The impact of tangled code changes on defect prediction models

Kim Herzig · Sascha Just · Andreas Zeller

Abstract When interacting with source control management system, developers often commit unrelated or loosely related code changes in a single transaction. When analyzing version histories, such tangled changes will make all changes to all modules appear related, possibly compromising the resulting analyses through noise and bias. In an investigation of five open-source JAVA projects, we found between 7 % and 20 % of all bug fixes to consist of multiple tangled changes. Using a multi-predictor approach to untangle changes, we show that on average at least 16.6 % of all source files are incorrectly associated with bug reports. These incorrect bug file associations seem to not significantly impact models classifying source files to have at least one bug or no bugs. But our experiments show that untangling tangled code changes can result in more accurate regression bug prediction models when compared to models trained and tested on tangled bug datasets—in our experiments, the statistically significant accuracy improvements lies between 5 % and 200 %. We recommend better change organization to limit the impact of tangled changes.

Keywords Defect prediction · Untangling · Data noise

K. Herzig (✉)
Microsoft Research, 21 Station Road, Cambridge CB1 2DZ, UK
e-mail: kimh@microsoft.com

S. Just · A. Zeller
Saarland University, Campus E1.1, 66123 Saarbücken, Germany

S. Just
e-mail: just@st.cs.uni-saarland.de

A. Zeller
e-mail: zeller@cs.uni-saarland.de

1 Introduction

A large fraction of recent work in empirical software engineering is based on mining version archives—analyzing which changes were applied to a repository, by whom, when, why, and where. Such mined information can be used to predict related changes (Zimmermann et al. 2004), to predict future defects (Zimmermann et al. 2007; Menzies et al. 2010; Herzig et al. 2013), to analyze who should be assigned a particular task (Anvik et al. 2006; Bhattacharya 2011), to gain insights about specific projects (Li et al. 2011), or to measure the impact of organizational structure on software quality (Nagappan et al. 2008).

Most of these studies depend on the *accuracy* of the mined information—accuracy that is threatened by noise. Such noise can come from missing associations between change and bug databases (Bird et al. 2009; Bachmann et al. 2010; Nguyen et al. 2010). One significant source of noise, however, are *tangled changes*.

What is a tangled change? Assume a developer is assigned multiple tasks T_A, T_B, and T_C, all with a separate purpose: T_A is a bug fix, T_B is a feature request, and T_C is a refactoring or code cleanup. Once all tasks are completed, the developer commits her changes to the source control management system (SCM) to share her changes with other developers and to start product integration. 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 or overlap with other changes. 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". Although the commit message suggests that there are multiple tasks addressed at once, the individual changes are merged together and cannot be tracked back to their original task.

Such tangled changes do not cause serious trouble in development. However, they introduce *noise* in any analysis of the corresponding version archive, thereby compromising the accuracy of the analysis. As the tangled change fixed a bug, all files touched by it are considered as being defective in the past, even though the tangled tasks *B* and *C* are not related to any 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 that will now be related with each other as well as the tasks T_A, T_B, and T_C.

The problem of tangled changes is not a theoretical one. In this paper, we present an empirical study on five open-source JAVA projects to answer these questions. We manually classified more than 7,000 change sets as being tangled or atomic. The result of this manual classification shows that tangled change sets occur frequently, with up to 15 % of all bug fixing change sets applied to the subject projects being tangled. Using an automated, multi-predictor untangling algorithm and comparing classic bug count datasets with bug count datasets derived after untangling tangled code changes, we show that between 6 % and 15 % of all fixes address multiple concerns at once—they are tangled and therefore introduce noise into any analysis of the respective change history. Considering only bug fixes, the ratio of tangled code changes lies between 7 % and 20 %. On average, at least 16.5 % of all source files are incorrectly associated with bug reports when ignoring the existence of tangled change sets. In terms of impact, this means that between 6 % and 50 % (harmonic mean: 17.4 %) of files originally marked as most defect prone do not belong to

this category.[1] Further, our experiments indicate that classification models predicting source files with at least one bug are not significantly impacted by tangled changes while regression models predicting the bug likelihood of source files are significantly impacted. In our experiments, regression models trained and tested on untangled bug datasets show a statistically significant precision improvement between 5 % and 200 % (median improvement at 16.4 %).

In this paper, we present an approach to untangle changes splitting tangled change sets into smaller *change set partitions*, whereby each partition contains a subset of change operations that are related to each other, but not related to the change operations in other partitions. The algorithm is based on static code analysis only and is fully automatic, allowing archive miners to untangle tangled change sets and to use the created change partitions instead of the original tangled change set. Our experiments on five open-source JAVA projects show that neither data dependencies, nor distance measures, nor change couplings, nor distances in call graphs serve as a one-size-fits-all solution. By combining these measures, however, we obtain an effective approach that untangles multiple combined changes with a mean precision of 58 %–80 %.

2 Background

A number of researchers have classified code changes, studied the relations between code changes and noise and bias in version archive datasets. This paper is an extended version of a previous study published in the Proceedings of the 10th Working Conference on Mining Software Repositories (Herzig and Zeller 2013).

2.1 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 only the closely related studies.

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 focused on classifying code changes based on their quality (Kim et al. 2008) or on their purpose (Hindle et al. 2008, 2009). Kim et al. (2008) developed a change classification technique 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 semantic difference and it would require a bias free software history. In addition, it would only work for maintenance of existing features, not with newly added features. Hindle et al. (2008, 2009) analyzed large change sets that touch a large number of files to automatically classify the maintenance category of the individual changes. Their 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. Mockus and Votta (2000) compared bug fixes against feature

---

[1]These findings confirm results of earlier research presented by Kawrykow and Robillard (2011), Kawrykow (2011).

implementing code changes and showed that bug fixes tend to be smaller than other changes. Herzig (2012) used change genealogies (a directed graph structure modeling dependencies between code changes) to classify code changes. He showed that bug fixing code changes modify fewer methods and changes mostly older code while feature implementations are based on newer code fragments. Similar findings were reported by Alam et al. (2009).

Stoerzer et al. (2006) used a change classification technique to automatically detect code changes contributing to test failures. Later, this work was extended by Wloka et al. (2009) 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. Even for changes covered by test cases, both approaches could be helpful to support untangling purposes; however, these approaches on their own are not designed to untangle tangled code changes. The fact that a partial code change does not cause any test failure has no implication on whether this partial code change is part of a bigger code change or whether it is a complete atomic set of changes.

Williams and Carver (2010) 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.

2.2 Refactorings

Changes that combine refactorings with code changes applying semantic differences to a program can be considered a sub-category of tangled changes. Thus, stripping refactorings from code changes can be considered a special case of the untangling problem. Murphy-Hill et al. (2009), Murphy-Hill and Black (2008) analyzed development activities to prove, or disprove, several common assumptions about how programmers refactor. Their results show that developers frequently do not explicitly state refactoring activities, which increases the bias potential discussed in this paper, even further. Later, Kawrykow and Robillard 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" (Kawrykow and Robillard 2011). We compare and discuss results presented by Kawrykow and Robillard (2011) in more detail in Section 6.

2.3 Change Dependencies

The problem that version archives do not capture enough information about code changes to fully describe them is not new. Robbes et al. (2007) showed that the evolutionary information in version archives may be incomplete and of low quality. Version archives treat software projects as a simple set of source files and sets of operations on these files. However, these operations have different purposes and are certainly not independent from each other, a fact not captured by version archives. As a partial solution, Robbes et al. (2007) 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.

2.4 Untangling Changes

To the best of our knowledge there exists only one other study that evaluated an untangling algorithm similar to the algorithm presented in this paper. In his master thesis, Kawrykow (2011) presented and evaluated a multi-heuristic untangling algorithm developed in parallel to the approach presented in this paper. Kawrykow based his untangling algorithm on statement level and ensures to return real patches that can be applied. In contrast, the approach presented in this paper was developed in order to show the impact of tangled changes. The untangling precision of Kawrykow's change operation lies slightly below the precision values reported in this paper. We compare and discuss results presented by Kawrykow and Robillard (2011) in more detail in Section 6.

2.5 Noise and Bias in Version Archive Datasets

In recent years, the discussion about noise and bias in mining datasets and their effect on mining models increased. Lately, Kawrykow and Robillard (2011) 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, their results are a strong indication that unrelated code changes applied together will have similar effects.

Dallmeier (2010) analyzed bug fix change sets of two open source projects minimizing bug fixes to a set of changes sufficient to make regression tests pass. On average only 50 % of the changed statements were responsible to fix the bug suggesting that these bug fixes were tangled–the remaining 50 % of the applied code changes applied changes that were not necessary to fix the program semantics.

The effects of bias caused by unbalanced data sets on defect prediction models were investigated by various studies (Bird et al. 2009; Bachmann et al. 2010; Nguyen et al. 2010). Bird et al. 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 generalization of hypotheses tested on biased data" (Bird et al. 2009). Kim et al. (2011) 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.

Lately, Herzig et al. (2013) manually classified more than 7,000 bug reports and showed that a significant number of bug reports do not contain bug descriptions but rather feature or improvement requests. This leads to noise when mapping these false bug reports to source files. The authors showed that on average 39 % of files marked as defective actually never had a bug–were associated solely with false bug reports. This misclassification introduces bias into heuristics automatically classifying code changes based on their associations with issue reports.

3 Research Questions

In the introduction, we briefly discussed the potential impact of tangled change sets when building defect prediction models. Counting the number of applied bug fixes per source file over a set of change sets that contain at least one tangled change set combining a bug fix with some other change (e.g. a feature implementation) over multiple source files may result in a wrong bug count. For tangled change set, it remains unclear, which source file changed

due to the bug fix and which one was changed in order to add the new feature. Not being able to tell which file was changed in order to fix the bug, means to either count one bug for all changed files or to choose (using some heuristic or random) a subset of files that will be associated with the bug fix. Both strategies may end up with a false positive and thus might introduce noise.

The research questions tackled by this paper are designed to determine whether tangled changes impact bug count and bug prediction models. Bug prediction models such as the BugCache algorithm (Kim et al. 2007) or classification models identifying the most defect prone code areas in a product (Zimmermann et al. 2007) use the number of bugs associated to code areas, such as source files, to support quality assurance processes. In general, there are two kind of recommender: those that use the absolute number of bugs associated with the code entity (regression models) and those that use a threshold value of bugs to mark code areas as defect prone (classification models). In case that tangled changes impact such statistical learners significantly, we should considered tangled code changes harmful. To achieve our goal, we have to complete four basic steps, each dedicated to research questions of lower granularity.

