line of code?
Answer with the All-to-All model
Answer with the One-to-One model
Actual corresponding lines (in bold)

Revision 1 by: Alice

```
01 Array a = {0,2,4,6,8};
02
03 Array b;
04 for (int i=20; i>0; i--)
05   b.add(i);
06
07 a = sortArray(a);
08 b = sortArray(b);
09 float m = 50; // max size
10 // Get the array union
11 Array u = getUnion(a, b, m);
12 print(u);
```

Revision 2 by: Chris

```
01 Array a;
02 for (int i=0; i<10; i+=2)
03   a.add(i);
04
05 Array b;
06 for (int i=20; i>0; i--)
07   b.add(i);
08
09 a = sortArray(a);
10 b = sortArray(b);
11 float m = 50; // max size
12 // Get the array union
13 Array u = getUnion(a, b, m);
14 print(u);
```

Revision 3 by: Chris

```
01 Array a = getFirstArray();
02 Array b = getSecondArray();
03
04 a = sortArray(a);
05 b = sortArray(b);
06 float m = 50; // max size
07 // Get the array union
08 Array u = getUnion(a, b, m);
09 print(u);
```

Revision 4 by: Bob

```
01 Array firstArray = getFirstArray();
02 Array secondArray = getSecondArray();
03 // Get union with maximum size
04 float maxSize = 50;
05 // Get the union of arrays
06 Array u = getUnion(
07   firstArray,secondArray,maxSize);
08 print(u);
```

Revision 5 by: Bob

```
01 Array firstArray = getFirstArray();
02 Array secondArray = getSecondArray();
03 // Get union with maximum size
04 Array u = getSetUnion(firstArray, secondArray, 50);
05 print(u);
```

Fig. 1: Code evolution example

This paper provides the following research contributions:

- A novel model (fuzzy history graph) and analysis (fuzzy history slicing) of fine-grained code history that preserve and account for the degree to which lines of code evolve.
- Novel algorithms to perform fuzzy, fine-grained modeling and analysis of code history that improve the accuracy of current fine-grained code history models and analyses.
- An evaluation, composed of two experiments, that show: that the fuzzy history graph improved the accuracy of existing fine-grained code-history models, and that such accuracy improvement led to higher accuracy in performing code-history analysis tasks.

II. Motivation and Background

Many software-engineering tasks benefit from fine-grained code-evolution analyses. Multiple studies found industry developers to often require the study of the history of fine-grained code selections [37], for a wide variety of purposes [12], such as identifying the rationale of code [35]. In addition, multiple automatic code-evolution analyses have been proposed to support developers in diverse tasks, such as automatic identification of changes in third-party APIs [52], automatic discovery of code clones [6], automatic identification of bug-introducing changes (SZZ) [49], automatic recommendations of developers to fix bugs (*e.g.,* [42], [48]), and automatic prediction of future bugs in code locations [41]. As such, the accuracy of code evolution models and analyses impacts both researchers and practitioners in a variety of tasks.

Running Example. To illustrate such a fine-grained multi-revision code-evolution analysis and the uses to which it could be applied, consider the source-code history included in Figure 1. This example program evolved through five revisions. Lines that changed between consecutive revisions are marked with connected rectangles.

The example task that we pose is answering the set of lines throughout the code history that contributed to the current revision of Line 4 in Revision 5. To perform such an analysis, each revision can be compared, and the corresponding lines in the previous revision may be identified through manual inspection. Those lines that match can be documented, and the pairwise analysis can continue: revision-pair by revision-pair, until the beginning of the history is reached or until all trajectories reach their earliest origin.

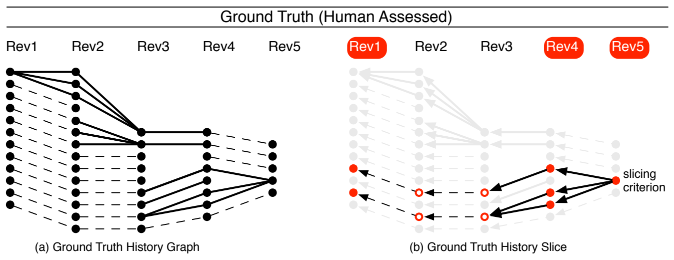

Ground Truth (Human Assessed)
Rev1
Rev2
Rev3
Rev4
Rev5
Rev1
Rev2
Rev3
Rev4
Rev5
slicing criterion
(a) Ground Truth History Graph
(b) Ground Truth History Slice

Fig. 2: Human-assessed history graph (a) and slice (b). On the slice (b), solid edges and solid nodes denote changes contributing to the slicing criterion; dashed edges and hollow nodes denote corresponding but unchanged lines.

History Models and History Analyses. In order to demonstrate and motivate the need for more descriptive models of source-code history and analyses thereof, we refer to our earlier work in which we defined an explicit model of source-code history (*i.e.,* a *history graph*) and a dependence analysis that operates on it (*i.e.,* *history slicing*). In past work, we defined the concept of *history slicing* [46], [47] to automate analysis of the correspondence of lines and changes across multiple revisions of code. The result — a *history slice* — of the *history slicing* process is defined as the complete and minimal history across all revisions of the queried set of lines of code. In keeping with the *slicing* tradition, the queried lines of code for a particular revision is called the *slicing criterion.*

For history slicing, a fine-grained model of the evolution of the code is needed. We call such a fine-grained, multi-revision model of the code history, a *history graph.* The history graph maps the corresponding lines between each two consecutive revisions, across any epoch of the code history. Once the history graph is constructed (either a priori or on-demand), the history slice can be computed for any slicing criterion by traversing the graph (in either time direction).

Example. Figure 2(a) depicts a history graph for the program history shown in Figure 1. This history graph was created without any automation and was assessed by a person from one of our empirical studies (described later in Section IV). Each column represents a revision, and in each column the nodes represent lines of code. Between each adjacent pair of revisions, edges are drawn to represent evolution of lines of the incident nodes. Solid edges are drawn to represent changed and corresponding lines, and dotted edges represent

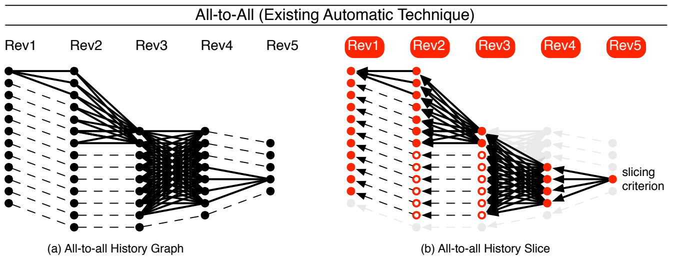

All-to-All (Existing Automatic Technique)
Rev1
Rev2
Rev3
Rev4
Rev5
Rev1
Rev2
Rev3
Rev4
Rev5
slicing criterion
(a) All-to-all History Graph
(b) All-to-all History Slice

Fig. 3: Existing all-to-all history graph and slice. Slice has high recall but low precision.

unchanged and corresponding lines.

On this graph, the slicing criterion is defined as Line 4 in Revision 5. The computation of the history slice is depicted in Figure 2(b). In this figure, the analysis is performed as a backward (in time) analysis, and the edges are now depicted as directional to convey the traversal of the reachability analysis. Nodes that represent changed lines are depicted as a solid, red dot — these constitute the resulting history slice. Nodes that did not change, but are on the trajectory of the slice, are depicted as open, red nodes.

As our example demonstrates, people can manually construct a history graph and perform the history-slicing analysis. Note that the code history may be assessed differently by another individual — such manual assessments are subject to both human fallibility and differences of opinion. In truth, often such histories can be subject to debate, and no authoritative truth is possible. More importantly, the manual task of computing the history of lines of code can be time consuming.

