The Impact of Tangled Code Changes

Kim Herzig
Microsoft Research†
Cambridge, UK
kimh@microsoft.com

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

***Abstract*—When interacting with version control systems, developers often commit unrelated or loosely related code changes in a single transaction. When analyzing the version history, 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 up to 15% 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. We recommend better change organization to limit the impact of tangled changes.**

***Index Terms*—Mining software repositories, tangled code changes, data quality, noise, bias**

I. INTRODUCTION

Many software quality related recommendation and prediction systems are based on mining software 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 [1], to predict future defects [2], [3], to analyze who should be assigned a particular task [4], [5], or to gain insights about specific projects [6].

Most of these studies depend on the *accuracy* of the mined information—accuracy which is threatened by noise. Such noise can come from missing associations between change and bug databases [7], [8], [9]. One significant source of noise, however, is *tangled changes*.

What is a tangled change? Assume a developer is assigned multiple tasks A, B, and C, all with a separate purpose: A is a bug fix, B is a feature request, and C is a refactoring or code cleanup. Once all tasks are completed, the developer commits her changes to the source code management system (SCM), such that her changes be visible to other developers and integrated into the product. However, when committing changes, developers frequently *group separate changes into a single commit,* resulting in a tangled change.

Tangled change sets do not cause trouble in development. However, they threaten any analysis of the corresponding history, thereby compromising the accuracy of the analysis. As the change set contains a bug fix (task A) and may even state so in the commit message, the tangled commit may be classified as a bug fix [2]. Consequently, files touched by the tangled change set will be marked as being fixed, even though the tangled tasks B and C have nothing to do with the actual bug fix. Likewise, all files will be marked as being changed together, which may impact recommender systems based on frequent change patterns (e.g. [10], [11]). Commit messages such "fixed typo" point to additional minor changes all over the code even if no specific task is referenced and that will now be related with each other as well as the tasks. From a developer's perspective fixing a typo may be part of the fix but the refactored code should not be considered as fixed.

How frequent are tangled changes, and how big is their impact? In this paper, we present an empirical study on five open-source 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 show 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 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. To facilitate the analysis of version histories, we recommend that version control systems and processes be set up to avoid tangling changes whenever possible.

II. BACKGROUND

A number of researchers have classified code changes, studied the relations between code changes and noise and bias in version archive datasets.

A. Classifying Code Changes

The work presented in this paper is closely related to many research approaches that analyze and classify code changes or development activities. In this section, we want to discuss 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 [12] or on their purpose [13], [14]. Kim et al. [12] 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,

†At the time of this study, Kim Herzig was PhD candidate at Saarland University. He is now employed at Microsoft Research Cambridge.

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. Hindle et al. [13], [14] 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.

Störzer et al. [15] used a change classification technique to automatically detect code changes contributing to test failures. Later, this work was extended by Wloka et al. [16] to identify committable code changes that can be applied to the version archive without causing existing tests to fail. Both approaches aim to detect change dependencies within one revision but require test cases mapped to change operations in order to classify or separate code changes. This will rule out the majority of change operations not covered by any test case or for which no test case is assigned.

Williams and Carver [17] present in their systematic review many different approaches on how to distinguish and characterize software changes. However, none of these approaches is capable of automatically identifying and separating combined source code changes based on their different characterization or based on semantic difference.

B. Refactorings

The combination of refactorings and semantic relevant code changes can be seen as a special case of the untangling problem. Murphy-Hill et al. [18], [19] 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" [20].

C. 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. [21] showed that the evolutionary information in version archives may be incomplete and of low quality. Storing historical data to explicit developer request fails to store important historic data fragments, while the nature of version archives leads to a view of software as being simply a set of files. As a solution, Robbes et al. [21] proposed a novel approach that automatically records all semantic changes performed on a system. An untangling algorithm would clearly benefit from such extra information that could be used to add context information for individual change operations.

D. 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 [22] 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 to retrieve real patches in the end. 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.

E. 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 [20] 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 [23] 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.

The effects of bias caused by unbalanced data sets on defect prediction models were investigated by various studies [7], [8], [9]. 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 generalizability of hypotheses tested on biased data" [7]. Kim et al. [24] 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.

III. Research Questions

Overall, the research question tackled by this paper is to determine whether tangled changes impact bug count models and thus should be considered harmful or whether they do not impact bug counting models and thus can be ignored. To achieve our goal, we have to complete three basic steps, each dedicated to research questions of lower granularity.

