Automatic Method Change Suggestion to Complement Multi-Entity Edits

Zijian Jiang[a], Ye Wang[a], Hao Zhong[b], Na Meng[a,*]

[a] *Virginia Polytechnic Institute and State University, Blacksburg VA* 24060*, USA*
[b] *Shanghai Jiao Tong University, Shanghai* 200240*, China*

Abstract

When maintaining software, developers sometimes change multiple program entities (*i.e.*, classes, methods, and fields) to fulfill one maintenance task. We call such complex changes *multi-entity edits*. Consistently and completely applying multi-entity edits can be challenging, because (1) the changes scatter in different entities and (2) the incorrectly edited code may not trigger any compilation or runtime error. This paper introduces CMSuggester, an approach to suggest complementary changes for multi-entity edits. Given a multi-entity edit that (i) adds a new field or method and (ii) modifies one or more methods to access the field or invoke the method, CMSuggester suggests other methods to co-change for the new field access or method invocation. The design of CMSuggester is motivated by our preliminary study, which reveals that co-changed methods usually access existing fields or invoke existing methods in common.

Our evaluation shows that based on common field accesses, CMSuggester recommended method changes in 463 of 685 tasks with 70% suggestion accuracy; based on common method invocations, CMSuggester handled 557 of 692 tasks with 70% accuracy. Compared with prior work ROSE, TARMAQ, and Transitive Association Rules (TAR), CMSuggester recommended more method changes with higher accuracy. Our research can help developers correctly apply multi-entity edits.

*Corresponding author
Email address: `nm8247@vt.edu` (Na Meng)

Keywords: Multi-entity edit; common field access; common method invocation; change suggestion

1. Introduction

Software maintenance is challenging and time-consuming. Christa et al. recently revealed that almost 70% of developers' time and resources were allocated to maintenance activities [1]. When maintaining software, developers may apply complex program changes by editing several program entities (*i.e.*, classes, methods, and fields) for one maintenance task (e.g., bug fix). For instance, a study by Zhong and Su [2] shows that developers fixed around 80% of real bugs by changing multiple program locations together. In this paper, we refer to a program commit as a *multi-entity edit* if it simultaneously changes multiple entities. Multi-entity edits can be difficult to apply consistently and completely. Park et al. once examined supplementary bug fixes—patches that were later applied to supplement or correct initial fix attempts [3]. These researchers found that developers sometimes failed to edit all program locations as needed for one bug, e.g., by inserting the value initialization of a newly added field to *some but not all* relevant methods.

Existing work is insufficient to help with such edit application [4, 5, 6, 7, 8, 9]. For example, ROSE mines software version history to identify change association rules like "*if method **A** is changed, method **B** should also be changed*" [4]. Given a program commit, ROSE checks the applied changes against identified rules to reveal any missing change. However, the accuracy of identified rules is low for two reasons. First, the *co-change* relationship between entities does not guarantee their *syntactic or semantic relevance*, so some rules identified in this way are actually false alarms. Second, some syntactically or semantically related entities were never changed together in history, so ROSE cannot reveal the entity relationship, causing false negatives.

LSDiff infers systematic structural change rules from a given program commit, and detects anomalies from systematic changes as exceptions to the inferred rules [6]. For instance, one representative inferred rule is "*All classes implementing type **A** delete method **B** except class **C**.*" LSDiff checks for consistent additions and deletions of entities, but helps little for entity updates (or changes). To handle entity updates, LASE infers a general context-aware edit script from two or more similarly changed methods, and exploits the inferred script to (1) search for other methods to change, and (2) suggest customized edits [7]. LASE can help apply similar edits to similar code; it does not help when co-changed entities have dissimilar content and require for distinct edits.

Our recent study on multi-entity edits reveals two frequently applied change patterns: ***CM→AF** and ***CM→AM** [10]. **AF** means *Added Field*; **AM** means *Added Method*; ***CM** represents one or more *Changed Methods*; and → denotes that one entity references or syntactically depends on another entity. These patterns show that when one field or method is added, developers usually change multiple methods together to access the field or invoke the method. As the co-changed methods usually contain different program contexts and experience divergent changes, developers may forget to change *all* relevant methods. This paper introduces a novel approach—CMSuggester—that suggests methods to co-change. 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** or ***CM→AM** edits. We found that the co-changed methods usually involve common fields or methods before an edit is applied. It indicates that **there are clusters of methods that access the same sets of fields or methods**. If one or more methods in a cluster are changed to access a new field or method, the other methods from the same cluster are likely to be co-changed for the new field access or method invocation.

Based on the preliminary study, we developed CMSuggester to recommend complementary changes for ***CM→AF** and ***CM→AM** 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 to be the *peer fields* of f_n, and locates any unchanged method accessing the peer fields to suggest changes. Similarly, given an added method (m_n) and one or more changed methods, CMSuggester extracts peer methods invoked by the changed methods. By identifying any unchanged method that also invokes the peer methods, CMSuggester recommends additional methods that need to be changed.

This paper makes the following contributions:

- We conducted an empirical study on ***CM→AF** and ***CM→AM** edits, and revealed that the co-changed methods for an added field or method usually access existing fields or methods in common. Our findings shed light on future research in automatic bug localization and program repair.

- We developed a novel approach CMSuggester that suggests complementary changes for ***CM→AF** and ***CM→AM** edits. Given an **AF** (or **AM**) and one or more **CM**s to access the field (or invoke the method), CMSuggester extracts peer fields (or methods) from those changed methods, and relies on the extracted information to predict other methods for change. Unlike existing tools, CMSuggester can recommend changes even if (1) there is no change history available and (2) the methods to co-change have totally different content and should go through divergent changes.

- We compared CMSuggester with three state-of-the-art tools: ROSE [4], TARMAQ [8], and Transitive Association Rules (TAR) [9]. We found that CMSuggester usually provided more suggestions with higher accuracy than all existing tools. Our results imply that CMSuggester complements these history-based mining tools when suggesting changes for ***CM→AF** and ***CM→AM** edits.

We envision CMSuggester to be integrated into Integrated Development Environments (IDE), code review systems, or version control systems. In this way, after developers make code changes, CMSuggester can help them detect and fix incorrectly applied multi-entity edits. Our programs and datasets are available at `https://data.mendeley.com/datasets/tmv2pp964r/3`.

This paper is an extended and revised version of our previous conference paper [11]. The main differences between this paper and our prior work are as follows:

- The original paper conducts a preliminary study for ***CM→AF** edits, while this paper includes an additional study for ***CM→AM** edits.
- In the original paper, CMSuggester only has the capability of suggesting changes for ***CM→AF** edits. For this paper, we extended the capability of CMSuggester such that it also suggests changes for ***CM→AM** edits.
- The original paper explores how sensitive CMSuggester is to different filter settings when dealing with ***CM→AF** edits, while this paper further investigates how sensitive CMSuggester is to filter settings when processing ***CM→AM** edits.
- In the original paper, our evaluation data set includes the software version history of four open-source projects. In this paper, the evaluation data set involves six open-source projects.
- The original paper only compares CMSuggester with ROSE, while this paper further compares CMSuggester with another two existing tools: TARMAQ, and TAR. To assess whether CMSuggester always works better than existing tools, we also conducted statistical testing based on the empirical measurements for individual change suggestion tasks.
- We expanded all sections to explain the additional work mentioned above. In the Related Work section, we added more discussion to comprehensively compare CMSuggester with existing work.