3.1 RQ1: How Popular are Tangled Changes?

First, we check whether tangled changes appear to be a theoretical problem or a practical one and if tangled changes do exist. Is the fraction of tangled changes large enough to threaten bug count models? If only one percent of all applied code changes appear to be tangled, it is unlikely that these tangled changes can impact aggregating bug count models. Further, we investigate how many individual tasks (*blob size*) make up common tangled changes. The more tasks get committed together, the higher the potential number of modified files and thus the higher the potential impact on bug count models ignoring tangled changes. The higher the blob size the more difficult it might be to untangle these changes.

3.2 RQ2: Can we Untangle Tangled Changes?

Knowing that there exist tangled changes and that they might impact quality related models is raising awareness but is no solution. There are two main strategies to deal with the issue of tangled change sets.

Removing tangled changes   and ignoring these data points in any further analysis. But this solution makes two major assumptions. First, one must be able to detect tangled change sets automatically; second, the fraction of tangled change sets must be small enough such that deleting these data points does not cause the overall data set to be compromised.

Untangling tangled changes   into separate change partitions that can be individually analyzed. This strategy not only assumes that we can automatically detect but also untangle tangled changes sets. But it makes no assumptions about the fraction of tangled changes and thus should be the preferred option.–

3.3 RQ3: How do Tangled Changes Impact Bug Count Models?

Although, we would like to answer this research question before *RQ2*—if tangled changes have no impact we do not need to untangle them—we can only measure the impact of tangled changes once we are able to compare corresponding models against each other.

Thus, we require two datasets; one dataset containing bug counts for code artifacts collected without considering the issue of tangled changes, and one dataset with tangled changes being removed. Removing tangled changes requires us to untangle them.

3.4 RQ4 How do Tangled Changes Impact Bug Prediction Models?

Even if tangled code changes impact bug count models (*RQ3*) it is still questionable whether bug prediction models will be impacted as well. Changes to bug counts might be distributed in such a way that bug prediction models might not show any significant accuracy difference. Such a scenario would imply that tangled changes are distributed equally or only affect bug counts of already bug prone files. Such a result would also mean that we could ignore tangled changes when building bug prediction models. Similar to *RQ3* we compare two groups of bug prediction models. One group trained and tested on tangled bug data. The other group trained and tested on untangled bug data. In fact, we reuse the bug data sets used to answer *RQ3* and combine the bug counts with network metrics (Zimmermann and Nagappan 2008). We use network metrics as independent variables to train classification and regression models to predict both, tangled and untangled bug counts. Comparing the accuracy results of both prediction model groups allows us to reason about the impact of tangled changes on defect prediction models.

4 Experimental Setup

To answer our four research questions, we conduct four experiments described in this section.

4.1 Measuring Bias Caused by Tangled Changes (RQ1)

We conducted an exploratory study on five open-source projects to measure how many tangled change sets exist in real world development SCMs. Overall, we manually classified more than 7,000 individual change sets and checked whether they address multiple (*tangled*) issue reports. More precisely, we classified only those change sets for which the corresponding commit message references at least one issue report (e.g. bug report, feature request, etc.) that had been marked as resolved. If the commit message clearly indicated that the applied changes tackle more than one issue report we classified the change set as tangled. This can either be commit messages that contain more than one issue report reference (e.g. "Fix JRUBY-1080 and JRUBY-1247 on trunk.") 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. Separate references to multiple issue reports marked as duplicate to each were considered as single reference. For example, if bug B_i is marked as a duplicate of bug B_k and both are mentioned in the commit message, we increase the bug count of corresponding source files by one.

To measure the amount of tangled changes, we conducted 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. 2007). Change sets linked to multiple bug reports, not marked as duplicates, are considered tangled.

2. Each change set from Step 1 was manually inspected and classified as atomic or tangled. During manual inspection, we considered the commit message and the actual applied code changes. In many cases, the commit message already indicated a tangled change set. Only if we had no doubt the change set targeted more than one issue or that additional changes (e.g. clean-ups) were applied we classified the change set as tangled. Similar, only if we had no doubt that the change set is atomic, we classified it as atomic. Any change set that we could not strictly mark as atomic or tangled were not classified and remained undecided. All undecided change sets are excluded from the presented results.

4.2 Untangling Changes (RQ2)

To answer *RQ2*, we developed a prototype of a heuristic-based untangling algorithm that expects an arbitrary change set as input and returns a set of *change set partitions*—a subset of changes applied by the original, tangled change set. The union of all returned change set partitions equals the original tangled change set.

In general, determining whether two code changes are unrelated is undecidable, as the halting problem prevents prediction whether a given change has an effect on a given problem.[2] Consequently, every untangling algorithm will have to rely on heuristics to present an approximation of how to separate two or more code changes. The aim of the presented algorithm is not to solve the untangling problem completely, but aims to verify whether untangling code changes is feasible and to evaluate the accuracy of such an algorithm. With a reasonable good accuracy, we may use the untangling algorithm to reduce the amount of bias significantly. The untangling algorithm itself is described in Section 5.

In Subsection 6.1 we show that a significant proportion of change sets must be considered as tangled. To evaluate any untangling algorithm we cannot rely on existing data. We cannot determine whether a produced change set partition is correct or not since there is no existing oracle for such an analysis. Additionally, we are not able to quantify the difference of the created partitioning from an expected result.

To determine a reliable set of atomic change sets—change sets containing only those code changes required to resolve exactly one issue—we use the manual classified atomic change sets derived from the experiments of RQ1 (Subsection 4.1) to generate *artificial tangled change sets* for which we already know the correct partitioning. As an alternative, we could manually untangle real tangled change sets to gain knowledge about the correct partitioning of real world tangled change sets. But manually untangling tangled change sets requires detailed project and source code knowledge and a detailed understanding of the intention behind all change operations applied within a change set. As project outsiders we know too little project details to perform a manual untangling and all wrongly partitioned tangled change sets added to the set of ground truth would bias our evaluation set.

In principal, combining atomic change sets into artificially tangled change sets is straightforward. Nevertheless, we have to be careful which atomic change sets to tangle.

---

[2]Since it is undecidable whether a program will ever terminate under arbitrary conditions, we are, in general, also unable to decide whether two code changes may influence each other during a possible infinite program run.

Combining them randomly is easy but would not simulate real tangled change sets. In general, developers act on purpose. Thus, we assume that in most cases, developers do not combine arbitrary changes, but 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 extent, we combined change sets using the following three tangling strategies:

Change close packages (pack)   Using this strategy we combine only change sets that contain at least two change operations touching source files that are not more than two sub-packages apart. As an example, assume we have a set of three change sets changing three classes identified using the full qualified name: $CS_1 = \{com.my.foo.intern.F_1\}$, $CS_2 = \{com.my.foo.extern.F_2\}$, and $CS_3 = \{com.your.foo.intern.F_3\}$. Each class is identified by its fully qualified name. In the example, the strategy would combine CS_1 with CS_2 but not CS_1 nor CS_2 with CS_3.

Frequently changed before (coupl.)   This strategy computes and uses change-coupling rules (Zimmermann et al. 2004). Two code changes are only tangled if in history at least two code artifacts changed by different change sets showed to be frequently changed together.

    For example, let CS_i and CS_j be a pair of atomic change sets and let CS_i be applied before CS_j. CS_i changed file F_s while CS_j changed file F_t. First, we compute all change coupling rules using the approach of Zimmermann et al. (2004) and call this set $\mathcal{S}$. The computed change coupling rules indicate how frequently F_s and F_t were changed together before CS_i was applied. We combine CS_i and CS_j only if $\mathcal{S}$ contains a file coupling rule showing that F_s and F_t had been changed in at least three change sets applied before CS_i. Further, we require that in at least 70 % of change sets applied before CS_i that changes either F_s or F_t the corresponding other file were changed as well.

Consecutive changes (consec.)   We combine consecutive change sets applied by the same author (not necessarily consecutive in the SCM). Consecutive change sets are change set that would have ended up in a tangled change set if the developer forgot to commit the previous change set before starting a new developer maintenance task.

For technical reasons, we limited all strategies to combine only atomic change sets that lie no more than 14 days apart. The untangling algorithm (described in Section 5) must be provided with a code base that must be compilable. Longer time periods between atomic change sets imply higher probability that merging change sets will lead to uncompilable code. Note, that we did not restrict change sets to be tangled on any other aspect. In particular, we also tangled atomic changes with overlapping sets of changed source files. These tangled code changes might be harder to untangle and might modify a shared state rather than individual variables. However, we did not apply any restriction on how to tangle changes over sets of files. In particular, the tangling strategies of combining files in code packages (pack) as well as consecutive changes by the same author (consec) create tangled changes with overlapping file sets.

Artificially tangled change sets may not necessarily represent all kinds of natural occurring tangled changes. However, the strategies to combine consecutive changes and changes modifying code entities in close proximity to each other simulate engineers that tend to produce less change sets and compile consecutive work items into single change sets. Verifying whether artificially tangled code changes represent naturally occurring tangled code changes would require to solve the untangling problem first and requires detailed knowledge about the code changes themselves.

To evaluate the accuracy of our untangling algorithm, we generate all possible artificially tangled change sets using all three tangling strategies described above (this may include duplicate tangled change sets). Since we know the origin of each change operation, we can compare the expected partitioning with the partitioning produced by the untangling algorithm (see Fig. 1). 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 tangled change sets $\mathcal{B}$, we define precision as

$$\text{precision} = \frac{\text{\# correctly assigned change operations}}{\text{total \# change operations} \in \mathcal{B}}$$

As an example for precision, consider Fig. 1. In the tangled change set, we have 9 change operations overall. Out of these, two are misclassified (the black one in the middle partition, and the gray one in the lower partition); the other seven change operations are assigned to the correct partition. Consequently, the precision is $7/9 = 77.\overline{7}\,\%$, implying that $2/9 = 22.\overline{2}\,\%$ of all changes need to be re-categorized in order to obtain the ground truth partitioning.

For each set of tangled change set, there exist multiple precision values. The precision depends on which change set partition is compared against which original atomic change set. Taking the example shown in Fig. 1, we would intuitively compare the created partitions with the ground truth set in the same order as shown in Fig. 1. However, we could also compare to created partition containing the two white pieces with the one containing the three black pieces from ground truth and vice versa. Both comparison yield different precision values. All together there exist $3! = 6$ different ways to compare created partitions with ground truth partitions. Precision values reported in this paper correspond to the partition association with the highest sum of *Jaccard indices* (Jaccard 1901). The higher the Jaccard index the higher the similarity of the created partitions when compared the the ground truth partition set. Thus, by maximizing the sum of Jaccard indices over a set of association permutations relating partitions with atomic change sets we chose the association permutation with the highest similarity of associated pairs. Short, we report the best precision value over all existing association permutations.