A. 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.

B. 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 assumption. 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. —

C. RQ3: How do tangled changes impact bug count models?

The last research question targets the actual impact of tangled code changes on 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. For removing tangled changes requires us to untangle them.

IV. Experimental Setup

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

A. 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.

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. [2].
2) Each change set from Step 1 was manually inspected and classified as atomic or non-atomic. 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 and therefore the change set was marked non-atomic. Only if we had no doubt that 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.

B. Untangling Changes (RQ2)

To answer *RQ2*, we developed a prototype of a heuristic-based untangling algorithm. 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. 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 V.

In Section VI-A 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 to evaluate our untangling algorithm, simply because we cannot determine whether a produced change set partition is correct and if not, how much it differs from an expected result.

To determine a reliable set of atomic and unbiased change sets—change sets containing only those code changes required to resolve exactly one issue—we use the manual classified atomic change sets (Section IV-A) 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 such 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. 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 located in source directories not more than two directory changes apart 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: $CS_1 = \{com.prod1.pack1.intern.F_1\}$, $CS_2 = \{com.prod1.pack2.extern.F_2\}$, and $CS_3 = \{com.prod2.pack1.intern.F_3\}$. Each class is identified by its fully qualified name. Using this strategy we combine CS_1 with CS_2 but not CS_1 nor CS_2 with CS_3.[1]

Frequently changed before (coupl.) This strategy computes and uses change coupling rules [10]. Two code changes get 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. [10] and call this set $\mathcal{S}$. The computed change coupling rules indicate how frequently F_s and F_t got changed together before CS_i got 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 got 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.

[1]This slightly penalizes CONFVOTERS that use package distances as a heuristic. However, we favored a more realistic distribution of changes over total fairness across all CONFVOTERS.

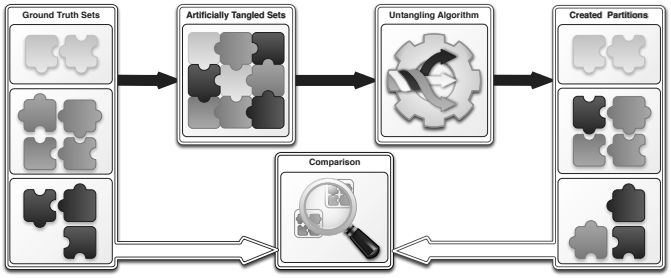

Fig. 1. Artificially tangled change sets are generated using manually classified atomic change sets to compare created partitions and desired output. In the example, two change operations are put into a wrong partition, and hence the success rate is $\frac{7}{9} = 77.\overline{7}\%$.

The untangling algorithm (described in Section V) 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.

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 Figure 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 Figure 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 recategorized 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. Precision values reported in this chapter correspond to the partition association with the highest sum of *Jaccard indices* [25]. The higher the Jaccard index the higher the similarity of the sets. 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 two, three, and four (tangled change sets with a bob size larger than four are rare, see Section VI-A).

C. Measuring the Impact of Tangled Changes (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*. 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 [26] and Hindle et al. [13], [14] 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. Both bug count sets get sorted in descending order using the distinct number of bug reports associated with the file (see Figure 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. Comparing the top x% of both file sets 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 Figure 2). The result is a number between zero and one where zero 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.

D. Study Subjects

All experiments are conducted on five open-source JAVA projects (see Table I). 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 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).

