Evaluating Untangling Tools

Anonymous Author(s)

ABSTRACT

Developers often make code changes that contain unrelated concerns such as both bug fixes and refactoring. These tangled concerns hinder code review. This problem has been addressed by untangling tools, which automatically partition unrelated concerns into coherent groups.

Unfortunately, we don't know how effective these untangling tools are in practice because they have rarely been directly compared to one another or evaluated on real commits. Instead, untangling tools have been evaluated on synthetic commits, which may differ from real commits made by developers. If so, assessments of untangling tools' performance, strengths, and weaknesses may be misleading.

We provide a methodology and evaluation framework for realistically comparing untangling tools. We evaluated three untangling techniques on real bug-fixing commits using quantitative and qualitative methods.

The granularity of a tool's representation of a diff (the tool's data structures: diff hunks, AST nodes, etc.) strongly affects the untangling performance and whether the generated groups are coherent for a human reader. Commit characteristics such as the size of the commit also have a statistically significant effect on the performance of the untangling tools. Moreover, nested and control-dependent changes degrades the performance of untangling tools.

CCS CONCEPTS

• **Software and its engineering → Software libraries and repositories**.

KEYWORDS

empirical evaluation, evaluation framework, untangling tools, mining software repositories, version control system

ACM Reference Format:
Anonymous Author(s). 2023. Evaluating Untangling Tools. In *Proceedings of 46th International Conference on Software Engineering (ICSE 2024).* ACM, New York, NY, USA, 10 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

A tangled code change is a patch (or commit, revision, pull request, etc.) containing multiple concerns, such as both a bug fix and a refactoring [7, 10, 14]. For code review, the multiple concerns induce a higher cognitive load [1, 18, 23], making reviews of tangled code more costly and error-prone. For AI assistants, code synthesis, and bug-fixing models, the multiple concerns create noise and bias in the learning dataset, reducing performance [7, 22].

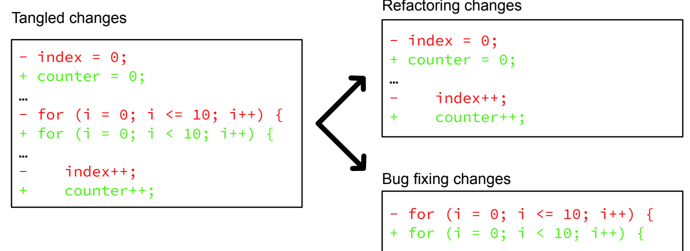

Figure 1: Tangled code changes untangled into two groups. One group contains a variable renaming and the other group contains a bug fix, where the bound of the for loop is changed.

Untangling tools aim to solve this problem by automatically clustering these concerns into distinct groups (fig. 1). Unfortunately, untangling tool evaluations have reliability and reproducibility concerns.

We found four reliability concerns in untangling tools evaluations: partial evaluations, incomparable performance metrics, synthetic commits, and small datasets. *Partial evaluations* denotes evaluations that only compare a tool to the tool they are based on, ignoring other approaches that have appeared in the literature [5, 6, 11, 12, 15, 21]. This is problematic, because it makes it difficult for researchers to compare different approaches, especially between programming languages. *Incomparable performance metrics* denotes evaluations that use different performance metrics, making it difficult to know which untangling approach is better [13]. *Synthetic datasets* denotes evaluations that use a heuristic process that artificially creates tangled commits [3, 7, 13, 16, 20]. These synthetic commits may not be representative of real tangled commits. We are unaware of work that evaluates the differences between real and synthetic commits. Developers and tool-builders using tools evaluated on synthetic datasets may be misled with false expectations about the untangling performance in practice. Additionally, past evaluations use different heuristics for creating the synthetic commits and different performance metrics to evaluate their tools, making it difficult to compare untangling tools from one evaluation to another. *Small datasets* denotes evaluations that use a small number of commits, making it difficult to generalize or perform quantitative analysis.

The reproducibility concerns we focus on are: disparate evaluation infrastructures and unavailable datasets. *Disparate evaluation infrastructures* denotes evaluations that use different evaluation infrastructures, making it difficult to compare the performance of the untangling tools. For example, some evaluations use a different programming language than others. *Unavailable datasets* denotes evaluations that use commits that are not available to the public,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICSE 2024, April 2024, Lisbon, Portugal
© 2023 Association for Computing Machinery.
ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

such as when the untangling tools are evaluated by using developers in a company [2, 5, 6, 20]. This makes it difficult for researchers to reproduce the evaluation and compare other untangling tools.

We address reliability concerns by contributing a methodology for realistically comparing untangling tools, including a performance metric to rank untangling tools, a quantitative evaluation of two untangling tools on 835 real bug-fixing commits, and a quantitative evaluation of how commit characteristics impact untangling performance.

We address reproducibility concerns by contributing an extendable evaluation framework for evaluating untangling tools, and a dataset containing the untangling results for the evaluated untangling tools. We also contribute the reimplementation of an untangling tool to enable the untangling of Java source code and C# source code.

Our experiments reveal a statistically and practically significant performance difference between untangling tools. We also find that the size of the commit measured by number of lines, number of files, and number of hunks has a statistically significant effect on the performance of the untangling tools, as well as the presence of tangled lines or tangled hunks in the commit. Additionally, the untangling performance for a tool can vary widely between synthetic and real commits.

To support open science, we provide our evaluation framework, scripts, and data https://zenodo.org/record/8206629.

2 METHODOLOGY

This section describes our methodology to evaluate untangling tools on bug-fixing commits.

2.1 Research Questions

We ask the following research questions:

RQ1 Which untangling tool performs best on real tangled commits?

RQ2 What characteristics of tangled commits affect untangling performance? This indicates where research should focus its effort.

2.2 Dataset

We evaluated the untangling tools on bug-fixing commits from the Defects4J [9] dataset, version 2.0.0. Defects4J is a collection of real bug-fixing commits that have been manually untangled to separate the bug fix from the rest of the changes in the commit. The dataset contains 835 real bug-fixing commits from 17 open-source projects.