2. A Motivating Example

Developers may incompletely apply multi-entity edits. Figure 1 shows a simplified program revision to Derby [12]—a Java-based relational database.

```
1.  public class SQLChar extends DataType implements
2.      StringDataValue, StreamStorable {
3.    …
4. +  protected Clob _clobValue;
5.    public int getLength() throws StandardException {
6. +    if (_clobValue != null) {
7. +      return getClobLength(); }
8.      if (rawLength != 1)
9.        return rawLength;
10.      if (stream != null) {
11.      …
12.  }
13.  public void restoreToNull() {
14.    value = null;
15.    stream = null;
16.    rawLength = -1;
17.    cKey = null;
18.  }}
```

Figure 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 [14].

In this revision, developers added a field `_clobValue` (line 4) and modified 12 methods in different ways to access the field (e.g., changing `getLength()` at lines 6-7). However, developers forgot to also change `restoreToNull()` (lines 13-18). 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()` [13]. It can be challenging for developers to examine or ensure the completeness of such edits. This is because when developers forgot to change all methods for the new field access, there is often no compilation error triggered, neither can existing bug detectors reveal the problem.

We developed CMSuggester, a tool that identifies complementary changes and helps 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, so CMSuggester considers them to be *peers* of the new field. CMSuggester then searches for any unchanged method that also accesses the peer fields. In this way, CMSuggester finds `restoreToNull()`—which accesses the peer fields in the same "*pure write*" mode—and suggests the method for change. With CMSuggester, developers can identify the change locations that they may otherwise miss when applying multi-entity edits.

3. A Preliminary Characterization Study

In our prior study [10], we analyzed 2,854 bug fixes from 4 popular open source projects to explore multi-entity edits, including Aries [15], Cassandra [16], Derby [12], and Mahout [17]. Our study shows that recurring change patterns commonly exist in all the projects. In particular, ***CM→AF** and ***CM→AM** are two of the most frequently applied patterns. Therefore, in this paper, we randomly sampled five commits in each project for each pattern, and manually analyzed the characteristics of co-changed methods.

Table 1 presents our inspection results for ***CM→AF** edits. 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 share no commonality. *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) paradigm, since OO emphasizes to group related data in the same structure to ease modification and understanding [18].

Table 2 shows the inspection results of ***CM→AM** edits. For each added method, there are 2-41 methods co-changed to invoke the method. We found that **in 16 of the 20 examined revisions, the co-changed methods commonly invoke existing method(s) before the edits are applied**, whereas the other 4 commits have co-changed methods to commonly access certain field(s). *Our observation indicates that when one or more methods in a cluster are changed to invoke a new method, the other methods from the same cluster are likely to be co-changed for the new method invocation.*

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

Table 2: Commonality inspection of 20 ***CM→AM** multi-entity edits

Project	**Commits**	**Added Method**	**# of Changed Methods**	**Commonality**
Aries	32aa11b	unableToApply(...)	3	Method invocation
	fc8fba6	selectMatchingConverter(...)	2	Method invocation
	9586d78	getTransaction()	4	Method invocation
	628523f	safeEndCoordination(...)	2	Field access
	50ca3da	containsKey(...)	2	Method invocation
Cas-sandra	1c9c47d	nextIsRangeTombstone()	3	Method invocation
	9170ea2	excise()	2	Field access
	e863c2b	getRpcAddress(...)	5	Method invocation
	af9b768	pagingFinished(...)	2	Method invocation
	8dfd75d	atomicMoveWithFallback(...)	2	Method invocation
Derby	643861	isConnectedToMaster()	2	Method invocation
	586052	privInitialDirContext(...)	2	Method invocation
	583691	calculateSlotFieldSize(...)	2	Method invocation
	329295	requiresTypeFromContext()	41	Method invocation
	421717	getDriverModule()	6	Method invocation
Mahout	d141c8e	recommend(...)	2	Method invocation
	c1d2cd1	inverse()	2	Field access
	c0f3d94	parameters()	13	Method invocation
	0833411	sparseVectorToString()	2	Method invocation
	1e3f7ae	invalidateCachedLength()	9	Field access

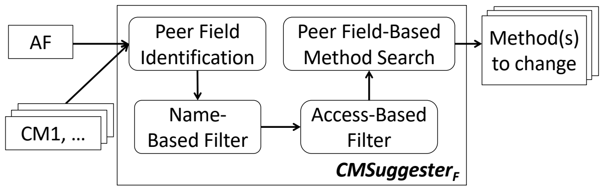

Figure 2: Overview of CMSuggester$_F$

4. Approach

Section 3 shows that *for some given methods, it is promising to suggest their co-changed methods based on the common field accesses/method calls.* Therefore, we developed CMSuggester to suggest method changes to complete ***CM→AF** and ***CM→AM** edits. Because we observed different characteristics for the two change patterns, the design of CMSuggester includes two parts: method change suggestion for ***CM→AF** edits (Section 4.1), and method change suggestion for ***CM→AM** edits (Section 4.2).

4.1. CMSuggester$_F$: Complementary Change Suggestion for ****CM→AF*** Edits

Figure 2 shows the overview of our approach. Given an edit that adds a field and changes one or more methods to access the field, CMSuggester$_F$ extracts peer fields from the changed method(s) (Section 4.1.1), filters the fields based on naming patterns and access modes (Sections 4.1.2 and 4.1.3), and searches for any unchanged method with the refined fields for change suggestion (Section 4.1.4).

4.1.1. Peer Field Identification

Given a new 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 new field is `_clobValue`. Thus, in method `getLength()`, CMSuggester$_F$ identifies `rawLength` and `stream` as peer fields. In our implementation, CMSuggester$_F$ traverses the Abstract Syntax Tree (AST) of each changed method's old version to locate all field accesses, creating a peer field set $PF = \{pf_1, pf_2, \ldots\}$.

4.1.2. Name-Based Filtering

We noticed that peer fields may have diverse power to indicate the usage of f_n. To ensure CMSuggester$_F$'s accuracy when suggesting methods for change, we refine the peer fields PF with two intuitive filters. The first filter uses 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 PF. Similarly, if f_n is a constant, we can use the constant peers to suggest f_n's usage, and remove variable peers from PF.

4.1.3. 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$_F$ scans the internal representation (IR) of each CM's old version created by WALA [19], 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$_F$ removes the field from PF.

4.1.4. Peer Field-Based Method Search

With the refined fields, CMSuggester$_F$ searches for methods to co-change by identifying any unchanged method that accesses at least two refined fields. In the search, CMSuggester$_F$ scans a large portion of code, because a program revision usually changes a small portion of code while keeping the majority of code unchanged [2]. To improve the search efficiency, we rely on the access

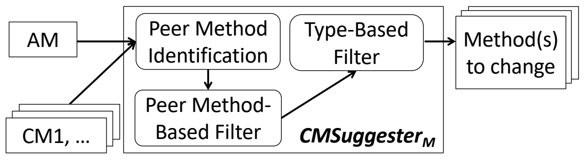

Figure 3: Overview of CMSuggester$_M$

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 invisible to 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, we cannot reduce the search space, so we scan all unchanged methods.

*4.2. CMSuggester$_M$: Complementary Change Suggestion for ***CM→AM** Edits*

With the observation that ***common method invocations indicate methods' co-change relations***, we designed CMSuggester$_M$ to recommend complementary changes for ***CM→AM** edits. As shown in Figure 3, given an edit that adds a method and changes one or more methods to invoke the method, CMSuggester$_M$ mines peer methods from the changed method(s) (Section 4.2.1), uses these peers to locate any unchanged method that should also be changed (Section 4.2.2), and refines the located methods via type checking (Section 4.2.3).

