Untangling Changes

Kim Herzig
Saarland University
Saarbrücken, Germany
herzig@cs.uni-saarland.de

Andreas Zeller
Saarland University
Saarbrücken, Germany
zeller@cs.uni-saarland.de

***Abstract*—When developers commit software changes to a version control system, they often commit unrelated changes in a single transaction—simply because, while, say, fixing a bug in module A, they also came across a typo in module B, and updated a deprecated call in module C. When analyzing such archives later, the changes to A, B, and C are treated as being falsely related. In an evaluation of five Java projects, we found up to 15% of all fixes to consist of multiple unrelated changes, compromising the resulting analyses through noise and bias. We present the first approach to *untangle* such combined changes after the fact. By taking into account data dependencies, distance measures, change couplings, test impact couplings, and distances in call graphs, our approach is able to untangle tangled changes with a mean success rate of 63–75%. Our recommendation is that such untangling be considered as a mandatory step in mining software archives.**

***Keywords*-bias; mining software repositories; software change analysis; untangling changes;**

I. Introduction

A large fraction of recent work in empirical software engineering is based on *mining version archives*—analyzing which changes were made to a system, by whom, when, and where. Such mined information can be used to predict related changes [1], to predict future defects [2], [3], to analyze who should be assigned a particular task [4], [5], or simply to gain insights about specific projects [6].

All these studies depend on the *accuracy* of the mined information—accuracy which is threatened by noise. Such noise can come from missing associations between change and bug databases [7]. One significant source of noise so far overseen, however, is *tangled changes*.

What is a tangled change? Let us assume we have a developer who is assigned multiple tasks A, B, and C. Let all these have a separate purpose; A may be a bug fix, B may be an extension, and C may be a refactoring. When the developer is done with the three tasks, she has to commit her changes to the version archive, such that the changes get propagated to other developers and can go into production. When committing her changes, she may be disciplined and group her changes into three individual commits, each containing the changes pertaining to each task and coming with an individual description. This separation is complicated, though; for instance, the tasks may require changes in similar locations. Therefore, it is more likely that she will commit all changes tangled in a single transaction, with a message such as "Fixed bug #334 in `foo.c` and `bar.c`; new feature #776 in `bar.c`; `qux.c` refactored; general typo fixes".

Table I
Proportion of bug-fixing changes that address more than one issue.

	# fixes	# tangled fixes	
ArgoUML	2,945	186	(6.3%)
GWT	809	115	(14.2%)
Jaxen	105	16	(15.2%)
JRuby	2,977	283	(9.5%)
XStream	312	39	(12.5%)

Such tangled changes do not cause serious trouble in development. However, they introduce *noise* in any analysis of the version archive, thereby compromising the accuracy of the analysis. As the tangled change fixed a bug, all files touched by it will now be marked as having had a defect—even though the tangled tasks B and C have nothing to do with a defect. Likewise, all files will be marked as being changed together—which may now induce a recommender to suggest changes to `qux.c` whenever `foo.c` is changed. Commit messages such as "general typo fixes" point to additional minor changes all over the code—locations which will now be related with each other as well as the tasks A, B, and C.

The problem of tangled changes is not a theoretical one. In an exploratory study on five open source projects, we manually classified more than 7,000 individual bug-fixing changes and checked whether these changes addressed multiple ("tangled") developer tasks. Table I summarizes our results: *Between 6% and 15% of all fixes address multiple concerns at once*—they are tangled and therefore introduce noise and bias into any analysis of the respective change history. (Section II has more on this study.)

The are two main approaches to overcome the problem of tangled changes. One solution is to detect tangled changes and to ignore these data points in any further analyses. But this solution makes two major assumptions. First, one must be able to detect tangled changes automatically; second, the fraction of tangled changes must be small enough that deleting these data points does not cause the overall data set

to be compromised. The second solution to deal with such tangled changes is to *untangle* them into separate changes which can be individually analyzed, thereby reducing the noise.

In this paper, we present the first approach to untangle changes. It splits tangled changes into smaller *partitions*, where each partition contains a subset of changes that are related to each other, but not related to the changes in other partitions. The algorithm is based on static code analysis only and is fully automatic, allowing archive miners to untangle tangled changes and to use the created change partitions instead of the original tangled change. Our experiments on five open-source projects show that neither data dependencies, distance measures, change couplings, test impact couplings, or distances in call graphs serve as a one-size-fits-all solution. By combining these measures, however, we obtain an effective approach which untangles multiple combined changes with a mean success rate of 63%–75%.

The remainder of this paper is organized as follows. To motivate untangling, we discuss the causes and effects of tangled changes in Section II. The basic principles of our untangling approach are presented in Section III and Section IV. Section V describes the evaluation setup, followed by the evaluation results in Section VI and the threats to validity (Section VII). Section VIII discusses related work and Section IX closes with conclusion and consequences.

II. Tangled Changes

Many approaches to mining version archives require historic quality data used to train machine learning models. Zimmermann et al. [2] describe one standard approach to generate their historic quality data sets mapping bug fixes to code artifacts. The idea is as follows: one requires a list of *bug fixes* applied to individual code artifacts (e.g. code files) during project history: The more bugs were fixed over time, the worse the code artifact's original quality. To retrieve such a bug fix count, one identifies bug fixing change sets by parsing commit messages. Once the change set is confirmed to reference a closed and fixed bug report, all code artifacts changed within the change set are marked as being fixed.

But such an approach relies on the basic assumption that change sets are *atomic*—each change set contains only those changes that are necessary to fix the referenced issue. However, as stated in Section I, there may be various reasons for developers to tangle multiple unrelated tasks into a single commit.

