CMSuggester: Method Change Suggestion to Complement Multi-Entity Edits

Ye Wang[1], Na Meng[1], and Hao Zhong[2]

[1] Virginia Tech, Blacksburg VA 24061, USA
{yewang16,nm8247}@vt.edu
[2] Shanghai Jiao Tong University, Shanghai 200240, China
zhonghao@sjtu.edu.cn

Abstract. Developers spend significant time and effort in maintaining software. In a maintenance task, developers sometimes have to simultaneously modify multiple program entities (*i.e.*, classes, methods, and fields). We refer to such complex changes as *multi-entity edits*. It is challenging for developers to apply multi-entity edits consistently and completely. Existing tools provide limited support for such edits, mainly because the co-changed entities usually contain diverse program contexts and experience different changes. This paper introduces CMSuggester, an automatic approach that suggests complementary changes for multi-entity edits. Given a multi-entity edit that adds a field and modifies one or more methods to access the field, CMSuggester suggests other methods to co-change for the new field access. CMSuggester is inspired by our previous empirical study, which reveals that *the methods co-changed to access a new field usually commonly access the same set of fields declared in the same class*. By extracting the fields accessed by the given changed method(s), CMSuggester identifies and recommends any unchanged method that also accesses those fields.
Our evaluation shows that CMSuggester recommends changes for 279 out of 408 suggestion tasks. With the recommended methods, CMSuggester achieves 73% F-score on average, while the widely used tool ROSE achieves 48% F-score. In most cases, as shown in our evaluation results, CMSuggester are useful for developers, since it recommend complete and correct multi-entity edits.

Keywords: Multi-entity edit, common field access, change suggestion.

1 Introduction

Developers spend almost 70% of time and resources in maintenance to fix bugs, add features, or refactor code [11]. Due to the complexity of modern software systems, developers sometimes apply complex changes by modifying multiple program entities (*i.e.*, classes, methods, and fields) for one maintenance task. Herzig *et al.* [15] report that more than half of maintenance issues are related to bug fixes, and Zhong and Su [32] report that developers fixed around 80% of real bugs by editing multiple program locations together. *A multi-entity edit is a* *program commit that simultaneously changes multiple entities.* It is challenging for developers to always apply multi-entity edits consistently and completely. Park et al. studied why some bug fixes failed to repair bugs [24], and observed that developers sometimes failed to identify all the edit locations relevant to a bug. For instance, they can forget to update the value initialization of a newly added field in certain methods.

Existing tools provide limited support for how to apply multi-entity edits [33, 30, 17, 22]. For instance, Zimmerman et al. [33] and Ying et al. [30] independently developed tools to mine the association rules between co-changed entities from software version histories, e.g., "*if method **A** is changed, method **B** is also changed*". Based on such rules, when developers change method **A**, method **B** is automatically suggested as a likely change. However, the suggestion accuracy of these tools is low for two reasons. First, they do not observe any syntactic or semantic relationship among co-changed entities. If some entities are *accidentally* co-changed in history, the resulting inferred rules are incorrect, and can produce false positives (e.g., false alarms) when predicting changes. Second, if some entity pairs were never co-changed in history, these tools cannot infer or predict any potential co-change of the entities in the future, causing false negatives.

Another related tool is LSDiff [17]. Given a textual diff, LSDiff infers systematic structural differences as logic rules, and detects anomalies from systematic changes as exceptions to the inferred logic rules. One sample rule is "*All classes implementing type **A** have method **B** deleted except class **C**.*" LSDiff mainly focuses on systematic entity additions and deletions, instead of entity changes (or updates). LASE infers a general program transformation from the exemplar edits in several similarly changed methods, and leverages the inferred transformation to (1) locate other methods for change and (2) suggest customized edits [22]. LASE is useful only when similar methods should be changed similarly; it does not help if distinct edits should be co-applied to dissimilar methods.

A recent study reveals a ***CM→AF** change pattern, which popularly exists in multi-entity edits [28]. **AF** means *Added Field*, while ***CM** represents one or more *Changed Methods*. The pattern shows that when one field is added, developers usually change multiple methods together to access the field. As the co-changed methods usually contain different program contexts and experience divergent changes, developers may forget to change *all* relevant methods to access the new field. This paper introduces a novel approach—CMSuggester—that suggests methods to co-change and helps developers completely apply such edits. Specifically, we first conducted a preliminary study (Section 3) to explore whether there is any syntactic or semantic relationship between the co-changed entities in ***CM→AF** edits. We found that the co-changed methods usually access common fields before an edit is applied. It indicates that **there are clusters of methods that access the same sets of fields**. If one or more methods in a cluster are changed to access a new field, the other methods from the same cluster are likely to be co-changed for the new field access.

Based on the observation, we developed CMSuggester to suggest complementary changes for ***CM→AF** multi-entity edits (Section 4). Specifically, given an added field (f_n) and one or more changed methods, CMSuggester first extracts existing fields accessed by the changed methods. If some of such fields (i) have the same naming pattern as f_n, and (ii) are accessed in the same way as f_n (*i.e.*, purely read, purely written, or read-written), CMSuggester considers them as the *peer fields* of f_n. CMSuggester then locates any unchanged method that accesses the peer fields, and suggests those methods as candidate change locations.

