Analyzing Closeness of Code Dependencies for Improving IR-based Traceability Recovery

Hongyu Kuang[1], Jia Nie[1], Hao Hu[1], Patrick Rempel[2], Jian Lü[1], Alexander Egyed[2], Patrick Mäder[3]

[1]State Key Lab for Novel Software Technology
Nanjing University
Nanjing, Jiangsu, China
{hector.khy, niejia.nju}@gmail.com
{myou, lj}@nju.edu.cn

[2]Institute for Software Systems Engineering
Johannes Kepler University
Linz, Austria
patrick.rempel@jku.at
alexander.egyed@jku.at

[3]Fakultät für Informatik und Automatisierung
Technische Universität Ilmenau
Ilmenau, Germany
patrick.maeder@tu-ilmenau.de

***Abstract*—Information Retrieval (IR) identifies traces based on textual similarities among software artifacts. However, the vocabulary mismatch problem between different artifacts hinders the performance of IR-based approaches. A growing body of work addresses this issue by combining IR techniques with code dependency analysis such as method calls. However, so far combined approaches considered each code dependency as equally helpful for traceability recovery, not taking full advantage of the code dependency analysis. In this paper, we combine IR techniques with closeness analysis on code dependencies to improve IR-based traceability recovery. Specifically, we quantify and utilize the "closeness" for each call and data dependency between two classes to improve rankings of traceability candidate lists. An empirical evaluation based on three real-world systems suggests that our approach outperforms baseline approaches.**

Keywords—Traceability Recovery, Information Retrieval, Closeness Analysis, Call Dependencies, Data Dependencies

I. INTRODUCTION

Software traceability is defined 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]. Traceability links can support stakeholders in development-related tasks. In a previous study [2] we discovered that requirements-to-code traceability strongly benefits developers in performing software maintenance tasks. We found that subjects with traceability data performed on average 24% faster on a maintenance task and created on average 50% more correct solutions as compared to maintenance tasks where traceability was not available. These findings were based on correct and complete traceability. Unfortunately, recent work has suggested that high quality traceability links are difficult to obtain [3-5] due to the typically large numbers of required traces, frequent changes in software artifacts (especially in code), and informal nature of the relationships.

Aiming at reducing the manual effort in traceability recovery by providing semi-automated tools, Information Retrieval (IR) has become perhaps the most widely accepted and applied technique in recent traceability research [4-18]. Typical IR-based approaches compute the textual similarity between two software artifacts (e.g., requirements and code) through IR models including Vector Space Model (VSM) [6], Latent Semantic Indexing (LSI) [7], and the probabilistic Jensen and Shannon model (JS) [8]. These approaches provide users with an automatically generated set of candidate traceability links to help narrow down the search space for potential links between artifacts. Due to the informal nature of textual software requirements and a typical vocabulary mismatch problem between requirements and code, performance of those pure IR-based approaches is significantly hindered. To address this issue, researchers have successfully improved IR-based approaches in a variety of different aspects, e.g., lexical analyses including hierarchies and clusters of artifact texts [4], feedback from users [4, 10], topic modeling [5], and code authors' contexts [9].

Meanwhile, another body of work focused on combining IR-based techniques with code dependency analysis to improve traceability recovery [13, 14, 16] and similar researches such as feature location [15], and concept location [17]. This kind of IR-based approaches utilizes structural information of the source code to be traced to complement textual analysis and has proven to be very useful [12-18]. In the following, we refer to this type of approach as *combined IR-based approach*. In general, a combined IR-based approach will first locate a set of candidate links by using IR techniques, and then either expand or filter the initial set of candidate links based on analyzing dependencies among code elements (e.g., classes or methods). Recent work successfully improved IR-based approaches by either introducing advanced code dependency analyses such as the PageRank algorithm [17] and by combining code analyses with user feedback [13].

However, we argue that these approaches did not fully explore available code information for two reasons. First, previous approaches solely considered direct code dependencies, i.e., calling relationships, class inheritance, and class usage. However, these approaches neglect the similarly important indirect code dependencies, i.e., data dependencies that exist when two code elements read or manipulate the same data [11, 19]. Second, previous approaches treated all code dependencies as equally important. We argue that this

assumption is not always true as demonstrated in the following example.