The version control system of the *Chart* project is incompatible with the SmartCommit untangling tool, so we excluded the 26 *Chart* commits, resulting in a dataset of 809 commits. 49.6% of these commits are tangled: they contain unrelated changes that the Defects4J authors manually untangled.[1] A single line may appear in both the big fix and the tangled changes, if the Defects4J authors judged that the line contained both types of changes. We call such a line a *tangled line*. A *tangled hunk* is a hunk contains either a tangled line or at least one bug-fixing line and one non-bug-fixing line.

The remaining 50.4% of bug-fixing commits in Defects4J are atomic — they do not include any extraneous changes. We include these commits in our evaluation to determine whether tools are able to recognize atomic commits and not untangle them.

The median size of the changes in the original commit (as it appears in a project's version control history) is 46 lines changed. The median size of the unrelated whitespace changes, including blank lines and documentation, is 12 lines. The median size of non-Java source code changes, including Test files, non-Java (.xml, README.md, etc.) files, is 16 lines.

2.3 Untangling Tools

We evaluate the Flexeme [16] and SmartCommit [20] untangling tools because they are recent, never evaluated on real commits, and partially compared to each other on artificially tangled commits [13]. Section 6 discusses other untangling tools.

We also include file-based untangling as a naive untangling tool. File-based untangling groups code changes based on the file they appear in. File-based untangling is a straightforward technique and has shown good performance in previous studies [20].

2.3.1 Flexeme. Flexeme [16] untangles at the AST node level. After parsing the program into an AST, the approach leverages program dependencies and name flows [4] to create groups containing syntactically and semantically related changes. Changes such as import statements, comments, and whitespace changes are ignored by this tool.

The original tool untangles C# source code. We reimplemented Flexeme to work on Java source code based on research papers [4, 16], online documentation, and correspondence with Flexeme's author.

2.3.2 SmartCommit. SmartCommit [20] untangles a commit by clustering a diff hunk graph. A diff hunk is a group of contiguous changed lines (added lines, deleted lines, and/or changed lines) in the differences between files [17]. SmartCommit uses graph partitioning to cluster the diff hunks containing code changes that are syntactically related. The tool untangles Java source code.

2.4 Measures

We wish to compare SmartCommit and Flexeme, but they have different granularity of untangling. SmartCommit untangles at the diff hunk level, while Flexeme untangles at the AST level, i.e., each node in the AST difference graph is labeled with a group.

To compare these two approaches, we translate the results of SmartCommit and Flexeme to the line level. We use line granularity because it is a ubiquitous baseline that is easy for developers to understand and that all untangling approaches results can be translated to. For SmartCommit, we label each line with the group of the hunk it belongs to. For Flexeme, we use the label of an AST node for the line that contains the corresponding source code. One line may have multiple labels in the Flexeme labeling.

We present an example of a tangled code change occurring in Cli 29 in listing 1. The line is tangled with a *bug fix* (the first argument of `str.substring()` is updated from 0 to 1) and a *refactoring*

[1] The Defects4J authors created a minimal patch that fixes the bug, rather than trying to determine the programmer's intent with each line. For example, their minimal patches omit changes to comments.

(`str.length()` is refactored into `length`). As a result, the entire hunk (which in this case consists of a single line) is considered tangled.

Listing 1: Tangled change in Cli 29

```
@@ -67,1 +68,1 @@ class Util
-   str = str.substring(0, str.length() - 1);
+   str = str.substring(1, length - 1);
```

Untangling tools ignore (some) documentation, whitespace, and other changes. Furthermore, the Defects4J authors did not include code comments or whitespace in the minimized bug fixes that we use as ground truth. To account for these facts, our evaluation uses a whitespace-free version of Defects4J. In the whitespace-free version, all blank lines, trailing whitespace, and code comments have been eliminated. Whitespace at the beginning of lines and within lines is not changed. This whitespace-free version compiles (which tree-based and graph-based untangling tools require), but it still contains changes that are outside the purview of (some) untangling tools. When computing untangling scores, we use *cleaned* differences that also exclude import statements, test files, and non-Java files. The cleaned differences only include Java program source code. We call lines that do not appear in the cleaned diff "noncode lines"; all lines that remain in the clean diff are called "code lines". Figure 2 shows our cleaning process and evaluation pipeline.

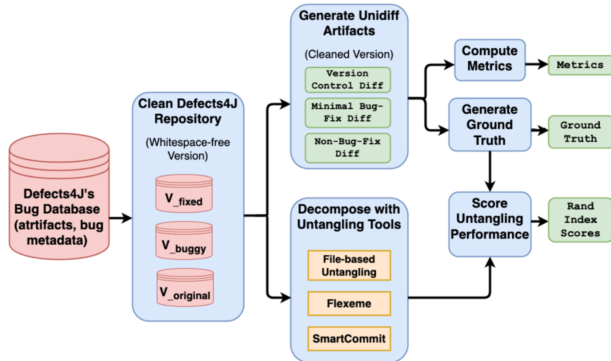

Figure 2: Evaluation Pipeline for Untangling Tools. The untangling tools are run on the whitespace-free Defects4J bug-fixing commit, while the ground truth, diff metrics, and Rand index scores are generated from the cleaned version.

2.4.1 Comparison of untangling tools (RQ1). For each bug in Defects4J, there is a ground-truth clustering created by the Defects4J developers, in which each diff line is classified into up to two groups: "fix" (minimal bug-fixing changes) and "other" (unrelated, non-bug-fixing changes). Atomic lines will be assigned to only one group, while tangled lines will be assigned to both groups.

We measure the performance of an untangling tool using the Rand index [19]. The Rand index is a measure of similarity for two clusterings of the same data. The Rand index ranges from 0 (worst) to 1 (best: the clusterings are identical). In our context (comparing against Defects4J), the Rand index gives a high score for groups containing only bug-fixing changes or non-bug-fixing changes, and penalizes groups containing both bug-fixing changes and non bug-fixing changes.