In this paper, we made the following contributions:

– We designed and implemented a novel approach CMSuggester that suggests complementary changes for ***CM→AF** edits. The approach is based on our empirical study [28], which reveals that the co-changed methods for an added field usually access existing fields in common. CMSuggester can be integrated to Integrated Development Environment (IDE) or version control systems to help developers completely apply multi-entity changes.
– We compared CMSuggester with a widely used tool ROSE [33], in terms of their change suggestion capability. We leveraged 106 real multi-entity edits from 4 open source projects to construct 408 change suggestion tasks. Within each task, a tool is given one added field and one related changed method as input to predict likely changes. CMSuggester recommends changes for 279 of these tasks. Among the 279 cases, CMSuggester's recommendation obtains 75% precision, 72% recall, and 73% F-score on average. Meanwhile, ROSE suggests changes in only 117 cases, obtaining 41% precision, 58% recall, and 48% F-score. The results indicate that CMSuggester effectively complements ROSE when suggesting changes for ***CM→AF** edits.
– We defined two filters in CMSuggester to ensure accurate method change suggestion. By disabling the filters, we implemented three variants of CMSuggester, and the evaluation results on these variants show our filters improve our f-scores by about 10%. Especially, the naming-based filter achieves a better trade-off between precision and recall than the access-based filter.

2 Motivating Example

Developers can fail to fully apply changes in tasks requiring multi-entity edits. Figure 1 shows a simplified program revision to Derby [4]—a Java-based relational database. The added code is colored with **blue** and marked with "+". In this revision, developers added a field `_clobValue` (line 5) and modified 12 methods in different ways to access the field (e.g., changing `getLength()` at lines 8-9). However, developers forgot to also change `restoreToNull()` (lines 15-20). Consequently, the multi-entity edit is incomplete. The inadvertently "*missed change*" remained in the software for more than two years, until developers finally inserted a statement `_clobValue = null;` to `restoreToNull()` [7].

It is challenging for developers to examine or ensure the completeness of such multi-entity edits. When a method fails to be changed to access a new field, compilation error are often not triggered, neither can any well-known bug detector reveal the problem. In this example, the missing field access in `restoreToNull()`

did not introduce any compilation errors, and was identified two years after it was first introduced.

In this paper, we developed CMSuggester, a tool that identifies complementary changes and helps developers avoid incomplete multi-entity edits. For this example, given the added field `_clobValue` and the changed method `getLength()`, CMSuggester identifies two existing fields accessed by `getLength()`: `rawLength` and `stream`. Similar to `_clobValue`, these fields are *purely read* by the method. Thus, CMSuggester considers both fields as *peers* of the new field. CMSuggester then searches for any method that accesses the peer fields but has not been changed to access the new field. In this way, CMSuggester finds `restoreToNull()`—which accesses the peer fields in the same "*pure write*" mode—and recommends the method for change. With CMSuggester, developers can identify the change locations that they may otherwise miss when applying multi-entity edits.

```
public class SQLChar extends
    DataType implements
    StringDataValue, StreamStorable{
  ...
+ protected Clob _clobValue;
  public int getLength() throws
      StandardException{
+   if ( _clobValue != null ) {
+     return getClobLength(); }
    if (rawLength != -1)
      return rawLength;
    if (stream != null) {
    ...
  }
  public void restoreToNull() {
    value = null;
    stream = null;
    rawLength = -1;
    cKey = null;
  }}
```

Fig. 1: A program revision requires 1 field addition and 13 method-level changes. However, developers changed only 12 of the 13 methods, ignoring `restoreToNull()` for change [6].

3 Our Empirical Finding

In our prior study [28], we analyzed 2,854 bug fixes from four popular open source projects to explore multi-entity edits, including Aries [2], Cassandra [3], Derby [4], and Mahout [5]. Our study shows that recurring change patterns commonly exist in all the projects. In particular, ***CM→AF** is one of the most popular patterns. Therefore, in this paper, we sampled five such commits in each project to manually analyze the co-changed methods for any newly added field.

Table 1 presents our inspection results. For each added field, there are 2-5 methods co-changed to access the field. We manually compared co-changed methods to identify any commonality between them. We found that **in 15 of the 20 examined revisions, the co-changed methods commonly access existing field(s) before the edits are applied.** Among the other five program commits, two commits have co-changed methods to commonly invoke certain method(s), while the remaining ones have no commonality among them. Our finding shows that when one or more methods in a cluster are changed to access a new field, the other methods from the same cluster are likely to be co-changed for the new field access. This finding is consistent with the Object Oriented (OO)

Table 1: Commonality inspection of 20 ***CM→AF** multi-entity edits

Project	**Commits**	**Added Field**	**# of Changed Methods**	**Commonality**
Aries	3d072a4	monitor	2	Field access
	50ca3da	properties	2	Field access
	5d334d7	BEAN	2	Method invocation
	95766a2	NS_AUTHZ	2	None
	9586d78	enlisted	3	Field access
Cassandra	0792766	validBufferBytes	3	Field access
	0963469	isStopped	2	Field access
	0d1d3bc	componentIndex	3	Field access
	1c9c47d	nextFlags	2	Field access
	266e94f	STREAMING_SUBDIR	2	Method invocation
Derby	f578f070	stateHoldability	2	Field access
	6eb5042	outputPrecision	2	Field access
	2f41733	MAX_OVERFLOW_ONLY_REC_SIZE	3	None
	099e28f	XML_NAME	3	Field access
	81b9853	activation	5	Field access
Mahout	0be2ea4	LOG	2	Field access
	0fe6a49	FLAG_SPARSE_ROW	2	Field access
	22d7d31	namedVector	2	Field access
	29af4d7	normalizer	2	Field access
	2f7f0dc	NUM_GROUPS_DEFAULT	2	None

paradigm, since OO emphasizes to group related data in the same structure to ease modification and understanding [25].