The number of individual tasks compiled into a tangled change set, called *blob size*, may vary. To check the untangling performance we generate artificial change sets of blob sizes
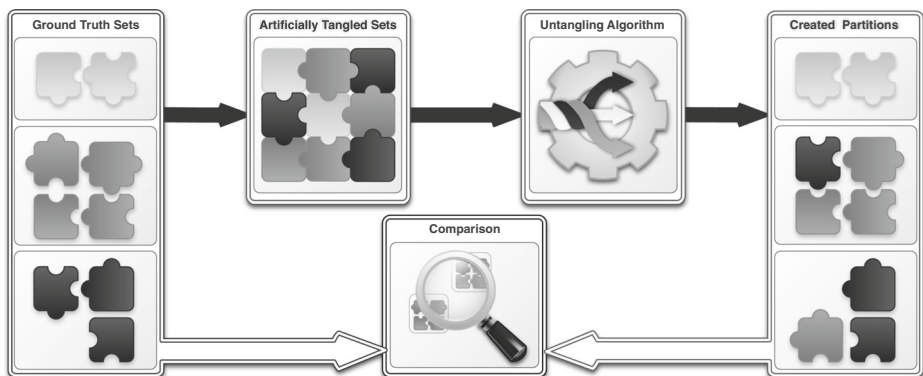

Fig. 1 Artificially tangled change sets are generated using manually classified atomic change sets to compare created partitions and desired output. In the example, two change operations are put into a wrong partition, and hence the success rate is $\frac{7}{9} = 77.\overline{7}\,\%$

two, three, and four (tangled change sets with a bob size larger than four are rare, see Section 6.1).

4.3 Measuring the Impact of Tangled Changes on Bug Count Models (RQ3)

To show the impact of tangled change sets on bug count models, we compare two different bug count datasets: the *original sets* against *untangled sets* produced in the experiments to answer RQ2 (Section 4.2). For the original reference dataset, we associate all referenced bug reports to all source files changed by a change set, disregarding whether we marked it tangled or not. For the untangled bug count set, we used our untangling algorithm to untangle manually classified tangled change sets. If the tangled change set references bug reports only, we assigned one bug report to each partition—since we only count the number of bugs, it is not important which report gets assigned to which partition. For change sets referencing not only bug reports we used an automatic change purpose classification model based on the findings of Mockus and Votta (2000) and Hindle et al. (2008, 2009) indicating that bug fixing change sets apply less change operations when compared to feature implementing change sets. Thus, we classify those partitions applying the fewest change operations as bug fixes. Only those files that were changed in the bug fixing partitions were assigned with one of the bug reports. In this study, we are only interested in a files bug count (the number of bug reports a file is associated to) rather than the actual association between files and the exact bug report–referring to bug #1234 rather than #1235 makes no difference. Thus, it suffices to assign any bug report to the file ignoring the unsolved problem of which bug report would be the one fixed by the code change. In general, this is remains an open problem, which is out of scope of this study. Both bug count sets contain all source files, also those that were not changed in tangled change sets, and are sorted in descending order using the distinct number of bug reports associated with the file (see Fig. 2).

The most defect-prone file is the top element in each bug count set. Both sets contain the same elements but in potentially different order. Although the rank correlation between those two ordered sets can tell us how big the influence of the tangled changes is when it comes to the actual ordering, it does not provide any insight about the most important changes. Typically, one is only interested in investing the top x % of defect prone entities and not their particular order. Hence, one wants to determine changes in the set of top x files. Thus, we compare the `cutoff`-difference of the top x% of both file sets, which allows us to reason about the impact of tangled change sets on models using bug counts to identify the most defect-prone entities. Since both `cutoffs` are equally large (the number of source files does not change, only their ranks), we can define the `cutoff_difference` as the size of the symmetric difference between the most frequently fixed files—once determined using the original dataset and once using the untangled dataset—normalized by the number of files in the top x% (see Fig. 2). The result is a number between zero and one where zero

$$
\begin{aligned}
\text{original} &:= \text{Files set ordered (descending) by original bug count.} \\
\text{untangled} &:= \text{File set ordered (descending) by bug count after untangling.} \\
\texttt{top}_x(\textit{files}) &:= \text{Top x\% of the } \textit{files}. \\
\texttt{cutoff}_x &:= \{\texttt{top}_x(\text{original}) \cup \texttt{top}_x(\text{untangled})\} \setminus \{\texttt{top}_x(\text{original}) \cap \texttt{top}_x(\text{untangled})\} \\
\texttt{cutoff\_difference}_x &:= \frac{|\texttt{cutoff}_x|}{|\{\texttt{top}_x(\text{original}) \cup \texttt{top}_x(\text{untangled})\}|}
\end{aligned}
$$

Fig. 2 Computing the `cutoff_difference`

indicates that both `cutoffs` are identical and where a value of one implies two `cutoffs` with an empty intersection. A low `cutoff_difference` is desirable.

4.4 Measuring the Impact of Tangled Changes on Bug Prediction Models (RQ4)

Similar to the previous experiment, we use again two competing groups of models: bug prediction models trained and tested on original, tangled bug data and models trained and tested on untangled bug data (see RQ3, Section 4.3). For this set of experiments, we used code dependency network metrics (Zimmermann and Nagappan 2008) as independent variables. These metrics express the information flow between code entities modeled by code dependency graphs. Zimmermann and Nagappan (2008) and others (Tosun et al. 2009; Premraj and Herzig 2011; Bird et al. 2009) showed that network code metrics have strong bug prediction capabilities. However, the choice of metrics is secondary as the focus of this work is to measure the impact of tangled code changes on prediction models rather than constructing the best prediction mode possible. The list of used network metrics including a brief description is shown in Table 1. The set of network metrics used in this work slightly differs from the original metric set used by Zimmermann and Nagappan (2008). We computed the used network metrics using the *R statistical software* (R Development Core Team 2010) and the *igraph* (Csardi and Nepusz 2006) package. Using *igraph*, we could not re-implement two of the 25 original network metrics: ReachEffciency and Eigenvector.

To measure the impact of tangled code changes on defect prediction models, we compare models trained and tested on tangled bug data against models trained and tested on untangled data. Further, we build two different kinds of models: classification models and regression models. Classification models simply decide whether a source file will have at least one bug or whether the source files will have no bugs. In contrast, regression models return a discrete number expressing the conditional expectation of bugs to be associated with the source file.

For both, regression and classification models, we perform a stratified repeated holdout setup to train and test models based on network metrics—the ratio of bug fixing change sets in the original data set is preserved in both training and testing data sets. This makes training and testing sets more representative by reducing sampling errors. Next, we split the training and testing sets into subsets. Each subset contains the complete set of all source files and all metrics. The difference between both subsets lies in the values of bug counts—the dependent variable. While one subset contains the original bug counts, noised by tangled changes, the other subset contains bug counts that reflect bug associations after untangling. Separating tangled and untangled dependent variables for already created training and testing sets ensures that we create pairs of prediction models using the exact same training and testing instances but using different dependent variables. We repeatedly sample the original data sets 100 times in order to generate 100 independent training (2/3 of original dataset) and testing (1/3 of original dataset) sets. For each cross-fold we then train the models using six different machine learners (see Table 2).

4.4.1 Prediction Accuracy for Classification Models

We measure the prediction accuracy for classification models using precision, recall, and F-measure for each cross-fold. To compare prediction performances for tangled and untangled models, we then compare the median precision, recall, and F-measure values, as well as the variance of these accuracy measures (different machine learners and different models for

Table 1 List of code dependency network metrics

Metric name	Description
Ego-network metrics (computed each for incoming, outgoing, and undirected dependencies; descriptions adapted from Z&N):	
Size	# nodes connected to the ego network
Ties	# directed ties corresponds to the number of edges
Pairs	# ordered pairs is the maximal number of directed ties
Density	% of possible ties that are actually present
WeakComp	# weak components in neighborhood
nWeakComp	# weak components normalized by size
TwoStepReach	% nodes that are two steps away
Brokerage	# pairs not directly connected. The higher this number, the more paths go through ego
nBrokerage	Brokerage normalized by the number of pairs
EgoBetween	% shortest paths between neighbors through ego
nEgoBetween	EgoBetween normalized by the size of the ego network
Structural metrics (descriptions adapted from Z&N):	
EffSize	# entities that are connected to an entity minus the average number of ties between these entities
Efficiency	Normalizes the effective size of a network to the total size of the network
Constraint	Measures how strongly an entity is constrained by its neighbors
Hierarchy	Measures how the constraint measure is distributed across neighbors. When most of the constraint comes from a single neighbor, the value for hierarchy is higher
Centrality metrics (computed each for incoming, outgoing, and undirected dependencies; descriptions adapted from Z&N):	
Degree	# dependencies for an entity
nDegree	# dependencies normalized by number of entities
Closeness	Total length of the shortest paths from an entity (or to an entity) to all other entities
Reachability	# entities that can be reached from a entity (or which can reach an entity)
alpha.centrality[†]	Generalization of eigenvector centrality (Bonacich 1987)
Information	Harmonic mean of the length of paths ending in entity
Betweenness	Measure for a entity in how many shortest paths between other entities it occurs
nBetweenness	Betweenness normalized by the number of entities

[†]Metrics not used by Z&N

individual cross-folds can vary; comparing the variances allows reasoning about the stability of the received models).

4.4.2 Prediction Accuracy for Regression Models

Predicted values of regression models should not be considered as absolute values but treated as indication how high the chances are that the corresponding source file is or will be associated with bugs. The higher the predicted regression value is, the higher the chances

Table 2 List of models used for classification experiments

Model[*]	Description
k-nearest neighbor (*knn*)	This model finds k training instances closest in Euclidean distance to the given test instance and predicts the class that is the majority amongst these training instances.
Logistic regression (*multinom*)	This is a generalized linear model using a logic function and hence suited for binomial regression, i.e. where the outcome class is dichotomous.
Recursive partitioning (*rpart*)	A variant of decision trees, this model can be represented as a binomial tree and popularly used for classification tasks.
Support vector machines (*svmRadial*)	This model classifies data by determining a separator that distinguishes the data with the largest margin. We used the radial kernel for our experiments.
Tree Bagging (*treebag*)	Another variant of decision trees, this model uses bootstrapping to stabilize the decision trees.
Random forest (*randomForest*)	An ensemble of decision tree classifiers. Random forests grow multiple decision trees each "voting" for the class on an instance to be classified.

[*]For a fuller understanding of these models, we advise the reader to refer to specialized machine learning texts such as by Witten and Frank (2002)

it being buggy. Thus, comparing observed and predicted values makes little sense. Instead, we can re-use the concept of `cutoff_difference` from Section 4.3 (*RQ3*).

