Using Frugal User Feedback with Closeness Analysis on Code to Improve IR-Based Traceability Recovery

Hongyu Kuang
State Key Lab for Novel Software Technology
Nanjing University
Nanjing, China
khy@nju.edu.cn

Hui Gao
State Key Lab for Novel Software Technology
Nanjing University
Nanjing, China
ghalexcs@gmail.com

Hao Hu
State Key Lab for Novel Software Technology
Nanjing University
Nanjing, China
myou@nju.edu.cn

Xiaoxing Ma
State Key Lab for Novel Software Technology
Nanjing University
Nanjing, China
xxm@nju.edu.cn

Jian Lü
State Key Lab for Novel Software Technology
Nanjing University
Nanjing, China
lj@nju.edu.cn

Patrick Mäder
Fakultät für Informatik und Automatisierung
Technische Universität Ilmenau
Ilmenau, Germany
patrick.maeder@tu-ilmenau.de

Alexander Egyed
Institute for Software Systems Engineering
Johannes Kepler University
Linz, Austria
alexander.egyed@jku.at

***Abstract*—Traceability recovery allows developers to extract and comprehend the trace links among software artifacts (e.g., requirements and code). These trace links can provide important support to software maintenance and evolution tasks. Information Retrieval (IR) is now widely accepted as the key technique of semi-automatic tools to recover candidate trace links based on textual similarities among artifacts. However, the vocabulary mismatch problem between different artifacts hinders the performance of these IR-based approaches. Thus, a growing body of enhancing strategies were proposed based on user feedback. They allow to adjust the textual similarities of candidate links after users accept or reject part of these links. Recently, several approaches successfully used this strategy to improve the performance of IR-based traceability recovery. However, these approaches require a large amount of user feedback, which is infeasible in practice. In this paper, we propose to improve IR-based traceability recovery by introducing only a small amount of user feedback into the closeness analysis on call and data dependencies in code. Specifically, our approach iteratively asks users to verify a chosen candidate link based on the quantified functional similarity for each code dependency (called closeness) and the generated IR values. The verified link is then used as the input to re-rank the unverified candidate links. An empirical evaluation based on five real-world systems shows that our approach can outperform four baseline approaches by using only a small amount of user feedback.**

Keywords—traceability recovery, information retrieval, closeness analysis, user feedback, code dependencies

I. Introduction

Software traceability is known as "the ability to interrelate any uniquely identifiable software engineering artifact to any other, maintain required links over time, and use the resulting network to answer questions of both the software product and its development process" [1]. These trace links (a.k.a. traces) can help stakeholders in development-related tasks, such as software maintenance and evolution. Recent work [2] showed that software quality is strongly affected by the completeness of software traceability. Another study [3] reported that subjects with correct and complete requirements-to-code traces can perform maintenance tasks on average 24% faster and created on average 50% more correct solutions as compared to the others without the traces. However, existing work also reported that high-quality traces are difficult to obtain [36] due to the large number of required traces, frequent changes in software artifacts such as code, and the informal nature of requirements.

Aiming at providing semi-automated tools to reduce the manual efforts, Information Retrieval (IR) is now the most widely accepted and applied technique in the research of traceability recovery [4-20]. In general, an IR-based recovery approach computes the textual similarity between two software artifacts through IR models, such as Vector Space Model (VSM) [4], Latent Semantic Indexing (LSI) [5], and the probabilistic Jensen and Shannon model (JS) [6]. Users then verify candidate traces along the automatically generated candidate lists sorted by IR values in descending order, instead of exploring all possible traces between any two given artifacts. Unfortunately, the accuracy of the IR-based approaches, i.e., the rankings of relevant traces in the candidate lists, remains unsatisfying to fully support the traceability recovery process. The reason is that different artifacts, such as requirements and code, often use different terms to denote the same concept. To address this so-called *vocabulary mismatch problem* for IR-based traceability recovery, researchers proposed many enhancing strategies from different perspectives, such as enhancing the lexical analyses [7, 8, 20], or combining with the code dependency analysis [9, 10].

Meanwhile, focusing on the semi-automatic nature of IR-based traceability recovery, a different body of enhancing strategies [11-14] are proposed based on the verified results of the candidate links by the users (either as relevant links or as false positives). This *user feedback* indicates users' judgements on whether the calculated IR values correctly reflect the actual trace links. Thus, when users start to verify an IR candidate list, their feedback on verified links can then be used to improve the ranking of the remaining list. Hayes et al. [11] proposed a pioneer approach that asks users to iteratively verify candidate links and uses the standard Rocchio algorithm [21] to modify the weights of the terms in requirements and code based on user feedback. However, follow-on work [12] demonstrated that improvements brought by the previously discussed approach are both limited and not always evident. To address this issue, Panichella et al. [14] proposed an adaptive version of the Rocchio algorithm that considers both the numbers of terms in software artifacts and the previously verified links. Meanwhile,

XXX-X-XXXX-XXXX-X/XX/$XX.00 ©20XX IEEE

instead of using user feedback in term re-weighting, Panichella et al. [13] proposed to iteratively use user-verified links as the input of their code dependency analysis to bonus the calculated IR values between requirements and code. However, to achieve the best performance of these two approaches (i.e., [13] and [14]), the user would have to verify each link in the candidate list until the last relevant trace for a given requirement is found. This requires great manual efforts in practice due to the low ranking of relevant traces and the large number of links in the candidate lists generated by IR techniques. Furthermore, the users will get tired when they discarded too many false positives during the verification process on the candidate lists [12].

To improve IR-based traceability recovery based on a small amount of user feedback, in this paper we propose an IR-based approach that combines user feedback with the closeness analysis on code dependencies. Closeness is a code measure (proposed by Kuang et al. [10]) that quantifies the degree of interaction based on direct (e.g., method calls, inheritance, and class usage) and indirect (e.g., reading or writing the same data) code dependencies among classes. We first use closeness to build separate regions in which code classes closely interact with each other based on their code dependencies. We argue that each region (named as *candidate region*) implicitly represents one unique part of the system functionalities. Thus, our approach is based on the assumption that *when the user is verifying the IR candidate lists, if one or more classes from a candidate region are verified as relevant to a given requirement, other classes in the region are also likely to be traced to the same requirement.*

In particular, our approach first uses IR techniques to generate ranked lists of candidate links between requirements and classes in code. Our approach then improves the ranking of the candidate list for each requirement in two steps: (1) it locates a small set of candidate regions and ask users to verify whether one or more classes that have high IR values in the region are relevant to the given requirement; (2) based on each verification result (relevant or irrelevant), our approach then either promote or demote the unverified links in which the classes have a composed high closeness measures to the class in the verified link. Eventually, the ranking of candidate lists is improved according to the composition of IR value, user feedback, and closeness measure. We evaluated our approach on five real-world systems and found that our approach statistically outperforms the pure IR-based approaches and four other baseline approaches [10, 13, 14] based on three mainstream IR models (VSM, JS, and LSI). The evaluation also showed that the improvements of our approach only require users to verify in average 6.33 classes for each requirement (from 1.45% to 6.84% of all code classes for the five evaluated systems).

The contribution of this paper is combining closeness analysis on call and data dependencies with user feedback to improve IR-based requirements-to-code traceability recovery. This work mainly targets functional requirements. We name our approach as **CLUSTER** (**CL**oseness-and-**US**er-feedback-based **T**rac**E**ability **R**ecovery). CLUSTER contains two novel features: (1) we build candidate regions of classes based on the closeness measure to represent one implicit aspect of the system functionalities; (2) we improve the accuracy of IR-based traceability recovery based on a small amount of user feedback and closeness analysis to re-rank IR candidate lists.

The rest of this paper is structured as follows. Section II discusses the research background and related work. Section III presents our approach. Section IV introduces the experiment and research question. Section V answers the research questions based on the experiment results. Section VI discusses possible threats. Section VII makes conclusions and refers to future work.

II. Background and Related Work

The focus of ongoing traceability researches [7-20] is to enhance the performance of IR techniques when tracing between source and target artifacts (e.g., requirements and code) to handle the vocabulary mismatch problem. Various enhancing strategies have been proposed from different perspectives, such as incorporating with execution tracing [18, 19], enhancing the advanced lexical analysis [7, 8, 20], combining with code dependency analysis [9, 10, 13], and using user feedback [11-17]. Specifically, Poshyvanyk et al. [18] proposed an affine transformation to combine execution tracing with IR technique in their feature location approach called PROMESIR. Dit et al. [19] further improved PROMESIR by defining a data fusion model that integrates IR, execution tracing, and web mining algorithms. From the advance lexical analysis perspective, Cleland-Huang et al. [7] proposed to introduce extra texts and to exclude keywords promoting wrongly retrieved traces when tracing requirements to code. Gethers et al. [8] used relational topic modeling to complement IR-based traceability recovery. De Lucia et al. [20] proposed to use smoothing filters to reduce the effect of textual noises in software artifacts for IR techniques. However, the use of advanced lexical analyses requires rich descriptions and documentations on both requirements and code. Unfortunately, in practice this is not always the case.