When analyzing such tangled change sets, it is not easy to determine which code artifacts were changed due to which developer task. In many cases, developers may be able to separate two unrelated changes and to review these changes separately. To the best of our knowledge, however, all approaches to mining version archives consider the change set as atomic. When it comes to defect prediction, for instance, this implies that a bug report is mapped to every single change in the associated change sets; consequently, this will introduce bias into the data set, which may spoil the results of any analysis based on this data.

The bias caused by tangled change sets is significant. Table I shows the number of issue fixing transactions for five open-source projects. We considered only those transactions that contained the keywords *fixed*, *resolved*, or *issue* and later manually validated that their commit message marks an issue within the source code (bug and feature requests). We also manually checked the commit messages of these fixes and marked a change set as tangled if the commit message clearly indicated that the applied changes tackle more than one developer task. This can either be commit messages that contain more than one issue report reference (e.g. "XSTR-93, XSTR-120, XSTR-170: Support for \r newline in strings.") or a commit message indicating extra work committed along the issue fix (e.g."Fixes issue #591[...]. Also contains some formatting and cleanup.")—mostly cleanups and refactorings. The fraction of tangled fixes lies between 6% and 15% (Table I) for issue fixing change sets.

These findings are supported by other studies reporting similar bias figures for change sets. Dallmeier [8] used delta debugging to minimize bug fixes of two open source projects to a minimal set of code changes letting a regression test pass. The results show that on average 50% to 60% of the code changes applied within a bug fix transaction had no effect on the result of the regression tests. Similarly, in a study of over 24,000 change sets from seven open-source projects [9], Kawrykow and Robillard found that 2% to 15% of all method updates were due to *non-essential differences*—code changes that did not change the semantics of the program. For our initial example of defect prediction models, this means that *up to 30% of all changes are falsely associated with bug reports; they either are tangled with other changes by coincidence, or have no impact on the semantics at all.* The effects of such biased data mining sets are significant [7], [9].

Up to 30% of all changes are falsely associated with bug reports.

III. The Untangling Algorithm

Many change sets contain code changes serving multiple developer tasks building bigger tangled change sets. Files touched by a single tangled change set may have been changed for different reasons and thus should be separated when analyzing them or mapping developer tasks to changed files. Otherwise, such tangled change sets lead to harmful bias impacting prediction models based on version archive training sets. To reduce this bias, it is necessary to *untangle* change sets into individual, corresponding change set partitions.

Generally, determining unrelated code changes applied together is undecidable, as the halting problem prevents

predicting whether a given code change has an effect on a given problem. Consequently, every untangling algorithm will have to rely on *heuristics* presenting an approximation of how to separate two or more code changes. We cannot solve the untangling problem completely, but we hope to reduce the amount of bias significantly.

An *untangling algorithm* should be fully automatic and simple at the same time. The algorithm proposed in this paper expects an arbitrary change set as input and returns a set of change set partitions such that each change set partition contains code changes that are related—ideally all necessary code changes necessary to resolve one developer task (e.g. fixing a bug)—while the union of all partitions equals the original change set. Instead of mapping developer tasks to all changed code artifacts within the corresponding original change set, one would assign individual developer tasks to those code artifacts changes within the corresponding change set partition.

A. Related changes

Each change set is a set of individual *change operations* (added, modified, and deleted method definitions and calls) to individual source code files. Using this terminology, our untangling algorithm will get one set of change operations as input and produces a set of sets of change operations (see Figure 1).

For each pair of change operations,the algorithm has to decide whether both change operations belong to the same partition (are related) or should be assigned to separate partitions (are not related). To determine whether two change operations are related or not, we have to determine the *relation distance* between two code changes such that the distance between two related change operations is lower than the distance between two unrelated change operations. Analyzing the change operations themselves and considering the project's history, there are *multiple ways to define distance* between two change operations. However, none of them seem to be powerful enough to capture the complexity of change operation relations. Here is an example: Considering data dependencies between two code changes, it seems reasonable that two change operations changing statements reading/writing the same local variable are very likely to belong together. But vice versa, two code changes not reading/writing the same local variable may very well belong together—because both change operations affect consecutive lines. As a consequence, our untangling algorithm should be based on a *feature vector* spanning multiple aspects describing the distances between individual change operations and should *combine these distance measures* to separate related from unrelated change operations.

In Section IV, we will discuss how to combine multiple distance measures to decide which code changes are likely to be related. But before that, let us discuss how the overall algorithm is designed.

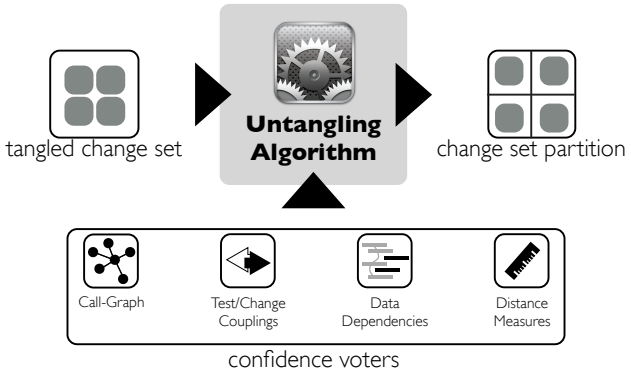

Figure 1. The untangling algorithm partitions change sets using multiple, configurable aspect extracted from source code. Gray boxes represent sets of change operations necessary to complete one developer task.