Existing Automated All-to-All History Analyses. To help automate such history analyses, tools and techniques have been developed and used. A commonly used tool — `diff`— can be used to determine which lines changed from one revision to another for any pair of files. Such continuous blocks of changed code are called *change hunks*. For example, `diff` computes this difference between Revision 3 and Revision 4:

```
1,8c1,7
< Array a = getFirstArray();
< Array b = getSecondArray();
<
< a = sortArray(a);
< b = sortArray(b);
< int m = 50; // max size
< // Get the array union
< Array u = getUnion(a, b, m);
---
> Array firstArray = getFirstArray();
> Array secondArray = getSecondArray();
> // Get union with maximum size
> int maxSize = 50;
> // Get the union of arrays
> Array u = getUnion(
>   firstArray,secondArray,maxSize);
```

Using such `diff` results, researchers created models of history and analyses on them by encoding the potential for all lines in the prior revision to have changed into all lines in the subsequent revision for each change hunk. For example, Zimmermann et al. [55] proposed a model, called *annotation graphs*. We generalize all such models that map all lines of the prior revision to all lines of the later revision of a change hunk — we refer to them as *All-to-all History Graphs*.

The all-to-all history graph reduces the number of *false negatives* (*i.e.,* not mapping lines of code that actually did evolve one into the other) by performing conservative mapping. As a consequence, such all-to-all models can be expected to provide *high recall*, although they may also provide *low precision*. Moreover, such conservative mapping can cause compounding

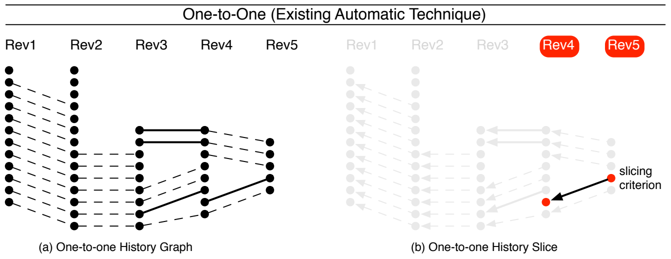

One-to-One (Existing Automatic Technique)
Rev1
Rev2
Rev3
Rev4
Rev5
Rev1
Rev2
Rev3
Rev4
Rev5
slicing criterion
(a) One-to-one History Graph
(b) One-to-one History Slice

Fig. 4: Existing one-to-one history graph and slice. Slice has high precision but low recall.

imprecision when performing reachability analyses on the graph for history slicing.

Example. For our running example, Figure 3(a) depicts the history graph constructed using an all-to-all strategy to map lines in the older revision to lines in the newer revision for each change hunk. Compared to the manually assessed (but time consuming) model, in Figure 3(b) we observe that the all-to-all analysis produced a result that wholly subsumes the correct lines in each revision. However, as expected, it includes many more lines in each revision, particularly for the oldest revision that contains all lines except one. Additionally, the number of revisions that would be recommended for examination is also an over-approximation. As a result, automatic analyses to assess risk for bugginess based on code history (*e.g.,* [41]) would overapproximate the risk of this code selection — since it would be reported as being changed five times, even though it was changed only three times. Similarly, automatic analyses to recommend expert developers for code based on its history (*e.g.,* [21]) would also overapproximate including Chris as an expert (as much as Bob), even if Chris never changed the code selection.

Existing Automated One-to-One History Analyses. To address the problems introduced by the over-approximate nature of all-to-all history models, researchers developed advanced line-mapping techniques that allow for the disambiguation of lines in change hunks (*e.g.,* [9]–[11], [44], [46], [47], [51]). The methods by which each such technique resolves which lines in the older revision correspond to which lines in the newer revision may differ, however a common technique is to use an *optimization algorithm* to find the most optimal one-to-one mapping. In this work, we generalize history models that are based on such one-to-one line mapping for change hunks — we refer to these as *One-to-one History Graphs*.

The one-to-one history graph reduces the number of *false positives* (*i.e.,* not mapping lines of code that are not truly associated). As a consequence, such one-to-one models can be expected to provide *high precision*, although they may also provide *low recall*. Moreover, the errors of under-approximation introduced by the one-to-one models can cause compounding false-negative errors when performing reachability analyses on the graph for slicing.

Example. For our running example, Figure 4(a) depicts the history graph constructed using a one-to-one strategy to map lines in the older revision to lines in the newer revision for each change hunk. Compared to the manually assessed model, in Figure 4(b) we observe that the one-to-one analysis produced a result that contains no spurious lines for any revision. However, as expected, it excludes many lines in each revision that should have been included. Additionally, the set of revisions that would be recommended for examination is also an under-approximation. Thus, automatic analyses to assess risk for bugginess based on code history (*e.g.,* [41]) would underapproximate the risk of this code selection — since it would be reported as being changed only once, even though it was changed three times. Likewise, automatic analyses to recommend expert developers for code based on its history (*e.g.,* [21]) would also underapproximate including only Bob as an expert — and therefore missing Alice, who also contributed to the selected code's history.

For each type of automatically generated history graph — all-to-all and one-to-one — errors are felt, and moreover, those errors tend to compound when used for multi-revision analyses such as history slicing. Given this background, our goal is to enable accurate multi-revision analyses.

III. Fuzzy History Analysis

In this paper, we follow the intuition that code evolution has a fundamental fuzzy nature. That is, there are varying degrees to which lines of code evolve in subsequent revisions — as opposed to the mapping of lines between revisions being binary (mapped or not). This intuition was further strengthened by the human-assessed, manual line mappings that we encountered. Moreover, there were sometimes uncertainty by individuals performing manual mapping, or dispute between multiple individuals as to the correct mapping.

As such, we created *Fuzzy History Slicing* as an automated analysis that is based on a fuzzy model of code evolution, which we naturally call the *Fuzzy History Graph*. These techniques have the goal of addressing the over-approximation errors found with all-to-all analyses and the under-approximation errors found with one-to-one analyses. We create a fuzzy approach to history slicing [46], [47] that can account for the indeterminate nature and degree of evolution of lines from one revision to the next. While past work showed the *productivity* improvements provided by history slicing [47], this paper studies the *accuracy* of fine-grained code-history analysis, contributing: (1) a novel approach to model fine-grained code-history that recognizes the fuzzy nature of code evolution; (2) a novel algorithm to compute such fuzzy code history; (3) a novel approach to analyze fine-grained code history that leverages the different extents to which lines of code evolve; (4) an algorithm to perform the fundamental code history operation of obtaining the fuzzy evolution of a set of lines of code; and evaluations of the accuracy improvement provided by the novel fuzzy code history analysis for: (5) single code evolutions, and (6) complete code histories.

Example. Before defining fuzzy history analysis, we return to our running example. Figure 5(a) depicts the fuzzy history graph that was constructed by assessing the degree to which each line in the older revision is similar to each line in the newer revision. Darker lines represent a strong correspondence, and lighter lines represent weaker correspondence. Figure 5(b) demonstrates a fuzzy history slice computed on the fuzzy history graph. The lines in each revision that are included in the fuzzy history slice are colored solid red, and their membership values are represented by the saturation of those nodes.We observe that among the lines in each revision with the highest membership, these more strongly agree with the human-assessed slice shown in Figure 2. Moreover, the revisions that would be recommended for examination also more strongly match the human-assessed revisions.