Let there be n changed lines $\{l_1, l_2, \ldots, l_n\}$ in the cleaned commit. To compute the Rand index of clusterings C_1 and C_2, first determine, for each pair of lines $\langle l_j, l_k \rangle$, whether the clustering puts the two lines in the same group or in different groups. Represent each clustering C_i as a mapping M_i from $\langle l_j, l_k \rangle$, where $j < k$, to {"same group", "different groups"}. Each mapping contains $\binom{n}{2}$ elements. The Rand index is the fraction of the mapping elements that agree:

$$\text{Rand index} = \frac{|M_1 \cap M_2|}{\binom{n}{2}} = \frac{\sum\limits_{1 \le j < k \le n} [M_1(l_j, l_k) = M_2(l_j, l_k)]}{\binom{n}{2}}$$

where $[P] = 1$ if P is true and $[P] = 0$ if P is false [8].

A commit might tangle more than two concerns (say, a bug fix, a refactoring, and a renaming). If so, the correct output is a clustering with three groups. However, the Defects4J authors were only concerned with the bug fix and put all other changes into a single group. To avoid penalizing a correct clustering with more than two groups, we postprocess the untangling tool output. We merge all groups that do not contain any bug-fixing line (according to the ground truth) together. We call this process "non-bugfix-merging".

We answer RQ1 using a quantitative analysis. We use a linear mixed-effects model to test whether the performance between the untangling tools is significantly different and we use Cohen's d to determine whether the difference is practically significant.

Linear mixed-effects are statistical models that combine fixed effects and random effects to account for the correlation of data points within nested or clustered groups. The dependent variable is the untangling performance measured by the Rand Index. The independent variable is the tool used to perform the untangling. The potential random effects are that the Defects4J bug-fixing commits address different types of bugs and that the bug-fixing commits belong to different projects. A linear mixed-effects model is suitable to answer this research question because the dependent variable is continuous and it supports the random effects that may interfere with the independent variables.

The null hypothesis (H_0) is that the untangling tools' performance is not different. We reject the null hypothesis if the p-value is less than 0.05.

2.4.2 What makes commits hard or easy to untangle? (RQ2). We measure the following 8 commit characteristics:

- **Number of code files updated**. The number of Java source code files added, deleted, or modified in the cleaned commit.
- **Number of noncode files updated**. The number of non-code files (Test.java, .xml, etc.) added, deleted, or modified in the original whitespace-free commit.
- **Number of code lines updated**. The number of Java source code lines added, deleted, or modified in the cleaned commit.
- **Number of noncode lines updated**. The number of import statements and non-Java lines added, deleted, or modified in the original whitespace-free commit.
- **Number of hunks**. The number of hunks (groups of nearby added, deleted, or modified lines) in the cleaned commit.

- **Average hunk size**. The average size of the hunks in the cleaned commit.
- **Tangled line**. Number of tangled lines in the cleaned commit.
- **Tangled hunk**. Number of tangled hunks in the cleaned commit.

We answer RQ2 using a statistical analysis. We use a linear model with random effects to test whether the measured commit metrics have a significant impact on the untangling performance, and we use Cohen's d to measure the effect size. As for RQ1, the mixed effects models accounts for the fact that the bug-fixing commits appear in different projects and that the projects are made by different developers.

The null hypothesis (H_0) is that the commit metrics have no impact on the untangling performance on real bug-fixing commits. We reject the null hypothesis if the p-value is less than 0.05.

2.5 Exploratory Manual Evaluation

To complement the quantitative analysis for RQ1 and RQ2, we manually evaluated a sample of the decompositions produced by SmartCommit and Flexeme to answer:

- What type of changes are tangled with the bug fix? Are there refactoring, formatting, co-located bugfix, maintenance, new features?
- In the untangling tool output, are the changes separated in groups that are understandable by a human?
- When the tools make mistakes, do the tools tend to over-cluster or under-cluster?
- Are there specific coding idioms that cause poor performance?

2.5.1 Procedure. We compared the results of the decompositions generated by the tool and the Defects4J ground truth against the original bug-fixing changes in the version control system. We looked at the decompositions generated by the tools on the original bug-fixing commit rather than the clean bug-fixing commit to emulate the experience a developer would have when using the tool in practice.

We viewed the original changes are in the unified diff format[2]. We retrieved them from each project repository using the commit id of the corresponding Defects4J bug-fixing commit. The decomposition results are in CSV format, where each row corresponds to a changed line denoted by its filename, its line number, and the group(s) determined by the untangling tools. The Defects4J ground truth is in the same CSV format as the decomposition results.

3 RESULTS

We ran the untangling tools on the 809 bug-fixing commits from our subset dataset of Defects4J. Flexeme untangled 680 commits and produced no decomposition on the remaining 129 commits. For 44 commits out of the 129 commits, Flexeme couldn't find a decomposition. For the remainder 85, Flexeme errored. SmartCommit untangled all 809 commits successfully. In case of an error, we use the trivial decomposition, in which all the changes belong to one group.

[2]https://en.wikipedia.org/wiki/Diff#Unified_format

3.1 Untangling Statistics

Table 1: The number of groups in the ground truth and generated by each tool after section 2.4.1 has been performed.

	Number of groups			
Treatment	Min.	Max.	Median	Std. dev.
File Untangling	1	16	1	1
Flexeme	1	21	2	2
Ground Truth	1	2	1	0
SmartCommit	1	9	1	0

Table 1 shows the number of groups generated by each tool. The median number of groups is 1 or 2, which concords with the ground truth. However, the tools have high maximums, especially Flexeme. It is difficult for developers to understand such untangling results.

Given that our dataset is composed of bug fixes, an untangling tool should produce 2 or 3 groups per commit: 1 group for the fixes, and 1 or 2 more groups for non-bug fix changes. The minimum of 1 group for file-based untangling and the ground truth is due to the fact that some Defects4J bugs have no test file changes, only one source file change. The maximum of 2 for the ground truth accounts for the representation of tangled lines.

Table 2: The size of the groups in the ground truth and generated by each tool.

	Size of groups			
Treatment	Min.	Max.	Median	Std. dev.
File Untangling	1	375	6	23
Flexeme	1	486	3	19
Ground Truth	1	483	5	24
SmartCommit	1	285	6	23

Table 2 shows the size of the groups for each tool, including the ground truth for reference. We can observe that the median and the standard deviation of SmartCommit, file-untangling and the ground truth are similar whereas Flexeme has a lower median and standard deviation. We hypothesize that Flexeme creates smaller groups compared to the other tools and the ground truth.