B. Using Multilevel Graph Partitioning

From the previous section we learned that the untangling algorithm has to iterate over pairs of change operations and needs to determine the likelihood that these two change operations are related and thus should belong to the same change set partition. Although we do not partition graphs, we reuse the basic concepts of a general *multilevel graph partitioning* (MGP) algorithm proposed by Karypis and Kumar [10]–[12]:

1) Build up a triangle matrix $\mathcal{M}$ of dimension $m \times m$ containing one row and one column for each of the m atomic change operations. Each column represents one change set partition; we start with the most fine granular partitioning of the original change set.
2) For each cell $[co_i, co_j]$ with $i, j \leq m$ of $\mathcal{M}$, we compute a *confidence value* indicating the likelihood that change operation co_i and change operation co_j are related and should belong to the very same change set partition (see Section IV for details on how to compute these confidence values). Figure 2 shows this step in detail.
3) After building up the triangle matrix (confidence value for $[co_i, co_j]$ equals the confidence value for $[co_j, co_i]$), we *collapse* the row and column with the highest confidence value within the corresponding cell. In other words we combine those partitions most likely being related.
4) We add a new column and row for the just newly generated partition co_{m+1} and compute confidence values between the new partition and all remaining partitions within the matrix. There are different strategies to compute the confidence values between two partitions P_1 and P_2 containing multiple change operations. For the results presented in this paper, we took the maximum of all confidence values between change operations stemming from different partitions:

$$Conf(P_1, P_2) = Max\{Conf(c_i, c_j) \mid c_i \in P_1 \wedge c_j \in P_2\}$$

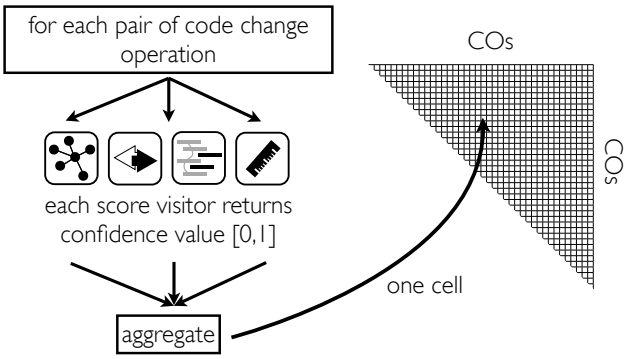

Figure 2. The procedure to build the initial triangle matrix used within the modified multilevel graph partitioning algorithm.

> The intention to use the maximum value here is that two code changes can be related even though they have very few properties in common but operate on the same data structure or object instance. In such cases, using the mean value instead would disregard such dependencies.

These steps ensure that the dimension of the matrix $\mathcal{M}$ decreases by one for each iteration. This is ensured by the fact that in each iteration, two already known partitions get deleted (two rows and two columns) while only one row and one column get added. We also ensure that we always combine those partitions that are most likely related to each other.

Without determining a stopping criterion, this algorithm would run until only one partition is left. This would be the original change set itself. Our algorithm can handle two different stopping strategies, both used for different purposes. Giving the algorithm a fixed number of partitions to be produced, it merges partitions until it reaches the desired number of partitions and returns. This strategy will be used in our experimental setup, but requires the user to specify the number of unrelated changes applied within the original change set. Besides our experimental setup, this strategy might be a good candidate for analyzing change sets of projects that follow a very strict commit message format. In these cases, it might suffice to scan the commit message to extract the expected number of partitions. If the number of partitions to be expected is unknown, the algorithm allows the user to specify a *confidence threshold* that must be exceeded in order to allow to partitions to be merged. If no partitions are left whose common cell exceeds this threshold, the algorithm terminates.

The untangling algorithm shown so far represents the partitioning framework used to merge change sets into partitions only. This part of the algorithm is general and makes no assumptions about source code, change operations, or any other aspect that estimates the relation between individual change operations.

Table II
LIST OF CONFVOTERS USED THROUGHOUT THE EXPERIMENTS.

ConfVoter	Description
FileDistance	The number of lines between two change operations if both change operations were applied to the same file. Will not be considered otherwise.
PackageDistance	If both change operations were applied to different code files, this ConfVoter will return the number of different package name segments within the package names of the changed files. Will not be considered otherwise.
CallGraph	Identifies the modified methods within the projects call graph and returns the distance between the two call graph nodes. The distance of a path within the call graph is defined as the sum of all edge weights along the path. An edge weight between method m_1 and method m_2 is defined as one divided by the number of method calls between m_1 and m_2.
ChangeCouplings	Returns the change coupling confidence as described by Zimmermann et al. [13]. Basically, it computes frequent change patterns and a confidence value indicating the probability that the change pattern will occur whenever one of the patterns components change.
DataDependency	Returns a value of one if both change operations read or write the same variable(s). Return a value of zero otherwise.

In the next sections, we will discuss how to derive the initial confidence values between change operations filling the cells of the initial triangle matrix $\mathcal{M}$ and how to combine multiple dependency measures into a single confidence value.

IV. CONFIDENCE VOTERS

To combine various dependency and relation aspects between change operations, the untangling framework itself does not decide which change operations are likely to be related but asks a set of so called *confidence voters* (ConfVoters) (see Figure 1). Each ConfVoter expects a pair of change operations and returns a *confidence value* between zero and one. A confidence value of one represents a change operation dependency aspect that strongly suggests to put both change operations into the same partition. Conversely, a return value of zero indicates that the change operations are unrelated according to this voter.

Using confidence voters, we can handle multiple relation dependency aspects within the untangling framework. Each ConfVoter represents exactly one dependeny aspect. Table II lists the set of ConfVoters used throughout our experiments. The current prototype of the untangling framework allows ConfVoters to be registered as plug-ins. This way, we can add or remove project-specific change operation dependency aspects within minutes.

To transform multiple confidence values—one per registered ConfVoter—into a single confidence values required

for the initial triangle matrix $\mathcal{M}$ (see Section III-B), we have to aggregate respecting the fact that different ConfVoters may have different importance (e.g. data dependency might be a stronger indication than change couplings). For this purpose, we train a *linear regression* model to determine a project's specific linear combination of dependency aspects that matter to separate related from unrelated change operations. Once such a model is trained, we can use it to determine a single confidence value (regression) providing all confidence values of all registered ConfVoters. For details on how to train the linear regression model see Section V-C.