4 Approach

Section 3 shows *it is promising to suggest methods for change based on the accessed fields by already-changed methods*. Inspired by that, we developed CMSuggester with the hypothesis that ***similar field usage indicates methods' co-change relationship***. Figure 2 shows the overview of the approach. Given an edit that adds a field and changes one or more methods to access the field, CMSuggester extracts peer fields from the changed method(s) (Section 4.1), filters the fields based on naming patterns and access modes (Section 4.2 and 4.3), and searches for any unchanged method with the refined fields for change suggestion (Section 4.4).

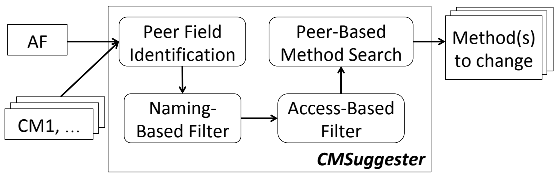

Fig. 2: CMSuggester's overview

4.1 Peer Field Identification

Based on our finding in Section 3, we believe that when a cluster of methods need to commonly access a cluster of fields to implement relevant functionalities, the fields are more likely to be defined in the same class. Given a newly added field f_n, we use **peer fields** to denote the existing fields that are (1) declared in the same class as f_n, and (2) accessed by one or more changed methods that also access f_n. For our motivating example, the newly added field is `_clobValue`, and we identify that in the method `getLength()`, `rawLength` and `stream` are its peer fields. CMSuggester traverses the AST of each changed method's old version to extract the accessed existing fields, obtaining a peer field set $P = \{p_1, p_2, \ldots\}$.

4.2 Naming-Based Filtering

We notice that peer fields may have diverse powers to indicate the usage of f_n. To ensure CMSuggester's accuracy when suggesting methods for change, we refined the peer fields P with two intuitive filters. The first filter leverages the heuristic that *similarly named fields are more likely to be used similarly than other fields.* This filter compares peer fields with f_n, and removes any field whose naming pattern is different from f_n's. We observed **two naming patterns** that developers usually followed when defining fields.

– **Pattern 1:** The names of constant fields (e.g., `static final`) capitalize all involved letters, such as `MAX_OVERFLOW_ONLY_REC_SIZE`.
– **Pattern 2:** The names of variable fields use lowercase or a combination of lowercase and uppercase letters, such as `outputPrecision`.

We rely on the naming patterns to classify fields as variables or constants. If f_n is a variable, it is likely to be similarly used to existing variable fields, so we filter out the constant peers in P. Similarly, if f_n is a constant, we can use the constant peers to suggest f_n's usage, and remove variable peers from P.

4.3 Access-Based Filtering

This filter implements another heuristic that *similarly accessed fields are more likely to have similar usage.* For each method, we classify the accessed fields into three access modes: **pure read**, **pure write**, and **read-write**, depending on how each field is accessed. For instance, if a method reads and writes a field, we put the field into the "*read-write*" category of that method. To implement the filter, CMSuggester scans the internal representation (IR) of each CM's old version created by WALA [10], and checks if an accessed field serves as a left or right value of each IR instruction. If the field serves as a right value, it is read by an instruction; otherwise, it is written. When a field's access mode is distinct from that of f_n, CMSuggester removes the field from P.

4.4 Peer-Based Method Search

With the refined fields, CMSuggester searches for methods to co-change by identifying any unchanged method that accesses at least two refined fields. In the search, CMSuggester scans a large portion of code, because a program revision usually changes a small portion of code while keeping the majority code unchanged [32]. To improve the search efficiency, we leveraged the access modifiers of f_n to reduce search space. Specifically, if f_n is a `private` field, only the methods declared by f_n's declaring class C are analyzed because f_n is not visible to

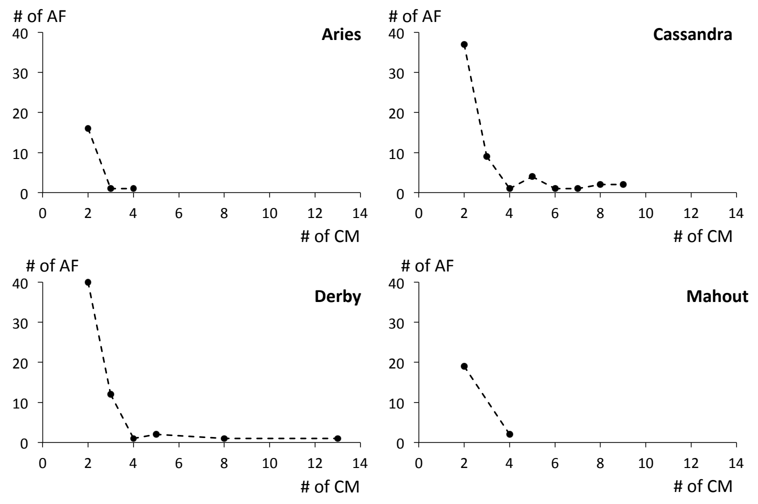

Fig. 3: The distribution of AFs based on the number of corresponding CMs

any method outside C. Similarly, if f_n is a `protected` field, only the methods declared in C and C's subclasses are analyzed. In the worst case, when a field f_n is a `public` field, it is visible to any method in the whole project; thus, we cannot reduce the search space, but instead scan all unchanged methods.