3.2 RQ1: Which untangling approach performs best on real tangled commits?

File-based untangling performs best with a median Rand index score of 0.93. SmartCommit has a median of 0.89, and Flexeme has 0.52. There is a wide performance gap between SmartCommit and Flexeme.

We hypothesize that file-based untangling has the best untangling performance because in the domain of bug-fixes, it is common that the fix is in a single file, which makes file-based untangling trivially perfect. Also, recall that our experimental methodology omits non-code files, including test files. In the original (non-whitespace-free) VCS changes (excluding test files changes because the test files are not manually untangled in Defects4J), file-based untangling has a Rand index score of 0.75, SmartCommit has 0.73, and Flexeme has

0.53. For this reason we encourage researchers to take a holistic approach when designing untangling tools and not just focus on source code changes. Other changes, such as documentation and comments, can have a significant impact on the performance of the untangling tools.

The ranking of the tools is the same when we drop the commits that errored to simulate a situation in which all the tools' bugs are fixed. In this scenario, Flexeme is 0.52. File-based and SmartCommit are unchanged because there were no errors.

The cleaned diffs have a smaller number of files changed on average. In this scenario, all the tools have a longer tail distribution as there is a large number of bug projects that have changes on only 1 file improving the tool's chance of getting a high decomposition score. To account for this we also report the average untangling performance. We find that file-based untangling gets a performance of 0.78, SmartCommit gets an average performance of 0.77, and Flexeme gets a performance of 0.58. While the average performance drops for SmartCommit and file-based untangling and increases for Flexeme, the ranking of the tools does not change.

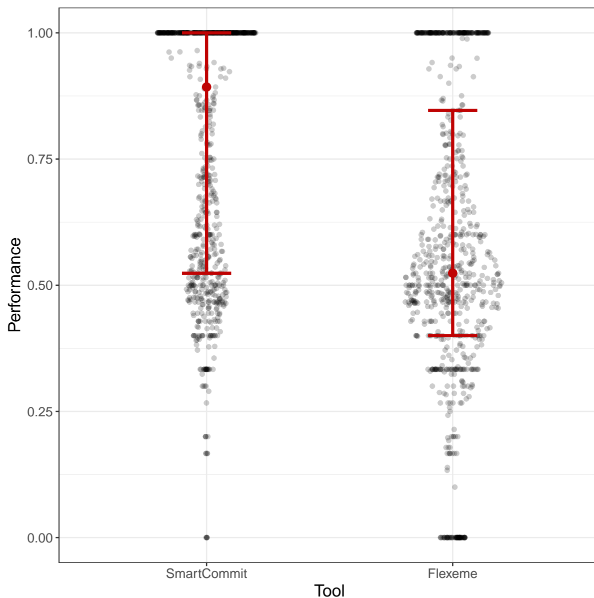

Figure 3: Performance comparison for SmartCommit and Flexeme. Each dot represents the Rand index for a Defects4J bug. The red whiskers represent the interquartile range.

Figure 3 shows the distribution of untangling performance. The performance difference is statistically significant ($p < 0.05$), has a medium effect size that is relatively significant (Cohen's d = 0.69), and an adjusted R^2 estimate of 0.107.

A score of 1 occurs when the tool decomposition is exactly the same as the ground truth. For many commits, both tools have a score of 1, which explains the dense cloud of points at the top of the graph for both Flexeme and SmartCommit.

A score of 0 occurs when the ground truth contains exactly two lines and the tool doesn't have the same clustering as the ground truth. For instance, in bug Closure 14, the ground truth contains two changed lines in the same group. Flexeme classifies only one changed line, so the other is added automatically in another group. This results in a Rand index of 0.

We determined the effect of project and bug by comparing two linear models. The first model has the untangling performance as the dependent variable and the tool as the independent variable. The second model has the same dependent and independent variable, and models the difference between bugs the project they belong to as random effects. We found that accounting for the random effects didn't change the statistical significance of the treatment effect and only increased the adjusted R^2 estimate by 0.03. We use the adjusted R^2 estimate to compare the models because the adjusted score takes into account the number of variables in the model, which is different between our two models. Table 3 summarizes the two models.

Table 3: Comparing two models shows that project and bug have no significant effect on the results. Both models have the same dependent and independent variables. The second model has the Bug Id and the project as random effects.

Model	Coefficient		P-Value	Adjusted R^2
	Flexeme	SmartCommit		
Without random effects	0.58	0.19	<2e-16	0.11
With random effects	0.59	0.19	<2e-16	0.14

The SmartCommit paper [20] reports two accuracies: a field study accuracy and a controlled experiment accuracy. They measured the accuracy using the Rand Index on diff hunks. The field study untangles real commits and the controlled experiment uses a synthetic dataset. For the field study, the SmartCommit paper reports a median accuracy of 74.70% and 70.45% for the two projects studied. For the controlled experiment, SmartCommit achieves a median accuracy per project in the range of 71.00 to 83.50%. Overall the median accuracy is 79.5%. In our evaluation, we observe a median performance of 89% which is also measured by the Rand Index, but at the line level. We hypothesize that the difference in performance is due to removing the non-code changes from the VCS as evidenced by the drop to 73% when untangling on all the changes (except tests). 73% is in the lower range of the accuracy reported by SmartCommit in their experiments. We hypothesize that the evaluation on a finer granularity identifies more tool mistakes, resulting in a lower Rand index score.

The Flexeme paper [16] reports a median accuracy of 0.81 on a synthetic dataset. The accuracy is measured using the node accuracy, which measures the percentage of nodes in the diffed AST that have been labeled with the correct group among the nodes that have changed. This is incomparable with the Rand index, so we cannot compare the two performances. However, our experimental results are qualitatively different from Flexeme's. The Flexeme paper reports *good* performance we observe only *average* performance.

Answer to RQ1: SmartCommit is significantly better at untangling real bug fixes than Flexeme.

Table 4: Commit characteristics effect on performance. Significant values are bolded. Coefficients are simplified to their sign. Cohen's d is also bolded if the effect size if at least medium (Cohen's $d > 0.5$) only if the characteristic is already statistically significant. Cohen d measures the effect size.