4.2.1. Peer Method Identification

Similar to peer field identification (Section 4.1.1), given a new method m_n, we use **peer methods** to refer to the existing methods that are (1) declared in the same class as m_n, and (2) invoked by one or more changed methods which also invoke m_n. CMSuggester$_M$ traverses the AST of each changed method's old version to extract the invoked existing methods, extracting a peer method set $PM = \{pm_1, pm_2, \ldots\}$.

4.2.2. Peer Method-Based Search

Similar to peer field-based method search (Section 4.1.4), with identified peer methods, CMSuggester$_M$ searches for any unchanged method that invokes at least one peer method. To improve efficiency, we also rely on the access modifiers of m_n to determine the search space. For instance, if m_n is private, the search scope is within the declaring class of m_n; if m_n is public, the search scope is the whole codebase. We denote the identified candidate method set with $MC = \{mc_1, mc_2, \ldots\}$.

4.2.3. Type-Based Filtering

Intuitively, *if a method should be changed to invoke m_n, this method is likely to contain the related calling context, i.e., properly typed variables that (1) pass values to m_n as parameters or (2) accept values returned by m_n*. With this intuition, we built a filter in CMSuggester$_M$ to improve the tool's change suggestion accuracy. Specifically, for any candidate method mc identified based on peer method invocation (see Section 4.2.2), CMSuggester$_M$ traverses the AST of mc to extract the list of used types L_{type}. Such type information is mined from the type binding of any field or local variable used in mc. Suppose that m_n has k parameters. If L_{type} has the return type of m_n (except `void`) and at least $(k-1)$ of those parameter types, mc is kept in MC; otherwise, mc is removed.

5. Evaluation

We conducted evaluations to explore the following four research questions:

- **RQ1:** *What is CMSuggester$_F$'s effectiveness to suggest complementary changes for* ****CM→AF** *edits, and how does it compare with prior tools?* We constructed an evaluation data set from ***CM→AF** edits (Table 3) and applied CMSuggester$_F$, ROSE, TARMAQ, and TAR to the suggestion tasks. Our results in Section 5.2 show that CMSuggester$_F$ achieved the highest coverage and accuracy. Our observations indicate that CMSuggester complements existing tools to recommend co-changes based on the syntactic or semantic relations between methods other than the history.

- **RQ2:** *What is the effectiveness comparison between CMSuggester$_M$ and prior tools when suggesting co-changes for* ****CM→AM** *edits*? We leveraged ***CM→AM** edits to create another evaluation data set (Table 4), and compared the change suggestions by CMSuggester$_M$, ROSE, TARMAQ, and TAR. The results in Section 5.3 show that CMSuggester$_M$

outperformed all three prior tools, especially in terms of coverage and precision.

- **RQ3:** *How does CMSuggester$_F$'s effectiveness vary with the two used filters?* By disabling one or both filters defined in CMSuggester$_F$, we built three variant approaches (Section 5.4). From the comparison between the variants and CMSuggester$_F$, we found that both filters improved the accuracy while sacrificing coverage, and the name-based filter obtained a better trade-off between accuracy and coverage than the other filter.

- **RQ4:** *How sensitive is CMSuggester$_M$'s effectiveness to the usage of its single filter?* We disabled the filter and created a variant approach (Section 5.5). Without the filter, CMSuggester$_M$ achieved higher coverage (95% vs. 82%) but lower accuracy (67% vs. 70%).

5.1. Setup

In this section, we introduce the data set (Section 5.1.1), our compared tools (Section 5.1.2), and our metrics (Section 5.1.3).

5.1.1. Data Set

In our study, we leveraged the multi-entity edits of six open-source projects:

- *Aries [15]* contains a set of pluggable Java components, which enable an enterprise OSGi application programming model.

- *Cassandra [16]* is a NoSQL database management system. It is designed to handle large amounts of data across many commodity servers, providing high availability with no single point of failure.

- *Derby [12]* is a relational database management system that can be embedded in Java programs and used for online transaction processing.

- *Mahout [17]* is a project to produce free implementations of distributed or otherwise scalable machine learning algorithms focused primarily in the areas of collective filtering, clustering, and classification.

- *ActiveMQ [20]* is a message broker written in Java together with a full Java Message Service (JMS) client. It provides "Enterprise Features" which foster the communication from more than one client or server.
- *UIMA [21]* is an unstructured information management application. The software system analyzes large volumes of unstructured information to discover knowledge that is relevant to an end user.

These Java projects are from the Apache software foundation, and built for different application domains. All projects have well-maintained issue tracking systems and version control systems. Many commit messages in these software repositories contain the corresponding issue IDs. In this paper, we mainly focus on the program commits that fix bugs. Therefore, given an issue labeled as "Bug Fix", we leveraged the issue ID to locate the corresponding program commit. Apart from issue IDs, we also collected bug-fixing commits based on the keywords like "bug" and "fix" in commit messages. This is because some applied bug fixes are not explicitly related to issues via the issue IDs.

Based on the collected data, we created two data sets to separately evaluate CMSuggester$_F$ and CMSuggester$_M$. To create the data set for CMSuggester$_F$, we searched for any ***CM→AF** edit that has (1) at least two methods co-changed for an added field, and (2) each changed method accessing at least two curren fields. In this way, we found 10 commits, 45 commits, 42 commits, 9 commits, 55 commits, and 14 commits separately in the revision data of Aries, Cassandra, Derby, Mahout, ActiveMQ, and UIMA. Each commit contains one or more ***CM→AF** edits. Similarly, to build the data set for CMSuggester$_M$, we searched for any ***CM→AM** edit that has (1) at least two methods co-changed for an added method, and (2) each changed method accessing at least one existing method declared in the same class of AM. We found 2 commits, 41 commits, 49 commits, 4 commits, 62 commits, and 26 commits from the 6 projects. Each commit has at least one ***CM→AM** edit.

For each AF (or AM), we constructed suggestion tasks by (i) providing the AF (or AM) and some of its co-applied CMs to CMSuggester as input, and

Table 3: Evaluation data set for CMSuggester$_F$

	Aries	**Cassandra**	**Derby**	**Mahout**	**ActiveMQ**	**UIMA**	**Total #**
# of program commits	10	45	42	9	55	14	175
# of 1AF1C suggestion tasks	39	172	151	46	197	80	685
# of 1AF2C suggestion tasks	9	237	168	12	181	63	670
# of 1AF3C suggestion tasks	4	379	366	8	252	32	1041

Table 4: Evaluation data set for CMSuggester$_M$

	Aries	**Cassandra**	**Derby**	**Mahout**	**ActiveMQ**	**UIMA**	**Total #**
# of program commits	2	41	49	4	62	26	184
# of 1AM1C suggestion tasks	7	141	237	27	205	75	692
# of 1AM2C suggestion tasks	3	165	1634	93	175	52	2122
# of 1AM3C suggestion tasks	0	204	17295	306	332	30	18167

(ii) using the remaining part as the oracle to evaluate CMSuggester's output. For instance, suppose that a commit has an added field f_n and two changed methods $M = \{m_1, m_2\}$. In one task, we provide f_n and m_1 as input, and check whether CMSuggester$_F$ suggests m_2 for change. Alternatively, we can provide f_n and m_2 as input, and check whether CMSuggester$_F$'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 (1AF1C)*** 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 (1AF2C)*** and ***one-AF-three-CM (1AF3C)*** tasks. As the majority of AFs (or AMs) in our data sets correspond to 2-4 CMs, our experiments focus on 1AF1C, 1AF2C, 1AF3C, 1AM1C, 1AM2C, and 1AM3C tasks, as shown in Table 3 and Table 4.