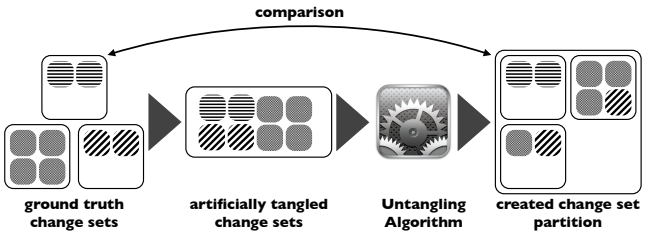

Figure 3. Artificially tangled change sets are generated using manually classified atomic change sets to compare created partitions and desired output. In the example, two change operations are put into a wrong partition, and hence the success rate is $\frac{6}{8} = 75\%$

V. Evaluation Setup

To determine the precision of our untangling algorithm, we ran experiments on five open-source Java projects (see Table III) with more than 50 months of active development history and more than 10 active developers. The projects differ in size (line of code) ranging from small (JRuby), medium (ArgoUML) to large (Google Web Toolkit). The different project sizes allow us to check whether the proposed untangling algorithm works on smaller projects as well as on large projects. The number of committed change sets ranges from 1,300 (Jaxen) to 16,000 (ArgoUML), and the number of bug fixing change sets ranges from 105 (Jaxen) to nearly 3000 (ArgoUML and JRuby).

In the next sections, we will discuss how to determine the expected outcome of an untangling procedure in order to measure its precision using a set of ground truths and a method to produce a set of artificially tangled change sets.

A. Ground Truth

In Section II, we discussed that a large proportion of change sets must be considered as tangled—combining multiple developer tasks into one version archive commit. For our experimental setup this means that we cannot rely on the existing data to evaluate our untangling algorithm, simply because we cannot determine whether a produced change set partition is correct and if not, how much it differs from an expected result.

To determine a reliable set of atomic (unbiased) change sets—change sets containing only those code changes to fulfill exactly one developer task—we used a two phase manual inspection of issue fixing change sets. The limitation to issue fixing change sets was necessary in order to understand the reason and to learn the purpose of the applied code changes. Without having a document describing the applied changes, it is very hard to judge whether a code change is tangled or not, at least for a project outsider.

1) We pre-selected change sets that could be linked to exactly one fixed and resolved bug report (similar to Zimmermann et al. [2]). All other change sets contained in the project's history were ignored in Step 2.
2) Each change set passed from Step 1 was manually inspected and classified as atomic or non-atomic. During manual inspection, we first read the commit message. In many cases, the commit message already indicated a tangled change set and therefore the change set was marked it non-atomic. If the commit message did not classify the change set as non-atomic, we inspected the actual code changes. Only if we had no doubt that the change set served only one developer task that was stated within the linked bug report (also no additional refactoring or code cleanup), we classified the change set as atomic. During classification, we tried to be as conservative as possible. If we had any doubt that the change set might not be atomic, we classified it as non-atomic.

The last row of Table III contains the number of manually classified atomic change sets per project. Their number depends on two project specific factors: the more bug fixing change sets contain a reference to the corresponding bug report, the more change sets rank for manual inspection; second, the smaller and cleaner a change set the easier its manual classification is. For the three smaller projects *Google Web Toolkit*, *Jaxen*, and *XStream*, we found between 30 and 40 atomic changes each. Considering the total amount of historic change sets, this number is small but does not represent the amount of atomic change sets within the project. Due to the very restrictive selection of change sets for manual classification, we have limited the number of atomic changes to a small selection. For the two larger projects *ArgoUML* and *JRuby*, we found more atomic change sets. In relation to their total number, the fraction of atomic change sets remains comparable: 0.8–2.4%. These figures represent the fraction of change sets that could be classified as atomic, whereas the figures presented in Table I are the fraction of change sets that could be classified as tangled. This implies that the majority of change sets could not be classified.

Table III
DETAILS OF PROJECTS USED DURING EXPERIMENTS.

	ArgoUML	Google Web Toolkit	Jaxen	JRuby	XStream
lines of code	164,851	266,115	20,997	101,799	22,021
#history months	150	54	114	105	90
#developers	50	120	20	67	12
Total #change sets	16481	5326	1353	11134	1756
#bug fixes	2,945	809	105	2,977	312
#atomic bug fixes	125 (4.2%)	44 (5.4%)	32 (30.5%)	200 (6.7%)	40 (12.8%)

B. Generating Artificial Tangled Change Sets

Combining atomic change sets into artificially tangled change sets is straight forward. Nevertheless, we have to be careful which atomic change sets to tangle. Combining them randomly is easy but would not simulate real tangled change sets. In most cases, developers do not combine arbitrary changes into one change set, but code changes that are close to each other (e.g. fixing two bugs in the same file or improving a loop while fixing a bug). To simulate such relations to some extend, we combined only change sets that contain at least two change operations touching files that *are not more than two sub-packages apart.* As an example, assume we have a set of three change sets: $CS_1 = \{a.b.c.d.F_1\}$, $CS_2 = \{a.b.c.e.F_2\}$, and $CS_3 = \{a.f.c.d.F_3\}$. Using our change set combination strategy explained above, we would combine CS_1 with CS_2 since they are only one sub-package apart. But we would not combine CS_1 with CS_3 nor CS_2 with CS_3.[1]

Furthermore, we limit the number of days allowed between two atomic change sets to 14 days (two weeks). The main reason for this limitation was to simulate real world code changes. On the other hand, this limitation was also necessary for technical reasons. Most of our ConfVoters require type resolution using the *partial program analysis* tool [14] which needs to compile the source code. Longer time periods between code changes imply higher probability that merging code changes leaves source code uncompilable. Applying our approach to real world tangled change sets, such a situation will never occur.