Metric	Flexeme			SmartCommit		
	p-value	Coefficient	Cohen's d	p-value	Coefficient	Cohen's d
Code files	0.77	+	-0.91	**4e-06**	-	**-0.73**
Noncode files	0.45	+	-0.47	0.9	-	-0.27
Code lines	0.07	+	-0.79	**0.004**	-	**-0.79**
Noncode lines	0.73	+	-0.64	0.05	-	-0.64
Number of hunks	**0.01**	+	**-0.71**	**5e-04**	-	**-0.68**
Average hunk size	0.83	-	N/A	0.5	-	N/A
Tangled line	0.40	-	0.04	**8e-09**	-	0.17
Tangled hunk	0.13	-	-0.05	**<2e-16**	-	0.15

3.3 RQ2: What characteristics of tangled commits affect untangling performance?

We computed eight commit characteristics on the whitespace-free and clean diffs.

Table 4 measures the impact of the commit characteristics on untangling performance. We used a linear model. A commit characteristics has a significant impact on performance if its p-value is smaller than 0.05 ($p < 0.05$).

We measured the impact of each commit characteristic against performance separately from the other metrics because we the results were slightly different when adding all the commit characteristics to the model at once. We speculate that model is not able to fit all the metrics at the same time for the amount of data points.

As the number of hunks increases, the performance increase for Flexeme and decreases for SmartCommit.

It is surprising that as the commit size grows, Flexeme's performance improves. SmartCommit's coefficients are negative, which is what we expected.

All the metrics for SmartCommit have a negative coefficient which mean, for example, that as the number of code files increases, the performance decreases. All the significant metrics also have a medium effect size as denoted by Cohen's d between 0.5 and 0.8, except for the tangled line and tangled hunk that have a negligible effect size and therefore are not practically significant. The negligible effect size might be due to the fact that these metrics are skewed at 0. Only 19% of the commits have tangled lines and 35% of the commits have tangled hunks.

These results suggest that for SmartCommit, larger commits (code files, code lines, and number of hunks) are harder to untangle than smaller commits. In addition, commits with tangled lines and tangled hunks are harder to untangle than commits with no tangled lines and no tangled hunks. However, there doesn't seem to be a large effect in practice on performance when the number of tangled lines and tangled hunks increases from, say, 1 to 2 or more.

Answer to RQ2: Commit characteristics have a significant impact on the performance of the untangling tools. Larger commits are statistically significantly harder to untangle, and the presence of tangled lines and tangled hunks also make commits harder to untangle. Moreover, different tools are affected by different commit characteristics.

3.4 Exploratory Manual Analysis

We sampled 10 bugs at random from Defects4J (table 5) and manually evaluated the results of the untangling tools.

3.4.1 When the tools make mistakes, do the tools tend to over-cluster or under-cluster?

SmartCommit consistently under-clusters the changes. When the changes contain files other than Java files, the non-Java files are always grouped together. i.e., one or more groups for the Java files and one group for the non-Java files. One the other hand, Flexeme consistently over-cluster the changes, often generating more than 2 groups for even a couple of changed lines such as for the MATH 3 bug-fixing commit, shown in listing 2, where it classified the line `if (len == 1 {)` in two groups and the line `return a[0] * b[0];` in three groups. The two lines have two overlapping groups. Only the last line is in a group that the first line is not in.

Listing 2: Diff of `MathArrays.java` for the MATH 3 bug-fixing commit.

```
@@ -818,6 +818,11 @@ public class MathArrays {
        throw new DimensionMismatchException(len,
            b.length);
    }

+   if (len == 1) {
+     // Revert to scalar multiplication.
+     return a[0] * b[0];
+   }
+

    final double[] prodHigh = new double[len];
    final double[] prodLow = 0;
```

These observations are supported by the statistics in table 1 and table 2. The high standard deviation, high median group count, and low standard deviation for group size for Flexeme indicates that Flexeme tends to over-cluster, generating many small groups. For SmartCommit, the high standard deviation and high median group

Table 5: Defects4J bug-fixing commits that we manually evaluated. The table shows the commit's project, its associated Defects4J bug id, the number of bug-fixing changed lines, the number of non-bug-fixing changed lines, the number of groups generated by SmartCommit and Flexeme, and the types of non-bug-fixing changes.

Project	Bug id	Bug fix changes	Other changes	Flexeme groups	SmartCommit groups	Non-bug fixing changes types
Cli	28	2	0	1	2	N/A
Cli	33	19	12	6	1	Documentation
Closure	64	8	2	1	2	Documentation
Csv	8	13	11	4	2	Documentation, Refactoring
Lang	8	1	16	4	2	Refactoring, Related changes
Lang	57	2	0	1	1	N/A
Math	3	3	1	5	2	Documentation
Math	16	23	23	10	1	Documentation, Related changes
Math	28	4	35	12	1	Documentation
Math	34	2	3	7	2	Documentation

size indicates that SmartCommit tends to under-cluster, creating few big groups.

Regarding the 2 atomic commits in our sample, both SmartCommit and Flexeme did not over-cluster or under-cluster the changes, even when more than one line changed.

From our observations, we speculate that the internal representations used in the clustering algorithms are the main cause for under-clustering or over-clustering when the tools make mistakes. SmartCommit groups elements at the hunk level, which constrains the number of groups possible. For untangling the bug fixes, this is advantageous as the sampled fixing changes were typically on contiguous lines. On the other hand, Flexeme groups elements at the AST-node level, enabling any node on the same line to be grouped differently, as evidenced by the high number of groups reported in table 1.

3.4.2 Are there specific coding idioms that cause poor performance? Both tools perform poorly when the changes were tangled with unrelated dependent code changes or unrelated nested code changes (for example, the bug fix changes an if statement and the body of the if statement contains unrelated changes). Untangling this type of tangled changes is difficult not only for tools but also for developers.

3.4.3 What are the type of changes tangled with the bug fix? Are there refactoring, formatting, co-located bugfix, maintenance, new features? Flexeme was better at untangling refactoring changes due to the number of groups it created while SmartCommit was limited at the hunk level. Related changes that were not part of the bug fix (maintenance) were the most difficult to untangle for both tools. For documentation, SmartCommit included documentation changes with the bug fix because the changes were often in the same hunk. Flexeme didn't group documentation changes since it classifies changes based on the AST nodes, which doesn't include commented lines.