TABLE I
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

```
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. 3. 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)*.

V. 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*. 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. 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. [27] 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 Figure 3 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 the definition or call was added to or deleted from. In our example the 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 Figure 4).

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.

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

Fig. 2. Computing the `cutoff_difference`.

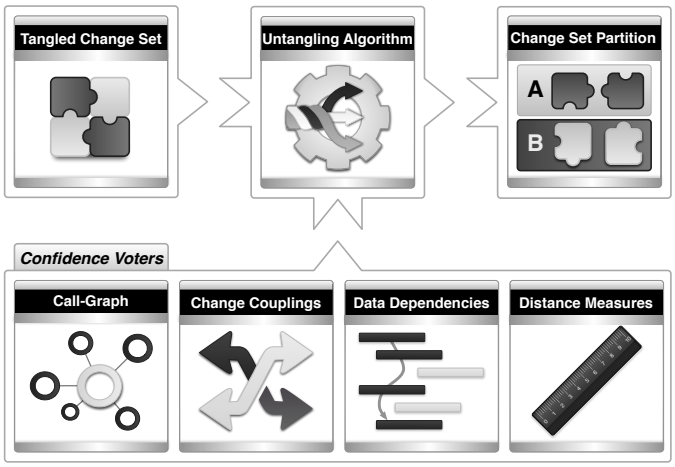

Fig. 4. The untangling algorithm partitions change sets using multiple, configurable aspect extracted from source code. Gray boxes represent sets of change operations necessary to resolve one issue.

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.

A. 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 Figure 4). Each CONFVOTER expects a pair of change operations and returns a *confidence value* between zero and one. A confidence value of one represents a change operation dependency aspect that strongly suggests to put both change operations into the same partition. Conversely, a return value of zero indicates that the change operations are unrelated according to this voter.

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 are applied to. 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.

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 nodes. 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.

ChangeCouplings The confidence value returned by this CONFVOTER is based on the concept of change couplings as described by Zimmermann et al. [10]. The CONFVOTER computes frequently occurring sets of code artifacts that got 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 Section V-B how to combine the confidence values of different CONFVOTERS.

B. 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

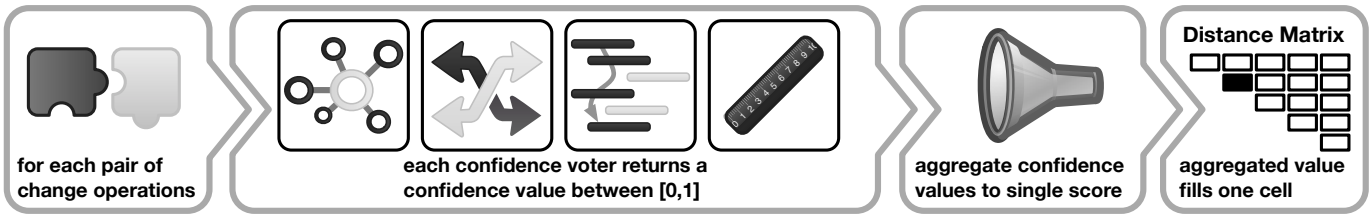

Fig. 5. The procedure to build the initial triangle matrix used within the modified multilevel graph partitioning algorithm.

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 [28], [29], [30]. 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 a $m \times m$ 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 Section V-A 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 5 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:

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

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 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 were 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.

TABLE II
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

VI. RESULTS

In this section we present the results of our three experimental setups as presented in Section IV.

A. Tangled Changes (RQ1)

The results of the manual classification process is shown in Table II. 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

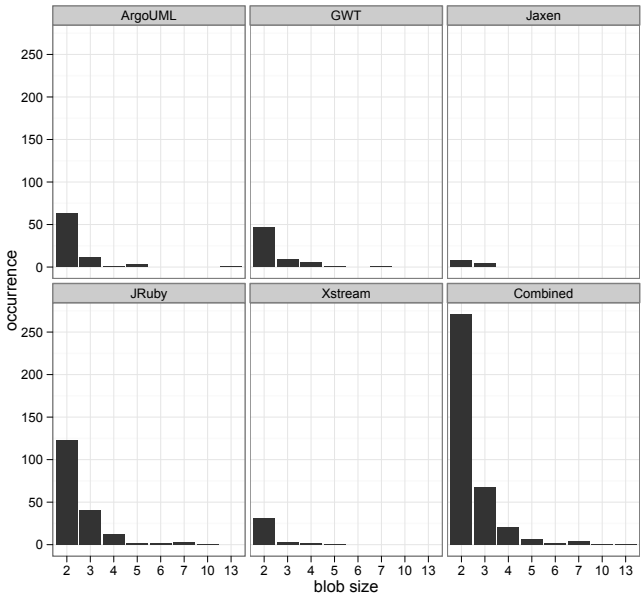

Fig. 6. Real world blob sized frequencies per project and combined.

last three rows in the table contain the same information as the upper three rows but dedicated to bug fixing change sets.

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

> ☛ *Up to 16% of all change sets that can be associated with bug reports address multiple concerns.*

Figure 6 shows the blob size of classified tangled change sets. Tangled change sets of blob size two are the most frequent ones. Change sets of blob size four or above are rare. Overall sets, change sets of blob size two make up 73% while 91% of all tangled change sets have a blob size lower than four.

> ☛ *73% of all tangled changes have a blob size of two.*

B. Untangling Changes (RQ2)

For *RQ2* we evaluate the proposed untangling algorithm and measure its untangling precision using artificially tangled change sets. Table III contains the number of generated artificially tangled change sets grouped by blob size and combination strategy (see Section IV-B). 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.

TABLE III
NUMBER OF ARITIFICIALLY 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
$\sum$	pack.	53	190	1.2M	248
	coupl.	0	20	101.1k	0
	consec.	1,596	170	710.0k	108

TABLE IV
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

The precision of our algorithm to untangle these artificially tangled change sets is shown in Table IV. 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 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.

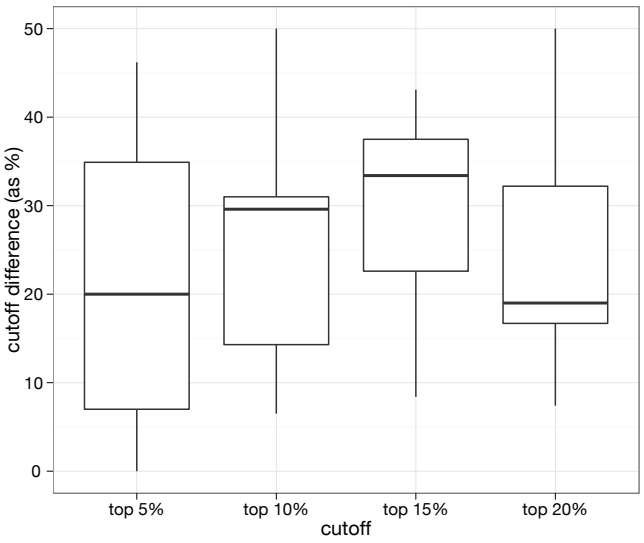