C. Training Confidence Voter Aggregation Model

In Section IV, we mentioned the use of a linear regression model to aggregate all ConfVoter confidence values for a pair of change operations into a single confidence value. To train the model, we used our ground truth set of change sets (see Section V-A). Thirty per cent of the generated artificially tangled change sets (see Section V-B) will be used for training purposes. For each pair of change operations within the training set, we compute all ConfVoter confidence values (as the untangling algorithm would do). Additionally, we add an expected result value indicating whether both change operations have been included within the very same change set: one for pairs containing change operations from the very same atomic change set, zero otherwise.

This set of confidence values and expected result values will then be used to train our aggregation model using the Weka tool [15]. To aggregate confidence values within the untangling algorithm itself, it suffices to let the model predict the result value based on the ConfVoter confidence values computed for each pair of change operations.

D. Success Rate

To test our untangling algorithm, we generate all possible artificially tangled change sets as described in Section V-A. Since we know the origin of each change operation, we now can compare the expected partitioning with the partitioning produced by the untangling algorithm (see Figure 3). We measure the difference between original and produced partitioning as the number of change operations that were put into a "wrong" partition.

For a set of artificially tangled change sets t, we define our *success rate* as

$$success\_rate = \frac{\#\ correctly\ assigned\ change\ operations}{total\ \#\ of\ change\ operations\ in\ t}$$

As an example for a success rate, consider Figure 3. In the artificially tangled change set, we have 8 changes overall. Out of these, two are misclassified (the diagonally striped one in the upper right corner, and the plain grey one in the lower left corner); the other six are assigned to the proper partition. Consequently, the success rate is $6/8 = 75\%$, implying that $2/8 = 25\%$ of all changes need to be recategorized in order to obtain the ground truth partitioning.[2]

VI. RESULTS AND DISCUSSION

The results presented in this section depend on the size of the artificial tangled change sets that is to be untangled—the so called *blob size*. The blob size represents the number of atomic change sets contained within the same artificial tangled change set. We did not generate artificially tangled change sets larger than five, although possible. During our

[1]This slightly favors the ConfVoter which uses package distances as a heuristic. However, we favored a more realistic distribution of changes over total fairness across all ConfVoters.

[2]Our computation of the success rate selects the *minimal* number of recombinations; this is in line with common differencing algorithms which also find the minimal number of operations to change a version into another.

Table IV
NUMBER OF GENERATED ARTIFICIALLY TANGLED CHANGE SETS SORTED BY BLOB SIZE.

	blob size				
	2	3	4	5	∑
ArgoUML	172	767	2,820	8,712	12,471
GWT†	128	413	940	1,559	3,040
Jaxen	240	1,573	7,470	26,892	36,175
JRuby	961	14,019	152,518	1,303,808	1,471,306
XStream	469	4,754	39,243	263,451	307,917

† GWT = *Google Web Toolkit*

manual change set inspection (see Section II) we rarely found commits that had or exceeded a blob size of five. We found that change sets with a blob size of two or three are most common.

A. Artificially Tangled Change Sets

Our results for generating artificially tangled change sets (see Section V-B) are listed in Table IV. The number of generated artificially tangled change sets highly depend on the project's history. While the number of artificially tangled change sets for JRuby and XStream are well above 1.4 million and 300K respectively, the numbers are much lower for the remaining projects. The ability to generate artificially tangled change sets depends on the number of atomic change sets (correlation of .5) but also depends on the files touched by these atomic change sets. The more atomic change sets touch source files with low file distance (see Section V-B) the more artificially tangled change sets will be generated. Increasing blob size allows more different combinations to be build. The Google Web Toolkit project is the project with the lowest number of generated artificially tangled change sets but also with one of the highest fraction of non-atomic bug fixes (see Table I) and it is also the project with the second smallest number of atomic change sets identified (see Table III). The project with the highest number of artificial tangled change sets is JRuby. Accordingly, it is the project with the second lowest number of tangled fixes (Table I) and the project with the highest number of atomic change sets.

B. Untangling Artificial Tangled Change Sets

The results of the actual untangling algorithm according to our experimental setup (Section V) are shown in Figure 4, grouped by project and blob size.

Overall, the success rates of our untangling algorithm lie between 60% and 91% for artificially tangled change sets of blob size two and between 50% and 84% for artificially tangled change sets of size five. Surprisingly, projects with higher number of generated artificially tangled change sets also show better untangling success rates (correlation .77). The more artificially tangled change sets, the higher the number of instances to train our linear regression aggregation model on (see Section V-C).

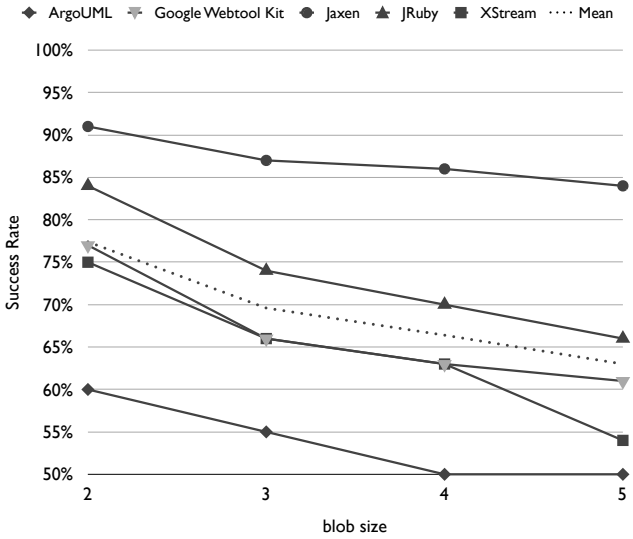

Figure 4. Untangling success rates per project and blob size.

Our approach untangles any two artificially tangled changes with a success rate of 60–91%.