We rank observed and predicted values in decreasing order and report the overlap of both sets top x %—disregarding the order within these sets. We report the relative size of the overlap as the models precision value `regression_precision` (Fig. 3). As for the classification models, we report the median precision values for each machine learner and study subjects as well as a box plot showing the diversity of precision values across cross-folds.

4.5 Study Subjects

All experiments are conducted on five open-source JAVA projects (see Table 3). We aimed to select projects that were under active development and were developed by teams for which at least 48 months of active history were available. We also aimed to have datasets that

$$\text{observed} := \text{Files set ordered (descending) by observed bug count.}$$
$$\text{predicted} := \text{File set ordered (descending) by predicted bug count.}$$
$$\texttt{top}_x(\textit{files}) := \text{Top x\% of the } \textit{files}.$$
$$\texttt{overlap}_x := \{\texttt{top}_x(\text{observed}) \cap \texttt{top}_x(\text{predicted})\}$$
$$\texttt{regression\_precision}_x := \frac{|\texttt{overlap}_x|}{|\{\texttt{top}_x(\text{predicted})\}|}$$

Fig. 3 Computing the `regression_precision`

Table 3 Details of projects used during experiments

	ArgoUML	GWT[†]	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
# Change Sets	16,481	5,326	1,353	11,134	1,756

[†]GOOGLE WEBTOOL KIT

contained a manageable number of applied bug fixes for the manual inspection phase. For all projects, we analyzed more than 50 months of active development history. Each project counts more than 10 active developers. 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 3,000 (ARGOUML and JRUBY).

5 The Untangling Algorithm

The untangling algorithm proposed in this paper expects an arbitrary change set as input and returns a set of *change set partitions*—a subset of changes applied by the original, tangled change set. Each partition contains code changes that are related closer to changes in the same partition than to changes contained in other partitions. Ideally, all necessary code changes to resolve one issue (e.g. a bug fix) will be in one partition and the partition only consists of changes for this issue. The union of all partitions equals the original change set. Instead of mapping issues or developer tasks to all changed code artifacts of a change set, one would assign individual issues and developer tasks to those code artifacts that were changed by code changes contained in the corresponding change set partition.

To identify related code changes we use the same model as Herzig et al. (2013) split each change set into a set of individual *change operations* that added or deleted method calls or method definitions. Thus, each change set corresponds to a set of change operations classified as adding or deleting a method definition (AD, DD) or a method call (AC, DC). Using an example change set that applied the code change shown in Fig. 4 we derive a set containing two change operations. One DC deleting *b.bar(5)* and one AC adding *A.foo(5f)*. Note that there exists no change operation changing the constructor definition *public C()* since the method signature keeps unchanged. All change operations are bound to those files and line numbers in which the the definition or call was added or deleted. In our example, the


```
3        public class C {
4                public C() {
5                        B b = new B();
6                        b.bar(5);          // fixes a wrong method
                         A.foo(5f);         // call in line 6 in class C
7                }
8        }
```

Fig. 4 Example change set printed as unified diff containing two change operations: one DC deleting the method call *b.bar(5)* and one AC adding the method call *A.foo(5f)*

DC and AC change operations are bound to line 6. Rename and move change operations are treated as deletions and additions. Using this terminology, our untangling algorithm expects a set of change operations and produces a set of sets of change operations (see Fig. 5).

For each pair of applied 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 significant lower than the distance between two unrelated change operations. The relation between change operations may be influence by multiple facts. 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.

5.1 Confidence Voters

To combine various dependency and relation aspects between change operations, the untangling framework itself does not decide which change operation are likely to be related but asks a set of so called *confidence voters* (CONFVOTERS) (see Fig. 5). Each CONFVOTER expects a pair of change operations and returns a *confidence value* between zero and one.
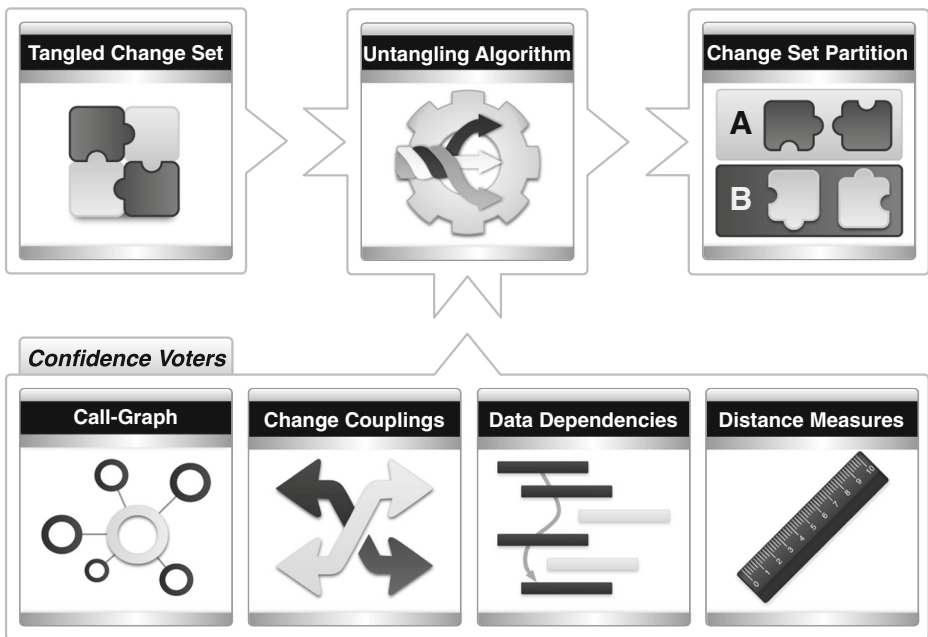

Fig. 5 The untangling algorithm partitions change sets using multiple, configurable aspect extracted from source code. *Gray boxes* represent sets of change operations necessary to resolve one issue

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.

CONFVOTERS can handle multiple relation dependency aspects within the untangling framework. Each CONFVOTER represents exactly one dependency aspect. Below we describe the set of CONFVOTERS used throughout our experiments.

FileDistance   Above we discussed that change operations are bound to single lines. This CONFVOTER returns the number of lines between the two change-operation-lines divided by the line length of the source code file both change operations were applied. If both change operations were applied to different files this CONFVOTER will not be considered.

PackageDistance   If both change operations were applied to different code files, this CONFVOTER will return the number of different package name segments comparing the package names of the changed files. This CONFVOTER will not be considered otherwise.[3]

CallGraph   Using a static call graph derived after applying the complete change set we identify the change operations and measure the call distance between two call graph nodes. The call graph distance between two change operations is defined as the sum of all edge weights of the shortest path between both modified methods. The edge weight between m_1 and m_2 is defined as one divided by the number of method calls between m_1 and m_2. This voter returns the actual distance value as described. The lower the value, the more likely the two code entities belong together and will have changed together.

ChangeCouplings   The confidence value returned by this CONFVOTER is based on the concept of change couplings as described by Zimmermann et al. (2004). The CONFVOTER computes frequently occurring sets of code artifacts that were changed within the same change set. The more frequent two files changed together, the more likely it is that both files are required to be changed together. The confidence value returned by this CONFVOTER indicates the probability that the change pattern will occur whenever one of the patterns components change.

DataDependency   Returns a value of one if both changes read or write the same variable(s); returns zero otherwise. This relates to any JAVA variable (local, class, or static) and is derived using a static, intra-procedural analysis.

We will discuss in Subsection 5.2 how to combine the confidence values of different CONFVOTERS.

---

[3]This CONFVOTERS is slightly penalized by the artificial blob generation strategy pack creating blobs by combining changes to files based on directory distance (see Subsection 4.2). However, we favored a more realistic distribution of changes over total fairness across all CONFVOTERS.

5.2 Using Multilevel Graph Partitioning

Our 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 algorithm proposed by Karypis and Kumar (1995a, b, 1998). We use a triangle partition matrix to represent existing untangling partitions and the confidence values indicating how confident we are that two corresponding partitions belong together. We will start with the finest granular partitioning and merge those partitions with the highest merge confidence value. After each partition merge we delete two partitions and add one new partition representing the partition union of the two deleted partition. Thus, in each partition merge iteration, we reduce the dimension of our triangle partition matrix by one. We also ensure that we always combine those partitions that are most likely related to each other. The algorithm performs the following steps:

1. Build an $m \times m$ triangle partition matrix $\mathcal{M}$ containing one row and one column for each change set partition. Start with the finest granular partitioning of the original change set—one partition for each change operation.
2. For each matrix cell $[P_i, P_j]$ with $i < j \leq m$ of $\mathcal{M}$, we compute a *confidence value* indicating the likelihood that the partitions P_i and P_j are related and should be unified (see Subsection 5.1 for details on how to compute these confidence values). The confidence value for matrix cell $[P_i, P_j]$ equals the confidence value for the partition pair (P_j, P_i). Figure 6 shows this step in detail.
3. Determine the pair (P_s, P_t) of partitions with the highest confidence value and with $s \neq t$. We then delete the two rows and two columns corresponding to P_s and P_t and add one column and one row for the new partition P_{m+1}, which contains the union of P_s and P_t. Thus, we combine those partitions most likely being related.
4. Compute confidence values between P_{m+1} and all remaining partitions within $\mathcal{M}$. For the presented results, we took the maximum of all confidence values between change operations stemming from different partitions:

$$Conf\ (P_x, P_y) = Max\{Conf(c_i, c_j) \mid c_i \in P_x \ \wedge \ c_j \in P_y\}.$$

The intention to use the maximum is that two partitions can be related but having very few properties in common.

Without determining a stopping criterion, this algorithm would run until only one partition is left. Our algorithm can handle two different stopping strategies: if a fixed number of partitions is reached (e.g. knowing the partitions from parsing the commit message) or if no
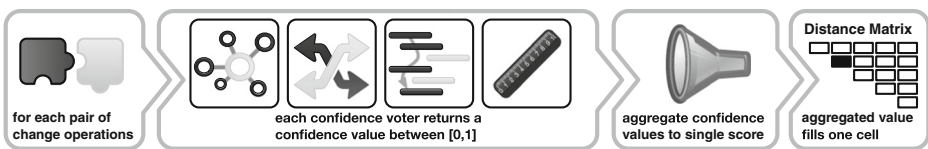

Fig. 6 The procedure to build the initial triangle matrix used within the modified multilevel graph partitioning algorithm

cell within $\mathcal{M}$ exceeds a specified threshold. In this paper, the algorithm is used to untangle manually classified tangled change sets, only. For each of these change sets we know the number of desired partitions. Thus, for all experiments we stopped the untangling algorithm once the desired number of partitions was created.