5.1.2. Compared Tools

In our evaluation, we compared CMSuggester with the three state-of-the-art co-change suggestion tools: ROSE [4], TARMAQ [8], and TAR [9]. We chose these tools because (1) ROSE has been popularly used and (2) TARMAQ and TAR were recently introduced. Although the three tools do not conduct so complicated analysis as CMSuggester, they all mine change patterns from revision histories, and we can align their inputs for the evaluation.

Specifically, ROSE mines the association rules between co-changed entities from software version histories, as shown below:

$$\begin{Bmatrix} \{(\_Qdmodule.c, func, GrafObj\_getattr())\} \Rightarrow \\ (qdsupport.py, func, outputGetattrHook()). \end{Bmatrix} \tag{1}$$

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 $support = 1$ and $confidence = 0.1$, because the ROSE paper [4] mentioned this setting multiple times.

Similar to ROSE, TARMAQ also mines association rules in software version history. However, given a query Q (i.e., a set of known *changed* entities), instead of using Q as is to suggest co-changes, TARMAQ first locates one or more program commits T that have the largest number of overlapping changed entities with Q. TARMAQ then treats the overlapping entities in each commit as a refined query Q' to suggest any co-change. Note that for 1AF1C and 1AM1C tasks, since there is only one known changed method (together with an added field or method), $Q' = Q = 1$ and TARMAQ performed identically to ROSE.

TAR is also similar to ROSE by suggesting co-changes based on software version history. However, different from ROSE, with the mined rules $E1 \Rightarrow E2$ and $E2 \Rightarrow E3$, TAR leverages transitive inference to further derive $E1 \Rightarrow E3$. Suppose that the confidence values of $E1 \Rightarrow E2$ and $E2 \Rightarrow E3$ are separately $c1$ and $c2$, then the confidence value of $E1 \Rightarrow E3$ is $c1 \times c2$. Same as ROSE, TARMAQ and TAR are also configured with $support = 1$ and $confidence = 0.1$.

To assess the capability of suggesting complementary changes, we used all the tools to complete the tasks mentioned in Tables 3 and 4.

5.1.3. 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 can 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.

$$C = \frac{\text{\# of tasks with a tool's suggestion}}{\text{Total \# of tasks}} \times 100\% \tag{2}$$

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:

$$P = \frac{\text{\# of correct suggestions}}{\text{Total \# of suggestions by a tool}} \times 100\% \tag{3}$$

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:

$$R = \frac{\text{\# of correct suggestions by a tool}}{\text{Total \# of expected suggestions}} \times 100\% \tag{4}$$

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:

$$F = \frac{2 \times P \times R}{P + R} \times 100\% \tag{5}$$

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

Table 5: CMSuggester$_F$ vs. existing tools for 1AF1C tasks (%)

Project	**CMSuggester$_F$**				**ROSE**				**TARMAQ**				**TAR**			
	C	**P**	**R**	**F**	**C**	**P**	**R**	**F**	**C**	**P**	**R**	**F**	**C**	**P**	**R**	**F**
Aries	51	68	85	76	31	35	39	37	31	35	39	37	31	27	56	36
Cassandra	69	81	75	78	38	53	71	61	38	53	71	61	38	50	74	60
Derby	71	71	68	69	22	25	42	31	22	25	42	31	24	23	47	36
Mahout	72	72	68	70	13	5	33	9	13	5	33	9	14	6	34	10
ActiveMQ	63	64	61	63	58	33	63	43	58	33	63	43	59	24	68	36
UIMA	77	73	60	64	18	22	58	32	18	22	58	32	18	22	58	32
WA	**68**	**72**	**68**	**70**	**42**	**37**	**61**	**47**	**42**	**37**	**61**	**47**	**45**	**31**	**66**	**43**

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

$$\Gamma_{overall} = \frac{\sum_{i=1}^{6} \Gamma_i \times n_i}{\sum_{i=1}^{6} n_i}. \tag{6}$$

In the formula, i varies from 1 to 6, representing Aries, Cassandra, Derby, Mahout, ActiveMQ, and UIMA in sequence. In particular, 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 $\Gamma_{overall}$.

5.2. RQ1. The Comparison between CMSuggester$_F$ and Prior Tools

Table 5 shows the results of CMSuggester$_F$, ROSE, TARMAQ, and TAR, for 1AF1C tasks. Overall, CMSuggester$_F$ obtained the highest coverage, precision, and accuracy values for all projects; it obtained the highest weighted average values in terms of all metrics. Although TAR derived the second highest weighted average value for coverage (i.e., 45%), its accuracy is the lowest (i.e., 43%) among all tools. Particularly for Mahout, CMSuggester$_F$ predicted changes for 72% of the tasks, while the other three tools provided predictions for 13%-14% of the tasks. Among the generated suggestions for Mahout, CMSuggester$_F$ achieved 72% precision, 68% recall, and 70% F-score; ROSE and TARMAQ obtained 5% precision, 33% recall, and 9% F-score; while TAR acquired 6% precision, 34% recall and 10% recall. For ActiveMQ, CMSuggester$_F$ acquired the lowest recall rate (61%), while the highest recall rate was acquired by TAR (68%).

To further explore whether CMSuggester$_F$ worked significantly better than other tools, we performed Mann-Whitney U test [22] and measured the Cliff's delta size [23]. The U test was applied to check whether two sample groups (e.g., precision rates for individual tasks reported by CMSuggester$_F$ and ROSE) have the same distribution. If the mean values of two groups are different and $p < 0.05$, we consider the two groups to have significantly different distributions; in such cases, the Cliff's delta size measures the magnitude of differences.

Table 6: Statistical significance tests for 1AF1C tasks

	CMSuggester$_F$ vs. ROSE			**CMSuggester$_F$ vs. TARMAQ**			**CMSuggester$_F$ vs. TAR**		
	P	**R**	**F**	**P**	**R**	**F**	**P**	**R**	**F**
Mean comparison	72% vs. 37%	68% vs. 61%	70% vs. 47%	72% vs. 37%	68% vs. 61%	70% vs. 47%	72% vs. 31%	68% vs. 66%	70% vs. 43%
p-value	< 2.2e-16	0.05	< 2.2e-16	< 2.2e-16	0.05	< 2.2e-16	< 2.2e-16	0.47	< 2.2e-16
Cliff's Δ	0.46 (medium)	-	0.40 (medium)	0.46 (medium)	-	0.40 (medium)	0.52 (large)	-	0.47 (medium)

As with prior work [24], we interpreted the computed Cliff's delta value v in the following way: (1) if $v < 0.147$, the effect size is "negligible"; (2) if $0.147 \leq v < 0.33$, the effect size is "small'; (3) if $0.33 \leq v < 0.474$, the effect size is "medium"; (4) otherwise, the effect size is "large".

Table 6 presents our statistical testing results. Note that we performed such testing for **P**, **R**, and **F**, but not for **C**. This is because the coverage formula is an accumulative function, producing a single number for a given group of tasks; while the other metrics are per-task formulae, reporting separate values for individual tasks. Therefore, with a group of values separately calculated for **P**, **R**, and **F**, we could perform statistical testing. According to the table, CMSuggester$_F$ obtained significantly higher precision and accuracy rates than other tools, with medium or large effect sizes. Although CMSuggester$_F$'s mean recall is higher than that of other tools, the difference is not significant. Overall, CMSuggester$_F$ significantly outperformed existing tools by predicting co-changes with higher accuracy.

Two major reasons can explain why CMSuggester$_F$ worked better. First, the three existing tools we evaluated all leverage the co-changed entities in version history to predict likely changes. When the history data is incomplete or some entities were never co-changed before, existing tools lack the evidence to predict some co-changes, obtaining lower coverage and recall rates in general. Second, the three experimented tools exploit no syntactic or semantic relationship between the co-changed entities. They can infer incorrect rules from co-changed but unrelated entities, deriving lower precision.