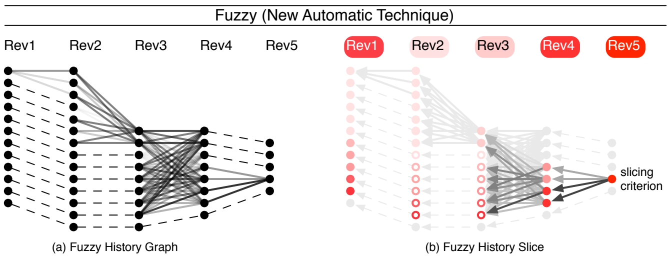


Fig. 5: Novel fuzzy history graph and slice. Slice allows for tuning precision and recall in a way that is more accurate than absolute techniques.

Fuzzy History Slicing Approach. The approach for performing fuzzy history slicing involves three processes.

Select Fuzzy History Slicing Criterion. In this process, a user or an automated analysis selects the *fuzzy history slicing criterion* that represents the lines of code and revisions of interest. The fuzzy history slicing criterion is specified as a fuzzy set that is composed of ⟨line, starting revision, ending revision⟩ tuples. Each tuple represents a line of code of interest that is characterized by the two revisions of the program between which the history analysis is requested for that line. A fuzzy history slicing criterion can include any set of lines of code, contiguous or not, from any set of files and revisions.

Build Fuzzy History Graph. This process creates our novel fine-grained model of code evolution, the *fuzzy history graph*, by analyzing the revision control system. Following our intuition that lines of code evolve into other lines *to varying degrees*, the fuzzy history graph includes a measure of *the degree to which* each older line became a new one.

As we described in Figure 5, and similarly to previous fine-grained code-evolution models, the fuzzy history graph has the shape of a multipartite graph. Each part represents a revision, each node represents a line of code, each edge represents a mapping between incident line nodes, and each edge has a weight assigned (or membership function) that estimates the *the degree to which* an older line became a new one. Edge weights can be computed in multiple ways, and different approaches to computing them can inform different analyses of fuzzy code evolution. The fuzzy history graph can be constructed once and then updated for every new revision of the code, and it can also be reused for different fuzzy-history-slice analyses.

Algorithm 1 Build Fuzzy History Graph

```
1: procedure FuzzyMapRevisions($RevisionL, RevisionR$)
2:   $Mapped \leftarrow mapUnchangedLines(RevisionL, RevisionR)$
3:   for each $\langle LinesL, LinesR\rangle \in getChangeHunks(RevisionL, RevisionR)$ do
4:     $Mapped \leftarrow Mapped \cup FuzzyMapLines(\langle LinesL, LinesR\rangle)$
5:   end for
6:   return $Mapped$
7: end procedure
8: procedure FuzzyMapLines($\langle LinesLeft, LinesRight\rangle$)
9:   $MappedL \leftarrow \{\emptyset\}$
10:  for each $lineL \in LinesLeft$ do
11:    $MappedL \leftarrow MappedL \cup FuzzyMapLine(lineL, LinesRight)$
12:  end for
13:  $MappedR \leftarrow \{\emptyset\}$
14:  for each $lineR \in LinesRight$ do
15:    $MappedR \leftarrow MappedR \cup FuzzyMapLine(lineR, LinesLeft)$
16:    for each $\langle leftR, rightR, weightR\rangle \in MappedR$ do
17:      if $\langle rightR, leftR\rangle \in MappedL$ then
18:        $\langle leftL, rightL, weightL\rangle \leftarrow MappedL.get(\langle rightR, leftR\rangle)$
19:        $MappedL.update(\langle leftL, rightL\rangle, max(weightL, weightR))$
20:      end if
21:    end for
22:  end for
23:  return $MappedL$
24: end procedure
25: procedure FuzzyMapLine($\langle l, LinesRight\rangle$)
26:  $Candidates \leftarrow \{\langle\{\emptyset\}, 0\rangle\}$
27:  for $i \leftarrow 0, MAX\_CONCATS$ do
28:    $NewCandidates \leftarrow \{\emptyset\}$
29:    for each $\langle candidate, weightC\rangle \in Candidates$ do
30:      for each $lineR \in LinesRight$ do
31:        if $lineR \notin candidate$ then
32:          $weightR \leftarrow 1 - Levenshtein(l, lineR)$
33:          $newCandidate \leftarrow concatenate(candidate, lineR)$
34:          $weightNewC \leftarrow 1 - Levenshtein(l, newCandidate)$
35:          if $(weightNewC > weightC) \& (weightNewC > weightR)$ then
36:            $NewCandidates \leftarrow NewCandidates \cup \{\langle newCandidate, weightNewC\rangle\}$
37:          end if
38:        end if
39:      end for
40:    end for
41:    $Candidates \leftarrow Candidates \cup NewCandidates$
42:  end for
43:  $Maps \leftarrow \{\emptyset\}$
44:  for each $lineR \in LinesRight$ do
45:    $Maps \leftarrow Maps \cup \{\langle l, lineR, maxWeight(\{c \in Candidates : lineR \in c\})\rangle\}$
46:  end for
47:  return $Maps$
48: end procedure
```

We created a technique to build the fuzzy history graph that uses a *branch* and *bound* optimization algorithm to minimize the textual difference between lines by using the *Levenshtein* distance [38]. Our intention with this technique is to estimate the degree to which lines of code evolved into others, and to be able to capture situations where any number of lines evolve into any other number of lines.

This technique is described in Algorithm 1 and can be divided into multiple steps. We include in parentheses the line number that implements each step. First, we map the lines that did not change at all between revisions with a weight of 1 (2). Second, we process the change hunks that remain (3–5). For each change hunk, we iterate through every line of code in the older revision (9–11) and try to map it to the most similar concatenation of lines from the newer revision (23–46). Whenever a candidate concatenation (*branch*) is less similar to the candidate line than any of its components, we stop adding lines to that concatenation — we *bound* our search (33–35). For our experiments, we also bounded our search to a maximum of three lines of code, although this setting is configurable. Third, we iterate through every newer line and assign it the maximum weight identified for a concatenation that contains it (42–44). Fourth, we perform the same operation in the opposite order, iterating every line of code in the newer revision to compare it to concatenations of lines in the older revision (12–13). Fifth, we iterate through every line pair between older and newer line, and assign it the maximum weight found for it by comparing the mappings obtained in both orders (14–19).

Algorithm 2 Slice Fuzzy History Graph

```
1: procedure FuzzyGetHistorySlice($Criterion$)
2:   $FuzzyHistorySlice \leftarrow \{\emptyset\}$
3:   for each $\langle startLine, startRevision, endRevision\rangle \in Criterion$ do
4:     $startNode \leftarrow getNode(startLine, startRevision)$
5:     $FuzzyHistorySlice \leftarrow FuzzyHistorySlice \cup \{startNode\}$
6:     $CurrentNodes \leftarrow CurrentNodes \cup \{startNode\}$
7:     $currentRevision \leftarrow startNode.Revision$
8:     while $currentRevision > endRevision$ do
9:       $VisitedNodes \leftarrow \{\emptyset\}$
10:      for each $current \in CurrentNodes$ do
11:        for each $adjacent \in current.getAdjacentNodesInPreviousRevision()$ do
12:          $adjacent.weight \leftarrow edge(adjacent, current).weight \times current.weight$
13:          if $adjacent \notin VisitedNodes$ then
14:            $VisitedNodes \leftarrow VisitedNodes \cup \{adjacent\}$
15:          end if
16:          $visited \leftarrow \{adjacent \in VisitedNodes\}$
17:          $VisitedNodes.update(adjacent, max(adjacent.weight, visited.weight))$
18:          $currentRevision \leftarrow adjacent.Revision$
19:        end for
20:      end for
21:      $FuzzyHistorySlice \leftarrow FuzzyHistorySlice \cup VisitedNodes$
22:      $CurrentNodes \leftarrow VisitedNodes$
23:    end while
24:  end for
25:  return $FuzzyHistorySlice$
26: end procedure
```

Slice Fuzzy History Graph. This process involves the analysis of the fuzzy history graph to obtain the complete history of the lines of code that were specified in the fuzzy history slicing criterion, between the revisions that were specified in it. In the process of slicing the fuzzy history graph, we also estimate the degree to which each analyzed line on each analyzed revision belongs to the history of the lines specified in the slicing criterion. Essentially, every time a new revision is visited, this operation answers in a fuzzy manner the question included in our example in Figure 1: *What are the corresponding lines, in other revisions, to a given line of code?* The traversal of the fuzzy history graph can be configured to compute a *minimal* fuzzy history slice — including only nodes with changes to the previously visited node (*e.g.,* only the solid nodes in Figure 5), or an *extended* fuzzy history slice — including both changed and unchanged nodes (*e.g.,* solid and open nodes in Figure 5).