3.4.4 Are the changes separated in groups that are understandable by a human? SmartCommit classifies the hunks so the changes are always contiguous lines, which is familiar to developers. The limitation is when tangled changes occurs in the same hunk as the bug fix, which happened 8 times out of 10 in our sample. SmartCommit will only produce one group. Flexeme, on the other hand can create many groups for the same lines, making it hard for developers to understand why there are multiple overlapping groups, and what the groups represent.

Overall, both tools are able to untangle the bug fixes with some limitations stemming from the internal representations used in the clustering algorithms. SmartCommit is limited to clustering tangled changes at the hunk level and cannot untangle changes that are in the same hunk. On the other hand, Flexeme can untangle changes at the AST node level, but generates too many groups that are incorrect. Even if the content of the groups generated by Flexeme were correct, the number of groups generated makes it hard for developers to understand the purpose of each group.

We recommend that untangling tools researchers consider how the untangling results will be presented to developers so they can read the changes, understand the purpose of each group and use the generated groups to create atomic commits. Future work should also investigate how to represent tangled changes on the same line.

3.5 Implications in practice

The effects of the statistically and practically significance untangling performance difference can be observed in the exploratory manual analysis table 1. SmartCommit creates cohesive groups that are close to the manual untangling done by the Defects4J authors, while Flexeme generates too many groups, containing too few lines and lines are often overlapping between groups, which is rarely the case in the manual untangling.

In the context of a bug-repair tool, this means that SmartCommit will create groups that are more likely to be correct than Flexeme, and more likely to reflect what developers would do if they were to untangling the bug-fixes, compared to the many groups generated by Flexeme that are not likely to be represent valid bug-fixes. If applied to pull request, this means that SmartCommit will create groups that are likely to be helpful to developers while Flexeme's groups won't be helpful to developers because the number of groups and the overlapping groups will make it difficult to understand what each group is supposed to represent.

Future work should investigate if the format and content of the untangling results produced by the untangling tools is coherent to a human reader. For example can a developer use the untangling results to help them in a code review? Future work should also investigate whether a significant performance difference on the

dataset is also significant for the developers in practice. It may be so that even a lower performing tool might still be practically useful for developers.

Additionally, we encourage researchers to take a holistic approach when designing untangling tools and take into account the use case of the tool. For example, if the tool is meant to be used in pull requests, the tool must be able to untangle noncode changes such as comments and documentation because it would be helpful to group these type of changes with their related code changes.

3.6 Comparison with the evaluation on the synthetic datasets

In the original synthetic SmartCommit benchmark [20], SmartCommit obtained a median untangling performance of 0.74. We expected the difference to be bigger between the synthetic and real commits. For Flexeme, while we cannot directly compare the performances because Flexeme uses a different metric than the rand index for its synthetic benchmark, we can speculate that there is a qualitative difference in performance between the original implementation of Flexeme and ours as mentioned in section 3.2.

There are multiple factors that could have increased or decreased the performance such as 1) the synthetic benchmark measures the untangling performance at the hunk level rather than line level. 2) the projects are different 3) the content of the commits are different.

Further investigation is needed to understand the difference between the synthetic and real commits and their effect on untangling performance. A potential direction for future work is to compute the untangling performance on the tools' original synthetic dataset using the rand index to compare the performance of the synthetic and real commits. Additionally, our community needs to investigate how the performance of the tools compares on the different synthetic datasets because each tool is evaluated on a synthetic dataset that is constructed using different heuristics.

4 LIMITATIONS

4.1 Java implementation of Flexeme

Flexeme was originally implemented for C#. We reimplemented Flexeme for Java. We based our implementation on the original paper, online documentation, and discussions with the original author. Unfortunately, we didn't have access to the specification or implementation of the component that reads the source code and generates a Program Dependency Graph, called the PDG generator.

We tested our Java implementation using a new synthetic dataset created with the same heuristic as the C# synthetic dataset. The projects selected for the Java dataset were commons-lang, joda-time, and commons-math. Flexeme's untangling performance on its synthetic dataset is measured using accuracy as described in the Flexeme's paper [16]. We present the results of the Java synthetic dataset in table 6.

For the Java synthetic dataset, we obtain an average accuracy of 0.5. The original authors obtained an average accuracy of 0.81 on the C# synthetic dataset. The accuracy results on these two synthetic datasets matches the qualitative performance on the real commits we discussed in section 3.2. However, unlike our evaluation on real commits, the evaluation on synthetic commit doesn't convert the granularity of the results so it's improbable that converting the untangling results from the AST format to the line-based format is the culprit in the difference of performance between the implementations. Additionally, we speculate that the difference of performance cannot be explained by the difference of programming language or projects but is due to differences in implementation.

Table 6: Average accuracy scores for Flexeme on synthetic Java dataset for individual projects and overall.

Project	Accuracy
Joda	0.45
Lang	0.53
Math	0.48
Overall	0.50

We previously investigated differences in implementation between the C# and Java PDG generators and corrected the differences we found. We found two issues. Firstly, the granularity of the PDG generated by the C# PDG generator and the Java PDG generator were different. The C# implementation generated a PDG with a statement granularity while the Java implementation generated a PDG with a granularity at the operation level, resulting in the generated Java PDG to be much larger than its C# counterpart which we thought might be interfering with Flexeme's clustering algorithm and lowering the performance. Secondly, we found that some nodes in the PDG produced when Flexeme merges the before-PDG and after-PDG (which represent respectively the state of the code before and after the bug-fixing changes.), are incorrectly labeled when the before-PDG and after-PDG were generated using the Java PDG generator. This issue isn't caused by the longer PDG graphs produced by the Java PDG generator. We speculate that the cause is either due to a subtle difference in the generated Java PDGs or due to a bug in the PDG merging algorithm in Flexeme. We didn't need to reimplement the PDG merging algorithm since the merging algorithm is language agnostic and not part of the PDG generator.

Unfortunately fixing these issues didn't improve the performance of Flexeme on the Java synthetic dataset. We generated PDGs on identical source code in Java and C# and verified that the generated PDGs were identical in format and content. We are still investigating whether a difference of implementation could be the cause of the performance difference rather than the difference of programming language and projects, even though fixing the existing implementation issues didn't improve the performance significantly.