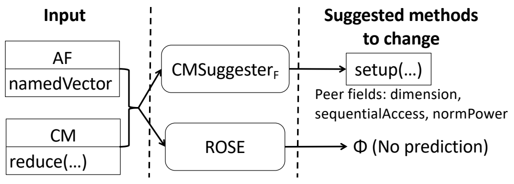
Input
Suggested methods to change
AF
namedVector
CM
reduce(...)
CMSuggester_F
ROSE
setup(...)
Peer fields: dimension, sequentialAccess, normPower
Φ (No prediction)

Figure 4: A task for which CMSuggester$_F$ outperformed ROSE

Figure 4 presents a task for which CMSuggester$_F$ outperformed ROSE. This task is extracted from the commit 22d7d31 [25] of Mahout. In the task, there is one AF `PartialVectorMergeReducer.namedVector` and one CM `PartialVectorMerge-Reducer.reduce(...)` provided as input, and another CM provided as the expected output. CMSuggester$_F$ successfully predicted `PartialVectorMergeReducer.setup(...)` based on 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(...)`.

Figure 5 shows a task for which ROSE worked better than CMSuggester$_F$. This task is from the commit f06e1d6 [26] of Cassandra. It provides one AF `Session.compactionStrategy` and one CM `Session.Session(...)` as input, and includes another CM as the oracle. CMSuggester$_F$ 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$_F$ can complement ROSE by suggesting co-changes in a different way.

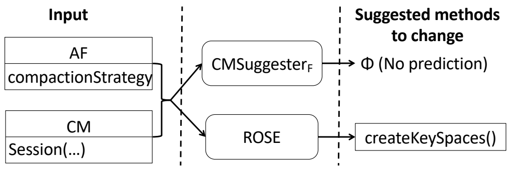

Figure 5: A task for which ROSE outperformed CMSuggester$_F$

Table 7: CMSuggester$_F$ vs. existing tools for 1AF2C tasks (%)

Project	**CMSuggester$_F$**				**ROSE**				**TARMAQ**				**TAR**			
	C	**P**	**R**	**F**	**C**	**P**	**R**	**F**	**C**	**P**	**R**	**F**	**C**	**P**	**R**	**F**
Aries	89	35	50	41	0	-	-	-	56	42	40	41	67	17	45	25
Cassandra	76	65	66	65	31	63	69	66	51	59	72	65	51	46	74	57
Derby	96	65	55	60	3	7	15	10	8	50	69	58	8	48	69	58
Mahout	100	35	39	37	0	-	-	-	25	0	0	-	25	0	0	-
ActiveMQ	86	49	63	55	24	42	49	45	79	39	50	41	82	22	52	31
UIMA	99	43	58	49	5	34	50	40	20	21	38	27	20	18	40	25
WA	**87**	**58**	**61**	**60**	**14**	**53**	**60**	**57**	**62**	**47**	**57**	**52**	**62**	**33**	**61**	**43**

Finding 1: *CMSuggester$_F$ significantly outperformed ROSE, TARMAQ, and TAR for 1AF1C tasks. This means that CMSuggester$_F$ complements these history-based tools by inferring co-changes from methods' common field accesses instead of from the history.*

In addition to 1AF1C tasks, we also compared CMSuggester$_F$ with the three tools for 1AF2C and 1AF3C tasks (see Tables 7 and 8), and observed similar phenomena in both tables. In particular, for 1AF2C tasks, CMSuggester$_F$ obtained the highest weighted average of F-score (60%); while ROSE, TARMAQ, and TAR separately obtained 57%, 52%, and 43%. This comparison implies that CMSuggester$_F$ achieved the best trade-off between precision and recall. More importantly, CMSuggester$_F$ acquired much higher coverage rates than other tools. In Table 7, for Aries and Mahout, CMSuggester$_F$'s coverage values are 89% and 100%, while the values by ROSE is 0%. Among the three history-based tools, ROSE acquired the lowest weighted average of coverage (14%), but highest weighted average of accuracy (57%). It indicates that ROSE is less applicable than TARMAQ and TAR, but manages to predict changes more accurately. One possible reason to explain this phenomenon is that both TARMAQ and TAR are based on ROSE, attempting to infer more rules from history and thus widen the application scope; nevertheless, such expansion of rule inference can also compromise the quality of change suggestion.

Table 8: CMSuggester$_F$ vs. existing tools for 1AF3C tasks (%)

Project	**CMSuggester$_F$**				**ROSE**				**TARMAQ**				**TAR**			
	C	**P**	**R**	**F**	**C**	**P**	**R**	**F**	**C**	**P**	**R**	**F**	**C**	**P**	**R**	**F**
Aries	100	12	25	16	0	-	-	-	50	30	100	46	100	9	50	16
Cassandra	75	56	62	59	33	57	64	60	59	45	81	58	60	41	82	55
Derby	100	66	61	63	0	0	0	-	3	13	56	22	3	7	80	13
Mahout	100	21	25	23	0	-	-	-	0	-	-	-	0	-	-	-
ActiveMQ	86	46	68	55	9	38	58	46	88	36	40	38	95	17	56	26
UIMA	100	37	64	47	1	0	0	-	32	10	30	15	32	6	30	10
WA	**88**	**57**	**63**	**60**	**16**	**54**	**62**	**58**	**71**	**40**	**60**	**48**	**77**	**28**	**69**	**40**

We made similar observations in Table 8. For 1AF3C tasks, CMSuggester$_F$ outperformed existing tools by covering more tasks and acquiring higher accuracy. Furthermore, by comparing the coverage among Tables 5, 7, and 8, we found that (1) ROSE always covered fewer tasks than the other three tools, and (2) when more CMs are provided, the gap between ROSE's coverage and that of other tools becomes larger. This finding can be explained with the internal mechanisms of different tools. Suppose that given an 1AF2C task, each tool can separately predict changes $M_1 = \{m_{1a}, m_{1b}, \ldots\}$ based on one changed method $CM1$, and predict changes $M_2 = \{m_{2a}, m_{2b}, \ldots\}$ based on the other changed method $CM2$. To improve the prediction precision, ROSE intersects the prediction sets of individual CMs and suggests $M_r = M_1 \bigcap M_2$ for co-changes. In comparison, CMSuggester$_F$, TARMAQ, and TAR predict changes based on the set union, i.e., $M = M_1 \bigcup M_2$. Consequently, ROSE is more conservative when predicting changes given multiple changed methods, while other tools are more likely to suggest changes in the same scenarios.

Finding 2: *For 1AF2C and 1AF3C tasks, when multiple CMs were provided as inputs, CMSuggester$_F$ outperformed existing tools by achieving better coverage and accuracy.*

5.3. RQ2. The Comparison between CMSuggester$_M$ and Prior Tools

The above evaluation shows that CMSuggester$_F$ outperformed ROSE, TARMAQ, and TAR, when suggesting complementary changes for ***CM→AF** edits.