The final output of the fuzzy history slicing process is a *fuzzy history slice*, *i.e.,* a fuzzy set of lines and revisions that are related to the lines in the fuzzy slicing criterion. The *fuzzy history slice* of a set of lines of code contains: (1) the revisions of the program that modified those lines, (2) the lines that correspond to them in such revisions, (3) a weight for each included line indicating the *degree* to which it belongs to the fuzzy history slice, and (4) the weighted edges that connect included lines in consecutive revisions, indicating the degree to which they evolved into each other.

As an example of a code-history-analysis technique that benefits from the fuzzy information stored in the fuzzy history graph, we created a novel fuzzy history slicing technique. We also used this technique for our experiments in Section IV. We implemented this technique as a *breadth-first-search* algorithm over the fuzzy history graph.

We describe this technique in Algorithm 2. We include in parentheses the line number that implements each step. First, it creates a working set of lines to visit that includes every line in the fuzzy history slicing criterion, and assigns them a weight of 1.0 (2,4,5). Second, it iterates through the working set of lines and obtains every line of code that belongs to a previous revision of it (10–11). Third, for each previous revision, it assigns each line's weight by multiplying the weight assigned to the line from which it was visited by the weight of the edge that connects them (12). Fourth, once it has performed this process for every line in the working set, it overwrites the working set of lines with the set of past revisions visited (23). Fifth, it filters the working set of lines by, when there are duplicate lines, keeping the line with the highest weight (13–21). Sixth, it also adds every line in the working set of lines to the fuzzy history slice (22). Seventh, it checks if the visited revision is the last one specified to visit (8). If it is not, it starts the process again for the working set of lines. If it is, it stops and it returns the fuzzy history slice.

IV. Evaluation

In order to evaluate how fuzzy history analysis mitigates the limitations of current models of fine-grained code history, we perform two separate experiments that complement each other. First, we study the accuracy improvement that the fuzzy history graph may provide over current models for representing code history. Then, we study the impact of such accuracy improvement over code-history analysis tasks.

Experiment 1: Fuzzy History Graph Accuracy. In our first experiment, we evaluate the accuracy with which the fuzzy history graph represents code evolution for individual changes. The goal of this experiment is to evaluate whether, and to what extent, fuzzy history graphs improve the accuracy (in terms of precision and recall) of current code-evolution models. Thus, we built fuzzy, One-to-One, and All-to-All history graphs for a set of real-world changes and measured their accuracy.

Subjects and Sampling. We used three real-world software projects to sample changes for our evaluation. We picked a diverse set of projects in terms of domain, size, and history: Apache Commons IO [2], which is a library to perform input and output functionality; Mozilla Rhino [40] is a JavaScript parser written in Java; and AspectJ [18] is an aspect-oriented programming framework for Java. Apache Commons IO has a size of 26 thousand lines of code (KLOC) and a history of 11 years, Mozilla Rhino is composed of 185 KLOC and has a history of 12 years, and AspectJ's size is around 510 KLOC and its history spans 10 years.

We randomly sampled 300 change hunks: 100 samples from each software project. We extracted the change hunks produced by all the changes in a software project by using `diff`. We sampled change hunks that had between 1 and 15 lines in the older or newer revision, so that they had a manageable size to ask developers to assess their evolution.

Human Assessment. Four participants independently and manually assessed the code change histories and defined their judgment of the correct and ideal mapping. The participants had 5–12 years of experience programming in Java to ensure that they were capable of assessing the ideal mapping.

Confirming our description in Section II, we observed the subjective nature of actual code evolution: 31% of the sampled change hunks obtained different assessments from different developers — each change hunk was assessed by 2–3 developers (see sample 127 [1] for an example of varied assessment). To reflect this fuzzy nature of code evolutions, for each older line that was assessed to evolve into a newer line, we assigned a weight to this evolution equal to the number of developers that assessed it to exist, divided by the total number of developers that assessed the change hunk. For example, if three developers assessed the change hunk from revision $r4$ to revision $r5$ of our example in Figure 1 and only one developer assessed line 4 from $r4$ as evolving into line 5 in $r5$, that mapping would have a weight of 0.33 in the human assessment for that change hunk [1].

Independent Variable 1: Change Hunk Type. As observed in Section II, the accuracy of current models is especially impacted when lines of code evolve into or from multiple other lines. In order to study both cases where current models are expected to have limitations and cases in which they are not, we classify the studied change hunks into four categories. The classification divided our sample into: 145 One-to-One change hunks, 36 One-to-Many change hunks, 34 Many-to-One change hunks, and 85 Many-to-Many change hunks.

One-to-One. Every line of code in the older revision evolved into at most one line of code in the newer revision.

One-to-Many. At least one line in the older revision evolved into multiple lines in the newer revision.

Many-to-One. Multiple lines of code in the older revision evolved into the same line of code in the newer revision.

Many-to-Many. At least one line in the older revision evolved into multiple lines in the newer revision, and more than one line in the older revision evolved into the same line in the newer revision.

Independent Variable 2: Evolution Model. We compare the accuracy of the fuzzy history graph to that of the other two existing evolution models — One-to-One and All-to-All History Graphs. We build the existing models for each change hunk by replicating a corresponding state-of-the-art technique. While other techniques could have been chosen for building each evolution model, they would similarly suffer the intrinsic limitations of modeling code evolution in a one-to-one or all-to-all fashion.

One-to-One History Graph. Each line in the older revision is modeled as evolving into at most one line in the newer revision. We build this evolution model using the technique proposed by Servant and Jones [47].

All-to-All History Graph. Every line of code in the older revision is modeled as evolving into every line of code in the newer revision. We build this evolution model using the technique proposed by Zimmermann et al. [55].

Fuzzy History Graph. Every line of code in the older revision is mapped as evolving into every line of code in the newer revision to a different extent, which is indicated by a weight. We build this evolution model using the technique described in Section III.

Independent Variable 3: Similarity Threshold. Most of the techniques that build a One-to-One model use a similarity threshold to avoid modeling evolutions of lines that are very dissimilar. In order to apply the same treatment to all the evaluated models, we discard mapped lines with similarity below the similarity threshold — calculated as one minus the Levenshtein [38] distance — after the One-to-One model and the Fuzzy History Graph are built. Because the All-to-All model does not account for the similarity of the modeled evolutions, it is not affected by the similarity threshold. We use ten different values for the similarity threshold: 0.9, 0.8, 0.7, 0.6, 0.5, 0.4, 0.3, 0.2, 0.1, and 0.0.