5 Evaluation

This section explains our data set (Section 5.1) and presents our defined metrics to evaluate CMSuggester's effectiveness (Section 5.2). The experiments discussed in Section 5.3-5.4 were designed to explore the following two research questions:

– **RQ1:** What is CMSuggester's effectiveness to predict methods for change, and how does it compare with ROSE?
– **RQ2:** How does CMSuggester's effectiveness vary with the two used filters?

5.1 Data Set

We created an evaluation data set based on the data of our prior work [28]. We searched for any ***CM→AF** edit that (1) contains at least two methods co-changed for an added field, and (2) has each changed method accessing at least two existing fields. In this way, we found 10 commits, 45 commits, 42 commits, and 9 commits separately in the revision data of Aries, Cassandra, Derby, and Mahout, among which each commit contains one or more ***CM→AF** edits. Figure 3 shows the distribution of added fields (AFs) based on the number of changed methods (CMs) corresponding to them. Each commit has one or more added AFs, in which each AF is related to 2-13 CMs. In particular, among the 9 commits from Mahout, there are 21 AFs applied. 19 of these AFs have 2 CMs co-applied; while each of the other 2 AFs are co-applied with 4 CMs.

For each AF, we constructed suggestion tasks by providing the AF and some of its co-applied CMs to CMSuggester as input, and using the remaining part as the oracle to evaluate CMSuggester's output. For instance, suppose that a commit has one added field f_n and two changed methods $M = \{m_1, m_2\}$. In one task, we provide f_n and m_1 to CMSuggester, and check whether CMSuggester suggests m_2 for change. Alternatively, we can provide f_n and m_2 to CMSuggester, and check whether CMSuggester's output is m_1. In this way, if a ***CM→AF** edit has one AF and n CMs ($n \geq 2$), we can create n ***one-AF-one-CM (1A1C)*** tasks based on the edit. In each task, only one AF and one CM are provided as input, and all the other CM(s) is/are treated as the expected output. Similarly, we can create ***one-AF-two-CM (1A2C)*** and ***one-AF-three-CM (1A3C)*** tasks. As the majority of AFs in our data set correspond to 2-4 CMs, our experiments mainly focus on 1A1C, 1A2C, and 1A3C tasks, as shown in Table 2.

Table 2: Evaluation data set

	Aries	**Cassandra**	**Derby**	**Mahout**	**Total #**
# of program commits	10	45	42	9	106
# of 1A1C suggestion tasks	39	172	151	46	408
# of 1A2C suggestion tasks	9	237	168	12	426
# of 1A3C suggestion tasks	4	379	366	8	757

5.2 Metrics

We defined and used four metrics to measure a tool's capability of suggesting methods for change: coverage, precision, recall, and F-score. We also defined the weighted average to measure a tool's overall effectiveness among all subject projects for each of the metrics mentioned above.

Coverage (C) measures the percentage of tasks for which a tool is able to provide suggestion. Given a task, a tool may or may not suggest any change to complement the already-applied edit, so this metric assesses a tool's applicability.

$$C = \frac{\text{\# of tasks with a tool's suggestion}}{\text{Total \# of tasks}} * 100\% \tag{1}$$

Intuitively, if a tool always suggests something given a task, its coverage is 100%, and thus the tool is widely applicable. All our later evaluations for precision, recall, and F-score are limited to the tasks covered by a tool. For instance, suppose that given 100 tasks, a tool can suggest changes for 8 tasks. Then the tool's coverage is 8/100 = 8%, and the evaluations for other metrics are based on these 8 tasks instead of the original 100 tasks.

Precision (P) measures among all methods suggested by a tool, how many of them are correct:

$$P = \frac{\text{\# of correct suggestions}}{\text{Total \# of suggestions by a tool}} * 100\% \tag{2}$$

This metric evaluates how precisely a tool suggests changes. If all suggestions by a tool are contained by the oracle or expected output, the precision is 100%.

Recall (R) measures among all the expected suggestions, how many of them are actually reported by a tool:

$$R = \frac{\text{\# of correct suggestions by a tool}}{\text{Total \# of expected suggestions}} * 100\% \tag{3}$$

This metric assesses how effectively a tool retrieves the expected outcome. Intuitively, if all expected suggestions are reported by a tool, the recall is 100%.

F-score (F) measures the accuracy of a tool's suggestion:

$$F = \frac{2 * P * R}{P + R} * 100\% \tag{4}$$

F-score is the harmonic mean of precision and recall. Its value varies within [0%, 100%]. The higher F values are desirable, as they demonstrate better trade-offs between the precision and recall rates.

Weighted Average (WA) measures a tool's **overall effectiveness** among all experimented data in terms of coverage, precision, recall, and F-score:

$$\Gamma_{overall} = \frac{\sum_{i=1}^{4} \Gamma_i * n_i}{\sum_{i=1}^{4} n_i}. \tag{5}$$

In the formula, i varies from 1 to 4, representing Aries, Cassandra, Derby, and Mahout in sequence. n_i represents the number of tasks built from the i^{th} project. Γ_i represents any measurement value of the i^{th} project for coverage, precision, recall, or F-score. By combining such measurement values of all projects in a weighted way, we are able to assess a tool's overall effectiveness $\Gamma_{overall}$.