Table 9: CMSuggester$_M$ vs. existing tools for 1AM1C tasks (%)

Project	**CMSuggester$_M$**				**ROSE**				**TARMAQ**				**TAR**			
	C	**P**	**R**	**F**	**C**	**P**	**R**	**F**	**C**	**P**	**R**	**F**	**C**	**P**	**R**	**F**
Aries	71	100	100	100	0	-	-	-	0	-	-	-	0	-	-	-
Cassandra	82	69	67	68	37	27	57	37	37	27	57	37	38	25	61	36
Derby	87	74	67	70	22	27	58	37	22	27	58	37	22	25	61	36
Mahout	85	88	91	90	0	-	-	-	0	-	-	-	0	-	-	-
ActiveMQ	75	66	68	67	44	17	52	26	44	17	52	26	44	15	57	24
UIMA	76	68	64	66	24	22	55	32	24	22	55	32	24	22	55	32
WA	**82**	**71**	**69**	**70**	**36**	**23**	**55**	**32**	**36**	**23**	**55**	**32**	**36**	**21**	**60**	**32**

We were also curious how CMSuggester$_M$ compares with these tools when recommending changes for ***CM→AM** edits. Thus, we also applied CMSuggester$_M$ and the three tools to the data set shown in Table 4.

Table 9 presents the experimental results of CMSuggester$_M$, ROSE, TARMAQ, and TAR, for 1AM1C tasks. Similar to what we observed in Section 5.2, CMSuggester$_M$ outperformed all the three tools in terms of all metrics. Specifically, CMSuggester$_M$ suggested changes for 82% of the tasks, while the other tools provided suggestions for 36% of those tasks. Among the provided suggestions, CMSuggester$_M$ achieved 71% precision, 69% recall, and 70% accuracy; ROSE and TARMAQ acquired 23% precision, 55% recall, and 32% accuracy; while TAR achieved 21% precision, 60% recall, and 32% accuracy.

Table 10: Statistical significance tests for 1AM1C tasks

	CMSuggester$_M$ vs. ROSE			**CMSuggester$_M$ vs. TARMAQ**			**CMSuggester$_M$ vs. TAR**		
	P	**R**	**F**	**P**	**R**	**F**	**P**	**R**	**F**
Mean comparison	71% vs. 23%	69% vs. 55%	70% vs. 32%	71% vs. 23%	69% vs. 55%	70% vs. 32%	71% vs. 21%	69% vs. 60%	70% vs. 32%
p-value	< 2.2e-16	0.004	< 2.2e-16	< 2.2e-16	0.004	< 2.2e-16	< 2.2e-16	0.016	< 2.2e-16
Cliff's Δ	0.58 (large)	0.15 (small)	0.50 (large)	0.58 (large)	0.15 (small)	0.50 (large)	0.59 (large)	-	0.51 (large)

We further conducted statistical testing to decide whether CMSuggester$_M$ worked significantly better than the three existing tools for 1AM1C tasks. As shown in Table 10, the p-values of **P** and **F** are less than $2.2e-16$, while the corresponding Cliff's delta values are at least 0.5. This means that CMSuggester$_M$ outperformed other tools by obtaining significantly higher precision and ac-

Table 11: CMSuggester$_M$ vs. existing tools for 1AM2C tasks (%)

Project	**CMSuggester$_M$**				**ROSE**				**TARMAQ**				**TAR**			
	C	**P**	**R**	**F**	**C**	**P**	**R**	**F**	**C**	**P**	**R**	**F**	**C**	**P**	**R**	**F**
Aries	100	33	100	50	0	-	-	-	0	-	-	-	0	-	-	-
Cassandra	94	58	85	69	26	38	88	53	36	33	80	47	35	26	80	39
Derby	99	84	77	80	3	36	80	50	11	21	34	26	6	41	52	46
Mahout	100	68	95	79	0	-	-	-	0	-	-	-	0	-	-	-
ActiveMQ	90	63	66	64	6	15	30	20	38	9	17	12	38	13	32	19
UIMA	83	43	59	50	12	2	17	4	45	12	48	18	45	12	48	18
WA	**98**	**81**	**77**	**79**	**14**	**34**	**75**	**47**	**24**	**21**	**40**	**28**	**26**	**27**	**53**	**35**

Table 12: CMSuggester$_M$ vs. existing tools for 1AM3C tasks (%)

Project	**CMSuggester$_M$**				**ROSE**				**TARMAQ**				**TAR**			
	C	**P**	**R**	**F**	**C**	**P**	**R**	**F**	**C**	**P**	**R**	**F**	**C**	**P**	**R**	**F**
Aries	0	-	-	-	0	-	-	-	0	-	-	-	0	-	-	-
Cassandra	100	57	94	71	18	32	89	47	24	32	92	48	24	42	92	58
Derby	100	84	86	85	0	40	83	54	2	24	49	32	2	25	53	34
Mahout	100	62	98	76	0	-	-	-	0	-	-	-	0	-	-	-
ActiveMQ	94	64	72	68	0	-	-	-	29	6	10	7	29	15	21	17
UIMA	87	24	54	33	0	-	-	-	57	3	36	5	57	7	59	13
WA	**99**	**82**	**85**	**84**	**8**	**37**	**85**	**51**	**24**	**20**	**45**	**28**	**24**	**24**	**51**	**33**

curacy. Additionally, CMSuggester$_M$ achieved significantly higher recall than ROSE and TARMAQ, with small Cliff's delta sizes; however, its recall is not significantly better than that of TAR.

In addition to 1AM1C tasks, we also compared CMSuggester$_M$ with the three tools for 1AM2C and 1AM3C tasks (see Tables 11 and 12). Similar to what we observed in Table 9, CMSuggester$_M$ obtained the highest values in terms of all metrics. Among the three existing tools, TARMAQ and TAR achieved higher coverage than ROSE, at the cost of sacrificing accuracy. According to the three tables (Table 9, 11, and 12), we observed that *as the number of provided CMs increases, the gap between ROSE's coverage and that of other tools increases.* For instance, given 1AM1C tasks, CMSuggester$_M$ obtained 82% coverage, while ROSE and the other tools obtained 36% coverage. Nevertheless, given 1AM3C tasks, ROSE's coverage became 8%, CMSuggester$_M$'s coverage was 99%, while TARMAQ and TAR achieved 24%. Such divergent trends are due to the tools' differences when handling tasks with multiple CMs provided. As mentioned in Section 5.2, if multiple CMs are provided, ROSE intersects the methods predicted based on individual CMs, while the other tools take the union of those predictions.

Finding 3: *Compared with ROSE, TARMAQ, and TAR, CMSuggester$_M$ obtained higher coverage and better accuracy, demonstrating better effectiveness when suggesting changes to complete* ****CM→AM*** *edits.*

5.4. RQ3. CMSuggester$_F$'s Sensitivity to Filter Setting

In the design of CMSuggester$_F$, there are two filters defined: name-based filter and access-based filter. To understand how each filter affects CMSuggester$_F$'s effectiveness, we built three variant approaches:

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

Table 13: CMSuggester$_F$ vs. its three variant approaches with filters enabled or disabled (%)

Project	**CMSuggester$_F$**				**VF$_o$**				**VF$_n$**				**VF$_a$**			
	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
ActiveMQ	63	64	61	63	97	64	59	61	87	63	58	60	76	63	61	62
UIMA	77	73	60	64	97	73	53	62	90	75	56	64	79	70	54	61
WA	**68**	**72**	**68**	**70**	**94**	**68**	**64**	**66**	**86**	**71**	**66**	**68**	**76**	**69**	**66**	**67**

Table 13 presents the effectiveness comparison between CMSuggester$_F$ and the variants. According to this table, CMSuggester$_F$ obtained the lowest overall coverage (68%), but the highest overall precision (72%), recall (68%), and F-score (70%). This is as expected, because CMSuggester$_F$ 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. VF$_o$ achieved the highest coverage (94%) but lowest F-score (66%). Since it did not refine peer fields before predicting changes, some of the included peer fields are used less similarly to the newly added fields, causing incorrect suggestions.