So far, the untangling algorithm represents a partitioning framework used to merge change operations. This part is general and makes no assumptions about code or any other aspect that estimates the relation between individual operations. It is important to notice that the partitions do not overlap and that change operations must belong to exactly one partition.

5.3 Remarks on Untangling Result

The output of the untangling algorithm is a change set partitioning. As discusses above, each created partition contains code changes that are related closer to changes in the same partition than to changes contained in other partitions. However, the algorithm does not suggest the purpose of an individual partition. Thus, it is not possible to determine which partition contains those code changes more likely to apply a bug fix or serving any other purpose. Deciding which partition applies a bug fix is based on heuristics. Findings of Mockus and Votta (2000) and Hindle et al. (2008, 2009) indicate that bug fixing change sets apply less change operations when compared to feature implementing change sets (see Subsection 4.3). Further, the untangling algorithm does not allow to judge which bug fix was applied in which partition. However, this knowledge is not required. The bug prediction models investigated in this study only count the number of distinct bug reports associated with a source file rather than considering the individual association pair Thus, the exact mapping between source files and bug reports is decisive.

6 Results

In this section, we present the results of our four experimental setups as presented in Section 4 and briefly discuss the results of this study with similar results provided by Kawrykow and Robillard (2011), Kawrykow (2011).

6.1 Tangled Changes (RQ1)

The results of the manual classification process are shown in Table 4. In total, we manually classified more than 7,000 change sets. Row one of the table contains the total number of change sets that could be associated with any issue report (not only bug reports). Rows two and three are dedicated to the total number of change sets that could be associated to any issue report and had been manually classified as tangled or atomic, respectively. The numbers show that for the vast majority of change sets we were unable to decide whether the applied change set should be considered atomic or tangled. Thus, the presented bias results in this paper must be seen as lower bound. If only one of the undecided change sets is actually tangled, the bias figures would only be increased. The last three rows in the table contain the same information as the upper three rows but dedicated to bug fixing change sets.

The numbers presented in Table 4 provide evidence that the problem of tangled change sets is not a theoretical one. Between 6 % and 12 % (harmonic mean:

Table 4 Proportion of tangled and atomic fix change sets. For those change sets not being classified as tangled nor as atomic, we could not decide whether the change set is tangled or atomic

	ArgoUML	GWT[†]	Jaxen	JRuby	Xstream
Number of issue fixes					
Total	2,944	809	160	2,977	312
Tangled	170 (5.8 %)	68 (8.4 %)	13 (8.1 %)	276 (9.3 %)	37 (11.9 %)
Atomic	125 (4.3 %)	22 (2.7 %)	32 (20.0 %)	200 (6.7 %)	40 (12.8 %)
Number of bug fixes					
Total	343	316	31	2,209	148
Tangled	68 (19.8 %)	47 (14.9 %)	5 (16.1 %)	156 (7.1 %)	22 (14.9 %)
Atomic	116 (33.8 %)	27 (8.5 %)	26 (83.9 %)	64 (1.9 %)	18 (12.2 %)

[†]GOOGLE WEBTOOL KIT

8.2 %) of all change sets containing references to issue reports are tangled and therefore introduce noise and bias into any analysis of the respective change history. The fraction of tangled, bug fixing change sets is even higher: between 7 % and 20 % of all bug fixing change sets are tangled (harmonic mean: 11.9 %). This result is consistent with previous reports on *non-essential changes* by Kawrykow and Robillard ([2011](#)), Kawrykow ([2011](#)). In their study on 10 open-source projects, the authors reported that 8.4 % of inspected code changes seem to be tangled, also showing a maximal portion of tangled changes around 20 % for the Spring project.

> ☞ *Up to 20% of all bug fixing changes are tangled.*
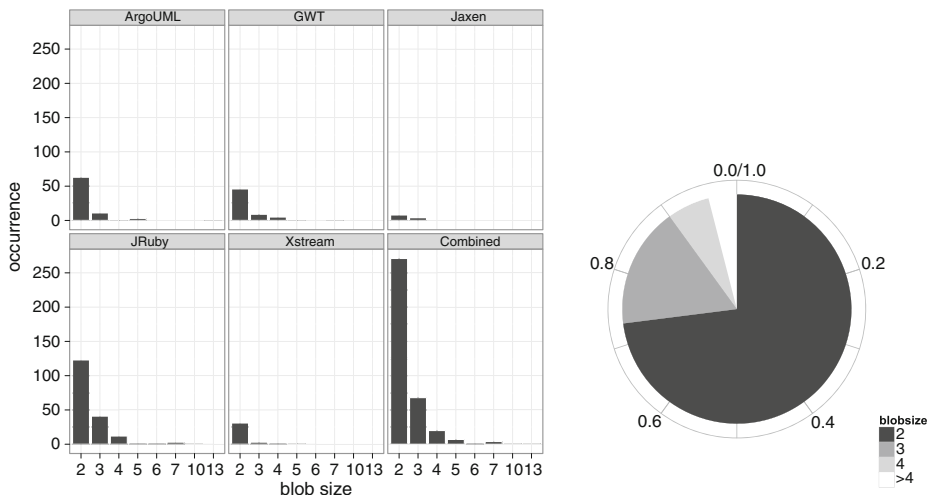
(a) Real world blob sized frequencies per project and combined.   (b) Percentage of real world blob over all projects combined.

Fig. 7 Tangled change sets discovered during manual inspection having a specific blob size

Figure 7 shows the blob size of classified tangled change sets. The fast majority (73 %) of manually inspected, tangled change sets have a blob size two. Change sets of blob size four or above are rare. 89 % of all tangled change sets showed a blob size below four.

> ☞ *73% of all tangled changes have a blob size of two.*
> ☞ *89% of all discovered tangled change sets had a blob size below four.*

The reasons for tangled code changes are manifold and not very well documented. From manual inspection and discussions with software engineers, there seem to be two main reasons for tangled code changes. Primarily, the separation of individual development activities into separate change sets is not always possible or practical. Many bug fixes include code improvements or depend on other bug fixes, without tackling the same underlying issue. Fixing a defect by rewriting an entire method or function may by nature leave to a fixed bug report and a resolved improvement request. In such a case, separating both changes is simply not possible. In other cases, such as fixing a bug while working on the same code area, it is not practical for the engineer to separate both code changes, especially since the engineer has no benefit from doing so. Which leads to the second main reason for tangled code changes: the lack of benefit for the engineer and the laziness of humans. Being under pressure to deliver productive code and to fix code defects, software development engineers tend to work as effective as possible, especially if reward systems do not acknowledge or rewards good practices. Usually, engineers are paid and rewarded for shipping new features or fixing severe issues. Sticking to engineering principles or following software development processes is usually not strictly enforced nor rewarded. Thus, the engineer has no benefit to follow these processes. For the engineer, the fact that the source control management system may be populated with tangled code changes has no effect on his work nor his rewards. Tangled changes are of primary concern for data analysts examining code changes after the fact, but not for the engineer applying the code change. Different development processes might prevent or cause even more tangled code changes, e.g. code reviews or highly distributed development activities. However, these effects have not been studies widely and remain speculative.

6.2 Untangling Changes (RQ2)

For *RQ2* we evaluate the proposed untangling algorithm and measure its untangling precision using artificially tangled change sets. Table 5 contains the number of generated artificially tangled change sets grouped by blob size and combination strategy (see Subsection 4.2). We could only generate a very small set of artificially tangled change sets for the JAXEN project. So we excluded JAXEN from this experiment.

The last three rows of the table contain the sum of artificially tangled change sets generated using different strategies but across different blob sizes. The number of artificially tangled change sets following the change coupling strategy (coupl.) is low except for JRUBY. The ability to generate artificially tangled change sets from project history depends on the number of atomic change sets, on the number of files touched by these atomic change sets, on the change frequency within the project, and on the number of existing change couplings.

The precision of our algorithm to untangle these artificially tangled change sets is shown in Table 6. The presented precision values are grouped by project, blob size, and tangling strategy. Rows stating $\overline{y}$ as strategy contain the average precision over all strategies for the corresponding blob size. The column $\overline{x}$ shows the average precision across different projects for the corresponding blob generation strategy. The cells (x, y) contain the average precision

Table 5 Number of artificially generated tangled change sets sorted by blob size and generation strategy

Blob size	Strategy	ARGOUML	GWT[†]	JRUBY	XSTREAM
2	pack.	40	110	1,430	32
	coupl.	0	20	590	0
	consec.	180	30	3,364	30
3	pack.	13	40	17.3k	133
	coupl.	0	0	19.2k	0
	consec.	673	70	11.4k	53
4	pack.	0	40	1.2M	83
	coupl.	0	0	81.9k	0
	consec.	743	70	695.3k	25
Σ	pack.	53	190	1.2M	248
	coupl.	0	20	101.1k	0
	consec.	1,596	170	710.0k	108

[†]GOOGLE WEBTOOL KIT

across all projects and blob generation strategies for the corresponding blob size. Table cells containing no precision values correspond to the combinations of blob sizes and generation strategies for which we were unable to produce any artificially tangled change sets. In many cases we were not able to produce any artificial blobs based on historic coupling (coupl.). This is mainly due to the restrictions we applied for this type of artificially tangling: we only allow changes that are not more than three change sets apart to be tangled and the historic coupling must show a confidence value of 0.7 or above (see Subsection 4.2). Relaxing these restrictions would allow us to produce more artificial tangled change sets, but would also reduce the quality of these artificially tangled change sets and raise concerns about whether these sets remain representative.

The algorithm performs well on all subject projects. Projects with higher number of generated artificially tangled change sets also show higher untangling precision. The

Table 6 Precision rates of the untangling algorithm sorted by blob size and generation strategy

Blob size	Strategy	ARGOUML	GWT[†]	JRUBY	XSTREAM	$\overline{x}$
2	pack.	0.79	0.67	0.91	0.81	0.80
	coupl.	—	0.75	0.93	—	0.84
	consec.	0.74	0.70	0.91	0.79	0.79
	$\overline{y}$	0.77	0.71	0.92	0.80	0.80
3	pack.	0.70	0.63	0.69	0.65	0.67
	coupl.	—	—	0.68	—	0.68
	consec.	0.62	0.57	0.70	0.66	0.64
	$\overline{y}$	0.66	0.60	0.69	0.66	0.66
4	pack.	—	0.58	0.62	0.50	0.57
	coupl.	—	—	0.63	—	0.63
	consec.	0.55	0.54	0.64	0.59	0.58
	$\overline{y}$	0.55	0.56	0.63	0.55	0.58

[†]GWT = GOOGLE WEBTOOL KIT

more artificially tangled change sets, the higher the number of instances to train our linear regression aggregation model. Overall, the precision values across projects show similar ranges and most importantly similar trends in relation to the chosen blob size.