Dependent Variables. We measure the accuracy with which a model represents each change hunk by measuring three dependent variables: Precision (Formula 3), Recall (Formula 4) and F-measure (Formula 5). We used the fuzzy variation of the precision and recall metrics, since we assessed the evolution of a change hunk as a fuzzy set of line mappings. However, we will refer to these metrics as simply *precision* and *recall* in the rest of the paper. The fuzzy-set definitions of precision and recall depend on fuzzy cardinality and intersection. The cardinality of a fuzzy set is defined as the sum of its membership degrees, as in Formula 1. The intersection of two fuzzy sets is defined as the set of all the elements that are in common, each of which is assigned a membership degree equal to its minimum membership degree from both sets, as in Formula 2.

$$|A| = \sum \mu_A(x), \forall x \in X \tag{1}$$

$$\mu_{A \cap B}(x) = \min\{\mu_A(x), \mu_B(x)\}, \forall x \in X \tag{2}$$

$$\text{Precision} = \frac{|\text{Modeled mappings} \cap \text{Human-assessed mappings}|}{|\text{Modeled mappings}|} \tag{3}$$

$$\text{Recall} = \frac{|\text{Modeled mappings} \cap \text{Human-assessed mappings}|}{|\text{Human-assessed mappings}|} \tag{4}$$

$$\text{F-measure} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{5}$$

Results. In the results of this experiment, we expected to observe the limitations in terms of precision and recall of existing code-evolution models that we described in Section II, as well as the fuzzy history graph mitigating such limitations. We depict in Figure 6 the mean precision, recall and f-measure provided for each type of change hunk by each model in a matrix of 15 graphs. From top to bottom, each row of graphs shows mean precision, mean recall, and mean f-measure scores, respectively. From left to right, each column of graphs shows the results provided by each model for One-to-One, One-to-Many, Many-to-One, and Many-to-Many change hunks, as well as the mean results for all kinds of change hunks. The vertical axis of each graph represents mean precision, mean recall, and mean f-measure scores, in the first, second, and third row, respectively. The horizontal axis of all graphs represents the similarity threshold used. Within each graph, we use orange crosses to represent the scores obtained by the One-to-One model, red squares for All-to-All, and green triangles for the fuzzy history graph.

The One-to-One model provided the highest mean precision and the lowest mean recall of all models for all types of change hunks and similarity thresholds. We expected the One-to-One model to provide high precision, because it includes at most one mapping per older line of code, and is therefore less likely to include false positives than other models. For the same reason, we also expected the One-to-One model to provide low recall. As the similarity threshold decreased, the recall provided by the One-to-One model increased, because it included more mappings. However, the recall stabilized below the similarity threshold of 0.3 value, since the One-to-One model did not include more than one mapping per line, regardless of the similarity threshold used. This characteristic also caused the One-to-One model to provide higher recall for the One-to-One change hunks than for the other types of change hunks. These results demonstrate the limitations of One-to-One models described in Section II.

The All-to-All model provided the lowest mean precision and the highest mean recall of all models for all types of change hunks and similarity thresholds. Because the All-to-All model mapped every older line of code to every newer line of code in a change hunk, it always reached a recall of 1.0. However, it modeled many false positives, which caused it to provide low precision for all change-hunk types, particularly for One-to-One change hunks. These results show the limitations of All-to-All models described in Section II.

The fuzzy history graph provided mean precision and mean recall values between those provided by existing models. As we anticipated, the fuzzy history graph always provided higher precision than the All-to-All model and higher recall than the One-to-One model. Moreover, precision was positively correlated with the similarity threshold, and recall was negatively related with the threshold. As such, the fuzzy history graphs allow a more flexible increase of the recall provided by the One-to-One model without sacrificing as much precision as the All-to-All model. We also observed that the potential for increasing recall is much higher for One-to-Many, Many-to-One and Many-to-Many change hunks, because the One-to-One model already provides a quite high recall for One-to-One change hunks. We also observed benefits of the fuzzy history graph in terms of the f-measure, which provides a balanced metric between precision and recall. In terms of f-measure, the fuzzy history graph reached a higher value than the existing models for most change-hunk types and most similarity thresholds, specifically for similarity threshold 0.6, which is the recommended value by most techniques that build One-to-One models, *e.g.,* [5], [9], [47].

Experiment 2: Impact of Accuracy Improvement in Code History Analysis Tasks. In our second experiment, we evaluate how the accuracy improvement provided by the fuzzy history graph impacts code history analysis tasks. We chose to study the popular code history analysis task of identifying bug-introducing changes as proposed by the SZZ approach [49], since SZZ is an application area that represents very well the kinds of applications that would benefit from fuzzy history slicing. SZZ is based on fine-grained code-history analysis and is therefore potentially limited by the compounding accuracy limitations of existing code-history models.

Since SZZ was introduced, it has been adopted both by researchers and practitioners. In the research literature, SZZ has been extensively applied to study the characteristics of bug-introducing changes, *e.g.,* [4], [7], [19], [49], [53], as well

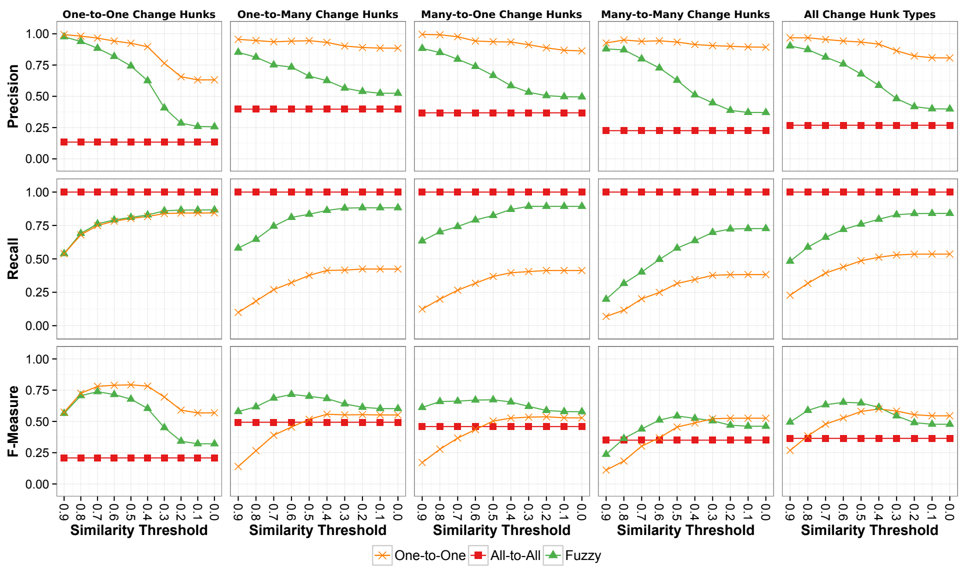


Fig. 6: Mean precision, recall, and f-measure for the different code-evolution models and change-hunk types (Higher is better).

as to provide automatic recommendations about the quality of code changes [30], [31], [34]. In addition, practitioners use SZZ to detect the origin of bugs in industrial systems [43].

We analyzed the fuzzy, One-to-One, and All-to-All history graphs for three software projects by applying fuzzy history slicing — defined in Section III — to obtain the history slice for a set of sampled lines and measured the accuracy with which we could obtain the originating line(s) for each sample.

Subjects, Sampling and Human Assessment. In this experiment, we used the same subjects as our previous experiment: APACHE COMMONS IO, MOZILLA RHINO, and ASPECTJ. For every subject, we randomly sampled slicing criteria of a line of code that experienced at least five changes in their prior history. For each slicing criterion, we manually traversed the history of the program and determined the version in which the selected line was originated by iteratively using GIT BLAME over the history of the program. We also manually assessed which lines of code corresponded to each sampled line in their originating revision by iteratively comparing each subsequent revision. Given the arduous nature of producing correct, manual assessments, we conducted this experiment with a total of 15 slicing criteria, randomly chosen, with five criteria per subject program. From slicing criterion to its first authorship, the length of the history slices spanned a median of five years of development, with a maximum length spanning nine years of development. These changed files, from criterion to first authorship, spanned on average over 100 revisions per criterion, which had to be manually inspected and traversed.