A different body of work focuses on code dependencies, which are the unique structural information of code, to improve IR-based traceability recovery [9, 10,13]. These approaches face two challenges: (1) code dependencies are not equally important to improve IR-based approaches; (2) incorrect links brought by IR techniques can even undermine the improvement [9]. To address the first issue, Kuang et al. [10] proposed the closeness measure to quantify the functional similarity for each call and data dependency and then use this measure to improve IR-based traceability recovery [10] and identifying outdated requirement [24]. Their work is based on two findings: (1) requirements are implemented in connected areas of code [22]; (2) call and data dependencies are complementary in understanding requirements traceability [23]. This idea of identifying code elements with strong connections is also used in mining design templates [40]. However, their approach [10] only uses the top-ranked class in the candidate list for each requirement as the input. This limits the effects of the proposed closeness analysis. To address the second issue, Panichella et al. [13] combined the code dependency analysis with user feedback. This approach first asks users to iteratively each candidate link. It then bonuses IR values of the unverified candidate links if they contain classes that can connect to the verified classes through direct code dependencies (i.e., calling relationship, class inheritance, and class usage). The ranking for each unverified candidate link in the remaining list will be changed accordingly after each verification. However, to achieve the best performance, this approach has to ask the user to verify all links in the candidate list until the last relevant trace for a given requirement is found.

User feedback on IR candidate lists is also an important perspective to improve IR-based traceability recovery [11-17]. In general, one kind of these approaches [15-17] uses a subset of relevant traces verified by the user as a training set, while the other kind [11-14] asks the user to iteratively verify each candidate link and returns this information to the approaches. Specifically, Antoniol et al. [15] used the training set as the input of a Bayesian Classifier to improve IR-based approaches. Di Penta et al. [16] also used this approach to recover traceability links between code and documents in systems with many COTS and middleware components. Recently, Guo et al. [17] proposed a neural network that uses word embedding and RNN with GRU to recover requirements-to-code traces by mining a training set of traces. The accuracy of this approach can be 41% and 32% higher than the pure IR-based approaches using VSM and LSI, respectively. However, this improvement needs 55% of verified traces for training and development [16]. This is infeasible in the recovery scenario because trace links are usually recovered from scratch. Meanwhile, Hayes et al. [11] proposed to ask users to iteratively verify each candidate link, rather than prepare a training set in advance. This approach then applies the feedback to the standard Rocchio algorithm [21] on VSM to modify the weights of the terms in requirements and code. However, follow-on work [12] demonstrated that the benefits provided by the previous approach are both limited and not always evident. As reported by Panichella et al. [14], this issue is caused by the fact that the queries (i.e., requirements) can contain more terms than the documents (i.e., code text) in IR-based traceability recovery. This fact violates the precondition of using the Rocchio algorithm, i.e., the queries contain only a few terms compared to the size of the documents to retrieve. The authors then proposed an adaptive version of the Rocchio algorithm that considers both the numbers of terms and the previously verified links. However, like the approach proposed by Panichella et al. [13] that combine user feedback with code dependency analysis, the performance of the adaptive Rocchio algorithm is still highly dependent to the number of user-verified candidate links.

Unlike the discussed IR-based approaches based on user feedback and/or code dependency analysis, our approach first iteratively locates a small set of candidate links for users to verify, based on both IR values and closeness measures. Using the verified links as input, our approach then amplifies and propagates users' valuable adjustments to the IR values of unverified links through the closeness analysis on code. Thus, our approach can improve the accuracy of IR-based traceability recovery by requiring only a small amount of user feedback.

III. Proposed Approach

We propose a three-step approach. First, we build a *Code Dependency with Closeness Graph (CDCGraph)* based on captured code dependencies with their calculated closeness measures (Step 1). Second, we use IR techniques to generate candidate links between requirements and classes (Step 2). Third, we build candidate regions in the CDCGraph and ask users to iteratively verify a small number of representative classes in each region for a given requirement, until the users exit the verification process; the ranking of IR candidate lists will be adjusted according to the verified links (Step 3). Details of each step are explained in the following subsections with a consistent excerpt adapted from the Maven system [26].

A. Step 1: Building CDCGraph with Calculated Closeness

In this subsection, we introduce how to capture code dependencies and calculate closeness measures (proposed by Kuang et al. [10]). We then create a Code Dependency with Closeness Graph (CDCGraph) as the basis of our approach.

1) Capturing and Organizing Code dependencies among classes. We consider four kinds of dependencies among classes: class call dependencies, class inheritance, class usage, and class data dependencies. A call dependency from class C_a to class C_b means that there is at least one method call from C_a to C_b. A class inheritance from class C_a to class C_b means that C_a is derived from C_b. A class usage from class C_a to class C_b means that C_b is a field of C_a. A class data dependency between two classes C_a and C_b exists if two methods $C_a.m_a$ and $C_b.m_b$ read or manipulate the same data. In this paper, we only focus on objects shared in program memory during runtime. However, this definition can be extended to other data, e.g., stored in the file system or database. However, Existing researches [10, 23] have shown that, although less obviously visible, class data dependencies can complement the first three code dependencies in understanding requirements-to-code traceability.

To capture the discussed four kinds of code dependencies, we used a dynamic analysis tool proposed by Kuang et al. [23]. This tool uses JVMTI (Java Virtual Machine Tool Interface) to capture method-level call and data dependencies. We chose this tool because: (1) this tool can capture all four code dependencies by running test cases in a single test run and correctly handle polymorphism; (2) potentially missed code dependencies caused by incomplete testing are tolerable for our approach (see Section VI). To be clear, we only keep the data types of shared objects to represent data dependencies by aggregating all shared objects of the same type. The previous case studies [23] reported that the data-type-level data dependencies can convey comparable information as object-level ones and save lots of computational efforts. The four kinds of class-level code dependencies are then derived based on the captured method-level dependencies: (1) the class call dependencies are abstracted from method call dependencies with the number of distinct method calls having the same calling direction; (2) the class data dependencies are abstracted from method data dependencies and keep all related data types; (3) the class usages are also abstracted from method data dependencies; and (4) the class inheritance is retrieved from method call dependencies, i.e., the constructor of a derived class calling the constructor of its base class.

By consulting typical IR-based approaches based on code dependency analysis (e.g., [10] and [13]), we treat class call dependency, class inheritance, and class usage as one kind of code dependencies (i.e., *direct code dependencies*), and *class data dependencies* as a different one. The first three code dependencies are combined because they are structurally similar (directed links from source classes to sink classes) while the class data dependencies are different (undirected links between two classes with shared data types). Meanwhile, class call dependency, class inheritance, and class usage largely overlap with each other, while class data dependency slightly overlaps with direct code dependencies (for more details please refer to Section VI). Thus, we calculate closeness measures for these two kinds of code dependencies separately.

Figure 1 depicts adapted samples of captured code dependencies from the Maven system. In the figure, a direct code dependency is represented by a solid line with an arrow and is labeled with the number of method calls and/or class usages. For example, a direct code dependency from `PluginDescriptor` to `MojoDescriptor` in Figure 1 consists of a class usage and a class call dependency. Meanwhile, a class data dependency is represented by a dashed line without an arrow and is labeled with the number of shared data types. For example in Figure 1, `PluginDescriptor` passes two data types as parameters to `MojoDescriptor`, i.e., `Plugin` and `CompnentDescriptor`. Thus, these two classes are also connected by a class data dependency based on the shared two data types.