It is not surprising that the blob size is negatively correlated with the success rate of the untangling algorithm. The more change operations to be included and the more partitions to be generated, the higher the likelihood of misclassifications. Increasing the blob size from two to three, success rates drop by approximately ten percent, across all projects. Increasing the blob size further has a negative impact on success rates, but the differences are lower than between blob size two and three. The number of samples to train the aggregation model is proportional to the blob size.

The project with the lowest success rates is ArgoUML—the only GUI application within this set of projects. Inspecting some of the falsely classified change operations unveiled that there are a number of code changes that are dependent considering the GUI level (e.g. window and fonts size) but are hard to detected on code level. Naturally, relations between code changes that are not evident on the source code level are hard to detect using code relation heuristics, only.

The mean values for all blob sizes (see dotted line in Figure 4) lie between 77% for blob size two and 63% for blob size five. Improving the untangling results for ArgoUML to a comparable level could increase the mean value by nearly 4%, ranging between 82% for blob size two and 66% for blob size five.

The mean success rate across all blob sizes is 63% (blob size five) to 73% (blob size two).

C. The importance of Confidence Voters

Using multiple ConfVoters to estimate the relation between two change operations raises questions about the

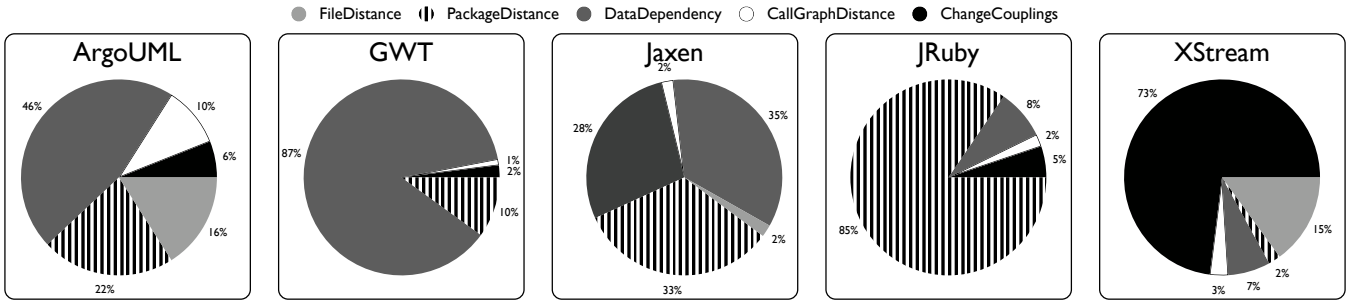

Figure 5. Relative importance for confidence for each ConfVoter used within the experimental setup. The importance is presented as percentage covering the proportion of variance explained by aggregation models. (GWT stands for Google Web Toolkit.)

contribution of each of the ConfVoter. During our experiments, we learned that the importance of the used set of ConfVoters is constant and independent of the project under investigation.

Figure 5 shows the relative importance for each of the aggregation model's ConfVoter components. The importance is measured as the average percentage of the aggregation models R^2 values—measured over all computed aggregation models. Surprisingly, the relative importance of single ConfVoters is highly dependent on the project under investigation. It seems that there exists no single combination of ConfVoters that describes change set dependency relations. For *ArgoUML* and *Google Web Toolkit* DataDependency is the most dominant factor followed by PackageDistance—for the other three projects it is of no major importance. For *Jaxen* and *XStream* ChangeCouplings are most dominant while for *JRuby* PackageDistance is the most important ConfVoter. Except for CallGraphDistance, all ConfVoters contribute to the aggregation prediction R^2 for at least one of the five projects. Thus we conclude, that the performance of individual ConfVoters is project specific.

Every project brings its own factors that determine how to untangle changes.

VII. Threats to Validity

Like any other empirical study of this kind, the approach presented in this paper has threats to its validity. We identified four noteworthy threats.

The change set classification process used in Section V-A involved manual code change inspection. We tried to be as conservative as possible when classifying atomic change sets, but the classification process was conducted by software engineers not familiar with the internal details of the individual projects. Thus, it is not unlikely that the manual selection process or the pre-filtering process mis-classified change sets. This could impact the number and the quality of generated artificially tangled change sets and thus could impact the untangling results, in general.

A second threat is given by the selected software projects used throughout our experimental setup. We cannot claim that the selected Java projects are representative in any way. We tried to include projects of different size and domains. With Google Web Toolkit, we also considered a project that is developed by an industrial company. Nevertheless, we have to be aware that untangling results for other projects may differ. The very same holds for the selection of ConfVoters. Choosing a different set of ConfVoters will impact untangling results.

The process of constructing artificially tangled change sets may not be simulating real life blobs caused by developers combining multiple developer tasks into single change sets. Thus, results of untangling real developer change sets may differ.

The aggregation process to transform multiple confidence values into a single confidence value includes a machine learner training phase. The data sets used to train these aggregation models are produced by random splits (see Section V-C). Using different random splits may impact the aggregation results significantly and thus may impact the overall untangling results.

As mentioned briefly, we use the partial program analysis tool [14] by Dagnais and Hendren within our untangling algorithm. Thus, the validity of our results depend on the validity of the used approach.

The untangling results presented in this paper are based on artificially tangled change sets derived using the ground truth set which contains issue fixing change sets, only. Thus, it might be that the ground truth set is not representative for all types of code changes.

VIII. Related Work

To the best of our knowledge, our approach to untangle unrelated changes from version archives is the first approach addressing this issue. Nevertheless, this work complements and makes use of existing research.

A. Classifying Code Changes

The work presented in this paper is closely related to many research approaches that analyze and classify code changes or development activities. In this section, we want

to discuss some of these approaches most closely related to our approach.