Independent Variable: Evolution Model. We used the same evolution models as in Experiment 1. Since the One-to-One and All-to-All history graphs produce discrete mappings, we assigned their edges a 1.0 weight.

Dependent Variable: SZZ Accuracy. We measured the accuracy of SZZ by evaluating the weighted lines contained in the originating revision with the Normalized Discounted Cumulative Gain (NDCG) metric [29]. NDCG is commonly used in information retrieval to assess the quality of weighted results. The NDCG metric was chosen because it provides a convenient and standard way to evaluate (potentially) weighted results in a way that accounts for positions of multiple correct and incorrect results, and accounts for the size of the result set. NDCG evaluates a weighted set as a recommendation that is sorted in terms of the elements' weights. NDCG results range from 0 — for the worst recommendation possible — to 1 — for the best recommendation possible. We used the NDCG formula proposed by Burges *et al.* [8], as in Equation 6.

$$\mathrm{NDCG}_p = \frac{\mathrm{DCG}_p(\mathrm{recomm.})}{\mathrm{DCG}_p(\mathrm{ideal\ recomm.})} \quad , \quad \mathrm{DCG}_p = \sum_{i=1}^{p} \frac{2^{\mathrm{rel}_i} - 1}{log_2(i+1)} \tag{6}$$

NDCG is calculated as the Discounted Cumulative Gain (DCG) score for a recommendation, divided by the DCG score of an ideal recommendation — in which the correct lines are contained in its top positions. In the DCG formula (Equation 6), i represents the top i^{th} position inside a recommendation, and rel_i represents the actual relevance of the item recommended in position i. We attributed a relevance of 1 to every line of code that we assessed as corresponding to the criterion. We used NDCG_{50} scores to accommodate even the largest produced recommendations in our study (50 lines). In some cases, multiple lines were recommended with the same weight — as was the case with the absolute, discrete models. In such cases, we computed the average case for the order of the recommendation (*i.e.,* as if the recommendation results were randomly ordered over an infinite number of orderings).

Results. In this experiment, the fuzzy history graph performed better than existing models. We observed the limitations of the One-to-One model and the All-to-All model propagating through multiple revisions of the code and thus decreasing the accuracy of the SZZ code-evolution analysis task. Figure 7 shows, for each subject, box plots that represent the distribution of NDCG_{50} scores provided by each history model.

The One-to-One model provided a median NDCG_{50} score of 0.0 for all subjects, which is displayed as a flat bar at the 0.0 score in Figure 7. For most of the studied lines of code, the One-to-One model did not capture their changes in enough revisions to track their history back to their originating revision — we illustrated this limitation in Figure 1. Since the One-to-One model applies a similarity threshold to decide which lines evolved into others, it prematurely ended code history.

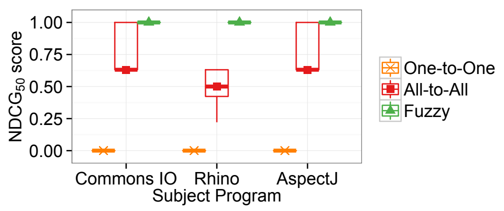


Fig. 7: Accuracy of SZZ for different code-history models. Higher scores are better.

As a result, SZZ returned no lines for the studied lines in their actual originating revisions.

The All-to-All model obtained a median NDCG$_{50}$ score of 0.5 in MOZILLA RHINO and a median NDCG$_{50}$ score of 0.63 — in APACHE COMMONS IO and ASPECTJ. Because the All-to-All model often contains multiple mappings for each line of code, the history of a single line of code included increasing numbers of lines as the traversal length increased over history. However, in the end, we were surprised to find that SZZ using the All-to-All history graph sometimes included the correct, human-assessed lines in the originating revision within a reasonably sized set of lines. We speculate that, at least for our randomly chosen slicing criteria, the compounding imprecision in the example program in Section II was found to a limited extent.

In contrast, the fuzzy history graph obtained a median NDCG$_{50}$ score of 1.0 in all subjects, which is displayed as a flat bar at the 1.0 score in Figure 7. For all the studied lines of code (except for one in ASPECTJ), SZZ recommended all their actual originating lines of code in the top position of its recommendation. The weights assigned to mappings in the fuzzy history graph allowed SZZ to select the paths with highest overall similarity in the history of the studied lines and thus recommend their actual corresponding lines in the top positions of its recommendation. Differently than the One-to-One model, the fuzzy history graph contains all potential mappings, so the history of lines of code did not end prematurely. Differently than the All-to-All model, the fuzzy history graph assigned weights to mappings to represent their strength, so SZZ could provide more informed recommendations by using the strength of the changes experienced by the line of code.

Discussion. Reflecting upon these results, we discuss the implications and other aspects of our evaluation.

Benefits of Accuracy Improvement for Code History Tasks. We observed the limitations of existing code-history models compounding as more past code revisions were included in the code-history analysis (as also illustrated in Figure 1). A representative case is one of our studied samples for RHINO, line 1018, revision 1.250 of `Interpreter.java`.

With the One-to-One model, SZZ identified only 3 past revisions containing only 1 line of code each. Since SZZ estimated that changes before those 3 past revisions modified the code too much, it stopped identifying revisions early — far from reaching the originating revision. With the All-to-All model, SZZ identified 23 past revisions, reaching the originating revision, but identifying 50 lines of code in it. Since SZZ accounted for every potential line-of-code evolution, it identified too many revisions and lines within them. With the fuzzy history graph, SZZ identified the same revisions and lines as the All-to-All model, but it assigned the highest relevance (*i.e.,* membership value) to the actual originating line of code that corresponded to the selected line.

As a consequence, users of SZZ would: (1) not find the originating revision for the selected code if using a One-to-One model, or (2) over-approximate it to a large amount of information if using an All-to-All model. For practitioners, this would imply: (1) incorrectly learning lessons from changes that were not actually bug-introducing, or (2) incorrectly learning that a bug was introduced by a large superset of changes. For researchers, such under and over approximations would introduce inaccuracies in SZZ-based research techniques, *e.g.,* estimating the quality of code changes [34]. In contrast, when using the fuzzy history graph, practitioners and researchers would correctly identify the bug-introducing changes at the top of the ranked lines returned by SZZ, and therefore would be able to learn from the correct changes.

Computational Efficiency. For all three approaches, for all subject programs, and for all history tasks, the results for each graph analysis were computed in less than a second — in a consumer laptop: 2.53 GHz Intel Core 2 Duo (P8700), 8GB RAM — since the graph is pre-computed in a database and it does not need to be kept in memory in full. Fuzzy history graphs are built offline, being a 1-time cost — with negligible-cost incremental updates for new code changes. The graph is then reused for all analyses. As such, none of these approaches (*i.e.,* One-to-One, All-to-All, nor Fuzzy) exhibited any challenges in terms of computational scalability, despite that all of our software subjects are real-world systems with over 10 years of active development. All such approaches were virtually equivalent in terms of computational time overhead.

Results Summary. In summary, our experiments revealed that the fuzzy history graph improved the accuracy of existing code-history models and the accuracy of an automatic technique to perform a code-history analysis task (SZZ).