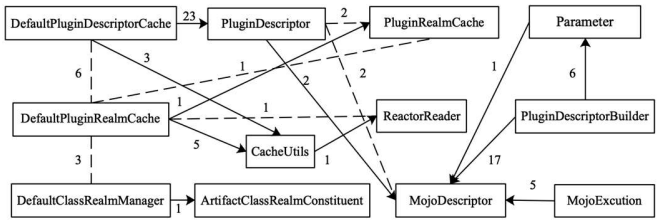

Fig. 1. Samples of captured class call dependencies (solid lines with an arrow) and class data dependencies (dashed lines without an arrow)

2) Closeness for direct code dependencies and class data dependencies. For a direct code dependency, it is intuitive that if two classes share more distinct method calls and class usages, they will interact more closely. Another important factor is the sink's in-degree and source's out-degree in a direct code dependency. The in-degree refers to the number of sources that reach the sink and the out-degree refers to the number of sinks that are reached by the source. A smaller sink's in-degree indicates that the sink class focuses more on serving the source class. A smaller source's out-degree means that the source class relies more on the service provided by the sink class. Thus, the $Closeness_{DC}$ for direct code dependencies is computed as:

$$\text{Closeness}_{\text{DC}} = \frac{2 \times \text{N}}{\text{WeightedInDegree}_{\text{Sink}} + \text{WeightedOutDegree}_{\text{Source}}} \quad (1)$$

where N refers to the number of distinct method calls and class usages in a given direct code dependency. When counting sink's in-degree and source's out-degree, the number of method calls and class usages from each direct code dependency are used as their extra weights. These two variables are denoted as $WeightedInDegree_{Sink}$ and $WeightedOutDegree_{Source}$ in the formula, respectively. $\text{Closeness}_{\text{DC}}$ is in the range [0, 1].

A data dependency exists because two classes share distinct data types with each other. We first use a factor named Inverse Data Type Frequency (idtf) to weigh the importance of each data type (proposed by Kuang et al. [23]). Besides, for a class data dependency between classes C_i and C_j, the ratio between the number of shared data types in this dependency and the number of all data types shared by C_i and C_j (from other data dependencies) is also important to calculate its closeness. Thus, the $Closeness_{CD}$ for class data dependencies is computed as:

$$idtf = \log \frac{N}{n_{dt}} \qquad \text{Closeness}_{\text{CD}} = \frac{\sum_{x \in \{DT_i \cap DT_j\}} \text{idtf}(x)}{\sum_{y \in \{DT_i \cup DT_j\}} \text{idtf}(y)} \quad (2)$$

where N is the number of all captured class data dependencies and n_{dt} is the occurrence of a given data type in all data dependencies. $DT_i \cap DT_j$ refers to the data types that C_i shares with C_j in the data dependency, and $DT_i \cup DT_j$ refers to all data types that C_i and C_j access in the entire code. $\text{Closeness}_{\text{CD}}$ is in the range [0, 1]. The idea of idtf is similar to the key weighing factor idf (Inverse Document Frequency) in IR techniques [34]. It reflects how a data type is uniquely shared among classes, thus indicating the degree of interaction between the two classes sharing the type. Moreover, before calculating $\text{Closeness}_{\text{CD}}$, a $\text{Threshold}_{\text{idtf}}$ is used to ignore data types with low idtf values. If all types in a class data dependency are ignored, this dependency will be ignored. The reason is that this kind of data dependencies weakens the effect of closeness analysis [10] (The threshold calibration is discussed in Section IV).

3) Generating the CDCGraph. We now create a Code Dependency with Closeness Graph (CDCGraph) as G=<V, E>. Each vertex V represents a class of the code and is annotated with the class name. Furthermore, we define two kinds of edges E in the graph: E_{DC} representing direct code dependencies and E_{CD} representing indirect class data dependencies between two classes. Finally, each code dependency is annotated with the calculated closeness measure. A derived CDCGraph based on the sample code dependencies in Figure 1 is shown in Figure 2.

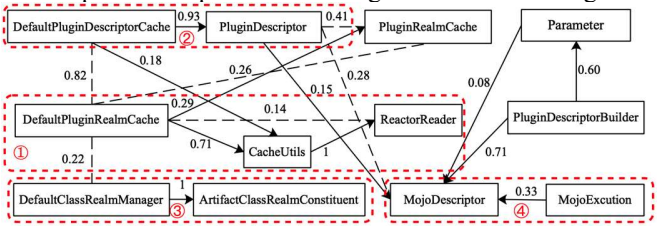

Fig. 2. Sample CDCGraph with code dependencies annotated by closeness and created candidate regions (numbered and surrounded by dashed lines)

B. Step 2: Generating IR candidate lists

Our IR technique consists of the following four steps:

- *Creating corpus.* Each class in code is extracted into one document containing its comments and identifiers including class name, method names and field names. For each requirement, we extract a document that includes its title and content (e.g., preconditions, main-flow, and sub-flows for structured use cases or pure text for unstructured requirements).
- *Normalizing corpus.* The documents of requirements and classes are normalized by standard pre-processing techniques for IR including splitting identifiers, special token elimination, stemming, and stop word removal.
- *Indexing corpus and computing textual similarity*. We use tf-idf for corpus indexing and three mainstream IR models to compute textual similarity: Vector Space Model (VSM) [4], Latent Semantic Indexing (LSI) [5], and the probabilistic Jensen and Shannon (JS) model [6].
- *Generating candidate links.* We rank IR candidate lists in descending order based on the IR values of candidate links.

Table I shows a candidate list generated through JS between an issue MNG-4194 ("API to safely release of plugin realms") of the Maven system and the twelve classes in Figure 2. The list is ranked by initial IR values ("IR_1") in descending order ("N_1"). An 'x' in column "Trace" marks an actual trace for MNG-4194 existed in the oracle of all traces for Maven.

TABLE I. A CANDIDATE LIST BETWEEN REQUIREMENT MNG-4194 AND TWELVE CLASSES IN MAVEN AFTER STEP 2 (N_1, IR_1) AND STEP 3 (N_2, IR_2)

N_1	Class	IR_1	Trace	IR_2	N_2
1	PluginRealmCache	0.298	x	0.298	1
2	CacheUtils	0.227	x	0.227	7
3	PluginDescriptor	0.113	x	0.284	6
4	DefaultClassRealmManager	0.113	x	0.179	8
5	Parameter	0.112		0.090	10
6	ReactorReader	0.054	x	0.298	2
7	DefaultPluginRealmCache	0.037	x	0.298	3
8	MojoExecution	0.032		0.029	11
9	MojoDescriptor	0.026		0.154	9
10	DefaultPluginDescriptorCache	0.025	x	0.298	4
11	ArtifactClassRealmConstituent	0. 018	x	0.298	5
12	PluginDescriptorBuilder	0. 016		0.013	12

C. Step 3: Locating Candiate Links for User Verification to Improve the Ranking of Candiate List

In this step, we first prune the CDCGraph into different connected areas as *candidate regions*. We assume that classes in the same region are likely to implement similar functionalities of the system. We then ask users to iteratively verify a small number of candidate links for each region. The verified links, either as relevant traces or false positives, are used to promote or demote the IR values of unverified links by analyzing the CDCGraph, respectively. Finally, we set up conditions for users to exit the verification process. The goal of this step is to guide users to verify a small but vital set of candidate links so that we can amplify and propagate these frugal but valuable judgements from users to improve IR-based traceability recovery.

1) Locating candidate links for user verification based on candidate regions. First, we use two separate thresholds based on calculated closeness measures to prune the CDCGraph, i.e, $Threshold_{DC}$ for direct code dependencies and $Threshold_{CD}$ for class data dependencies. After the pruning, we choose the created connected areas that contain at least two or more classes as candidate regions. For example in Figure 2, we propose a $Threshold_{DC}$ of 0.7 and a $Threshold_{CD}$ of 0.9 and get four candidate regions shown in the figure (annotated with numbers 1 to 4). Second, for a given requirement, we choose the class that has the highest IR value in each region as its *representative class*. We then ask the user to iteratively verify each candidate region in descending order based on the IR values between its representative class and the given requirement. Specifically, if the representative class is verified as traced to the requirement, we mark all the other classes in the region as relevant. On the other hand, if the representative class is verified as not traced, We ask the user to further verify whether the other classes in the same region are traced to the requirement according to their IR values in *descending* order. Once another class is verified as not traced, we mark all remaining unverified classes in the region as irrelevant. Whenever a class in the region is verified or marked, the relevant class is used to give bonuses to the IR value of unverified links, while the irrelevant class is used to give penalties. All IR values will be updated in each iteration of user verification. Thus, the ranking of unverified candidate regions will also be updated before the user starts to verify the next candidate region. How to give bonuses or penalties will be discussed in the next sub-step. The idea of this sub-step is similar to the double-elimination tournament. The reason is that the IR values cannot guarantee to reflect the actual trace link between the representative class of a candidate region and a requirement. So we give the region a second chance to continue user verification on it until another class is verified as irrelevant. It is also worth-while noticing that we assume the user to always make correct verifications on the IR candidate links. We argue that this assumption is reasonable because: (1) it is widely used in the research of using user feedback to improve IR-based traceability [11-17]; (2) we aim to use only a small amount of user feedback to improve IR-based approaches, so we argue that the assumption is also viable for our approach in practice.