Consider a medical care system and a class named `MonitorAdverseEvent`. This class has calling relationships with to two other classes: `MonitorAdverseEventAction` and `AuthDAO`. The first calling relationship is abstracting three distinct method calls between the two classes while the second one is abstracting only one method call. Furthermore, `MonitorAdverseEventAction` is the only callee of `MonitorAdverseEvent`, while `AuthDAO` will also be called by other classes. Comparing the two calling relationships, the former represents a stronger interaction between two classes than the latter. This observation has already been used as important heuristics in improving automated techniques for extracting class refactoring based on class cohesion and class coupling analysis [23-25].

In this paper, we propose that *the degree of interaction for each code dependency between two given classes in the code is essential for further improving combined IR-based approaches.* With our previous finding showing that indirect data dependencies in the code are also helpful for understanding requirements-to-code traceability [11, 19], we developed a code dependency concept, called *closeness*, to quantify the degree of interaction based on direct and indirect code dependencies among classes.

Based on this closeness measure, we further propose a combined IR-based approach for traceability recovery. This approach first utilizes IR techniques to generate candidate links between requirements and source code classes. The result is a ranked list of candidate links. This ranking is then changed in two steps: (1) we combine the results of IR and closeness analysis for all candidate links, and (2) we enhance the candidate list by propagating links to code classes that have a high closeness measure to classes that are identified in the IR analysis. Eventually, the set of candidate links is re-ranked according to the combined information of IR and code analysis. We evaluated our approach in an empirical study and found that our approach statistically significant outperforms pure IR-based approaches (VSM, JS, and LSI) as well as two previously proposed combined IR-based approaches [14, 17]. These results were obtained on three real-world systems.

The remainder of this paper is structured as follows. Section II introduces the research background and discusses the related work. Section III presents our proposed approach. Section IV introduces our research questions and how we set up the experiments based on three software systems for answering those questions. Section V reports the results of our experiments and answers the research questions. Section VI refers to limitations and threats of our work. Finally, Section VII concludes this paper.

II. BACK GROUND AND RELATED WORK

In this section we discuss related work in IR-based and combined traceability approaches.

IR techniques are a widely studied and applied technology in traceability research [4-18]. However, an important issue hindering the performance of IR techniques when applied to traceability recovery is the vocabulary mismatch problem between source and target artifacts (e.g., requirements and code). This problem remains the focus of ongoing research in the field with various approaches proposed. For example, Cleland-Huang et al. [4] presented three strategies to enhance the matching results generated by their IR model based on probabilistic networks. The key idea of those strategies is introducing extra information when matching requirements (such as the section name of a given requirement) and code elements (such as the package name of a given class or method) and to exclude keywords promoting wrongly retrieved traces. Diaz et al. [9] extracted additional "author contexts" (code snippets that are commented by the same author) from code to improve IR-based traceability recovery. When executing a query with a high-level artifact, first the principle developer of the artifact is located based on the textual similarity between the query and the author context. Then rankings of classes authored by this principle developer will be increase by an adaptive bonus in the IR candidate list. Gethers et al. [5] proposed an IR-based approach that integrates orthogonal information generated by relational topic modeling, which defines a comprehensive method for modeling interconnected networks of documents in order to achieve a complementary effect for improving traceability recovery. However, these approaches require rich requirements descriptions and well-documented code, which in practice is not always the case. Furthermore, these approaches did not consider dependencies among code elements.

In earlier work [3, 22], we studied calling relationships between methods and found that requirements are typically implemented in connected areas of the source code rather than being randomly distributed. In a follow-on study [11, 19] we found that indirect method data dependencies complement method call dependencies in understanding requirements traceability. The previous work formed the foundation for the proposed approach in this paper.

There is work that incorporates code dependency analysis to improve IR-based approaches for traceability recovery and related research topics [12-18]. The majority of these approaches focused on analyzing direct code dependencies (i.e., method calls, class inheritance, and class usage). Scanniello et al. [17] introduced the PageRank algorithm to compute relative importance for each code method based on their direct code dependencies in the combined IR-based approach for concept location. The methods in IR candidate lists are then re-ranked by the product of their IR values and their relative importance values. Panichella et al. [13] extended their previous work on combining direct code and IR analysis [14] by utilizing user feedback, which had previously been demonstrated helpful for traceability recovery [4, 10]. Once, a user confirms a link in the IR candidate list between a requirement and a method, the rankings of methods connected to the chosen method by direct code dependencies will increase by an adaptive bonus. Two previous studies extended code analysis to indirect data dependencies [12, 16]. McMillan et al. [12] created an approach called Exemplar to find highly relevant software projects (similar to traceability recovery) from large archives of applications based textual similarity between user queries and project descriptions, the API calls used in the projects, and the

data flow among these API calls. Eaddy et al. [16] developed a so-called Prune Dependency Analysis based on dependencies among methods, fields, and types in their approach which also combines IR and execution tracing techniques.