5.3 Comparison with ROSE

To assess CMSuggester's capability of suggesting complementary changes, we used CMSuggester to complete the tasks mentioned in Table 2 (*i.e.*, 1A1C, 1A2C, and 1A3C). To understand how CMSuggester is compared with prior work, we also executed the state-of-the-art co-change suggestion tool, ROSE [33], for the same tasks. ROSE mines the association rules between co-changed entities from software version histories. Below presents an exemplar rule mined by ROSE [33]:

$$\begin{aligned} &\{(\_Qdmodule.c, func, GrafObj\_getattr())\} \Rightarrow \\ &\{\,(qdsupport.py, func, outputGetattrHook()).\,\} \end{aligned} \tag{6}$$

This rule means that whenever the function `GrafObj_getattr()` in a file `_Qdmodule.c` is changed, the function `outputGetattrHook()` in another file `qdsupport.py` should also be changed. We configured ROSE with support=1, confidence=0.1, because the paper [33] mentioned this setting more often than other settings.

Table 3 shows the results of CMSuggester and ROSE for 1A1C tasks. Overall, CMSuggester obtained higher measurement values for all the

Table 3: CMSuggester vs. ROSE for 1A1C tasks (%)

Project	CMSuggester				ROSE			
	C	P	R	F	C	P	R	F
Aries	51	68	85	76	31	35	39	37
Cassan-dra	69	81	75	78	38	53	71	61
Derby	71	71	68	69	22	25	42	31
Mahout	72	72	68	70	13	5	33	9
WA	**68**	**75**	**72**	**73**	**29**	**41**	**58**	**48**

projects than ROSE. Particularly for Mahout, CMSuggester predicted likely changes for 72% of the tasks, while ROSE provided predictions for 13% of the tasks. Among the generated suggestions, CMSuggester achieved 72% precision, 68% recall, and 70% F-score; while ROSE obtained 5% precision, 33% recall, and 9% F-score. CMSuggester's weighted average values for coverage, precision, recall, and F-score are 68%, 75%, 72%, and 73%, while ROSE's corresponding weighted average values are 29%, 41%, 58%, and 48%.

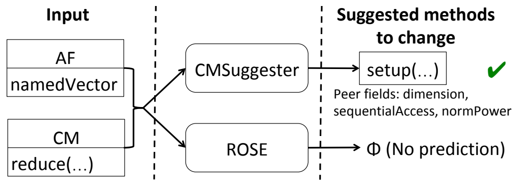
Input
AF
namedVector
CM
reduce(...)
CMSuggester
ROSE
Suggested methods to change
setup(...)
✔
Peer fields: dimension, sequentialAccess, normPower
Φ (No prediction)

Fig. 4: A task for which CMSuggester outperformed ROSE

Two major reasons can explain why CMSuggester outperformed ROSE. First, ROSE uses the co-changed entities in version histories to predict likely changes. When the history data are incomplete or some entities were never co-changed before, ROSE lacks the evidence to predict some co-changes, obtaining lower coverage and recall rates. Second, ROSE does not leverage any syntactic or semantic relation between the co-changed entities. ROSE can infer incorrect rules from co-changed but unrelated entities, achieving lower precision.

Figure 4 presents a task for which CMSuggester outperformed ROSE. This task is extracted from the commit 22d7d31 [9] of Mahout. In the task, there is one AF `PartialVectorMergeReducer.namedVector` and one CM `PartialVectorMergeReducer.reduce(...)` provided as input, and another CM provided as the expected output. CMSuggester succesfully predicted `PartialVectorMergeReducer.setup(...)` based on the three peer fields extracted from the given CM. However, ROSE could not predict any method, because the version history did not manifest any association rule between `reduce(...)` and `setup(...)`.

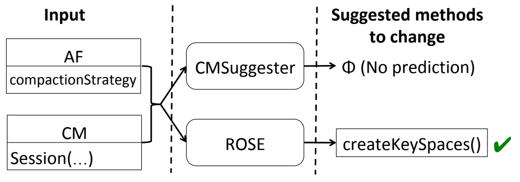

Fig. 5: A task for which ROSE outperformed CMSuggester

Figure 5 shows a task for which ROSE worked better than CMSuggester. This task is from the commit f06e1d6 [1] of Cassandra. It provides one AF `Session.compactionStrategy` and one CM `Session.Session(...)` as input, and includes another CM as the oracle. CMSuggester predicted nothing, because the identified peer fields in `Session(...)` are not commonly used by any unchanged method. However, ROSE correctly suggested one method `Session.createKeySpaces(...)`. Our results show that CMSuggester can complement ROSE by suggesting co-changes in a different way.

Finding 1: *CMSuggester outperformed ROSE in many 1A1C tasks. This means that CMSuggester complements ROSE by inferring co-changes from methods' common field accesses instead of from the history.*

In addition to 1A1C tasks, we also compared CMSuggester with ROSE for 1A2C and 1A3C tasks, as shown in Tables 4 and 5. The tools have similar F-scores. For 1A2C tasks, CMSuggester obtained 61% F-score, while ROSE achieved 62%. For 1A3C tasks, the F-score comparison is 61% vs. 60%. More importantly, CMSuggester obtained much higher coverage rates than ROSE for both types of tasks. In Table 4, the coverage comparison is 85% vs. 8%. For Aries and Mahout, CMSuggester achieved 89% and 100%, meaning that it predicted changes for the majority of tasks. However, ROSE predicted nothing for either project. In Table 5, CMSuggester's coverage is 88%, while ROSE's is 17%. Especially for Aries, Derby, and Mahout, CMSuggester achieved 100% coverage, while ROSE covered 0% of the tasks.