V. RELATED WORK

Researchers proposed techniques to analyze the multi-revision evolution of code for specific purposes and at different granularities; *e.g.,* Kim *et al.* [33] and Duala-Ekoko and Robillard [16] track the history of code fragments that contain code clones to study their evolution. Herzig and Zeller [27] analyze multiple revisions of methods to predict defects. Hassan and Holt [25] analyze the evolution of methods to infer change rationale. In contrast, history slicing facilitates multi-purpose, multi-revision analyses of code evolution at the granularity of a line of code. Moreover, in this paper, we seek to improve the effectiveness of such multi-revision code analyses by representing evolution more descriptively.

A number of researchers proposed line-mapping techniques. One example of line-mapping technique that produces an All-to-All model is the annotation graph, proposed by Zimmermann *et al.* [55]. Many approaches have been proposed to model code evolution in a One-to-One fashion. Canfora *et al.* [9], [10], Chen *et al.* [11], Williams and Spacco [51] use a line-mapping technique that first performs an inexact difference of revisions, and then refines it by using an optimization algorithm. Reiss [44] proposed a group of line mapping techniques, some of which considered adjacent lines. Asaduzzaman *et al.* [5] proposed a language-independent line-mapping technique that also detects lines that evolve into multiple others, although only when they change little and are contiguous. In prior work the authors of this paper also proposed a one-to-one history graph [47] using Levenshtein [38] distance and the Kuhn-Munkres [36] algorithm. In this current work, we describe the first explicit weighted model of code evolution.

Other models represent code evolution at different granularities. Hassan and Holt model code evolution at the method-level [24], [25]. Hata *et al.* [26] proposed a model for tracking the history of methods and fields that accounts for renames. Godfrey and Zou [23] and Wu *et al.* [52] also track the history of methods and fields and detect their splits and merges. Zimmermann *et al.* [54], Fluri *et al.* [20] and Spacco and Williams [50] capture differences at the statement level. Girba and Ducasse [22] proposed a code-evolution meta-model at multiple levels of granularity. Other researchers proposed algorithms that perform the mapping over models of the program, *e.g.,* [3], [39] allowing the detection of moved code, *e.g.,* [15], [28] or providing techniques for specific domains, *e.g.,* [17]. Some techniques capture code changes by monitoring the IDE, *e.g.,* [15], [45], to model evolution with high accuracy when all developers always use the required IDE. Our model is applicable to such models. Davies *et al.* [13], [14] proposed "software bertillonage" to track the evolution of releases of code outside the revision-control system. Finally, Kim and Notkin [32] presented a survey of techniques that track program elements between revisions. To the extent of our knowledge, the model proposed in this paper is the first fine-grained code-evolution model to quantify the evolution of code and preserve it as a fuzzy measure to augment the model itself, thus enabling multi-revision analyses of code evolution at the line-of-code granularity to leverage such fuzzy measure.

VI. Threats to Validity

An external threat to validity is whether our proposed technique may capture some complex changes, such as movements of code between files, since it is based on textual differencing. We intend to study in future work the accuracy improvements that may be provided by modeling fine-grained code history with semantic approaches, *e.g.,* [3], in a fuzzy manner. In this paper, however, our goal is to study the accuracy improvements provided by a fuzzy approach to modeling and analyzing code history. The limitation for capturing moved code between files affects both our proposed textual differencing technique and all the other techniques studied in our experiments. In that respect, we believe that this limitation did not affect the results of our experiments. Additionally, the fuzzy history graph may address this limitation, since it allows its construction with other line mapping techniques, *e.g.,* [15].

Another possible external threat to validity is whether our technique would scale to other code bases. We studied software systems of up to 510 KLOC in size and up to 12 years of development, and all over 10 years of development. In all cases, for all techniques, the computational cost was well under 1 second, which demonstrates that the computational cost of such querying is negligible.

VII. Conclusions and Future Work

In this paper, we presented a novel model of source-code history at the line-of-code granularity (fuzzy history graph) and novel multi-revision analysis based upon it (fuzzy history slicing) that can improve the accuracy of the modeling and analysis of code history. The fuzzy history graph allows for a more expressive representation of code history by including a measure of the degree to which lines of code evolve.

We provided techniques to build fuzzy history graphs and analyze them (fuzzy history slicing). We demonstrated their accuracy over three real-world software projects, each having over a decade of history.

We evaluated the benefits provided by fuzzy history graphs for representing and analyzing code history. The fuzzy history graph provided a tunable balance between precision and recall, and moreover provided a higher f-measure score than afforded by existing code-history models. Also, we found that the fuzzy history graph improved the accuracy of an automatic technique for a code-history analysis task (identifying bug-introducing changes with SZZ [49]) over existing code-history models.

In practice, these results mean that software engineers could more accurately identify past bug-introducing changes to learn from them [43], and automatic SZZ-based techniques would more accurately predict the quality of code changes, *e.g.,* [30], [31], [34]. Moreover, and more importantly, this paper provides a novel theoretical fuzzy framework for modeling and analyzing code history, which also opens the door for other future fuzzy code-history analyses to emerge as well — potentially even by adapting other existing discrete approaches to a fuzzy model, *e.g.,* automatic recommendations of developers to fix bugs [42], [48], or detection of code clones [16].

In the future, we plan to experiment with additional algorithms to build the fuzzy history graph that may provide yet other benefits to the model and client analyses. We also plan to design additional code-history analysis techniques that can benefit from processing the weights of the fuzzy history graph, such as an analysis technique to identify the most relevant changes in the history of a method.

VIII. Replication

We provide our experimental dataset as a resource for future research and for experimental replication [1].

Acknowledgment

This work is supported by the National Science Foundation under award CAREER CCF-1350837.

REFERENCES

[1] Fuzzy History Slicing Experimental Package. http://people.cs.vt.edu/fservant/replication/17-ICSE-Servant-Jones.zip.

[2] Apache Software Foundation. Apache Commons IO. http://commons.apache.org/proper/commons-io, 2002.

[3] T. Apiwattanapong, A. Orso, and M. J. Harrold. JDiff: A Differencing Technique and Tool for Object-Oriented Programs. *Automated Software Engineering*, 14:3–36, 2007.

[4] M. Asaduzzaman, M. C. Bullock, C. K. Roy, and K. A. Schneider. Bug introducing changes: A case study with android. In *Working Conference on Mining Software Repositories*, pages 116–119, 2012.

[5] M. Asaduzzaman, C. K. Roy, K. a. Schneider, and M. D. Penta. LHDiff: A Language-Independent Hybrid Approach for Tracking Source Code Lines. In *International Conference on Software Maintenance*, pages 230–239, 2013.

[6] T. Bakota. Tracking the Evolution of Code Clones. In *International Conference on Current Trends in Theory and Practice of Computer Science*, pages 86–98, 2011.

[7] M. L. Bernardi, G. Canfora, G. A. Di Lucca, M. Di Penta, and D. Distante. Do developers introduce bugs when they do not communicate? the case of eclipse and mozilla. In *European Conference on Software Maintenance and Reengineering*, pages 139–148, 2012.

[8] C. Burges, T. Shaked, E. Renshaw, A. Lazier, M. Deeds, N. Hamilton, and G. Hullender. Learning to Rank using Gradient Descent. In *International Conference on Machine learning*, pages 89–96, 2005.

[9] G. Canfora, L. Cerulo, and M. Di Penta. Tracking Your Changes: a Language-independent Approach. *IEEE Software*, 26:50–7, 2009.

[10] G. Canfora, L. Cerulo, and M. D. Penta. Identifying Changed Source Code Lines from Version Repositories. In *International Workshop on Mining Software Repositories*, pages 14–21, 2007.