Our approach is different from the discussed ones in two aspects: (1) we analyze both direct and indirect code dependencies in the code while most combined IR-based approaches focused on one type solely (except for [12, 16]), and (2) we propose a closeness measure for each code dependency while other approaches treated all code dependencies similarly. An exception is [17] where authors propose to compute each method's relative importance based on its topology in a code dependency graph. However, in the graph all code dependencies are still treated equally.

III. PROPOSED APPROACH

We propose a four-step approach. First, we capture and organize code dependencies between classes (Step 1). Second, we calculate *closeness* for captured code dependencies and built a graph structure, called *Code Dependency Graph (CDCGraph)*, which combines captured code dependencies and their calculated closeness measures (Step 2). Third, we use IR techniques to generate candidate links between requirements and classes (Step 3). Fourth, we re-rank and enhance the candidate list generated in the previous step based on the code analysis (Step 4). Each step is explained in more detail in the following subsections. We use adapted excerpts of the iTrust system [20] for illustration of relevant concepts.

A. Step 1: Capturing and Organizing Code Dependencies

1) Code dependencies among classes

We consider four kinds of dependencies between classes: class call dependencies, class inheritance, class usage, and class data dependencies. A call dependency between two classes C_a and C_b means that there is at least one method call between C_a and C_b. Figure 1 shows an iTrust excerpt covering three classes: `EmailUtil`, `MonitorAdverseEventAction`, and `SendMessageAction`. A class usage exists between `MonitorAdverseEventAction` and `EmailUtil`. In method `MonitorAdverseEventAction.sendEmail()`, an object of type `Email` is initialized and passed through method calls to `SendMessageAction.saveReceiver()` and `EmailUtil.sendEmail()`. No call, class inheritance, and class usage dependencies exist between `EmailUtil.sendEmail()` and `SendMessageAction.saveReceiver()`.

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 piece of information [11]. Our focus currently restricts to data that is read and manipulated at runtime in an application's program memory. However, one could extend that concept to other data, e.g., stored in the file system or on the web. Methods may access that data directly or via a transitive chain of aliases or pointers. This complex formulation is necessary as the same underlying data is often accessed through references or even chains of references, which are less obviously visible than direct code dependencies. Figure 1 shows a data dependency example. There are the obvious usage and call dependencies between `MonitorAdverseEventAction` and

```
class MonitorAdverseEventAction{
   private EmailUtil emailUtil;…
   public String sendEmail() {
     Email mail = new Email();
     SendMessageAction messenger
              = new SendMessageAction();
     messenger.saveReceiver(mail);
     emailUtil.sendEmail(mail);
     …}
}
class EmailUtil{
   public void sendEmail(Email email){
     factory.getFakeEmailDAO()
              .sendEmailRecord(email);
   }…
}
class SendMessageAction{
   private MessageDAO messageDAO;…
   public void saveReceiver (Email info){
     messageDAO.addEmailMessage
              (info.getReceiver());
     …}
}
```

Fig. 1. Code snippets from iTrust illustrating code dependencies

`EmailUtil`. However, there exists also a less obvious class data dependency between `SendMessageAction` and `EmailUtil`. `SendMessageAction.saveReceiver()` and `EmailUtil.sendEmail()` both take the same `Email` object as a parameter. Thus, all three classes (including `MonitorAdverseEventAction` which also accesses the `Email` object) access the same data object implying that all three are data dependent based on the `Email` data type.

2) Capturing code dependencies

To capture the discussed four kinds of code dependencies, we used our previously proposed dynamic analysis tool [11] (available at http://www.sea.jku.at/tools), which relies on JVMTI (Java Virtual Machine Tool Interface) to capture method-level call and data dependencies. We decided to use this tool: (1) since it instruments the JVM (Java Virtual Machine) during runtime, we capture actually executed code dependencies and handle polymorphism correctly; (2) we capture all dependencies simultaneously by running test cases in a single test run; and (3) potentially missing code dependencies caused by incomplete testing do not jeopardize our approach (further discussed in Section VI).