> ☞ *The untangling algorithm shows comparable results across all subject projects.*

For all projects, the precision is negatively correlated with the used blob size. The more change operations to be included and the more partitions to be generated, the higher the likelihood of misclassification. Figure 7b shows that tangled change sets with a blob size of two are most common (73 %). The results in Table 6 show that for the most popular cases our untangling algorithm achieves precision values between 0.67 and 0.93—the harmonic mean lies at 79 %. When the blob size is increased from two to three the precision drops by approximately 14 %, across all projects and from 80 % to 66 % on average. Increasing the blob size further has a negative impact on precision. For each project and blob size, the precision values across different strategies differ at most by 0.09 and on average by 0.04.

> *We can untangle*
> - ☞ *artificially tangled change sets with a mean precision between 0.58 (blob size four) and 0.79 (blob size two),*
> - ☞ *any two artificially tangled change sets with a precision between 0.67 and 0.93.*

These results are consistent with untangling results reported by Kawrykow and Robillard (2011), Kawrykow (2011). The authors reported an untangling overall precision of 80 % and an overall recall value of 24 %. The similarity is not surprising as both untangling algorithms follow very similar strategies, although developed in parallel without knowledge of each other. However, the influence of blob-size on the precision of untangling is not reported by Kawrykow and Robillard (2011), Kawrykow (2011).

Table 7 Relative untangling file error

Blobsize	Strategy	ArgoUML	GWT†	JRuby	Xstream	$\overline{x}$
2	pack.	0.00	0.00	0.10	0.30	0.00
	coupl.	—	0.25	0.15	—	0.20
	consec.	0.26	0.30	0.22	0.32	0.28
	$\overline{y}$	0.13	0.18	0.16	0.31	0.19
3	pack.	0.00	0.36	0.32	0.34	0.34
	coupl.	—	—	0.42	—	0.42
	consec.	0.38	0.40	0.37	0.45	0.40
	$\overline{y}$	0.19	0.38	0.37	0.40	0.29
4	pack.	—	0.50	0.47	0.41	0.46
	coupl.	—	—	0.48	—	0.48
	consec.	0.44	0.40	0.45	0.47	0.44
	$\overline{y}$	0.44	0.45	0.47	0.44	0.45

Table 6 shows that the precision values for artificially tangled change sets are stable across tangling strategies. For each project and blob size, the precision values across different strategies differ at most by 0.09 and on average by 0.04.

> ☛ *The performance of our untangling algorithm is stable across all tangling strategies.*

The above-discussed results show that the proposed untangling algorithm is capable of separating change operation applied to fix different code bugs with a reasonable precision. However, the results do not indicate the fraction of source files that would be assigned a false positive bug fix count. To show this impact, we measured the relative file error during untangling tangled change sets. For each untangled change set, we lift the level for computing precision values to the file level measuring the number source code files correctly or falsely associated with an untangling result partition. Thus, the relative file error reports the proportion of source files that would be falsely assigned to a developer maintenance task. The relative file error is a value between 0 and 1, where a value of 0 is desirable as it corresponds to cases in which no file was wrongly associated to any developer maintenance task. Table 7 shows the results for all subject projects grouped by tangling strategy and blob size. Similar to Table 6, the $\overline{y}$ column contains the average error rates over all strategies: $\overline{x}$ rows contain the average error rates across different projects for the corresponding blob generation strategy. The cells $(\overline{x}, \overline{y})$ contain the average error rates across all projects and blob generation strategies for the corresponding blob size.

Overall, the average file error rates over all projects and strategies (red colored cells) correspond to the overall precision values in Table 6. For a blob size of two, the average relative file error rate lies at 0.19. Consequently, untangling change sets of blob size two reduces the average number of source files falsely associated to development tasks by at least 81 %.[4] For a blob size of four, the average reduction rate of falsely associated files drops to 55 %. Like untangling precision, the relative file error is positively correlated with the blob size. The higher the blob size the lower the untangling precision and the higher the relative file error rates. Although the relative file error rates raise to a value of 0.5 for blobs of size four, we still reduce the number of source files falsely associated to development tasks by at least 50 %.

> ☛ *Untangling code changes reduces the number of source files falsely associated to development tasks by 55% (blob size four) to 81% (blob size two).*

This result cannot be compared to the closely related study by Kawrykow and Robillard (2011), Kawrykow (2011). In their study, the authors did not investigate the impact of tangled or non-essential changes on bug counts for source files or other codeentities.

6.3 The Impact of Tangled Changes (RQ3)

Remember that we untangled only those change sets that we manually classified as tangled change sets (see Table 4). The fraction of tangled change sets lies between 6 % and

---

[4]Since we are analyzing artificially tangled change sets only, the file mapping error rate without untangling lies at 100 %. Having a error rate after untangling of 19 %, the result is a reduction rate of 100 % − 19 % = 81 %.
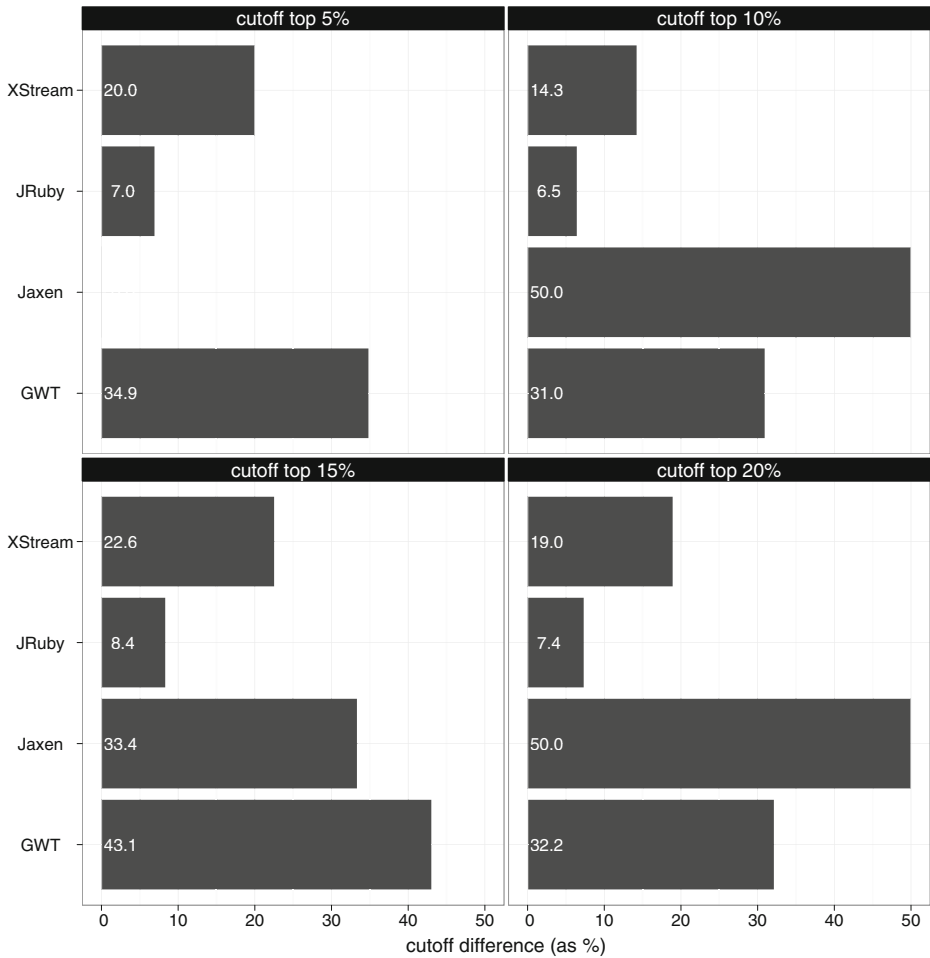
Fig. 8 The `cutoff_differences` caused by real tangled change sets

15 %. Figure 8 shows that untangling these few tangled change sets already has a significant impact on the set of source files marked as most defect prone. The impact of untangling lies between 6 % and 50 % (harmonic mean: 17.4 %). The `cutoff_difference` and the fraction of tangled change sets is correlated. JRUBY has the lowest fraction of blobs and shows the smallest cutoff_differences. JAXEN has the highest tangled change set fraction and shows the highest cutoff_differences. We can summarize that untangling tangled change sets impacts bug counting models and thus are very likely to impact more complex quality models or even bug prediction models trained on these data sets.

We further observed that in total between 10 % and 38 % and on average (harmonic mean) 16.6 % of all source files we assigned different bug counts

when untangling tangled change sets. Between 1.5 % and 7.1 % of the files originally associated with bug reports had no bug count after untangling.

Tangled change sets severely impact bug counting models.

- ☞ *Between 6% and 50% (harmonic mean: 17.4%) of the most defect prone files do not belong in this category.*
- ☞ *On average, at least 16.6% of all source files are incorrectly associated with bug reports.*

6.4 The Impact on Bug Prediction Models (RQ4)

6.4.1 Impact on Classification Models

Figure 9 compares precision, recall, and F-measure values for all models (across different machine learners and across different cross-folds). The black line in the middle of each box plot indicates the median value of the corresponding distribution. Larger median values indicate better performance on the metric set for the project based on the respective evaluation measure. Note that the red colored horizontal lines connecting the medians across the box plots do not have any statistical meaning—they have been added to aid visual comparison of the performance of the metrics set. An upward horizontal line between two box plots indicates that the metrics set on the right performs better than the one of the left and vice versa.

The results in Figure 9 show no significant difference. As discussed in Subsection 6.2, only a small fraction of source files (1.5 %–7.1 %) originally associated with bug reports had no bug count after untangling. All other changes to bug counts, i.e. files with a reduced number of associated bugs but still at least one association, cannot impact classification models using a threshold of one bug to decide between positive and negative instances. As the results show, these changes have no impact on the classification performances.