Fig. 7. The `cutoff_differences` caused by real tangled change sets.

Projects with higher number of generated artificially tangled change sets also show higher untangling precision. Overall, the precision values across projects show similar ranges and most importantly similar trends in relation to the chosen blob size. 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 misclassifications. Figure 6 shows that tangled change sets with a blob size of two are most common (73%). The results in Table IV 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.*

C. The Impact of Tangled Changes (RQ3)

Remember that we untangled only those change sets that we manually classified as tangled change sets (see Table II). The fraction of tangled change sets lies between 6% and 15%. Figure 7 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 `cutoff_difference` for the top 5%, 10%, 15%, and 20% of files with the highest distinct number of associated bug reports. 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 not unlikely 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.*

VII. Threats to Validity

Empirical study 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. This 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.

Internally our untangling algorithm uses the partial program analysis tool [31] by Dagenais and Hendren. The validity of our results depends on the validity of the used approach.

VIII. Conclusion and Consequences

Tangled changes introduce noise in change data sets: In this paper, we found up that up to 20% of all bug fixes to consist of multiple tangled changes. This noise can severely impact bug counting 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. These numbers are the main contribution of this paper, and they demand for action.

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. Therefore, 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

ACKNOWLEDGMENTS

Sascha Just and Jeremias Rößler provided constructive feedback on earlier versions of this work. We thank the reviewers for their constructive comments.

REFERENCES

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

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

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

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

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

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

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

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

[9] T. H. Nguyen, B. Adams, and A. E. Hassan, "A Case Study of Bias in Bug-Fix Datasets," in *2010 17th Working Conference on Reverse Engineering*. IEEE Computer Society, 2010, pp. 259–268.

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

[11] G. Canfora, M. Ceccarelli, L. Cerulo, and M. Di Penta, "Using multivariate time series and association rules to detect logical change coupling: An empirical study," in *Proceedings of the 2010 IEEE International Conference on Software Maintenance*, ser. ICSM '10. IEEE Computer Society, 2010, pp. 1–10.

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

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

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

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

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

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

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

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

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

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

[22] D. Kawrykow, "Enabling precise interpretations of software change data," Master's thesis, McGill University, August 2011.

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

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

[25] P. Jaccard, "Étude comparative de la distribution florale dans une portion des Alpes et des Jura," *Bulletin del la Société Vaudoise des Sciences Naturelles*, vol. 37, pp. 547–579, 1901.

[26] A. Mockus and L. G. Votta, "Identifying reasons for software changes using historic databases," in *Proceedings of the International Conference on Software Maintenance (ICSM'00)*, ser. ICSM '00. IEEE Computer Society, 2000, pp. 120–.

[27] K. Herzig, S. Just, and A. Zeller, "It's not a Bug, It's a Feature: How Misclassification Impacts Bug Prediction," Tech. Rep., August 2012, accepted for ICSE 2013.

[28] G. Karypis and V. Kumar, "Analysis of multilevel graph partitioning," in *Proceedings of the 1995 ACM/IEEE conference on Supercomputing*, ser. Supercomputing 1995. ACM, 1995.

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

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

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