Based on the captured method-level dependencies, we derive the discussed four kinds of class-level code dependencies. First, class call dependencies are abstracted from method call dependencies with the number of distinct method calls having the same calling direction. Second, class data dependencies are abstracted from method data dependencies and keep all data types in those method data dependencies (e.g., the `Email` data type). Third, class usages are abstracted from method data dependencies. Finally, class inheritance dependencies are retrieved through method call dependencies, i.e., the constructor of a derived class calling its base class's constructor.

3) Organizing code dependencies

By consulting typical combined IR-based approaches (e.g., [14] and [17]), we treat class call dependency, class inheritance, and class usage as one kind of code dependencies (namely *direct code dependencies*) while the captured class data dependencies as a different one. The reason why combining the first three code dependencies is that these kinds of code dependencies are structurally similar (directed links from source classes to sink classes) while our captured *class data dependencies* are different in structure (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 (more details refer to Section VI). So we calculate closeness measures for these two kinds of code dependencies separately.

Figure 2 depicts samples of captured direct code dependencies and class data dependencies. In the figure a direct code dependency is represented by a solid line with arrow and is labeled with the number of method calls and/or class usages while a data dependency is represented by a dashed line without arrow which is labeled with the number of shared data types.

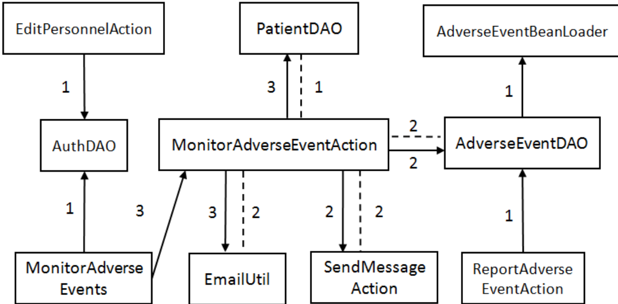

Fig. 2. Samples of captured class call (solid lines with arrow) and class data dependencies (dashed lines) between classes

B. Step 2: Calculating Closeness and Creating CDCGraph

We now calculate the closeness measure for captured code dependencies. After that we create our *Code Dependency with Closeness Graph (CDCGraph)* for the follow-up steps.

1) Calculating closeness for direct code dependencies

In the introduction, we gave an example to demonstrate that different code dependencies can indicate different degrees of interactions between classes. The two discussed direct code dependencies between the three involved classes: `MonitorAdverseEventAction`, `MonitorAdverseEvents`, and `AuthDAO` are shown in Figure 2. It seems intuitive that if two classes share multiple distinct method calls and class usages among each other then these two classes interact more closely. Less straightforward, but equally important, is the sink's in-degree (number of classes which reach the sink) and source's out-degree (number of classes which are reached by the source) in a direct code dependency. Specifically, a smaller sink's in-degree indicates that the sink class is more concentrate to serve the source class, instead of providing a common service. Meanwhile, smaller source's out-degree means that the source class focuses more on the service provided by the sink class. Based on these two observations, we define $Closeness_{DC}$ for direct code dependencies:

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

where N represents the number of distinct methods calls and class usages from a given direct code dependency between two classes, $WeightedInDegree_{Sink}$ represents sink's in-degree, and $WeightedOutDegree_{Source}$ represents source's out-degree. Both are weighted by the number of methods and class usages, i.e., N of each related direct code dependency.

For the example in Figure 2, the direct code dependency from `MonitorAdverseEvents` to `AuthDAO` has a closeness of 2*3 / ((1 + 3) + 3) which equals to 0.86, while the direct code dependency from `MonitorAdverseEvents` to `MonitorAdverseEventAction` has a closeness of 2*1 / ((1 + 1) + (1 + 3)) which equals to 0.33. The former is higher than the latter, showing that `MonitorAdverseEvents` interacts more closely with `MonitorAdverseEventAction` than with `AuthDAO` based on our analysis.