Table 4: CMSuggester vs. ROSE for 1A2C tasks (%)

Project	**CMSuggester**				**ROSE**			
	C	**P**	**R**	**F**	**C**	**P**	**R**	**F**
Aries	89	35	50	41	0	-	-	-
Cassandra	76	65	66	65	31	63	69	66
Derby	96	65	55	60	3	7	15	10
Mahout	100	35	39	37	0	-	-	-
WA	**85**	**63**	**60**	**61**	**8**	**59**	**66**	**62**

The coverage comparisons in Tables 3, 4, and 5 show that given an arbitrary task, CMSuggester is more likely to provide suggestions than ROSE, and the suggestion accuracy is at least comparable to ROSE's. Two reasons can explain it. First, ROSE is limited by the available historical co-change data. If certain methods have never been changed together in history, ROSE can not find potential co-changes between the methods. Second, when more CMs are provided, CMSuggester can detect more peer fields from more methods, and leverage the fields to predict more. Suppose CMSuggester can predict changes $M1 = \{m_{1a}, m_{1b}, \ldots\}$ based on CM1, and predict changes $M2 = \{m_{2a}, m_{2b}, \ldots\}$ based on CM2. Given CM1 and CM2, CMSuggester predicts the joint set of M1 and M2 by outputting $M_s = \{m_{1a}, m_{2a}, m_{1b}, m_{2b}, \ldots\}$. However, ROSE always intersects the prediction sets of individual CMs to ensure its prediction precision. Thus, with the CM1 and CM2, ROSE outputs $M_r = M1 \cap M2$, which covers less methods than M_s.

Table 5: CMSuggester vs. ROSE for 1A3C tasks (%)

Project	**CMSuggester**				**ROSE**			
	C	**P**	**R**	**F**	**C**	**P**	**R**	**F**
Aries	100	12	25	16	0	-	-	-
Cassandra	75	56	62	59	33	57	64	60
Derby	100	66	61	63	0	0	0	-
Mahout	100	21	25	23	0	-	-	-
WA	**88**	**61**	**61**	**61**	**17**	**57**	**63**	**60**

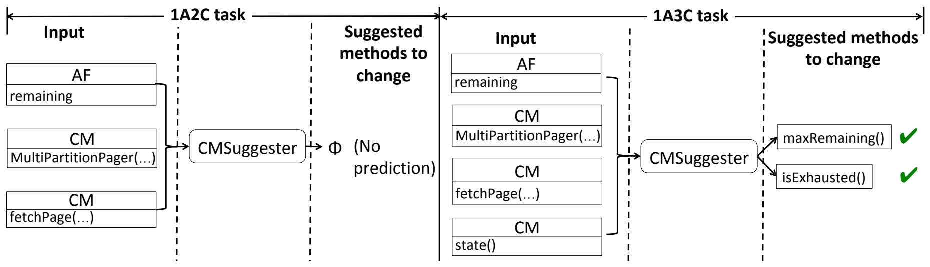

Fig. 6: CMSuggester predicts better when more CMs are provided as input

Figure 6 presents two tasks from Cassandra, showing that when more CMs are provided as input, CMSuggester can predict better. These tasks are from the commit 7c32ffb [8], in which one AF and five CMs were applied by developers. For the 1A2C task, CMSuggester was given one AF and two CMs. Since CMSuggester could not identify enough peer fields from the methods to predict changes, it predicted nothing. In contrast, when CMSuggester was given one more CM in the 1A3C setting, it was able to identify enough peer fields from the third method `MultiPartitionPager.state()` and correctly suggested two methods.

Finding 2: *For 1A2C and 1A3C tasks, when multiple CMs were provided as input, CMSuggester outperforms ROSE by achieving better coverage and at least comparable accuracy. Overall, CMSuggester works better than ROSE when suggesting complementary changes for* ****CM→AF*** *edits.*

5.4 Sensitivity to Filter Settings

Two filters were used in CMSuggester to refine the peer fields. To understand how each filter affects CMSuggester's effectiveness, we built three variant approaches:

– CMSuggester$_o$: We disabled both filters, and leveraged all detected peer fields in the input CM(s) to predict changes.
– CMSuggester$_n$: We only used the naming-based filter to refine peer fields but disabled the access-based filter.
– CMSuggester$_a$: We refined peer fields only with the access-based filter while turning off the naming-based filter.

We applied all three variants to the 1A1C tasks. Table 6 presents the effectiveness comparison between CMSuggester and the variants. According to this table, CMSuggester obtained the lowest overall coverage (68%), but the highest overall precision (75%), recall (72%), and F-score (73%). This is as expected, because CMSuggester applied two filters to refine the detected fields as much as possible. As a result, fewer fields passed both filters and suggested fewer but more accurate changes. CMSuggester$_o$ achieved the highest coverage (91%) but lowest F-score (68%). Since it did not refine peer fields before predicting changes,

Table 6: CMSuggester vs. its three variant approaches with filters on or off (%)

Project	**CMSuggester**				**CMSuggester$_o$**				**CMSuggester$_n$**				**CMSuggester$_a$**			
	C	**P**	**R**	**F**	**C**	**P**	**R**	**F**	**C**	**P**	**R**	**F**	**C**	**P**	**R**	**F**
Aries	51	68	85	76	77	70	83	76	72	70	86	77	56	67	86	75
Cassandra	69	81	75	78	88	78	76	77	80	81	74	77	75	79	76	77
Derby	71	71	68	69	97	63	60	61	94	66	63	64	73	67	64	65
Mahout	72	72	68	70	96	6	57	56	74	72	68	70	93	56	57	56
WA	**68**	**75**	**72**	**73**	**91**	**69**	**68**	**68**	**84**	**73**	**70**	**71**	**75**	**71**	**70**	**70**

some of the included peer fields are used less similarly to the newly added fields, causing incorrect suggestions.