5 THREATS TO VALIDITY

5.1 Defects4J

We evaluated untangling tools only on bug-fixing commits. Other tangled commits might have different characteristics. The bug-fixing commits were untangled by the Defects4J authors to create minimal bug fixes, and might not reflect the conceptually cleanest decomposition of the tangled commit.

The Defects4J authors selected bug fixing commits from 17 projects. Moreover, Defects4J authors curate each bug-fixing commit to guarantee the commit is indeed fixing a bug. Other approaches find bug-fixing commits by looking for corrective keywords in the commit messages, which doesn't guarantee that the commit is a bug-fixing commit [6, 15, 24].

Defects4J manual untangling sometimes induces tangled lines, such as in Lang 8 where the field `mTimeZone` is renamed as `zone` to make the bug-inducing patch compile, since the other changes in the bug-fix also calls `zone`. These tangled changes exist purely to minimize the bug-fix patch according to the Defects4J Bug Minimization Guidelines, and we comply with the Defects4J rules to identify such tangled lines.

5.2 Performance Metric

Calculating the untangling performance with a metric that works across tools requires to convert the results of the untangling tools, potentially affecting their performance. We use a line-based metric that has a granularity between the diff hunk of SmartCommit and the AST of Flexeme. However, converting the AST results to line-based results might induce bias due to the fact that multiple AST nodes belong to one changed line, creating multiple labels for that line.

We mitigate this issue by measuring the pair-wise agreement for labels using the Rand Index. This causes identical labels for a lines to be counted multiple times with no penalty and lines different labels to be counted as agreement if the line has also different labels in the ground truth.

5.3 Analysis

We accounted for the random effects caused by having different bug-fixes and the bug-fixes belong to different project in our quantitative analysis. However, we didn't check whether the identity of the developer that committed the original bug-fix has an impact on performance. Given that the aforementioned random effects didn't have an effect on the model's performance, we don't expect the developer identity to have an impact either.

6 RELATED WORK

Previous evaluations of untangling tools suffer from one or more of the following issues: use of synthetic commits, use of unavailable commits, small datasets, partial evaluations, and use of different performance metrics.

Evaluations on **synthetic commits** requires no manual work to establish ground truth (which is the synthetic commit's constituent original commits) [3, 7, 13, 16, 20]. In addition to the fact that synthetic commits may have different characteristics than real commits, these evaluations are often not comparable because they use different heuristics to create the synthetic commits.

Evaluations on **unavailable commits** is problematic because it limits the reproducibility of the evaluation. These evaluations are done in companies with developers acquainted to the project, but where the commits are not publicly available, preventing the researchers to release the commits as part of their dataset [2, 5, 6, 20].

Evaluations using real commits use a **small dataset**. Evaluating on real commits is challenging because the ground truth has to be untangled manually, by outsiders to the software project. In consequence, the resulting dataset is too small for significance analysis [24].

Partial evaluations give comparative improvements from one version of an untangling tool to the next [3, 7, 16, 20] or, may not compare to other tools entirely [5, 6, 11, 12, 15, 21]. In both cases, these evaluations ignore other tools that are developed in parallel or that use different technical approaches.

Another issue is the **different performance metrics** used to evaluate the untangling tools that is often tied to the internal representation of the untangling tool. For example a hunk-based tool [20] may use a different performance metric than an AST-based tool [16] to evaluate how the changes are grouped, making it difficult to determine if one tool is better than another [13].

Compared to previous work such as Li et., al. (UTango) [13], we use the same performance metric to compare Flexeme and SmartCommit, making it possible to rank the tools. We also we evaluate on real commits, rather than synthetic commits that may not reflect the performance of the untangling tools in practice. Our evaluation uses open-source commits, which enables future researchers to reproduce our results. We evaluate unrelated untangling tools that use different technical approaches and target different programming languages.

In addition, compared to the methodology of other papers, Defects4J is closer to reality than synthetic datasets [7] or datasets generated by filtering the commit message by corrective keywords [6, 15, 24]. Filtering commits by keywords in the commit message is used to generate bug-fixing datasets, but has many false alarms or misses.

7 CONCLUSION AND FUTURE WORK

In this paper, we presented a methodology for evaluating untangling tools on a dataset of real bug-fixing commits. We conducted a quantitative and qualitative evaluation of the performance of the Flexeme and SmartCommit tools.

We found that SmartCommit is significantly more performant than Flexeme and that a naive file-based approach is better than both of these tools when untangling bug fixes code changes. Additionally, we found that the size of the commit has a statistically significant impact on the untangling performance. We found that as the size of the commit increases, the performance decreases. Additionally we found that the presence of tangled lines or tangled hunks in the commit has a statistically significant impact on the untangling performance.

Our manual exploratory analysis of the untangling results showcases how SmartCommit performed better than Flexeme by creating more cohesive groups due to its internal representation at the hunk level compared to Flexeme's fine grain AST-based representation that created many small overlapping groups. In addition to the groups generated by Flexeme being incorrect, the number of groups and overlapping lines are challenging to understand for developers, and we speculate that they would be difficult to leverage for downstream tasks such as code review or automated bug repair. Thus, we encourage researchers to take a holistic approach

1045 when building untangling tools and to be mindful of the internal
1046 representation of the untangling tool and how the granularity of
1047 the decompositions impacts the developers and the downstream
1048 tasks.