2) Calculating closeness for class data dependencies

As we discussed in Section III.A, part 1, a data dependency between two classes exists due to shared data types among each other. The three shared data types in Figure 2 are shown in the rows of Table I. The "Occurrences" column shows how often a data type occurred across all class data dependencies of the iTrust system. The table shows that the data type `DAOFactory` is occurring much more often than the other two data types. A closer look at the source code shows that this class is responsible for all database accesses of iTrust and shared by many classes of the system (a typical J2EE pattern), indicating that `DAOFactory` is too general to be helpful for analyzing closeness between classes. Therefore, the closeness of classes based on data dependencies should consider the importance of each shared data type. We introduce a weighting factor called *Inverse Data Type Frequency (idtf)* [11] as:

$$\text{idtf} = \log \frac{N}{n_{dt}} \quad (2)$$

where N is the total number of captured data dependencies and n_{dt} is the occurrence of a given data type in all data dependencies. The calculated idtf values for each data type in the sample are also shown in Table I (the value of N is 4844).

TABLE I. DATA TYPES SHARED IN THE EXAMPLE OF FIGURE 2

#	Data type	Occurrences	idtf Value
1	`Email`	9	2.7310
2	`Java.lang.String`	1118	0.6368
3	`DAOFactory`	4478	0.0341

By setting a $Threshold_{idtf}$, commonly shared data types, such as `DAOFactory` in the example, do not negatively impact the analysis result anymore. Data types with idtf values lower than the $Threshold_{idtf}$ will be ignored and if all data types in a class data dependency are ignored the whole data dependency is ignored for analysis. Threshold calibration will be discussed in Section IV. Assuming a threshold of 0.6 for the example in

Figure 2 the data dependency between `PatientDAO` and `MonitorAdverseEventAction` would be ignored.

Like the idf concept used in IR [21], the idtf value reflects how much a data type is shared globally across the source code. A higher idtf means that a data type is more uniquely shared between two classes, indicating a stronger interaction. Besides, for a class data dependency between two 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 these two classes (from other data dependencies) can represent how these two classes share data types with each other "locally", like the tf concept [21] in IR. The higher ratio represents a more diversified data sharing between two classes.

We define $Closeness_{CD}$ for class data dependencies as:

$$Closeness_{CD} = \frac{\sum_{x \in \{DT_i \cap DT_j\}} idtf(x)}{\sum_{y \in \{DT_i \cup DT_j\}} idtf(y)} \quad (3)$$

where *idtf(x)* represents the idtf value for data types larger than $Threshold_{idtf}$ and DT_i and DT_j represent the sets of all data types that C_i and C_j share respectively.

For the example in Figure 2, the class data dependency between `MonitorAdverseEventAction` and `EmailUtil` has a closeness of 2.73 / (2.73 + 0.64) which equals to 0.81, while the class data dependency between `AdverseEventDAO` and `MonitorAdverseEventAction` has a closeness of 0.64 / (0.64 + 2.73) which equals to 0.19. The former is higher than the latter, showing that `MonitorAdverseEventAction` interacts more closely with `EmailUtil` than with `AdverseEventDAO` based on class data dependencies.

3) Generating the CDCGraph

With all details described above, we can now create a Code Dependency Graph (CDCGraph) as G=<V, E>. Each vertex V represents a class of the analyzed source code and being annotated with its name. Furthermore, we distinguish 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. Furthermore, each code dependency is annotated with the calculated closeness measure. A derived CDCGraph based on the sample code of Figure 2 is shown in Figure 3.

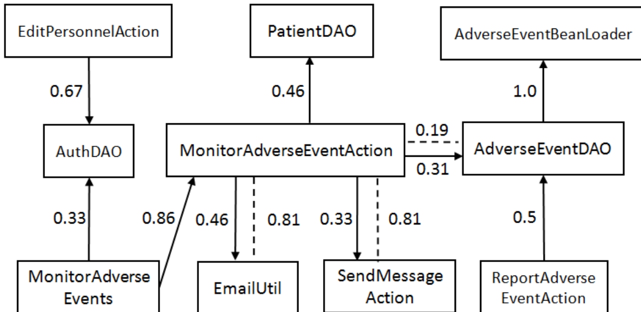

Fig. 3. Sample CDCGraph showing direct code dependencies as solid arcs with arrow and class data dependencies as dashed arcs, both annotated with the computed closeness measure

C. Step 3: Generating IR candidate lists

We use IR techniques to generate traceability link candidates between a given requirement and the code. Specifically, we perform the following steps to gain traceability link candidates:

- *Creating corpus.* Each class of the source code is extracted into one document including its identifiers, name, method names, property names, and comments. For each requirement we extract a document that includes its title and content (e.g., preconditions, main-flow, and sub-flows if structured use case and pure text otherwise).
- *Normalizing corpus.* The documents of both requirements and classes are normalized by standard pre-processing techniques 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 different IR models to compute textual similarity: Vector Space Model (VSM) [6], Latent Semantic Indexing (LSI) [7], and the probabilistic Jensen and Shannon (JS) model [8].
- *Generating candidate links.* In candidate lists per requirement, we rank classes by their textual similarity to a requirement in descending order.