Compared with CMSuggester$_a$, CMSuggester$_n$ obtained better coverage (84% vs. 75%), better precision (73% vs. 71%), equal recall (both 70%), and better F-score (71% vs. 70%). This is out of our expectation. Although the naming-based filter seems more intuitive and is easier to implement than the access-mode filter, it obtained a better trade-off among coverage, precision, and recall. This may indicate that developers usually name fields in meaningful ways. Thus, the similarity in fields' names can more effectively indicate methods' co-change relationship than the similarity in access modes. When some fields are named similarly, even though they are accessed divergently by one or more CMs, the fields' co-occurrence can still effectively predict methods for change.

Finding 3: *Both filters used in CMSuggester effected to improve F-score at the cost of coverage. Especially, the naming-based filter achieved a better balance between F-score and coverage than the access-based filter.*

6 Threats to Validity

Threats to External Validity. Our evaluation results show that CMSuggester outperforms ROSE, as far as only one change pattern is concerned. However, as our prior work [28] found more patterns, it is feasible to extend CMSuggester, and we believe that CMSuggester can outperform ROSE in more cases. Furthermore, based on our manual inspection, we believe that our patterns are not specific to only Apache projects, so CMSuggester can outperform ROSE, even if we select projects from other open source communities as the subjects. In the future, we will support more patterns and evaluate the tools on subjects from more software repositories. We also plan to develop a hybrid approach of ROSE and CMSuggester. By relating methods based on common field accesses and historic co-change relationship, the hybrid approach is guaranteed to suggest changes when either tool predicts something, and may provide more precise suggestions if both tools' outputs can cross-validate each other.

Threats to Construct Validity. When we prepared the golden standards, we constructed suggestion tasks from manual fixes. Yin *et al.* [29] show that a bug fix can be not fully correct, which can lose useful co-changes. It is possible that developers made mistakes when making some multi-entity edits. Therefore, the imperfect evaluation data set based on developers' edits may affect our assessment for both CMSuggester and ROSE. We share this limitation with prior work [33, 22, 27]. In the future, we plan to mitigate the problem by conducting user studies with developers. By carefully going through the edits made by developers and the complementary changes suggested by CMSuggester or other tools, we can further evaluate the usefulness of different tools' suggestions.

7 Related Work

Our research is related to co-change mining, automatic change recommendation, and automatic program repair.

Co-Change Mining. Tools were built to mine version histories for co-change patterns [12, 13, 26, 33, 30]. Specifically, Gall et al. mined release data for the co-change relationship between subsystems [12] and classes [13]. Shirabad et al. trained a machine-learning model to predict whether two given files should be changed together [26]. However, none of these approaches analyze any syntactic or semantic relationship between co-changed modules. Hassan et al. created a framework to predict change propagation based on the historical co-changes, caller-callee relationship of methods, def-use relationship of fields, and/or entities' co-occurrence in the same file [14]. They found that the historic co-changes had better prediction capability than other types of information. Instead of mining software repositories, CMSuggester identifies co-changed methods based on the commonly accessed fields, and complemented above-mentioned approaches when the revision history is limited or unavailable.

Change Recommendation Systems. Researchers built tools to recommend code changes [19, 17, 23, 22]. For instance, PR-Miner was created to mine the implicit API invocation rules (e.g., `lock()` and `unlock` should be called together), to detect any code violating the rules, and to suggest changes that complement existing API invocations [19]. Clever is a tool tracking all clone groups in software and monitoring for edits on clones [23]. If one clone is detected to be updated, Clever lists all its clone peers, and recommends relevant changes. These approaches recommend changes based on either the co-occurrence of APIs or code similarity. In comparison, CMSuggester recommends changes based on the common field accesses between methods.

Automatic Program Repair (APR). There are tools proposed to generate candidate patches for certain bugs, and automatically check patch correctness using compilation and testing [18, 16, 21, 20, 31]. For example, GenProg [18] generated candidate patches by replicating, mutating, or deleting code randomly from the existing programs. Genesis trained a machine-learning model by extracting features from existing bug fixes, and suggesting candidate patches accordingly [20]. CMSuggester is different from APR in two aspects. First, CMSuggester focuses on multi-entity changes by suggesting method changes to complement already-applied edits. However, APR focuses on single-entity changes by creating single-method updates from scratch. Second, CMSuggester locates methods to change, while APR approaches generate concrete and applicable statement-level changes as a candidate fix. We believe that CMSuggester is valuable because it is challenging to locate places to change in large codebases, and it needs to locate such places, before APR tools can generate changes.

8 Conclusion

When developers change multiple entities simultaneously for one maintenance task, it can be challenging for them to identify all relevant entities to edit. This paper presents CMSuggester, a novel approach that suggests complementary changes for multi-entity edits. Different from prior work that relates co-changed methods based on their historic co-change relationship or similar program contexts, CMSuggester takes a different perspective by modeling the common field accesses between methods. Our evaluation shows that CMSuggester outperforms ROSE when suggesting complementary changes for ***CM→AF** edits—a type of frequently applied complex changes. Since ROSE can work well in certain scenarios where CMSuggester does not suggest changes, we plan to explore a hybrid approach between the tools in the future. To better characterize the strengthens and weaknesses of different tools, we will also apply CMSuggester and other tools to suggest changes for more types of multi-entity edits.