2) Adjusting IR Values According to User Feedback. First, when a candidate link is verified (by the user) or marked (by our approach) as a relevant trace, we use separate strategies on two kinds of code dependencies to give bonuses to the IR values for each class C_{UNVER} in the unverified links. For direct code dependency, we try to find a path from C_{UNVER} to the class C_{VER} in the verified or marked link. A valid path can only have one direction, meaning C_{UNVER} transitively reaches or is reached by C_{VER}. For class data dependency, we consider whether C_{UNVER} can directly (non-transitive) connect to C_{VER}. C_{UNVER} can get bonuses according to both kinds of code dependencies. The IR value updated with a bonus (IR_{bonus}) for C_{UNVER} is computed as:

$$ADJ_{DC} = \prod_{x \in PATH} Closeness_{DC}(x)$$

$$IR_{bonus} = IR_{current} + IR_{top}(ADJ_{DC} + Closeness_{CD}(x)) \quad (3)$$

where $IR_{current}$ represents C_{UNVER}'s current IR value, IR_{top} represents the highest IR value between the given requirement and all classes, *PATH* represents the set of direct code dependencies in a discovered path between C_{UNVER} and C_{VER}, $Closeness_{DC}(x)$ represents the closeness measure for each direct code dependency in the path, and $Closeness_{CD}(x)$ represents the closeness measure of the class data dependency that directly connects C_{UNVER} and C_{VER}. It is possible that there are multiple paths between C_{UNVER} and C_{VER}. We only keep the path that can maximize ADJ_{DC} for each C_{UNVER} in the unverified link.

Unlike giving bonuses, when a candidate link is verified or marked as a false positive, we use a rather conservative way to give penalties to the IR values for C_{UNVER} in the unverified links by considering the discussed one-direction paths based on direct code dependencies only. This is because previous work [13] has reported that when using the same bonuses to give penalties according to the verified false positives, this enhancing strategy based on user feedback was not able to improve IR-based traceability recovery. Thus, the IR value updated with a bonus ($IR_{penalty}$) for C_{UNVER} is computed as:

$$IR_{penalty} = IR_{current}\ (1 - IR_{top} \times ADJ_{DC}) \quad (4)$$

where $IR_{current}$ represents C_{UNVER}'s current IR value, IR_{top} represents the highest IR value between the given requirement and all classes, and ADJ_{DC} is defined in Formula (3). The entire process of Step 3 is descripted in Algorithm 1, where *list* represents the candidate list for a given requirement *req*. For example, when the user starts to verify the initial candidate list

Algorithm 1 Improving the Ranking of IR Candidate List for each Requirement (*req, list, Threshold$_{DC}$, Threshold$_{CD}$*)

```
1: regions <- CDCGraph.prune (Threshold_DC, Threshold_CD);
2: while not (stopping criterion) do
3:    region_top <- regions.getTopRegion (list, req);
4:    class_maxIR <- Region_top.getMaxIRClass(req);
5:    The user verifies the link (req, class_maxIR)
6:    if (req, class_maxIR) is a relevant trace then
7:       foreach class in region_top do
8:          giveBonusToUnverifiedLink (class, list);
9:       end foreach
10:   else
11:      givePenaltyToUnverifiedLink (class_maxIR, list);
12:      Hide class_maxIR from region_top
13:      class_maxIR <- region_top.getMaxIRClass(req);
14:      while (req, class_maxIR) is a relevant trace do
15:         giveBonusToUnverifiedLink (class_maxIR, list);
16:         Hide class_maxIR from region_top
17:         class_maxIR <- region_top.getMaxIRClass(req);
18:      end while
19:      foreach class in region_top.remainingClasses() do
20:         givePenaltyToUnverifiedLink (class, list);
21:      end foreach
22:   end if
23:   Hide region_top from regions
24:   Hide verified links and reorder list
25: end while
```

shown in Table I, the top ranked class `PluginDescriptor` has highest IR value is 0.298. Because this class is not in any candidate region, our approach asks user to verify `CacheUtils` in Region 1. With a verified relevant trace, our approach uses all three classes in Region 1 to give bonuses to other unverified classes. Similarly, the user then verifies Region 2 and 3 and gives bonuses. However, when `MojoDescriptor` in Region 4 is verified as a false positive, other unverified classes received penalties (such as `Parameter`). The reordered candidate list is also shown in Table II (i.e., columns "IR_2" and "N_2").

Exiting the user verification. Previous work [12] has reported that when met too many false positives, the user of an IR-based recovery approach will be exhausted and likely to be error-prone during the verification. Thus, we set up a stopping criterion that allows users to exit the verification process when they meet only five false positives for each requirement. The reason for this criterion is trying to use minimum user feedback to improve IR-based traceability recovery with the help of closeness analysis. However, the user can choose to verify more candidate links for better accuracy (discussed in Section V).

IV. Experimental Setup

We now introduce our experimental setup to evaluate our approach. Section IV.A introduces the five evaluated systems. Section IV.B defines metrics for evaluating the performance of our approach and baseline approaches. Section IV.C discusses the threshold calibration for our approach. At last, Section IV.D discusses our research questions and the design of experiments.

A. Evaluated Systems

Our evaluation is based on five real-world software systems: iTrust [25], Maven [26], Pig [27], GanttProject[35] and Infinispan [33]. We chose these systems because of their availability of both requirements specifications with developer maintained test cases and their Requirements-to-code Trace Matrices (RTM). For GanttProject, high-quality requirements-to-code traces are gained by recruiting the original developers. The RTM of iTrust contains method-level traces maintained by original developers and is publicly available [25]. However, the RTMs of the other four systems are at class-level. To keep our experiment consistent at the same granularity, we propagated the method-level traces of iTrust to class-level traces by aggregating all traces to methods of a class on the class-level.

TABLE II. Overview of the Five Evaluated Systems

#	iTrust [25]	Maven [26]	Pig [27]	Infinispan [33]	GanttProject[35]
Version	13.0	3.5.2	0.17.0	9.2.0	2.0.9
Programming language	Java	Java	Java	Java	Java
KLoC	43	101	365	521	45
Executed classes	138	94	236	388	124
Evaluated requirements	34	36	68	237	16
Ave. number of classes tracing to a requirement	8 (1-17)	4 (1-18)	5 (1-38)	6 (1-79)	20 (4-38)
Direct code dependencies	274	182	1998	2126	617
Class data dependencies	4792	1812	5405	6076	1788
Relevant Traces in RTM	255	155	356	1515	315

Meanwhile, Maven, Pig, and Infinispan come from the dataset named IlmSeven [28]. This dataset consists of seven open source software projects that are implemented in Java. By analyzing both the issue-tracking tool Jira [29] and GitHub over the seven projects, this dataset records a large number of development artifacts and links between them, including the commit logs and issues. When the text of a commit log contains a unique id of an issue, indicating that these code changes are committed to address the mentioned issue, we can retrieve actual traces created during daily development between the issue and the classes modified by this commit (i.e., the changed Java files).

Furthermore, to make sure that the issues linked with commit logs can be treated as meaningful functional requirements (the main target of this paper), we use the following five heuristics to filter and merge the issues based on the IlmSeven dataset: (1) we ignore issues with terms of "testing" or "testcase"; (2) the resolution of issues must be "Fixed"; (3) the priority of issues must be "Major" or "Critical"; (4) we merge the issues if each two of them have the explicit issue link as "part-of"; (5) we only keep issues that have or contain (for the merged issues) the issue types as "New feature" (or "Feature request"). The reason for the last two heuristics is that contributors of an open source project usually start a "New feature" issue to apply for a new system functionality and then improve the implementation of this functionality through "Improvement", "Bug", or even "New feature" issues gradually. Meanwhile, we capture high-quality call and data dependencies by using the dynamic analysis tool [23] during the running of the sample systems (for Maven and Infinispan) or the test cases (for iTrust, GanttProject, and Pig) maintained by original developers. We randomly inspected about 15% of both the filtered and merged issues and the captured code dependencies for each evaluated system. We found no contradictories in either the organized issues or the code dependencies. Unfortunately, because collecting code dependencies and preparing linked issues with an acceptable quality requires manual efforts, at this moment we can only

perform our experiments on Maven, Pig, and Infinispan from the IlmSeven data set. We are currently working on using the other four systems to further evaluate our approach. Table II lists basic information about the five evaluated systems. The entire dataset is open at: https://dataverse.harvard.edu/dataverse/CLUSTER.