Untangling changes can be seen as a code change classification problem. The untangling algorithm classifies code changes as related or unrelated. Prior work on code classification mainly tried to classify code changes based on their quality [16] or on their purpose [17], [18]. Sunghun et al. [16] developed a change classification technique, based on machine learning, comparing software changes to earlier applied software changes. Their approach is able to classify changes as "buggy" or "clean" with a precision of 75% and a recall of 65% (on average). Despite their good classification results, their approach cannot be used to untangle code changes. Comparison of current and past code changes does not help to determine a possible semantical difference and it would require a bias free software history. Hindle et al. [17], [18] analyzed large change sets that touch a large number of files to automatically classify the maintenance category of the individual changes. The results indicate that large change sets frequently contain architectural modifications and are thus important for the software's structure. In most cases, large commits were more likely to be perfective than corrective.

Störzer et al. [19] used a change classification technique to automatically detect code changes contributing to test failures. Later, this work was extended by Wloka et al. [20] to identify committable code changes that can be applied to the version archive without causing existing tests to fail. Both approaches aim to detect change dependencies within one revision but require test cases mapped to change operations in order to classify or separate code changes. This will rule out the majority of change operations not covered by any test case or for which no test case is assigned.

Williams and Carver [21] present in their systematic review many different approaches on how to distinguish and characterize software changes. However, none of these approaches is capable of automatically identifying and separating combined source code changes based on their different characterization or based on semantic difference.

B. Refactorings

The combination of refactorings and semantic relevant code changes can be seen as a special case of the untangling problem which has been topic for many research papers. Murphy-Hill et al. [22], [23] analyzed thousands of development activities to prove, or disprove, several common assumptions about how programmers refactor. Their results also show that developers frequently do not indicate refactoring activity in commit logs, which would increase the bias potential discussed in Section II, even further. Later, Kawrykow and Robillard [9] investigated over 24,000 open-source change sets and found "that up to 15.5% of a system's method updates were due solely to non-essential differences affecting the association rules that can be mined from change data".

C. Change Dependency

The problem that version archives do not capture enough information about code changes to fully describe them is not new. Robbes et al. [24] showed that the evolutionary information contained within version archives such as CVS and SVN is incomplete and of low quality. Storing historical data as reaction due to explicit developer request fails to store important historic data fragments, while the nature of version archives leads to a view of software as being simply a set of files. As a solution, Robbes et al. [24] proposed a novel approach that automatically records all semantic changes performed on a system. An untangling algorithm would clearly benefit from such extra information that could be used to add context information for individual change operations.

D. Bias in Version Archives

In recent years, the discussion about bias and noise in data sets produced by mining version archives and their effect on mining models increased. Lately, Kawrykow and Robillard [9] showed that bias caused by non-essential changes severely impacts mining models based on such data sets. Considering the combination of non-essential changes and essential changes as an untangling problem, the result of Kawrykow and Robillard are a strong indication that unrelated code changes applied together will have similar effects.

Dallmeier [8] analyzed bug fix change sets of two open source projects and used delta debugging to minimize bug fixes to a set of code changes that is sufficient to make regression tests pass. He found out that on average only 50% of the changed statements were responsible to fix the bug.

The effects of bias caused by unbalanced data sets on defect prediction models were investigated by Bird et al. [7]. The authors conclude that "bias is a critical problem that threatens both the effectiveness of processes that rely on biased datasets to build prediction models and the generalizability of hypotheses tested on biased data".

Kim et al. [25] showed in an empirical study that the defect prediction performance decreases significantly when the data set contains 20%-35% of both false positives and false negatives noises. The authors also present an approach that allows automatic detection and elimination of noise instances. But removing data points from data sets also means less data points to learn from. Untangling changes can be seen as an attempt to resolve possible noise factors before aggregation and further processing, allowing prediction models to benefit from possible noise instances instead of removing them.

E. Used Frameworks

In this section, we give credit to those frameworks used in order to accomplish the presented work.

As described in Section III-B, we use a simplified version of the multilevel graph partitioning algorithm as proposed by Karypis and Kumar [12]. Although we do not work on any graph structure, our untangling algorithm follows the basic principles of the coarsening phase of a multilevel graph partitioning algorithm — with minor modifications. The uncoarsening phase of the original partition algorithm is not considered within this work.

To compute the ConfVoters confidence values that build the fundament of our untangling algorithm, we have to extract single change operations analyzing change sets that might leave the source code in an uncompilable state. To accomplish this task, many of the mentioned ConfVoters use static analysis for partial Java programs, as proposed by Dagenais and Hendren [14].

IX. Conclusion and Future Work

In this paper we propose an untangling algorithm that helps to reduce the amount of bias within data mining sets, caused by version archive commits, combining changes that were committed due to multiple developer tasks. To the best of our knowledge, this study is the first one to quantify the extent of tangled changes in real world projects. Our results show that the fraction of tangled changes may be substantial, causing a serious threat to empirical findings based on version archives.

For the five open-source projects used within our experiments, the algorithms showed an average success rate between 63% and 77%, depending on the number of partitions to be created. Basing empirical findings on untangled changes will make them more precise, and less threatened by bias and noise.

Our results indicate that untangling changes is a surprisingly difficult task, leaving lots of room for future improvements. Our future work will focus on the following topics:

- **Evaluate tangling effect.** As a next step, we plan to fully integrate our untangling algorithm into state of the art prediction or recommendation models and to show the effect of the untangling project. Given our approach, the challenge here is not so much to untangle the changes, but to *untangle the commit message.* While we can easily identify the purpose of the tangled change overall (from the message committed to the version archive), we can no longer identify the purpose of the untangled subchanges—we know they are independent, but we no longer know which ones are still tied to the explicitly stated purpose, and which ones are not. We are currently investigating classification models that determine the semantic change applied by code changes.
- **Non-essential changes.** We also plan to integrate the work on non-essential changes, as proposed by Kawrykow and Robillard [9]. We can use the concept of non-essential changes to filter out non-essential changes before untangling, putting all non-essential changes into a separate change set partition. We believe that such an integration of both approaches would improve the success rate of the untangling algorithm.
- **Recommendations to developers.** In the long run, one could also present untangled change sets as recommendations to users—either at the time they are committed, or at the time they would be manually analyzed. Our current focus, however, is to reduce data noise and the resulting threats to analysis of version archives.