Acknowledgment

We thank anonymous reviewers for their valuable comments on our earlier version of the paper. This work was supported by NSF Grant CCF-1565827, National Basic Research Program of China (973 Program) No. 2015CB352203, the National Nature Science Foundation of China No. 61572313, and the grant of Science and Technology Commission of Shanghai Municipality No. 15DZ1100305.

References

1. Support of compaction strategy option for stress.java. `https://github.com/apache/cassandra/commit/f06e1d63a2006aa95d36636c56561158c8758a3c`
2. Apache Aries. `http://aries.apache.org` (2018)
3. apache/cassandra. `https://github.com/apache/cassandra` (2018)
4. apache/derby. `https://github.com/apache/derby` (2018)
5. apache/mahout. `https://github.com/apache/mahout` (2018)
6. DERBY-2201: Allow scalar functions to return LOBs. `https://github.com/apache/derby/commit/638f1b48afc27c094c7f34a6254778c1a4ad9608` (2018)
7. DERBY-5162: Null out the wrapped Clob when resetting a SQLClob to NULL. `https://github.com/apache/derby/commit/e9737b6` (2018)
8. Fix infinite loop when paging queries with IN. `https://github.com/apache/cassandra/commit/7c32ffb` (2018)
9. MAHOUT-401: Use NamedVector in seq2sparse. `https://github.com/apache/mahout/commit/22d7d31` (2018)

10. WALA. `http://wala.sourceforge.net/wiki/index.php/Main_Page` (2018)
11. Christa, S., Madhusudhan, V., Suma, V., Rao, J.J.: Software maintenance: From the perspective of effort and cost requirement. In: Proc. ICDECT. pp. 759–768 (2017)
12. Gall, H., Hajek, K., Jazayeri, M.: Detection of logical coupling based on product release history. In: Proc. ICSM. pp. 190–198 (1998)
13. Gall, H., Jazayeri, M., Krajewski, J.: CVS release history data for detecting logical couplings. In: Proc. IWPSE. pp. 13–23 (2003)
14. Hassan, A.E., Holt, R.C.: Predicting change propagation in software systems. In: Proc. ICSM. pp. 284–293 (2004)
15. Herzig, K., Just, S., Zeller, A.: It's not a bug, it's a feature: how misclassification impacts bug prediction. In: Proc. ICSE. pp. 392–401 (2013)
16. Kim, D., Nam, J., Song, J., Kim, S.: Automatic patch generation learned from human-written patches. In: Proc. ICSE. pp. 802–811 (2013)
17. Kim, M., Notkin, D.: Discovering and representing systematic code changes. In: Proc. ICSE. pp. 309–319 (2009)
18. Le Goues, C., Nguyen, T., Forrest, S., Weimer, W.: Genprog: A generic method for automatic software repair. IEEE Trans. Softw. Eng. **38**(1) (2012)
19. Li, Z., Zhou, Y.: PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code. In: Proc. ESEC/FSE. pp. 306–315 (2005)
20. Long, F., Amidon, P., Rinard, M.: Automatic inference of code transforms for patch generation. In: Proc. ESEC/FSE. pp. 727–739 (2017)
21. Long, F., Rinard, M.: Automatic patch generation by learning correct code. In: Proc. POPL. pp. 298–312 (2016)
22. Meng, N., Kim, M., McKinley, K.: Lase: Locating and applying systematic edits. In: Proc. ICSE. pp. 502–511 (2013)
23. Nguyen, T.T., Nguyen, H.A., Pham, N.H., Al-Kofahi, J.M., Nguyen, T.N.: Clone-aware configuration management. In: Proc. ASE. pp. 123–134 (2009)
24. Park, J., Kim, M., Ray, B., Bae, D.H.: An empirical study of supplementary bug fixes. In: Proc. MSR. pp. 40–49 (2012)
25. Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorensen, W.: Object-oriented Modeling and Design. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1991)
26. Shirabad, J.S., Lethbridge, T.C., Matwin, S.: Mining the maintenance history of a legacy software system. In: Proc. ICSM. pp. 95–104 (2003)
27. Tan, Ming: Online Defect Prediction for Imbalanced Data. Master's thesis, University of Waterloo (2015)
28. Wang, Y., Meng, N., Zhong, H.: An empirical study of multi-entity changes in real bug fixes. In: Proc. ICSME (2018)
29. Yin, Z., Yuan, D., Zhou, Y., Pasupathy, S., Bairavasundaram, L.: How do fixes become bugs? In: Proc. ESEC/FSE. pp. 26–36 (2011)
30. Ying, A.T.T., Murphy, G.C., Ng, R.T., Chu-Carroll, M.: Predicting source code changes by mining change history. IEEE Trans. Software Eng. **30**(9), 574–586 (2004)
31. Zhong, H., Mei, H.: Mining repair model for exception-related bug. The Journal of Systems & Software **141**, 16–31 (2018)
32. Zhong, H., Su, Z.: An empirical study on real bug fixes. In: Proc. ICSE. pp. 913–923 (2015)
33. Zimmermann, T., Weisgerber, P., Diehl, S., Zeller, A.: Mining version histories to guide software changes. In: Proc. ICSE. pp. 563–572 (2004)