Table II shows part of a candidate list generated using VSM for one use case UC36 ("Monitor adverse event") of the iTrust system and the ten classes also shown in Figure 3. The list is ranked based on probability in descending order. An 'X' in column "Is trace" marks an actual trace between a class and the use case based on the oracle of correct traces for iTrust.

TABLE II. A SAMPLE CANDIDATE LIST BETWEEN REQUIREMENT UC36 AND TEN CLASSES OF THE ITRUST SYSTEM AFTER STEP 3

#	Class	IR Value	Is Trace
1	MonitorAdverseEventAction	0.3492	X
2	ReportAdverseEvent	0.2723	
3	MonitorAdverseEvents	0.2568	X
4	EditPersonnelAction	0.2349	
5	AdverseEventDAO	0.2286	X
6	AdverseEventBeanLoader	0.1926	X
7	SendMessageAction	0.1338	X
8	EmailUtil	0.0964	X
9	PatientDAO	0.0529	X
10	AuthDAO	0.0369	

D. Step 4: Reordering Candidate Lists based on Closeness

We aim to improve the initial IR-based ranking traceability candidate lists based on the computed closeness measure. This step is inspired by the following two findings: (1) top ranked classes in IR candidate lists are likely to be traced to a given requirement [4]; and (2) requirements are implemented in connected areas of the source code, so-called requirement regions, rather than being randomly distributed [3]. Based on these two observations, we developed a re-rank algorithm involving two steps. First, we compute an *initial requirement region* in the CDCGraph and promote all classes in the region.

Second, we re-rank other classes that are not in the initial region by giving bonuses based on their code dependencies to classes that are in the initial region.

1) Computing initial requirement region

To establish an initial requirement region for a given requirement, we choose the top ranked class of the IR candidate list as seed for the initial requirement region. We then identify close classes based on code dependencies in CDCGraph. This region is being established for each requirement using two strategies based on two kinds of code dependencies. Considering direct code dependencies only, if the top ranked class reaches (or is reached by) other classes through direct code dependencies with closeness measures equal or higher than a Threshold_{DC}, we add these sinks (or sources) in the initial region and restart the same process from each newly added class until no more sinks (or sources) are added. Meanwhile, considering class data dependencies only, classes with data dependencies that have closeness measures equal or higher than Threshold_{CD} and directly relate to the top ranked class will be added to the initial requirement region. If the top ranked class has no neighboring classes through either direct code dependencies or class data dependencies with closeness measures higher than the two thresholds, we put the next ranked class from the candidate list in the initial region and repeat the process until we find a class with neighbor classes. The IR values of newly added neighbor classes will be set to the same value of the top ranked class with neighbor classes (other than top ranked class without neighbor classes). We are using the following algorithm to establish initial requirement region based on IR candidate lists and CDCGraph:

Algorithm 1 Establishing Initial Region

```
initialRegion ← ∅;
prunedGraph ← CDCGraph.setPruning(Threshold_DC,Threshold_CD);
topLink ← candidateList.next();
while prunedGraph.hasNoNeighbors(topLink.class) do
    initialRegion.add(topLink.class);
    topLink ← candidateList.next();
end
initialRegion.add(topLink.class);
initialRegion.topIRValue ← topLink.IRValue;
reachedClasses ← ∅;
reachedClasses.add(prunedGraph.getTransitiveCallers(topLink.class));
reachedClasses.add(prunedGraph.getTransitiveCallees(topLink.class));
reachedClasses.add(prunedGraph.getNeighborsByData(topLink.class));
foreach link in candidateList do
    if reachedClasses.contains(link.class) then
        link.IRValue ← initialRegion.topIRValue;
        initialRegion.add(link.class);
    end
end
candidateList.reorderByIRValue();
```

To show how Algorithm 1 works, for the example candidate list in Table II, the top ranked class for requirement UC36 is `MonitorAdverseEventAction`. We assume a Threshold_{DC} of 0.7 and a Threshold_{CD} of 0.8. According, `MonitorAdverseEvents` will be added to the initial region based on direct code dependencies while `EmailUtil` and `SendMessageAction` are added to the region based on indirect class data dependencies (see also Figure 3). The reordered IR candidate list after this step is shown in Table III.