[11] A. Chen, E. Chou, J. Wong, A. Y. Yao, Q. Zhang, S. Zhang, and A. Michail. CVSSearch: Searching through Source Code using CVS Comments. In *International Conference on Software Maintenance*, pages 364–373, 2001.

[12] M. Codoban, S. S. Ragavan, D. Dig, and B. Bailey. Software history under the lens: a study on why and how developers examine it. In *International Conference on Software Maintenance*, pages 1–10. IEEE, 2015.

[13] J. Davies, D. M. German, M. W. Godfrey, and A. Hindle. Software bertillonage: Finding the provenance of an entity. In *Working Conference On Mining Software Repositories*, pages 183–192, 2011.

[14] J. Davies, D. M. German, M. W. Godfrey, and A. Hindle. Software bertillonage. *Empirical Software Engineering*, 18(6):1195–1237, 2013.

[15] D. Dig, K. Manzoor, R. Johnson, and T. N. Nguyen. Refactoring-Aware Configuration Management for Object-Oriented Programs. In *International Conference on Software Engineering*, pages 427–436, 2007.

[16] E. Duala-Ekoko and M. P. Robillard. Tracking code clones in evolving software. In *International Conference on Software Engineering*, pages 158–167. IEEE Computer Society, 2007.

[17] A. Duley, C. Spandikow, and M. Kim. A Program Differencing Algorithm for Verilog HDL. In *International Conference on Automated Software Engineering*, pages 477–486, 2010.

[18] Eclipse Foundation. AspectJ. http://www.eclipse.org/aspectj/, 2001.

[19] J. Eyolfson, L. Tan, and P. Lam. Do time of day and developer experience affect commit bugginess? In *Working Conference on Mining Software Repositories*, pages 153–162, 2011.

[20] B. Fluri, M. Wursch, M. PInzger, and H. C. Gall. Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction. *IEEE Transactions on Software Engineering*, 33(11):725–743, 2007.

[21] T. Fritz, G. C. Murphy, and E. Hill. Does a programmer's activity indicate knowledge of code? In *Foundations of Software Engineering*, pages 341–350, 2007.

[22] T. Gîrba and S. Ducasse. Modeling History to Analyze Software Evolution. *Journal of Software Maintenance and Evolution: Research and Practice*, 18(3):207–236, 2006.

[23] M. W. Godfrey and L. Zou. Using origin analysis to detect merging and splitting of source code entities. *IEEE Transactions on Software Engineering*, 31(2):166–181, 2005.

[24] A. E. Hassan and R. C. Holt. C-REX: an Evolutionary Code Extractor for C. In *CSER Meeting*, 2004.

[25] A. E. Hassan and R. C. Holt. Using Development History Sticky Notes to Understand Software Architecture. In *International Workshop on Program Comprehension*, pages 183–192, 2004.

[26] H. Hata, O. Mizuno, and T. Kikuno. Historage: Fine-grained Version Control System for Java. In *IWPSE-EVOL'11*, pages 96–100, 2011.

[27] K. Herzig and A. Zeller. Mining cause-effect-chains from version histories. In *International Symposium on Software Reliability Engineering*, pages 60–69, 2011.

[28] J. J. Hunt and W. F. Tichy. Extensible Language-Aware Merging. In *International Conference on Software Maintenance*, pages 511–520, 2002.

[29] K. Järvelin and J. Kekäläinen. IR Evaluation Methods for Retrieving Highly Relevant Documents. In *International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 41–48, 2000.

[30] Y. Kamei, S. Matsumoto, A. Monden, K.-i. Matsumoto, B. Adams, and A. E. Hassan. Revisiting common bug prediction findings using effort-aware models. pages 1–10, 2010.

[31] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering*, 39(6):757–773, 2013.

[32] M. Kim and D. Notkin. Program element matching for multi-version program analyses. In *International Workshop on Mining Software Repositories*, pages 58–64, 2006.

[33] M. Kim, V. Sazawal, D. Notkin, and G. Murphy. An empirical study of code clone genealogies. In *European Software Engineering Conference Held Jointly with International Symposium on Foundations of Software Engineering*, pages 187–196, 2005.

[34] S. Kim, E. J. Whitehead Jr, and Y. Zhang. Classifying software changes: Clean or buggy? *IEEE Transactions on Software Engineering*, 34(2):181–196, 2008.

[35] A. J. Ko, R. DeLine, and G. Venolia. Information Needs in Collocated Software Development Teams. In *International Conference on Software Engineering*, pages 344–353, 2007.

[36] H. W. Kuhn. The Hungarian Method for the Assignment Problem. *Naval Research Logistics Quarterly*, 2(1-2):83–97, 1955.

[37] T. D. LaToza and B. A. Myers. Hard-to-Answer Questions about Code. In *Evaluation and Usability of Programming Languages and Tools*, pages 8:1–8:6, 2010.

[38] V. I. Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady*, 10:707, 1966.

[39] J. I. Maletic and M. L. Collard. Supporting Source Code Difference Analysis. pages 210–219, 2004.

[40] Mozilla Foundation. Rhino. https://developer.mozilla.org/en-US/docs/Rhino, 1997.

[41] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. In *International Conference on Software Engineering, 2005*, pages 284–292. IEEE, 2005.

[42] M. Perscheid, M. Haupt, R. Hirschfeld, and H. Masuhara. Test-driven fault navigation for debugging reproducible failures. *Information and Media Technologies*, 7(4):1377–1400, 2012.

[43] L. Prechelt and A. Pepper. Why software repositories are not used for defect-insertion circumstance analysis more often: A case study. *Information and Software Technology*, 56(10):1377–1389, 2014.

[44] S. P. Reiss. Tracking Source Locations. In *International Conference on Software Engineering*, pages 11–20, 2008.

[45] R. Robbes and M. Lanza. Improving code completion with program history. *Automated Software Engineering*, 17(2):181–212, 2010.

[46] F. Servant and J. A. Jones. History Slicing. In *International Conference on Automated Software Engineering*, pages 452–455, 2011.

[47] F. Servant and J. A. Jones. History Slicing: Assisting Code-Evolution Tasks. In *International Symposium on the Foundations of Software Engineering*, pages 43:1–43:11, 2012.

[48] F. Servant and J. A. Jones. WhoseFault: Automatic Developer-to-Fault Assignment through Fault Localization. In *International Conference on Software Engineering*, pages 36–46, 2012.

[49] J. Sliwerski and T. Z. A. Zeller. When do changes induce fixes? *Working Conference on Mining Software Repositories 2005*, 1(1.18):1–5, 2005.

[50] J. Spacco and C. Williams. Lightweight Techniques for Tracking Unique Program Statements. In *International Working Conference on Source Code Analysis and Manipulation*, pages 99–108, 2009.

[51] C. C. Williams and J. W. Spacco. Branching and Merging in the Repository. In *International Working Conference on Mining Software Repositories*, pages 19–22, 2008.

[52] W. Wu, Y.-G. Guéhéneuc, G. Antoniol, and M. Kim. Aura: A hybrid approach to identify framework evolution. In *International Conference on Software Engineering*, pages 325–334, 2010.

[53] H. Yang, C. Wang, Q. Shi, Y. Feng, and Z. Chen. Bug inducing analysis to prevent fault prone bug fixes. In *International Conference on Software Engineering and Knowledge Engineering*, pages 620–625, 2014.

[54] T. Zimmermann. Fine-Grained Processing of CVS Archives with APFEL. In *OOPSLA workshop on Eclipse Technology eXchange*, pages 16–20, 2006.

[55] T. Zimmermann, S. Kim, A. Zeller, and E. J. Whitehead, Jr. Mining Version Archives for Co-changed Lines. In *International Workshop on Mining Software Repositories*, pages 72–75, 2006.