Compared with VF$_a$, VF$_n$ obtained better coverage (86% vs. 76%), better precision (71% vs. 69%), equal recall (both 66%), and better F-score (68% vs. 67%). This is out of our expectation. Although the name-based filter seems more intuitive and is easier to implement than the access-based filter, it obtained a better trade-off among coverage, precision, recall, and accuracy. 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. In many cases, 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 4: *Both filters effected to improve CMSuggester$_F$'s accuracy at the cost of coverage. Especially, the name-based filter achieved a better trade-off between accuracy and coverage than the access-based filter.*

5.5. RQ4. CMSuggester$_M$'s Sensitivity to Filter Setting

In our design of CMSuggester$_M$, there is a type-based filter to refine candidate methods via type checking. We were curious how sensitive CMSuggester$_M$ is to this filter, so we created a variant approach—VM$_o$—by disabling the filter in CMSuggester$_M$. We also applied VM$_o$ to the 1AM1C tasks. Table 14 shows the effectiveness comparison between CMSuggester$_M$ and VM$_o$. Compared with CMSuggester$_M$, VM$_o$ obtained higher coverage (i.e., 95% vs. 82%), lower precision (i.e., 66% vs. 71%), lower recall (i.e., 68% vs. 69%), and lower F-score (i.e., 67% vs. 70%). This is understandable because without type checking,

Table 14: CMSuggester$_M$ vs. VM$_o$ for 1AM1C tasks (%)

Project	**CMSuggester$_M$**				**VM$_o$**			
	C	**P**	**R**	**F**	**C**	**P**	**R**	**F**
Aries	71	100	100	100	100	100	100	100
Cassandra	82	69	67	68	94	63	63	63
Derby	87	74	67	70	95	70	66	68
Mahout	85	88	91	90	96	71	97	82
ActiveMQ	75	66	68	67	93	59	68	63
UIMA	76	68	64	66	96	66	70	68
WA	**82**	**71**	**69**	**70**	**95**	**66**	**68**	**67**

CMSuggester$_M$ suggests changes purely based on the invocation of peer methods, even if some candidate methods do not contain the necessary program context (e.g., variables with matching types) for invoking any new method.

By default, we set CMSuggester$_M$ to include the type-based filter even if this filter can reduce the tool's coverage and compromise its applicability. The reason is that we want to achieve higher accuracy of CMSuggester$_M$'s predictions. Users of CMSuggester$_M$ can always disable this filter as they like.

Finding 5: *The type-based filter in CMSuggester$_M$ worked to improve F-score accuracy while reducing the coverage. VM$_o$ obtained 95% coverage and 67% accuracy for 1AM1C tasks.*

6. Threats to Validity

(a) Threats to External Validity: Our evaluation shows that CMSuggester outperformed existing tools when recommending co-changes for both ***CM→AF** and ***CM→AM** edits. This experimental conclusion may not hold when we apply these tools in other scenarios, where edits do not add any field or method. To overcome this limitation, we will revisit the frequently applied change patterns revealed by our prior work [10], and extend CMSuggester to recommend changes even though no new entity is inserted. Additionally, our experiment results can be different if all tools were applied to another set of open source projects or to a set of closed source software. In the future, we will evaluate these tools on program data from more software repositories. We also plan to develop a hybrid approach of CMSuggester, ROSE, TARMAQ, and

TAR. By relating methods based on common field accesses, common method invocations, and historical co-change relationship, the hybrid approach is guaranteed to suggest changes when either tool predicts something, and may provide more precise suggestions if tools' outputs can cross-validate each other.

(b) Threats to Construct Validity: When we prepared the golden standards, we constructed suggestion tasks from manual fixes. Yin *et al.* [27] show that a bug fix may be partially 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 existing tools. We share this limitation with prior work [4, 8, 9, 7, 28]. 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 tools, we can further evaluate the usefulness of different tools' suggestions.

7. Related Work

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

Co-Change Mining. Tools were built to mine version histories for co-change patterns [29, 30, 31, 4, 5, 8, 9, 32, 33, 34, 35, 36, 37]. Specifically, Gall et al. mined release data for the co-change relationship between subsystems [29] and classes [30]. Shirabad et al. trained a machine-learning model to predict whether two given files should be changed together [31]. Several other research groups developed tools (e.g., ROSE) to mine the association rules between co-changed entities and suggest possible changes accordingly [4, 5, 8, 9, 32, 36, 37]. Recently, some hybrid approaches are built with information retrieval (IR)-based techniques and association rule mining [33, 34, 35]. Specifically given a software entity E, these approaches leverage IR-based techniques to (1) extract terms from E and any other entity and (2) rank those entities based on their term overlapping with E. Meanwhile, these tools also apply association rule mining to commit history to rank entities based on the co-change frequency.

Given a new commit, these tools combine the two ranked lists in various ways to reveal any missing change. However, none of the approaches mentioned above 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 [38]. They found that the historical 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 or invoked methods, and complemented above-mentioned approaches when the revision history is limited or unavailable. Yamauchi et al. and Kreutzer et al. separately clustered similar code changes based on either string similarity or common usage of identifier names [39, 40]. In particular, Yamauchi et al. relied on the commonly used identifiers to summarize semantics of program changes [39]. CMSuggester does not cluster similar changes, neither does it summarize program semantics. However, CMSuggester (1) relies on the def-use relationship between program entities to infer the syntactic relevance and (2) leverages the commonly accessed fields or methods to infer the semantic relevance.

Change Recommendation Systems. Researchers built tools to recommend various code changes [41, 6, 42, 7, 43]. 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 [41]. Clever is a tool tracking all clone groups in software and monitoring for edits on clones [42]. 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 or method invocations between methods. In Model-Driven Engineering, ReVision repairs incorrectly updated models by (1) extracting change patterns from version history, and (2) matching the incorrect updates against those patterns to suggest repair operations [43]. CMSuggester shares similar methodology with ReVision, but focuses on code changes instead of model changes.

Automatic Program Repair (APR). There are tools proposed to generate candidate patches for certain bugs, and automatically check patch correctness using compilation and testing [44, 45, 46, 47, 48]. For example, GenProg [44] generates candidate patches by replicating, mutating, or deleting code randomly from the existing programs. Genesis trains a machine-learning model by extracting features from existing bug fixes, and suggesting candidate patches accordingly [47]. 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 for change in large codebases, and such places need to be located before APR tools can generate changes.

8. Conclusion

It is challenging for developers to completely apply multi-entity edits, because some missing changes may not trigger any compilation error or fail any test case. Particularly for ***CM→AF** and ***CM→AM** edits, after adding a field or method, developers may forget to change all related methods to access the added entity. In this paper, we introduced CMSuggester, an approach to recommend complementary changes for multi-entity edits. Compared with prior work that recognizes missing changes based on the historical co-change relationship between entities or program content similarity, CMSuggester recommends complementary method changes if any unchanged method shares common field accesses or method invocations with the already-changed method(s).

There are two parts of CMSuggester: (1) CMSuggester$_F$ that helps with ***CM→AF** edits and (2) CMSuggester$_M$ which facilitates ***CM→AM** edits. We conducted a comprehensive evaluation for both parts by (i) applying them to different sets of suggestion tasks, (ii) comparing them with ROSE, TAR-MAQ, and TAR, and (iii) varying the filtering configurations. Our evaluation shows that both CMSuggester$_F$ and CMSuggester$_M$ outperformed the three existing tools, providing better suggestions in more scenarios. All filters used in CMSuggester effectively helped improve the accuracy of change suggestion. In the future, we plan to investigate ways to integrate CMSuggester with existing tools, so that more high-quality code change suggestions can be provided to complete more multi-entity edits.