TABLE III. REORDERED CANDIDATE LIST BETWEEN REQUIREMENT UC36 AND TEN CLASSES OF THE ITRUST SYSTEM AFTER ESTABLISHING INITIAL REQUIREMENTS REGION (ADAPTED RANKING VALUES IN BOLD)

#	Class	Ranking Value	Is Trace
1	`MonitorAdverseEventAction`	0.3492	X
2	`MonitorAdverseEvents`	**0.3492**	X
3	`SendMessageAction`	**0.3492**	X
4	`EmailUtil`	**0.3492**	X
5	`ReportAdverseEvent`	0.2723	
6	`EditPersonnelAction`	0.2349	
7	`AdverseEventDAO`	0.2286	X
8	`AdverseEventBeanLoader`	0.1926	X
9	`PatientDAO`	0.0529	X
10	`AuthDAO`	0.0369	

2) Re-ranking candidate links outside initial region

We now re-rank candidate links outside initial requirement regions by enhancing IR values of those links based on how their classes interact with classes in the initial regions according to the CDCGraph and their original IR values. Similar to the previous sub-step, we use two different strategies on direct code dependencies and class data dependencies separately for the enhancement.

Considering direct code dependencies only, for an outside candidate link we start from its class C_{out} to traverse the CDCGraph. We try to find a path from C_{out} to a class C_{in} that is already in the initial region. A valid path needs to satisfy two conditions: (1) it can only have one direction, meaning C_{out} transitively reaches or is transitively reached by C_{in}; (2) it cannot contain more than one class in initial region (to avoid duplicates). If a valid path is found, we calculate the geometric mean of the closeness measures for all direct code dependencies in this path and use the following formula to recalculate IR value for C_{out} (IR_{DC}):

$$IR_{DC} = IR_{origin} + (IR_{top} - IR_{origin}) \sqrt[|PATH|]{\prod_{x \in PATH} Closeness_{DC}(x)} \quad (4)$$

where IR_{origin} represents the original IR value of C_{out}, IR_{top} represents the promoted IR value of C_{in}, $PATH$ represents the set of direct code dependencies in a discovered path between C_{out} and C_{in}, and $Closeness_{DC}(x)$ represents the closeness measure for each direct code dependency in the path. It is possible that there are multiple paths to the same C_{in} and we only keep the one to maximize the enhanced IR value.

On the other hand, considering class data dependencies only, if the class of an outside link C_{out} can directly connect to a class C_{in} in the initial region, we use the following formula to recalculate bonuses based on class data dependencies (IR_{CD}):

$$IR_{CD} = IR_{origin} + \left(IR_{top} - IR_{origin}\right) Closeness_{CD}(x) \quad (5)$$

where IR_{origin} represents the original IR value of C_{out}, IR_{top} represents the promoted IR value of C_{in}, and $Closeness_{CD}(x)$ represents the closeness measure of the class data dependency that directly connect C_{in} and C_{out}.

If paths to multiple C_{in}s (or multiple direct neighboring classes in the initial region) are found based on direct code dependencies (or class data dependencies), the IR value of the candidate link can be increased multiple times by taking the enhanced value as the original one. The IR value of the outside

candidate link can be increased through both direct code dependencies and class data dependencies but not higher than C_{in}'s IR value. We are using the following algorithm to reorder candidate links outside initial requirement region:

Algorithm 2 Re-rank Links outside Initial Region

```
topIRValue ← initialRegion.topIRValue;
foreach link in candidateList do
    if !initialRegion.contains(link.class) then
        foreach c in initialRegion do
            pathList ← findValidPaths(link.class, c);
            gMean ← 0;
            foreach path in pathList do
                gMean ← max(GeometricMean(Closeness_DC(path)),
                gMean);
            end
            link.IRValue ← link.IRValue + gMean
            × (topIRValue − link.IRValue);
            if hasDataDependencies(c, link.class) then
                link.IRValue ← link.IRValue + (topIR-
                Value - link.IRValue) × Closeness_CD(c,
                link.class);
            end
        end
        if link.IRValue > topIRValue then
            link.IRValue ← topIRValue;
        end
    end
end
candidateList.reorderByIRValue();
```