B. Metrics

We first leveraged two well-known metrics for our evaluation, i.e., recall and precision:

$$recall = \frac{|relevant \cap retrieved|}{|relevant|}\% \quad precision = \frac{|relevant \cap retrieved|}{|retrieved|}\% \quad (5)$$

where *relevant* is the set of relevant links and *retrieved* is the set of all links retrieved by traceability recovery approaches.

A common way to evaluate the accuracy of IR techniques is to compare the precision values obtained at different recall levels, resulting in a set of precision-recall points displayed as curves. We then leveraged the following two metrics: average precision (AP) and mean average precision (MAP). These two metrics are widely used to evaluate IR-based approaches for traceability recovery. AP and MAP are computed as:

$$AP = \frac{\sum_{r=1}^{N}(Precision(r) \times isRelevant(r))}{|RelevantDocuments|} MAP = \frac{\sum_{q=1}^{Q} AP(q)}{Q} \quad (6)$$

where r is the rank of the target artifact in an ordered list of links, *Precision(r)* represents its precision value, *isRelevant()* is a binary function assigned 1 if the link is relevant or 0 otherwise, N is the number of all documents, q is a single query, and Q is the number of all queries. AP measures how well relevant documents of all queries (requirements) are ranked to the top of the retrieved links. Meanwhile, MAP uses the average of the AP scores of all queries to measure how well relevant documents for each query are ranked to the top of the retrieved links.

C. Threshold Calibration

We need to calibrate four thresholds for our approach: $Threshold_{idtf}$, $Threshold_{DC}$, $Threshold_{CD}$, and the k value for LSI. According to the previous case studies [10, 23], we used a $Threshold_{idtf}$ of 1.4 to ignore data types with small idtf value (discussed in Section III.B). We then follow the same process proposed by Kuang et al. [10] to calibrate both $Threshold_{DC}$ and $Threshold_{CD}$. We first used the 3σ criterion to filter out outliers (closeness measure three times higher or lower than the standard deviation σ) from the set of $Closeness_{DC}$. We then rescaled closeness measures into [0, 1] by min-max normalization. Filtered abnormally high closeness measures were set to 1 and abnormally low closeness measures were set to 0 in the rescaled range. We used the same process to calibrate $Threshold_{CD}$. The two processes led to a $Threshold_{DC}$ of 0.7 and a $Threshold_{CD}$ of 0.9 (the same two thresholds used in the previous work [10]) based on our experiment results. These two thresholds are only used to build candidate regions for all evaluated systems, we still use the original closeness measures for Algorithm 1. For the k value of the LSI method, we found that k = 85 provided the best accuracy for iTrust, Maven, and GanttProject, while for Pig and Infinispan k = 220 is the best. The two separate LSI k values are necessary because the number of executed classes of iTrust, Maven, and GanttProject are 138, 94, and 124 while the executed classes for Pig and Infinispan are 236 and 388, respectively. We use the same four thresholds for all systems to avoid biases. For new systems we suggest using the same calibration process before fine-tuning them to optimize the performance.

D. Research Questions

In this paper, we aim to study whether the combination of a small amount of user feedback and the closeness analysis on code dependencies is able to improve IR-based traceability recovery. Thus, we formulated our research question as follows:

Can our approach outperform baseline approaches for IR-based traceability recovery?

To study the RQ, we use the following four baseline approaches: (1) the pure IR-based approach (*IR-ONLY*); (2) the approach combining code dependency analysis with user feedback: *User-Driven Combination of Structural and Textual Information* (*UD-CSTI* [13]); (3) the approach using closeness analysis on code only: *Traceability Recovery by Information retrieval and ClosEness analysis* (*TRICE* [10]); and (4) the approach using an adaptive version of the Rocchio Algorithm: *Adaptive Relevance Feedback (ARF* [14]*)*. Our approach is named as *CLUSTER*. We planned to use three mainstream IR models, i.e., VSM, LSI, and JS, to compare CLUSTER with the four baseline approaches. However, ARF is proposed on VSM because the Rocchio Algorithm works on the text vectors [21]. Furthermore, applying the Rocchio Algorithm to LSI or JS is not trivial. Specifically, De Lucia et al. [12] used the Rocchio algorithm before and after LSI, i.e., decomposing the term-by-document matrix, while Salton and Buckley [37] reported that the Rocchio Algorithm is not as competitive on the probabilistic model as on VSM. Since the main focus of this paper is not to optimize ARF on LSI and JS, to avoid any biases, we only compare CLUSTER with ARF through VSM. For the other three baselines, the comparison with CLUSTER is based on all three IR models. To find out whether CLUSTER is able to improve IR-based traceability recovery with a small amount of user feedback, for UD-CSTI and ARF we make all candidate links to be verified by users to reach the best performance of the two approaches (default settings according to related papers [13, 14]), while CLUSTER still use the stopping criterion that the verification stops when five false positives are met. Based on this comparison, we expect to find out whether a small amount of user feedback with the help of closeness analysis on code can reach or even exceed the improvement brought by verifying the entire candidate lists. Because we assume that the user always makes correct verifications (discussed in Section III.D), we simulate this verification process by referring to the known RTM of the evaluated systems for CLUSTER, UD-CSTI and ARF.

Besides the proposed metrics in Section IV.B, we used a statistical significance test to check whether the performance of CLUSTER is significantly better than the performance of the baseline approaches. By consulting the significance test used in [10] and [31], we use the F-measure at each recall point as the single dependent variable of our study. We use the F-measure because we want to know whether CLUSTER improves both precision and recall. The F-measure is computed as:

$$F = \frac{2P \times R}{P+R} \quad (7)$$

where P represents precision and R represents recall and F is the harmonic mean of P and R. A higher F-measure means that both precision and recall are high. We then use the Wilcoxon rank sum test [32] to test the following null hypothesis:

H_0: There is no difference between the performance of CLUSTER and baseline approaches

TABLE III. THE NUMBER OF COMPUTED AP, MAP, P-VALUE, AND CLIFF'S D EVALUATING EACH APPROACH FOR ALL TWENTY EXPERIMENT VARIATIONS (EVALUATED SYSTEMS ITRUST, MAVEN, PIG, GANTTPROJECT, AND INFINISPAN COMBINED WITH IR MODELS VSM, LSI, AND JS)

		VSM				**JS**				**LSI**			
		AP	***MAP***	***p-value***	***Cliff's delta***	***AP***	***MAP***	***p-value***	***Cliff's delta***	***AP***	***MAP***	***p-value***	***Cliff's delta***
iTrust	IR-ONLY	42.55	56.55	**<0.01**	0.29	38.28	55.99	**<0.01**	0.31	41.59	54.63	**<0.01**	0.28
	UD-CSTI	45.75	59.08	**<0.01**	0.22	43.26	62.99	**<0.01**	0.19	46.04	58.26	**<0.01**	0.18
	TRICE	45.12	58.46	**<0.01**	0.24	44.49	60.77	**<0.01**	0.19	44.63	56.67	**<0.01**	0.23
	ARF	44.12	58.35	**<0.01**	0.31				-				**-**
	CLUSTER	**54.12**	**65.72**	-	**-**	**49.08**	**64.31**	**-**	-	**51.08**	**63.07**	-	**-**
Maven	IR-ONLY	14.54	29.46	**<0.01**	0.51	15.86	35.03	**<0.01**	0.47	14.25	35.13	**<0.01**	0.44
	UD-CSTI	15.14	30.40	**<0.01**	0.46	17.95	36.72	**<0.01**	0.35	15.46	36.65	**<0.01**	0.32
	TRICE	14.15	29.16	**<0.01**	0.49	15.62	35.44	**<0.01**	0.46	14.51	35.52	**<0.01**	0.39
	ARF	**24.25**	31.34	0.86	0.01	**-**	**-**	**-**	-	**-**	**-**	**-**	**-**
	CLUSTER	22.96	**38.07**	**-**	**-**	**23.09**	**39.74**	**-**	-	**20.36**	**39.79**	**-**	**-**
Pig	IR-ONLY	22.05	42.38	**<0.01**	0.23	15.94	35.65	**<0.01**	0.25	20.39	41.83	**<0.01**	0.19
	UD-CSTI	24.25	43.52	**<0.01**	0.15	19.26	37.92	**<0.01**	0.15	23.05	42.99	**0.03**	0.09
	TRICE	16.91	41.75	**<0.01**	0.26	113.94	36.68	**<0.01**	0.21	16.08	41.01	**<0.01**	0.21
	ARF	**35.07**	44.95	0.56	0.03	**-**	**-**	**-**	-	**-**	**-**	**-**	-
	CLUSTER	24.75	**45.12**	**-**	**-**	**20.34**	**39.81**	**-**	-	**22.95**	**44.38**	**-**	-
Gantt Project	IR-ONLY	43.17	49.79	**<0.01**	0.39	36.50	46.76	**<0.01**	0.41	43.94	51.70	**<0.01**	0.38
	UD-CSTI	47.86	55.66	**<0.01**	0.24	45.91	58.39	**<0.01**	0.26	50.23	58.58	**<0.01**	0.16
	TRICE	46.86	53.81	**<0.01**	0.32	39.95	49.50	**<0.01**	0.38	43.39	51.91	**<0.01**	0.32
	ARF	**56.16**	60.71	**<0.01**	0.22	**-**	**-**	**-**	-	**-**	**-**	**-**	**-**
	CLUSTER	55.41	**65.46**	**-**	**-**	**56.30**	**63.83**	**-**	-	**54.39**	**63.53**	**-**	**-**
Infinispan	IR-ONLY	8.23	23.51	**<0.01**	0.20	6.43	24.25	**<0.01**	0.18	8.62	25.00	**<0.01**	0.22
	UD-CSTI	9.31	24.18	**<0.01**	0.12	7.80	25.22	**0.02**	0.05	10.70	26.14	**<0.01**	0.06
	TRICE	6.65	22.79	**<0.01**	0.24	5.72	23.82	**<0.01**	0.18	7.32	23.77	**<0.01**	0.24
	ARF	**13.58**	24.75	**<0.01**	0.14	**-**	**-**	**-**	-	**-**	**-**	**-**	-
	CLUSTER	11.47	**26.22**	**-**	**-**	**9.67**	**26.87**	**-**	-	**12.63**	**28.29**	**-**	-