The R package *caret* (Kuhn 2011) allows computing the importance of individual metrics using the *filterVarImp* function. The function computes a ROC curve by first applying a series of cutoffs for each metric and then computing the sensitivity and
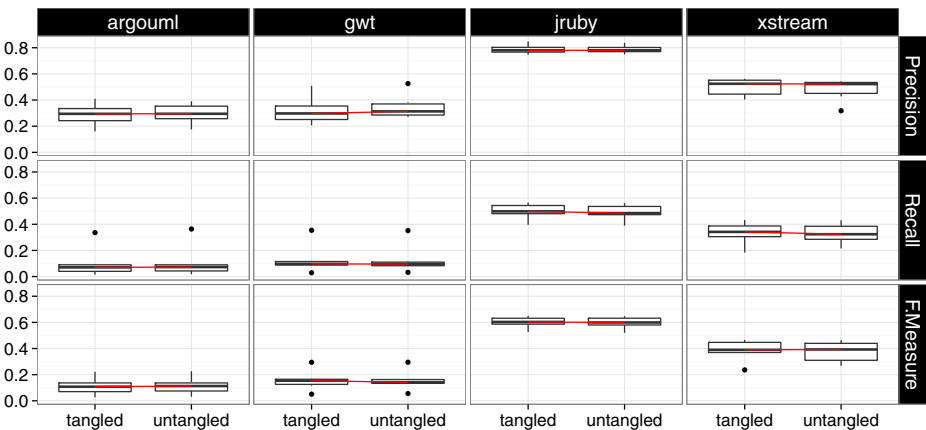

Fig. 9 Comparing classification model results for tangled and untangled data sets

specificity for each cutoff point. The importance of the metric is then determined by computing the area under the ROC curve. We compared the top-10 most influential metrics for models trained and tested on tangled and untangled bug datasets and compared these top-10 most influential metrics between tangled and untangled models. Similar to the prediction performance results, there exists no difference in the most influential metrics. Only GOOGLE WEBTOOL KIT and XSTREAM showed one out of ten metrics to be different—a difference that might also be due to splitting and cross-fold differences.

> ☞ *Tangled change sets seem to have no impact on prediction models identifying source files that have at least one bug fix applied to it.*

6.4.2 Impact on Regression Models

As described in Subsection 4.3, we measured the impact of tangled changes using `regression_precision`$_x$—the intersection between the top x% of files with the most observed and predicted number of bugs, respectively. The higher the overlap of these two subsets, the higher the precision of the regression models in predicting the files that are the likeliest ones to contain bugs.

The results of this experimental setup are shown in Fig. 10. The box plots in the figure compare `regression_precision`$_x$ values for x values 5 %, 10 %, 15 % and 20 %
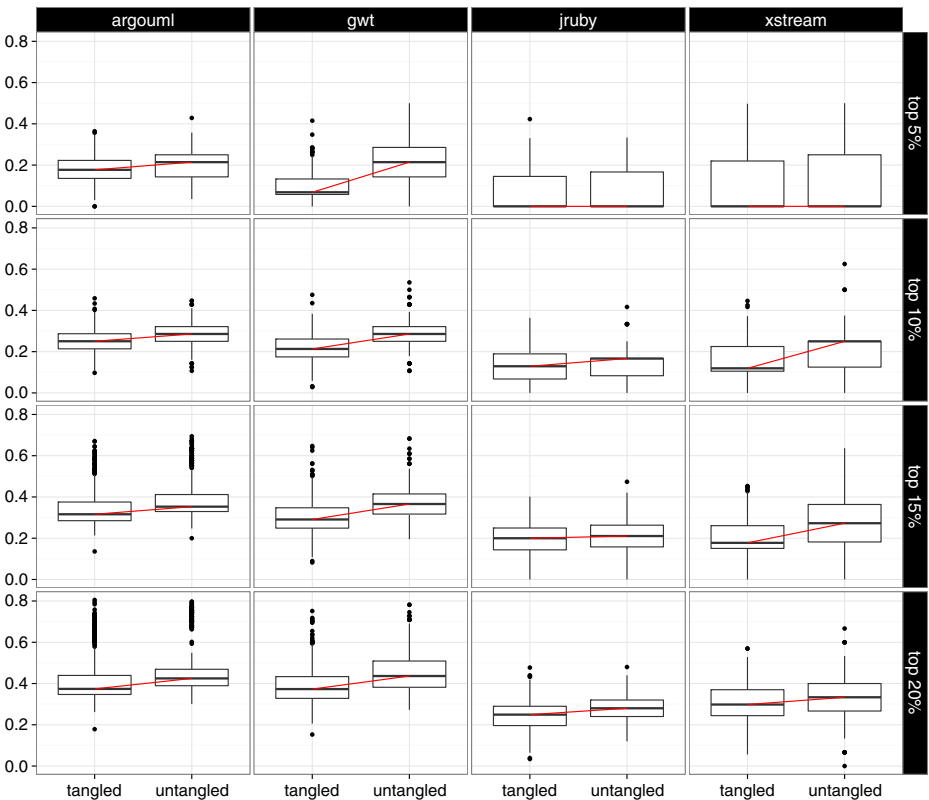

Fig. 10 Comparing regression model results for tangled and untangled data sets.

as well as the distribution of prediction results across the used machine learners (see Sub-section 4.3). The black line in the middle of each box plot indicates the median value of the corresponding distribution (also shown in Table 8). Larger median values indicate better performance on the corresponding bug data set (tangled or untangled) for the project. Note that the red colored horizontal lines connecting the medians across the box plots do not have any statistical meaning—they have been added to aid visual comparison of the performance of the metrics set. An upward horizontal line between two box plots indicates that the prediction models trained and tested on untangled bug data perform better than the ones trained and tested on tangled bug data and vice versa. Additionally, we performed a non-parametric statistical test (Kruskal-Wallis) to statistically compare the results. Table 8 contains the median values of the prediction models and the relative improvement when comparing models based on untangled bug data with models based on tangled bug data. A positive relative improvement value in Table 8 reflects a better result for untangled prediction models (corresponding upward red horizontal line in box plots). Column $\overline{x}$ in Table 8 contains the median precision improvement across different projects for the corresponding cutoff size. Row $\overline{y}$ contains the median precision values for individual software projects across all cutoff values.

Both, Table 8 and Fig. 10 show that regression models trained and tested on untangled bug data perform better than models based on tangled bug data. Except for the top 5 % `regression_precision` for JRUBY and XSTREAM untangling tangled code changes improves the prediction accuracy of regression models—the prediction accuracy for `regression_precision`$_{0.05}$ on JRUBY and XSTREAM is zero for both cases. The prediction accuracy improvement seems to be unrelated to project size of the number of tangled changes. XSTREAM, the project with the highest fraction of tangled changes, shows a high improvement rate (30 %), while GOOGLE WEBTOOL KIT, the project with the second lowest relative number of tangled change shows the highest improvement. Interestingly, the impact of tangled changes on regression models for JRUBY is the lowest. This is surprising as the relative number of tangled changes with higher blob sizes (see Fig. 7) for JRUBY is higher than for any other project. This indicates that the number and size of tangled changes is secondary. We suspect the relative number of tangled changes combining bug fixes with

Table 8 Median regression prediction model precision values and the relative improvement due to untangling tangled changes

cutoff	measure	ARGOUML	GWT†	JRUBY	XSTREAM	$\overline{x}$
	tangled	.18	.07	.00	.00	
top 5%	untangled	.21	.21	.00	.00	
	rel. improvement	**16.7%**	**200%**	**0.0%**	**0.0%**	8.4%
	tangled	.25	.21	.13	.12	
top 10%	untangled	.29	.28	.17	.25	
	rel. improvement	**16.0%**	**33.3%**	**30.8%**	**108.3%**	32.0%
	tangled	.32	.29	.20	.18	
top 15%	untangled	.35	.37	.21	.27	
	rel. improvement	**9.4%**	**38.1%**	**5.0%**	**50.0%**	23.8%
	tangled	.37	.37	.25	.30	
top 20%	untangled	.42	.44	.28	.33	
	rel. improvement	**13.5%**	**18.9%**	**12.0%**	**10.0%**	12.8%
$\overline{y}$		14.8%	35.7%	8.5%	30.0%	16.4%

other change types to be dominant. Such tangled changes are likely to cause heuristics commonly used to map bug reports with source files to create false associations and thus to impact bug counts.

Across all projects, the prediction accuracy improvement is highest (median value) for cutoff values of 10 %. Overall, we measured a prediction accuracy improvement of 16.4 % (median value) across all projects and cutoffs. A non- parametric statistical test (Kruskal-Wallis) showed that the difference in regression accuracy is statistically significant ($p < 0.05$), except for top 5 % JRUBY and top 5 % XSTREAM.

Tangled change sets significantly impact regression bug prediction models.

☞ *Models trained and tested on untangled bug data showed a median accuracy improvement of 16.4% when compared to models trained and tested on tangled bug data.*

---

7 Threats to Validity

Empirical studies of this kind have threats to their validity. We identified the following noteworthy threats.

The change set classification process involved manual code change inspection. 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 misclassified change sets. Please note that we ignored all change sets that could not we were unable to classify behind doubt. The main reason for this is the fact that the persons conducting the manual classification were project outsiders and thus were lacking expertise to make a decisive call. Due to the indecisiveness about many commits, the presented numbers of tangled code changes should be considered a lower bound. Increasing the precision and corpus of manual classified change sets could impact the number and the quality of generated artificially tangled change sets and thus the untangling results in general.

The selected study subjects may not be representative and untangling results for other projects may differ. Choosing CONFVOTERS differently may impact untangling results.

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 change sets. The process of constructing these artificially tangled change sets may not simulate real life tangled change sets caused by developers.

The results in this paper are produced using our untangling algorithm using a stopping criteria that assumes the number of partitions to create is known. The number of partitions to be creates may not be known in all cases. We can use hints in the commit messages to approximate that value. The petter the development process and the more detailed the commit messages the more accurate such a heuristic might be. Please note that using the alternative stopping criteria using CONFVOTER thresholds would be much more realistic. However, please also note that the purpose of this paper is not to build a fully automatic and reliable process of untangling processes but rather assesses the impact of tangled code changes on empirical version archive datasets and potential applications that consume such data.

Our analysis tool relies on the Eclipse Java Development Tools (JDT) and thus can only be applied to JAVA projects. Hence, we cannot generalize our findings for projects written in other programming languages.

Internally our untangling algorithm uses the partial program analysis tool (Dagenais and Hendren 2008) by Dagenais and Hendren. The validity of our results depends on the validity of the used approach.

8 Conclusion and Consequences

Tangled changes introduce noise in change data sets: In this paper, we found that up to 20 % of all bug fixes to consist of multiple tangled changes. This noise can severely impact bug counting models and bug prediction models. When predicting bug-prone files, on average, at least 16.6 % of all source files are incorrectly associated with bug reports due to tangled changes. Although, in our experiments tangled changes showed no significant impact on classification models classifying source files as buggy or not buggy, bug prediction regression models trained and tested on untangled bug datasets showed a statistically significant precision improvement of, on average, 16.4 %. These numbers are the main contribution of this paper, and they demand for action, at least when studying bug prediction regression models. The reason why simpler bug classification models seem to be unaffected by tangled changes is quite simple. Only between 1.5 % and 7.1 % of code files were falsely associated with any bug. Only for those cases, the binary decision about the defect-proneness of the corresponding source file changes and makes a difference for the classification model. The fast majority of different bug counts (on average 16.6 % of source files) affect a bug count larger than one and thus make no difference for any classification model, but certainly changes the dependent variable for regression models.