1049 In future work, we plan to evaluate the untangling performance
1050 on downstream applications such as code review and automated
1051 bug repair. We also plan to develop an untangling tool that leverages
1052 past code changes to address the limitations stemming from using
1053 only the code changes from the current commit to untangle changes.
1054

1055 # REFERENCES

1056 [1] Mike Barnett, Christian Bird, João Brunet, and Shuvendu K Lahiri. 2015. Helping
1057 developers help themselves: Automatic decomposition of code review changesets.
1058 In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 134–144.
1059 [2] Mike Barnett, Christian Bird, João Brunet, and Shuvendu K. Lahiri. 2015. Helping Developers Help Themselves: Automatic Decomposition of Code Review
1060 Changesets. In *2015 IEEE/ACM 37th IEEE International Conference on Software
1061 Engineering*, Vol. 1. 134–144. https://doi.org/10.1109/ICSE.2015.35
1062 [3] Siyu Chen, Shengbin Xu, Yuan Yao, and Feng Xu. 2022. Untangling composite commits by attributed graph clustering. In *Internetware 2022: 13th Asia-Pacific
1063 Symposium on Internetware (Internetware)*. Hohhot, China, 117–126.
1064 [4] Santanu Kumar Dash, Miltiadis Allamanis, and Earl T. Barr. 2018. RefiNym:
1065 Using Names to Refine Types. In *ESEC/FSE 2018: The ACM 26th joint European Software Engineering Conference and Symposium on the Foundations of Software
1066 Engineering (ESEC/FSE)*. Lake Buena Vista, FL, USA, 107–117.
1067 [5] Martín Dias, Alberto Bacchelli, Georgios Gousios, Damien Cassou, and Stéphane
1068 Ducasse. 2015. Untangling fine-grained code changes. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*.
1069 IEEE, 341–350.
1070 [6] Bo Guo, Young-Woo Kwon, and Myoungkyu Song. 2019. Decomposing composite changes for code review and regression test selection in evolving software.
1071 *Journal of Computer Science and Technology* 34 (2019), 416–436.
1072 [7] Kim Herzig, Sascha Just, and Andreas Zeller. 2016. The impact of tangled code changes on defect prediction models. *Empirical Software Engineering* 21, 2 (2016),
1073 303–336.
1074 [8] K. E. Iverson. 1962. *A Programming Language*. Wiley, New York.
1075 [9] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: A Database of existing faults to enable controlled testing studies for Java programs. In *ISSTA
1076 2014, Proceedings of the 2014 International Symposium on Software Testing and
1077 Analysis*. San Jose, CA, USA, 437–440. Tool demo.
1078 [10] David Kawrykow and Martin P. Robillard. 2011. Non-essential changes in version histories. In *2011 33rd International Conference on Software Engineering (ICSE)*.
1079 351–360. https://doi.org/10.1145/1985793.1985842
1080 [11] Hiroyuki Kirinuki, Yoshiki Higo, Keisuke Hotta, and Shinji Kusumoto. 2016. Splitting commits via past code changes. In *2016 23rd Asia-Pacific Software Engineering
1081 Conference (APSEC)*. IEEE, 129–136.
1082 [12] Patrick Kreutzer, Georg Dotzler, Matthias Ring, Bjoern M Eskofier, and Michael
1083 Philippsen. 2016. Automatic clustering of code changes. In *Proceedings of the 13th International Conference on Mining Software Repositories*. 61–72.
1084 [13] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2022. UTANGO: Untangling Commits
1085 with Context-Aware, Graph-Based, Code Change Clustering Learning Model. In
1086 *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Singapore, Singapore)
1087 *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA,
1088 221–232. https://doi.org/10.1145/3540250.3549171
1089 [14] Emerson Murphy-Hill, Chris Parnin, and Andrew P. Black. 2009. How we refactor, and how we know it. In *ICSE 2009, Proceedings of the 31st International Conference
1090 on Software Engineering*. Vancouver, BC, Canada, 287–297.
1091 [15] Hoan Anh Nguyen, Anh Tuan Nguyen, and Tien N Nguyen. 2013. Filtering noise
1092 in mixed-purpose fixing commits to improve defect prediction and localization. In *2013 IEEE 24th international symposium on software reliability engineering
1093 (ISSRE)*. IEEE, 138–147.
1094 [16] Profir-Petru Pârtachi, Santanu Kumar Dash, Miltiadis Allamanis, and Earl T. Barr.
1095 2020. Flexeme: Untangling Commits Using Lexical Flows. In *ESEC/FSE 2020: The ACM 28th joint European Software Engineering Conference and Symposium on the
1096 Foundations of Software Engineering (ESEC/FSE)*. Sacramento, CA, USA, 63–74.
1097 [17] GNU Project. accessed 2023-04-26. *GNU Diffutils Manual: Hunks*. https://www.gnu.org/software/diffutils/manual/html_node/Hunks.html
1098 [18] Achyudh Ram, Anand Ashok Sawant, Marco Castelluccio, and Alberto Bacchelli.
1099 2018. What makes a code change easier to review: an empirical investigation on
1100 code change reviewability. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of
1101 Software Engineering*. 201–212.
1102

[19] William M. Rand. 1971. Objective criteria for the evaluation of clustering methods. 1103
Journal of the American Statistical association 66, 336 (1971), 846–850. 1104
[20] Bo Shen, Wei Zhang, Christian Kästner, Haiyan Zhao, Zhao Wei, Guangtai Liang, 1105
and Zhi Jin. 2021. SmartCommit: A graph-based interactive assistant for activity-oriented commits. In *ESEC/FSE 2021: The ACM 29th joint European Software 1106
Engineering Conference and Symposium on the Foundations of Software Engineering 1107
(ESEC/FSE)*. Athens, Greece, 379–390.
[21] Sarocha Sothornprapakorn, Shinpei Hayashi, and Motoshi Saeki. 2018. Visualizing a tangled change for supporting its decomposition and commit construction. 1108 1109
In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, Vol. 1. IEEE, 74–79. 1110
[22] Shin Hwei Tan and Abhik Roychoudhury. 2015. relifix: Automated Repair of 1111
Software Regressions. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. 471–482. https://doi.org/10.1109/ICSE.2015.65 1112
[23] Yida Tao and Sunghun Kim. 2015. Partitioning composite code changes to 1113
facilitate code review. In *2015 IEEE/ACM 12th Working Conference on Mining 1114
Software Repositories*. IEEE, 180–190. 1115
[24] Min Wang, Zeqi Lin, Yanzhen Zou, and Bing Xie. 2019. CoRA: Decomposing 1116
and Describing Tangled Code Changes for Reviewer. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1050–1061. 1117
https://doi.org/10.1109/ASE.2019.00101

1118 1119 1120 1121 1122 1123 1124 1125 1126 1127 1128 1129 1130 1131 1132 1133 1134 1135 1136 1137 1138 1139 1140 1141 1142 1143 1144 1145 1146 1147 1148 1149 1150 1151 1152 1153 1154 1155 1156 1157 1158 1159 1160