We use $\alpha = 0.05$ to accept or refute the null hypothesis. We also use a non-parametric effect size measure for ordinal data, i.e., Cliff's *d* [38], to compute the magnitude of the effect of our approach compared to the baseline approaches as follows:

$$d = \left| \frac{\#(x_1 > x_2) - \#(x_1 < x_2)}{n_1 n_2} \right| \qquad (8)$$

where x_1 and x_2 are F-measure values of CLUSTER and the baseline approaches, and n_1 and n_2 are the sizes of the sample groups. The effect size is considered small for *d* in the range [0.15, 0.33), medium for *d* in the range [0.33, 0.47) and large if *d* equals or is larger than 0.47.

V. RESULTS AND DISCUSSION

Table III shows the results of the five evaluated systems (rows). For each system and each IR model (columns), we compared the performance of four baseline approaches with CLUSTER. We leveraged the introduced performance metrics AP and MAP (sub column 1 and 2). Sub column 3 shows the p-value of the F-measure significance test for CLUSTER and sub column 4 shows the Cliff's *d*. In 48 out of 50 cases, the F-measure for the results of CLUSTER is significantly higher than the F-measure of the compared four baseline approaches (*p-value* < 0.05) at each level of recall, indicating that CLUSTER significantly outperforms baseline approaches in most cases. Specifically, CLUSTER outperforms IR-ONLY, UD-CSTI, and TRICE on both AP and MAP in all cases. When compared with ARF, CLUSTER performs worse in AP (0.89 on average) but outperforms in MAP (4.43 on average). We need to point out that the performance of CLUSTER is achieved by tolerating only five candidate links verified as false positives for each requirement, while the performance of UD-CSTI and ARF relies on all inks in the candidate lists to be verified. In particular, the average user-verified links over all requirements for iTrust, Maven, Pig, Infinispan, and GanttProject are 6.10, 5.74, 5.72, 5.62, and 8.48, respectively. The ratio between user-verified classes and all classes in the code (i.e., "executed classes" in Table II) ranges from 1.45% to 6.84% for each requirement. This observation demonstrates that by combining with closeness analysis on code dependencies, a small amount of user feedback can be amplified and propagated to improve IR-based traceability. We argue that this observation is very beneficial to our approach for improving IR-based traceability recovery because in practice the user feedback can be valuable but frugal. Figure 3 shows and compares the precision-recall curves for the four approaches grouped by each system and IR model.

We now use the adapted excerpt from the Maven system (discussed in Section III) to demonstrate why CLUSTER is able to outperform TRICE and UD-CSTI with a small amount of user feedback. As we discussed, TRICE only uses the top-ranked class inside a code region from the candidate list for each requirement as the input. So according to Figure 2 and Table II, TRICE will start its bonusing only from the second-ranked `CacheUtils` which is in the created Region1. So its improvement is quite limited. To make things worse, if the initial IR values are not reliable, TRICE will give bonuses to candidate links that are actually irrelevant to a given requirement and thus even decrease its performance. This can be observed from Table III and Figure 3 that when the initial IR results are low (e.g., Pig-VSM), TRICE performs the worst compared to the other four approaches. In contrast, CLUSTER uses both the candidate region and initial IR values of classes in the region to locate candidate links for users to verify. It further asks the user to iteratively verify multiple classes from different candidate regions. In the Maven sample, Region 1, Region 2 and Region3 will be verified as relevant regions. So even the class with small IR values, such as `ArtifactClassRealmConstituent`, can also get multiple bonuses because it has code dependencies to

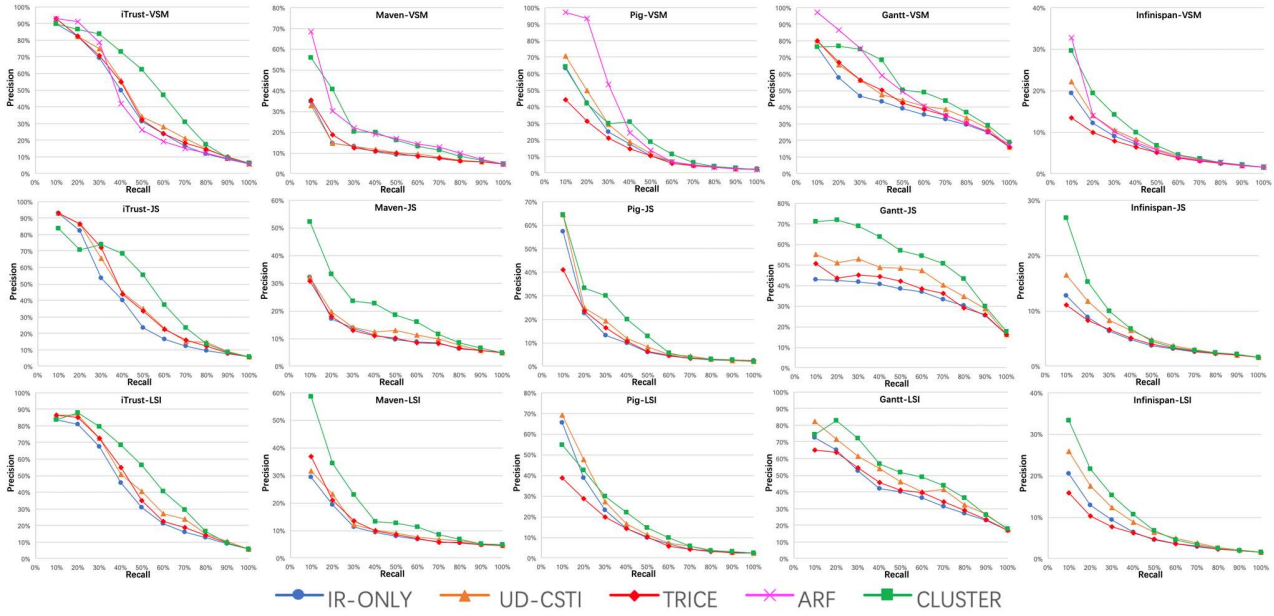

Fig. 3. Precision/Recall curves grouped by evaluated systems (iTrust, Maven, Gantt, Infinispan, Pig) and IR models (VSM, LSI, JS)

all relevant regions. Furthermore, the candidate links verified as false positives are also used to improve the ranking of candidate lists by giving penalties to IR values of unverified links. On the other hand, although UD-CSTI is able to eventually bonus all relevant classes based on the code dependency analysis without making big mistakes, it requires great manual efforts to verify lots of candidate links, especially for giving bonuses to the relevant class with small IR values. On the contrary, CLUSTER is able to amplify user-verified links through the candidate regions through the closeness analysis on code dependencies.