9. Acknowledgements

This work was supported by NSF-1845446, National Key R&D Program of China under Grant No. 2018YFB1004800, and National Natural Science Foundation of China under Grant No. 61572313.

References

[1] S. Christa, V. Madhusudhan, V. Suma, J. J. Rao, Software maintenance: From the perspective of effort and cost requirement, in: Proc. ICDECT, 2017, pp. 759–768.

[2] H. Zhong, Z. Su, An empirical study on real bug fixes, in: Proc. ICSE, 2015, pp. 913–923.

[3] J. Park, M. Kim, B. Ray, D.-H. Bae, An empirical study of supplementary bug fixes, in: Proc. MSR, 2012, pp. 40–49.

[4] T. Zimmermann, P. Weisgerber, S. Diehl, A. Zeller, Mining version histories to guide software changes, in: Proc. ICSE, 2004, pp. 563–572.

[5] A. T. T. Ying, G. C. Murphy, R. T. Ng, M. Chu-Carroll, Predicting source code changes by mining change history., IEEE Trans. Software Eng. 30 (9) (2004) 574–586.

[6] M. Kim, D. Notkin, Discovering and representing systematic code changes, in: Proc. ICSE, 2009, pp. 309–319.

[7] N. Meng, M. Kim, K. McKinley, Lase: Locating and applying systematic edits, in: Proc. ICSE, 2013, pp. 502–511.

[8] T. Rolfsnes, S. D. Alesio, R. Behjati, L. Moonen, D. W. Binkley, Generalizing the analysis of evolutionary coupling for software change impact analysis, in: Proc. SANER, 2016, pp. 201–212.

[9] M. A. Islam, M. M. Islam, M. Mondal, B. Roy, C. K. Roy, K. A. Schneider, [research paper] detecting evolutionary coupling using transitive association rules, in: Proc. SCAM, 2018, pp. 113–122.

[10] Y. Wang, N. Meng, H. Zhong, An empirical study of multi-entity changes in real bug fixes, in: Proc. ICSME, 2018, pp. 287–298.

[11] Y. Wang, N. Meng, H. Zhong, Cmsuggester: Method change suggestion to complement multi-entity edits, in: Proc. SATE, 2018, pp. 137–153.

[12] Apache Derby, `https://github.com/apache/derby` (2018).

[13] DERBY-5162: Null out the wrapped Clob when resetting a SQLClob to NULL., `https://github.com/apache/derby/commit/e9737b6` (2018).

[14] DERBY-2201: Allow scalar functions to return LOBs., `https://github.com/apache/derby/commit/638f1b48afc27c094c7f34a6254778c1a4ad9608` (2018).

[15] Apache Aries, `http://aries.apache.org` (2018).

[16] Apache Cassandra, `https://github.com/apache/cassandra` (2018).

[17] Apache Mahout, `https://github.com/apache/mahout` (2018).

[18] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen, Object-oriented Modeling and Design, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991.

[19] WALA, `http://wala.sourceforge.net/wiki/index.php/Main_Page` (2018).

[20] Apache ActiveMQ, `http://activemq.apache.org` (2019).

[21] Apache UIMA, `http://uima.apache.org` (2019).

[22] P. E. McKnight, J. Najab, Mann-Whitney U Test, American Cancer Society, 2010, pp. 1–1.

[23] Cliff's Delta Calculator: A non-parametric effect size program for two groups of observations, Universitas Psychologica 10 (2011) 545 – 555.

[24] S. Wang, T.-H. Chen, A. E. Hassan, Understanding the factors for fast answers in technical Q&A websites, Empirical Software Engineering 23 (3) (2018) 1552–1593.

[25] MAHOUT-401: Use NamedVector in seq2sparse, `https://github.com/apache/mahout/commit/22d7d31` (2018).

[26] Support of compaction strategy option for stress.java, `https://github.com/apache/cassandra/commit/f06e1d63a2006aa95d36636c56561158c8758a3c`.

[27] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, L. Bairavasundaram, How do fixes become bugs?, in: Proc. ESEC/FSE, 2011, pp. 26–36.

[28] Tan, Ming, Online defect prediction for imbalanced data, Master's thesis, University of Waterloo (2015).

[29] H. Gall, K. Hajek, M. Jazayeri, Detection of logical coupling based on product release history, in: Proc. ICSM, 1998, pp. 190–198.

[30] H. Gall, M. Jazayeri, J. Krajewski, CVS release history data for detecting logical couplings, in: Proc. IWPSE, 2003, pp. 13–23.

[31] J. S. Shirabad, T. C. Lethbridge, S. Matwin, Mining the maintenance history of a legacy software system, in: Proc. ICSM, 2003, pp. 95–104.

[32] H. Kagdi, J. I. Maletic, B. Sharif, Mining software repositories for traceability links, in: Proc. ICPC, 2007, pp. 145–154.

[33] H. H. Kagdi, M. Gethers, D. Poshyvanyk, Integrating conceptual and logical couplings for change impact analysis in software, Empirical Software Engineering 18 (2012) 933–969.

[34] M. Gethers, B. Dit, H. Kagdi, D. Poshyvanyk, Integrated impact analysis for managing software changes, in: Proc. ICSE, 2012, pp. 430–440.

[35] M. B. Zanjani, G. Swartzendruber, H. Kagdi, Impact analysis of change requests on source code based on interaction and commit histories, in: Proc. MSR, 2014, pp. 162–171.

[36] L. L. Silva, M. T. Valente, M. de Almeida Maia, Co-change clusters: Extraction and application on assessing software modularity, Trans. Aspect-Oriented Software Development 12 (2015) 96–131.

[37] T. Rolfsnes, L. Moonen, S. D. Alesio, R. Behjati, D. Binkley, Aggregating association rules to improve change recommendation, Empirical Software Engineering 23 (2) (2018) 987–1035.

[38] A. E. Hassan, R. C. Holt, Predicting change propagation in software systems, in: Proc. ICSM, 2004, pp. 284–293.

[39] K. Yamauchi, J. Yang, K. Hotta, Y. Higo, S. Kusumoto, Clustering commits for understanding the intents of implementation, in: Proc. ICSME, 2014, pp. 406–410.

[40] P. Kreutzer, G. Dotzler, M. Ring, B. M. Eskofier, M. Philippsen, Automatic clustering of code changes, in: Proc. MSR, 2016, pp. 61–72.

[41] Z. Li, Y. Zhou, PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code, in: Proc. ESEC/FSE, 2005, pp. 306–315.

[42] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, T. N. Nguyen, Clone-aware configuration management, in: Proc. ASE, 2009, pp. 123–134.

[43] M. Ohrndorf, C. Pietsch, T. Kehrer, ReVision: A tool for history-based model repair recommendations, in: Proc. ICSE-Companion, 2018, p. 105.

[44] C. Le Goues, T. Nguyen, S. Forrest, W. Weimer, Genprog: A generic method for automatic software repair, IEEE Transaction on Software Engineering 38 (1).

[45] D. Kim, J. Nam, J. Song, S. Kim, Automatic patch generation learned from human-written patches, in: Proc. ICSE, 2013, pp. 802–811.

[46] F. Long, M. Rinard, Automatic patch generation by learning correct code, in: Proc. POPL, 2016, pp. 298–312.

[47] F. Long, P. Amidon, M. Rinard, Automatic inference of code transforms for patch generation, in: Proc. ESEC/FSE, 2017, pp. 727–739.

[48] H. Zhong, H. Mei, Mining repair model for exception-related bug, The Journal of Systems & Software 141 (2018) 16–31.