We are committed to make the entire untangling framework and all training examples publicly available. We expect to release this package early 2012 at the project web site:

`www.st.cs.uni-saarland.de/softevo/untangling/`

Acknowledgment

This work was conducted while Kim Herzig was a research intern at Google Inc. This work was funded by a Google Research Award “Predicting the Risk of Changes”. Yana Mileva provided constructive feedback on earlier versions of this work.

References

[1] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller, “Mining version histories to guide software changes,” in *Proceedings of the 26th International Conference on Software Engineering*. IEEE Computer Society, May 2004, pp. 563–572.

[2] T. Zimmermann, R. Premraj, and A. Zeller, “Predicting defects for Eclipse,” in *Proceedings of the Third International Workshop on Predictor Models in Software Engineering*, ser. PROMISE ’07. IEEE Computer Society, 2007.

[3] T. Menzies, Z. Milton, B. Turhan, B. Cukic, Y. Jiang, and A. Bener, “Defect prediction from static code features: current results, limitations, new approaches,” *Automated Software Engg.*, vol. 17, pp. 375–407, December 2010.

[4] J. Anvik, L. Hiew, and G. C. Murphy, “Who should fix this bug?” in *Proceedings of the 28th international conference on Software engineering*. ACM, 2006, pp. 361–370.

[5] P. Bhattacharya, “Using software evolution history to facilitate development and maintenance,” in *Proceeding of the 33rd international conference on Software engineering*. ACM, 2011, pp. 1122–1123.

[6] P. L. Li, R. Kivett, Z. Zhan, S.-e. Jeon, N. Nagappan, B. Murphy, and A. J. Ko, “Characterizing the differences between pre- and post-release versions of software,” in *Proceeding of the 33rd international conference on Software engineering*. ACM, 2011, pp. 716–725.

[7] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu, "Fair and balanced? Bias in bug-fix datasets," in *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ser. ESEC/FSE '09. ACM, 2009, pp. 121–130.

[8] V. Dallmeier, "Mining and checking object behavior," Ph.D. dissertation, Universität des Saarlandes, August 2010.

[9] D. Kawrykow and M. P. Robillard, "Non-essential changes in version histories," in *Proceeding of the 33rd international conference on Software engineering*, ser. ICSE '11. ACM, 2011, pp. 351–360.

[10] G. Karypis and V. Kumar, *MeTis: Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 2.0*, 1995. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.38.376

[11] ——, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM J. Sci. Comput.*, vol. 20, pp. 359–392, December 1998.

[12] ——, "Analysis of multilevel graph partitioning," in *Proceedings of the 1995 ACM/IEEE conference on Supercomputing*, ser. Supercomputing 1995. ACM, 1995.

[13] T. Zimmermann, P. Weisgerber, S. Diehl, and A. Zeller, "Mining version histories to guide software changes," in *Proceedings of the 26th International Conference on Software Engineering*, ser. ICSE '04. IEEE Computer Society, 2004, pp. 563–572.

[14] B. Dagenais and L. Hendren, "Enabling static analysis for partial Java programs," in *Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, ser. OOPSLA '08. ACM, 2008, pp. 313–328.

[15] I. H. Witten and E. Frank, *Data Mining: Practical Machine Learning Tools and Techniques, Third Edition*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011.

[16] S. Kim, E. J. Whitehead, Jr., and Y. Zhang, "Classifying software changes: Clean or buggy?" *IEEE Trans. Softw. Eng.*, vol. 34, pp. 181–196, March 2008.

[17] A. Hindle, D. M. German, and R. Holt, "What do large commits tell us? A taxonomical study of large commits," in *Proceedings of the 2008 international working conference on Mining software repositories*, ser. MSR '08. ACM, 2008, pp. 99–108.

[18] A. Hindle, D. German, M. Godfrey, and R. Holt, "Automatic classication of large changes into maintenance categories," in *Program Comprehension, 2009. ICPC '09. IEEE 17th International Conference on*, may 2009, pp. 30–39.

[19] M. Stoerzer, B. G. Ryder, X. Ren, and F. Tip, "Finding failure-inducing changes in java programs using change classification," in *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, ser. SIGSOFT '06/FSE-14. ACM, 2006, pp. 57–68.

[20] J. Wloka, B. Ryder, F. Tip, and X. Ren, "Safe-commit analysis to facilitate team software development," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09. IEEE Computer Society, 2009, pp. 507–517.

[21] B. J. Williams and J. C. Carver, "Characterizing software architecture changes: A systematic review," *Information and Software Technology*, vol. 52, no. 1, pp. 1–51, 2010.

[22] E. Murphy-Hill, C. Parnin, and A. P. Black, "How we refactor, and how we know it," *Software Engineering, International Conference on*, vol. 0, pp. 287–297, 2009.

[23] E. Murphy-Hill and A. Black, "Refactoring tools: Fitness for purpose," *Software, IEEE*, vol. 25, no. 5, pp. 38 –44, sept.-oct. 2008.

[24] R. Robbes, M. Lanza, and M. Lungu, "An approach to software evolution based on semantic change," in *Fundamental Approaches to Software Engineering*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2007, vol. 4422, pp. 27–41.

[25] S. Kim, H. Zhang, R. Wu, and L. Gong, "Dealing with noise in defect prediction," in *Proceeding of the 33rd international conference on Software engineering*, ser. ICSE '11. ACM, 2011, pp. 481–490.