We now focus on comparing the differences between the number of false positives made by CLUSTER, IR-ONLY, and ARF at different levels of recall for twenty experiment variations (shown in Table IV). The sub column 1 (denoted as IR-ONLY) shows that CLUSTER is able to bring a large reduction of 4250 retrieved false positives at the 80% level of recall on Infinispan-LSI. This saves notable efforts to the user when using CLUSTER instead of IR-ONLY. Such an improvement is particularly evident when the recall is between 20% and 80%. Furthermore, we compare CLUSTER with ARF that uses all verified links as its input (shown in sub column 2, denoted as ARF). We found that in the majority of cases, CLUSTER is still able to make less false positives compared to ARF, especially around recall levels between 40% to 80%. The overall observation demonstrates that CLUSTER is useful to save great manual efforts for the user who aims to recover requirement-to-code traces by IR-based approaches in practice. However, we also noticed that ARF is able to achieve good precision at the lower level of recall (less than twenty percent). It is even more interesting that ARF can perform better on the systems with low text quality (e.g., Maven and Pig) instead of those with good text quality (e.g., iTrust). The reason might be that ARF will not apply its adaptive Rocchio Algorithm unless the text length of queries (requirements) is smaller than the text length of documents (classes). The use cases in iTrust are well documented with text size larger than the code texts in most cases. So ARF can hardly be activated when applied to iTrust. On contrary, the issue texts in Maven and Pig is short compared to their class texts. Thus, ARF is able to adjust the text vectors created from these issues multiple times. To further combine our approach with ARF is one of our future work.

We made four additional observations. First, the results of CLUSTER can vary with different IR models for the same system. The reason is that re-ranking IR candidate lists in CLUSTER relies on giving bonuses to generated IR values. These values can be very different if they are computed by different IR models. Second, we also tried to let all created candidate regions verified to optimize CLUSTER. We found that for all five systems, the biggest increase of performance can be achieved when the user is asked to verify about thirteen classes from different candidate regions. So if the user aims to recover more relevant traces at a higher level of recall (such as 60%-80%), we suggest she verify around thirteen classes on CLUSTER for a given requirement. Third, the increase of the performance brought by CLUSTER is not significant for Infinispan and Pig (see related Cliff's delta in Table III). This is probably because the requirements we elicited from the filtered and merged issues in IlmSeven dataset are too fine-grained compared to the requirements maintained by developers of iTrust and GanttProject. This can also be observed from Table II where the numbers of average classes that traces to a requirement for iTrust and GanttProject are 8 and 12, while for Infinispan and Pig the numbers are 6 and 5, respectively. In future work we plan to use text clustering techniques to further enhance the quality of derived traces from issues linked to commit logs, which is relevant to the work proposed by Palomba et al. [30]. Fourth, we found that although CLUSTER is built on three different perspectives, i.e., textual similarities, code structural information, and user feedback, our approach still cannot achieve acceptable precision at higher level recall (e.g., higher than 80%), even if we use all verified candidate links as input on the evaluated systems with good text quality (e.g.,

TABLE IV. REDUCED FALSE POSITIVES AT DIFFERENT LEVELS OF RECALL (CLUSTER COMPARED WITH IR-ONLY AND ARF)

		Recall (20%)		Recall (40%)		Recall (60%)		Recall (80%)		Recall (100%)	
		IR-ONLY	*ARF*	*IR-ONLY*	*ARF*	*IR-ONLY*	*ARF*	*IR-ONLY*	*ARF*	*IR-ONLY*	*ARF*
		FP	*FP*	*FP*	*FP*	*FP*	*FP*	*FP*	*FP*	*FP*	*FP*
iTrust	VSM	**-6**	0	**-65**	**-103**	**-375**	**-529**	**-578**	**-519**	**-189**	**-145**
	JS	+3	-	**-123**	-	**-532**	-	**-736**	-	**-257**	-
	LSI	**-7**	-	**-74**	-	**-395**	-	**-419**	-	**-57**	-
Maven	VSM	**-233**	**-62**	**-504**	**-110**	**-646**	**-3**	**-822**	+385	+60	0
	JS	**-160**	-	**-483**	-	**-775**	-	**-770**	-	+5	-
	LSI	**-115**	-	**-390**	-	**-915**	-	**-705**	-	**-7**	-
Pig	VSM	**-29**	+64	**-409**	**-173**	**-1456**	**-1312**	**-1393**	**-661**	+208	**-6**
	JS	**-145**	-	**-964**	-	**-1164**	-	**-645**	-	+110	-
	LSI	**-45**	-	**-493**	-	**-941**	-	**-690**	-	**-118**	-
Gantt Project	VSM	**-27**	+9	**-121**	**-43**	**-159**	**-92**	**-226**	**-195**	**-294**	**-131**
	JS	**-67**	-	**-136**	-	**-178**	-	**-294**	-	**-169**	-
	LSI	**-21**	-	**-93**	-	**-137**	-	**-266**	-	**-109**	-
Infinispan	VSM	**-1124**	**-815**	**-2641**	**-2221**	**-4172**	**-2472**	**-4143**	**-1847**	+17	+35
	JS	**-1556**	-	**-3742**	-	**-3027**	-	**-2295**	-	**-368**	-
	LSI	**-1123**	-	**-3671**	-	**-4241**	-	**-4250**	-	**-2**	-

iTrust). This situation can also be observed in many IR-based traceability recovery approaches [8-16]. We plan to refine CLUSTER in future work to help users recover the majority of relevant traces by discarding the least false positives.

VI. THREATS TO VALIDITY

Internal Threats. A possible threat is the incomplete code dependencies due to the incompleteness nature of dynamic analysis. However, compared with static analysis, our approach prefers concise but correct descriptions of system behavior (e.g., to correctly handle polymorphism) to work because CLUSTER mainly targets functional requirements. We further tried to alleviate this problem by extensively executing the test cases or sample systems maintained by original developers. Meanwhile, we treat class call dependency, class inheritance, and class usage as the same direct code dependencies and we treat class data dependency as a different kind based on their structures in the CDCGraph (discusses in Section III.A). We further investigated how these code dependencies overlap with each other. We found that for all evaluated systems the class inheritance and class usage are 100% overlapped with class call dependencies. The reasons are: (1) for class inheritance, the constructor of a derived class will eventually call the constructor of its base class; (2) for class usage, a class usually uses the fields of another class through the get-set methods unless the fields are publicly visible, which is rare in practice. On the contrary, only 3% to 23% of class data dependencies overlap with direct code dependencies for the evaluated systems, indicating a significant difference. Finally, by consulting previous work [11-14, 39], the user feedback we used in this paper is simulated by referring to the known RTM of the evaluated systems. Evaluating how CLUSTER works under real-world user verification process is in our future work.

External Threats. At this moment we only collected code dependencies and linked issues with acceptable qualities on three out of seven systems in the IlmSeven dataset. However, we consider our findings still relevant since we evaluated five real-world systems in total that are either widely studied or used in practice from different domains (iTrust: J2EE medical care system, Maven: project dependency management, Pig: big-data processing, Infinispan: distributed in-memory data store, and GanttProject: project planning). Furthermore, we combined the evaluated systems with three mainstream IR models (i.e., VSM, LSI, and JS) to extend our experiments to twenty variations in total (e.g., iTrust-VSM and Pig-JS). Meanwhile, because of our heuristics to filter and merge issues from the IlmSeven dataset, our evaluation is only based on part of RTMs of Maven, Pig, and Infinispan. However, we use the same RTMs to compare our approach with the baselines for each system, thus making no bias. The experiment results of pure IR-based approaches on Maven, Pig and Infinispan shows that our heuristics can create traces from their issues with acceptable text quality. Finally, we re-implemented all baseline approaches because we cannot find available implementations. However, all related papers provided detailed algorithms and discussions [10, 13, 14] that help us to avoid big deviations when re-implementing baseline approaches.

VII. CONCLUSIONS AND FUTURE WORK

User feedback on candidate links created by IR technique has been used to improve IR-based traceability recovery. However, the performance of these approaches is highly dependent on the number of user-verified links. Thus, we use the closeness measure to quantify the degree of interactions for code dependencies. We then proposed an approach that combines the closeness analysis with user feedback to improve IR-based traceability recovery. An empirical study based on five real-world systems showed that our approach uses a small amount of user feedback to outperform four baselines. Our future work is to find out how to guide users to recover most relevant traces with least candidate links to be verified when using IR-based recovery approaches. The dataset in this paper is now publicly available at: https://dataverse.harvard.edu/dataverse/CLUSTER.