What can one do to prevent this? Tangled changes are natural and from a developer's perspective, tangled changes make sense and should not be forbidden. Refactoring a method name while fixing a bug caused by a misleading method name should be considered as part of the bug fix. However, we think that the extend of tangled changes could be reduced. Preventing tangled code changes requires to change software engineers' behavior. Changing human behavior is a difficult task and often requires changes to the overall development process, e.g. rewarding good development practices. Although there exists no clear evidence, making code reviews mandatory might help to minimize unnecessary tangled changes. Reviewers might be offended by large unrelated code changes, which may lead to longer review processes. However, more research is needed to investigate such dependencies and the driving factors of tangled code changes. In the meantime, version archive miners should be aware of tangled changes and their impact. Untangling algorithms similar to the algorithm proposed in this paper may help to untangle changes automatically and thus to reduce the impact of tangled changes on mining models.

In our future work, we will continue to improve the quality of history data sets. With respect to untangling changes, our work will focus on the following topics:

Automated untangling.   The automated algorithms sketched in this paper can still
                               be refined. To evaluate their effectiveness, though, one
                               would require substantial ground truth—i.e., thousands of
                               manually untangled changes.

Impact of change organization.　Our results suggest that extensive organization of software changes through branches and change sets would lead to less tangling and consequently, better prediction. We shall run further case studies to explore the benefits of such organization.

To learn more about our work, visit our Web site:

http://softevo.org/untangling_changes

References

Alam O, Adams B, Hassan AE (2009) Measuring the progress of projects using the time dependence of code changes. In: ICSM, pp 329–338

Anvik J, Hiew L, Murphy GC (2006) Who should fix this bug? In: Proceedings of the 28th international conference on Software engineering. ACM, pp 361–370

Bachmann A, Bird C, Rahman F, Devanbu P, Bernstein A (2010) The missing links: bugs and bug-fix commits. In: Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering. ACM, pp 97–106

Bhattacharya P (2011) Using software evolution history to facilitate development and maintenance. In: Proceeding of the 33rd international conference on Software engineering. ACM, pp 1122–1123

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

Bird C, Nagappan N, Gall H, Murphy B, Devanbu P (2009) Putting it all together: Using socio-technical networks to predict failures. In: Proceedings of the 2009 20th International Symposium on Software Reliability Engineering, ISSRE '09. IEEE Computer Society, Washington, pp 109–119. doi:10.1109/ISSRE.2009.17

Bonacich P (1987) Power and centrality: a family of measures. American journal of sociology

Csardi G, Nepusz T (2006) The igraph software package for complex network research. InterJournal Complex Systems:1695. http://igraph.sf.net

Dagenais B, Hendren L (2008) Enabling static analysis for partial Java programs. In: Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications, OOPSLA '08. ACM, pp 313–328

Dallmeier V (2010) Mining and checking object behavior. Ph.D. thesis, Universität des Saarlandes

Herzig K (2012) Mining and untangling change genealogies. Ph.D. thesis, Universität des Saarlandes

Herzig K., Just S., Rau A., Zeller A. (2013) Predicting defects using change genealogies. In: Proceedings of the 2013 IEEE 24nd international symposium on software reliability engineering, ISSRE '13. IEEE Computer Society

Herzig K, Just S, Zeller A (2013) It's not a bug, it's a feature: how misclassification impacts bug prediction. In: Proceedings of the 2013 international conference on software engineering, ICSE '13. IEEE Press, Piscataway, pp 392–401

Herzig K., Zeller A. (2013) The Impact of Tangled Code Changes. IEEE Press, Piscataway, pp 121–130

Hindle A, German D, Godfrey M, Holt R (2009) Automatic classication of large changes into maintenance categories. In: Program comprehension, 2009. ICPC '09. 17th International Conference on IEEE, pp. 30–39

Hindle A, German DM, Holt R (2008) What do large commits tell us? A taxonomical study of large commits. In: Proceedings of the 2008 international working conference on Mining software repositories, MSR '08. ACM, pp 99–108

Jaccard P (1901) Étude comparative de la distribution florale dans une portion des Alpes et des Jura. Bulletin del la Société Vaudoise des Sciences Naturelles 37:547–579

Karypis G, Kumar V (1995) Analysis of multilevel graph partitioning. In: Proceedings of the 1995 ACM/IEEE conference on Supercomputing, Supercomputing 1995. ACM

Karypis G, Kumar V (1995) MeTis: unstructured graph partitioning and sparse matrix ordering system. Version 2:0. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.38.376

Karypis G, Kumar V (1998) A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM J Sci Comput 20:359–392

Kawrykow D. (2011) Enabling precise interpretations of software change data. Master's thesis, McGill University

Kawrykow D, Robillard MP (2011) Non-essential changes in version histories. In: Proceeding of the 33rd international conference on Software engineering, ICSE '11. ACM, pp 351–360

Kim S, Whitehead Jr. EJ, Zhang Y (2008) Classifying software changes: Clean or buggy. IEEE Trans Softw Eng 34:181–196

Kim S, Zhang H, Wu R, Gong L (2011) Dealing with noise in defect prediction. In: Proceeding of the 33rd international conference on Software engineering, ICSE '11. ACM, pp 481–490

Kim S, Zimmermann T, Whitehead Jr. EJ, Zeller A (2007) Predicting faults from cached history. In: Proceedings of the 29th International Conference on Software Engineering, ICSE '07. IEEE Computer Society, Washington, pp 489–498. doi:10.1109/ICSE.2007.66

Kuhn M (2011) caret: classification and regression training. R package version 4.76. http://cran.r-project.org/web/packages/caret/caret.pdf

Li PL, Kivett R, Zhan Z, Jeon SE, Nagappan N, Murphy B, Ko AJ (2011) Characterizing the differences between pre- and post-release versions of software. In: Proceeding of the 33rd international conference on Software engineering. ACM, pp 716–725

Menzies T, Milton Z, Turhan B, Cukic B, Jiang Y, Bener A (2010) Defect prediction from static code features: current results, limitations, new approaches. Automated Software Engg 17:375–407

Mockus A, Votta LG (2000) Identifying reasons for software changes using historic databases. In: Proceedings of the international conference on software maintenance (ICSM'00), ICSM '00. IEEE Computer Society, pp 120–130

Murphy-Hill E, Black A (2008) Refactoring tools: fitness for purpose. IEEE Software 25(5):38–44

Murphy-Hill E, Parnin C, Black AP (2009) How we refactor, and how we know it. Int Conf Softw Eng 287–297

Nagappan N, Murphy B, Basili V (2008) The influence of organizational structure on software quality: an empirical case study. In: Proceedings of the 30th international conference on Software engineering, ICSE '08. ACM, New York, pp 521–530. doi:10.1145/1368088.1368160

Nguyen TH, Adams B, Hassan AE (2010) A case study of bias in bug-fix datasets. In: 2010 17th Working Conference on Reverse Engineering. IEEE Computer Society, pp 259–268

Premraj R, Herzig K (2011) Network versus code metrics to predict defects: a replication study. In: Proceedings of the 2011 international symposium on empirical software engineering and measurement, ESEM '11. IEEE Computer Society, Washington, pp 215–224. doi:10.1109/ESEM.2011.30

R Development Core Team (2010) R: a language and environment for statistical computing. R Foundation for Statistical Computing

Robbes R, Lanza M, Lungu M (2007) An approach to software evolution based on semantic change. In: Fundamental approaches to software engineering, Lecture notes in computer science, vol 4422. Springer, Berlin, pp 27–41

Stoerzer M, Ryder BG, Ren X, Tip F (2006) Finding failure-inducing changes in java programs using change classification. In: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering. ACM, pp 57–68

Tosun A, Turhan B, Bener A (2009) Validation of network measures as indicators of defective modules in software systems. In: Proceedings of the 5th international conference on predictor models in software engineering, PROMISE '09. ACM, New York, pp 5:1–5:9. doi:10.1145/1540438.1540446

Williams BJ, Carver JC (2010) Characterizing software architecture changes: a systematic review. Information and Software Technology 52(1):1–51

Witten IH, Frank E (2002) Data mining: practical machine learning tools and techniques with java implementations. SIGMOD Rec 31(1):76–77. doi:10.1145/507338.507355

Wloka J, Ryder B, Tip F, Ren X (2009) Safe-commit analysis to facilitate team software development. In: Proceedings of the 31st International Conference on Software Engineering, ICSE '09. IEEE Computer Society, pp 507–517

Zimmermann T, Nagappan N (2008) Predicting defects using network analysis on dependency graphs. In: Proceedings of the 30th international conference on Software engineering, ICSE '08. ACM, New York, pp 531–540. doi:10.1145/1368088.1368161

Zimmermann T, Premraj R, Zeller A (2007) Predicting defects for eclipse. In: Proceedings of the third international workshop on predictor models in software engineering, PROMISE '07. IEEE Computer Society

Zimmermann T, Weisgerber P, Diehl S, Zeller A (2004) Mining version histories to guide software changes. In: Proceedings of the 26th international conference on software engineering. IEEE Computer Society, pp 563–572

Kim Herzig is an engineer and researcher in the Tools for Software Engineers group at Microsoft, closely collaborating with the Empirical Software Engineering group at Microsoft Research. His work is focused on development and testing process analytics. Prior to Microsoft, Kim was Researcher at the Software Engineering Chair at Saarland University where he mined the change and bug archives of SAP, of IBM's Jazz and Eclipse projects, of Google, and of dozens of open source projects.

Sascha Just is currently working as a researcher at the Software Engineering chair at Saarland University, Saarbrücken, Germany under the supervision of Prof. Dr. Andreas Zeller. His research focus lies on empirical software engineering and mining software repositories, such as version archives and test execution databases, to guide software development processes and tools. Since 2015, Sascha is sponsored through a PhD scholarship from Microsoft Research in Cambridge, UK.

Andreas Zeller is a full professor for Software Engineering at Saarland University in Saarbrücken, Germany. His research concerns the analysis of large software systems and their development process; his students are funded by companies like Google, Microsoft, or SAP. In 2010, Zeller was inducted as Fellow of the ACM for his contributions to automated debugging and mining software archives. In 2011, he received an ERC Advanced Grant, Europe's highest and most prestigious individual research grant, for work on specification mining and test case generation. In 2013, he co-founded Testfabrik AG, a start-up for automatic testing of Web applications.