ACKNOWLEDGMENT

We are funded by the National Natural Science Foundation of China (Grant Nos. 61690204 and 61802173), the Collaborative Innovation Center of Novel Software Technology and Industrialization, the German Ministry of Education and Research (BMBF) grant: 01IS16003B and by DFG grant: MA 5030/3-1, and Pro2Future, a COMET K1-Centre of the Austrian Research Promotion Agency (FFG), grant no. 854184.

REFERENCES

[1] CoEST: Center of excellence for software traceability, http://www.CoEST.org

[2] P. Rempel and P. Mäder, "Preventing Defects: The Impact of Requirements Traceability Completeness on Software Quality," IEEE Transactions on Software Engineering (TSE), vol. 43, no. 8, pp. 777-797, 2017.

[3] P. Mäder and A. Egyed, "Assessing the effect of requirements traceability for software maintenance," in the 28th IEEE International Conference on Software Maintenance (ICSM), Riva del Garda, Italy, 2012, pp. 171-180.

[4] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo, "Recovering Traceability Links between Code and Documentation," IEEE Transactions on Software Engineering(TSE), vol. 28, no. 10, pp. 970-983, 2002.

[5] A. Marcus and J I. Maleti, "Recovering documentation-to-source-code traceability links using latent semantic indexing," in the 25th IEEE International Conference on Software Engineering (ICSE), 2003, pp. 125-135.

[6] A. Abadi, M. Nisenson, and Y. Simionovici, "A Traceability Technique for Specifications," in the Proceedings of the 16th IEEE International Conference on Program Comprehension (ICPC), 2008, pp. 103-112.

[7] J. Cleland-Huang, R. Settimi, C. Duan and X. Zou, "Utilizing supporting evidence to improve dynamic requirements traceability," in the 13th IEEE International Conference on Requirements Engineering (RE), 2005, pp.135-144.

[8] M. Gethers, R. Oliveto, D. Poshyvanyk, and A. De Lucia, "On integrating orthogonal information retrieval methods to improve traceability recovery," in the 27th IEEE International Conference on Software Maintenance (ICSM), 2011, pp. 133-142.

[9] C. McMillan, D. Poshyvanyk, and M. Revelle, "Combining textual and structural analysis of software artifacts for traceability link recovery," in Proceedings of the International Workshop on Traceability in Emerging Forms of Software Engineering, 2009, pp. 41–48.

[10] H. Kuang, J. Nie, H. Hu, P. Rempel, J. Lü, A. Egyed, and P. Mäder, "Analyzing closeness of code dependencies for improving IR-based Traceability Recovery," in the 24th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), 2017, pp. 68-78.

[11] J. H. Hayes, A. Dekhtyar, and S. K. Sundaram, "Advancing candidate link generation for requirements tracing: The study of methods," IEEE Transactions on Software Engineering(TSE), vo1. 32, no. 1, pp. 4–19, 2006.

[12] A. De Lucia, R. Oliveto, and P. Sgueglia, "Incremental approach and user feedbacks: a silver bullet for traceability recovery," in Proceedings of the 22nd IEEE International Conference on Software Maintenance(ICSM), 2006, pp. 299–309.

[13] A. Panichella, C. McMillan, E. Moritz, D. Palmieri, R. Oliveto, D. Poshyvanyk, and A. De Lucia, "When and How Using Structural Information to Improve IR-Based Traceability Recovery", in the 17th European Conference on Software Maintenance and Reengineering (CSMR), 2013, pp. 199-208.

[14] A. Panichella, A. De Lucia, and A. Zaidman, "Adaptive user feedback for ir-based traceability recovery," in Proceedings of 8th International Symposium on Software and Systems Traceability (SST), 2015, pp. 15–21.

[15] M. Di Penta, S. Gradara, G. Antoniol, "Traceability Recovery in RAD Software Systems," in Proceedings of 10th International Workshop on Program Comprehension, Paris, France, 2002, pp. 207-216.

[16] J. Guo, J. Cheng, and J. Cleland-Huang, "Semantically Enhanced Software Traceability Using Deep Learning Techniques," in Proceedings of the 39th International Conference on Software Engineering(ICSE), 2017, pp. 3-14.

[17] G. Antoniol, G. Casazza, and A. Cimitile, "Traceability Recovery by Modelling Programmer Behaviour," in Proceedings of 7th Working Conference on Reverse Engineering, Brisbane, Queensland, Australia, 2002, pp. 240-247.

[18] Denys Poshyvanyk, Yann- Gaël Guéhéneuc, Andrian Marcus, Giuliano Antoniol, "Václav Rajlich: Feature Location Using Probabilistic Ranking of Methods Based on Execution Scenarios and Information Retrieval," in IEEE Trans. Software Eng, 2007, vol. 33, no. 6, pp. 420-432.

[19] B. Dit, M. Revelle, D. Poshyvanyk, "Integrating information retrieval, execution and link analysis algorithms to improve feature location in software," in Empirical Software Engineering(EMSE), 2013, vol. 18, no. 2, pp. 277-309.

[20] A. De Lucia, M. Di Penta, R. Oliveto, A. Panichella, and S. Panichella, "Improving ir-based traceability recovery using smoothing filters", in Proceedings of the 19th International Conference on Program Comprehension(ICSM), 2011, pp. 21–30.

[21] J. Rocchio, Relevance feedback in information retrieval, G. Salton, Ed. Englewood Cliffs, NJ: Prentice-Hall. 1971.

[22] B. Burgstaller and A. Egyed, "Understanding where requirements are implemented," in 26th IEEE International Conference on Software Maintenance (ICSM), Timişoara, Romania, 2010, pp. 1-5.

[23] H. Kuang, P. Mäder, H. Hu, A. Ghabi, L. Huang, J. Lü, and A. Egyed, "Can method data dependencies support the assessment of traceability between requirements and source code?", Journal of software: Evolution and Process (J. Softw. Evol. and Proc.), 2015, vol. 27, no. 11, pp. 838–866.

[24] H. Kuang, J. Nie, H. Hu, and J.Lü, "Improving Automatic Identification of Outdated Requirements by Using Closeness Analysis Based on Source Code Changes," in Software Engineering and Methodology for Emerging Domains: the proceedings of the 15th National Software Application Conference (NASAC English Track), 2016, pp. 52-67.

[25] iTrust System: http://agile.csc.ncsu.edu/iTrust/wiki/doku.php

[26] Maven System: http://maven.apache.org/

[27] Pig System: https://pig.apache.org/

[28] M. Rath, P. Rempel, and P. Maeder, "The IlmSeven Dataset", in the Proceedings of the 25th IEEE International Requirements Engineering Conference (RE), 2017, pp. 516 - 519

[29] Jira Tool: https://www.atlassian.com/software/jira

[30] F. Palomba, P. Salza, A. Ciurumelea, S. Panichella, H. Gall, F. Ferrucci, and A. De Lucia, "Recommending and localizing change requests for mobile apps based on user reviews," in Proceedings of the 39th International Conference on Software Engineering (ICSE), 2017, pp. 106-117.

[31] N. Ali, Z. Sharafi, and Y. Gueheneuc, "An empirical study on the importance of source code entities for requirements traceability," Empirical Software Engineering, 2014, vol. 20, no. 2, pp. 442-478.

[32] W. J. Conover, Practical Nonparametric Statistics (3rd edn). Wiley: Hoboken, New Jersey, USA, 1998.

[33] Infinispan System: http://infinispan.org/

[34] R. Baeza-Yates and B. Ribeiro-Neto, "Modern information retrieval," New York: ACM press, 1999.

[35] GanttProject System: http://www.ganttproject.biz

[36] A. Egyed, F. Graf, and P. Grunbacher, "Effort and quality of recovering requirements-to-code traces: Two exploratory experiments," in 18th IEEE International Conference on requirements engineering (RE), 2010, pp. 221–230.

[37] G. Salton and C. Buckley, "Improving Retrieval Performance by Relevance Feedback", Jour. of the American Society for Information Science, vol. 41, no.4, 1990, pp. 288-297.

[38] G. Macbeth, E. Razumiejczyk, and R. D. Ledesma, "Cliff's delta calculator: a non-parametric effect size program for two groups of observations", Univ Psychol 10(2):545–555, 2011

[39] Y. Lin, J. Sun, Y. Xue, Y. Liu, and J. Dong, "Feedback-based debugging," in Proceedings of the 39th International Conference on Software Engineering(ICSE), 2017, pp. 3-14.

[40] Y. Lin, G. Meng, Y. Xue, Z. Xing, J. Sun, X. Peng, Y. Liu, W. Zhao, and J. Dong, "Mining implicit design templates for actionable code reuse". In Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2017), 2017, pp. 394-404.