To show how Algorithm 2 works, For the example in Figure 3, `AdverseEventDAO` is outside the initial region. This class has one path to `MonitorAdverseEventAction` and is also directly connected to it by a class data dependency. So IR_{DC} for `AdverseEventDAO` is 0.23 + (0.35 - 0.23) * 0.31 which equals to 0.27. Furthermore, IR_{CD} for this class is 0.27 + (0.35 - 0.27) * 0.19 which equals to 0.28. Meanwhile, there is a path from `MonitorAdverseEventAction` to `AdverseEventBeanLoader` containing two direct code dependencies. The geometric mean of the closeness measures for the two dependencies is 0.56 and IR_{DC} for `AdverseEvent BeanLoader` is 0.19 + (0.35 - 0.19) * 0.56 which equals to 0.28. The re-ranked list after this sub-step is shown in Table IV.

TABLE IV. REORDERED CANDIDATE LIST BETWEEN REQUIREMENT UC36 AND NINE CLASSES OF THE ITRUST SYSTEM AFTER REORDERING LINKS OUTSIDE INITIAL REGION (ADAPTED RANKING VALUES SHOWN IN BOLD)

#	Class	IR Value	Is Trace
1	`MonitorAdverseEventAction`	0.3492	X
2	`MonitorAdverseEvents`	0.3492	X
3	`SendMessageAction`	0.3492	X
4	`EmailUtil`	0.3492	X
5	`AdverseEventDAO`	**0.2818**	X
6	`AdverseEventBeanLoader`	**0.2798**	X
7	`ReportAdverseEvent`	0.2723	
8	`EditPersonnelAction`	0.2349	
9	`PatientDAO`	**0.1892**	X
10	`AuthDAO`	**0.1400**	

IV. EXPERIMENTAL SETUP

In this section, we introduce our experimental setup to evaluate the proposed approach. In Section IV.A, we introduce the three evaluated systems and discuss the three studied IR models. In Section IV.B, we define metrics for evaluating the performance of the proposed approach. In Section IV.C we refer to threshold calibration for the proposed approach and finally in Section IV.D we discuss our research questions and the design of experiments.

A. Evaluated Systems and IR Models

Our evaluation is based on three real-world, medium-sized software systems: iTrust [20], GanttProject [27], and jHotDraw [28]. Table V lists basic metrics about the three systems. The three systems comprised 160 kLoC. We chose these systems because of the availability of requirements specifications and, more significantly, "gold standard" requirements-to-code traces. For jHotDraw and GanttProject, we gained high quality requirements-to-code traces by recruiting the original developers. For iTrust high quality requirements-to-code traces were already available [20]. The RTM of iTrust contains method-level traces while the RTMs of jHotDraw and GanttProject are at class-level. To keep our experiment consistent at the same trace granularity, we propagated the method-level traces of iTrust to class-level. This means that we aggregated all traces to methods of a class on the class-level. To ensure the generalizability of our results, we involved three well-accepted IR models, namely VSM, LSI, and JS.

TABLE V. OVERVIEW OF THE THREE EVALUATED SYSTEMS

	iTrust [20]	Gantt Project[27]	jHotDraw [28]
Version	13.0	2.0.9	7.2
Programming language	Java	Java	Java
KLoC	43	45	72
Executed classes	131	124	144
Evaluated requirements	34	16	16
Direct code dependencies	274	452	691
Class data dependencies	4844	1788	1815
Trace links in RTM	248	315	221

B. Metrics

To evaluate the performance of different traceability recovery approaches, we leveraged two well-known metrics:

$$\text{recall} = \frac{|\text{correct} \cap \text{retrieved}|}{|\text{correct}|}\% \qquad \text{precision} = \frac{|\text{correct} \cap \text{retrieved}|}{|\text{retrieved}|}\% \qquad (6)$$

where *correct* represents the set of correct links and *retrieved* is the set of all links retrieved by traceability recovery approaches. A common way for evaluating the performance of IR methods is to compare the precision values obtained at different recall levels resulting in a set of precision-recall points displayed as graphs. We further leveraged the following two metrics to measure the approach's performance: average precision (AP) and mean average precision (MAP). These metrics are widely used to evaluate IR-based approaches for traceability recovery (e.g., [13] and [8]). AP measures how well relevant documents of all queries (requirements) are ranked to the top of the retrieved links and it is computed as:

TABLE VI. NUMBER OF COMPUTED METRICS AND *P*-VALUE EVALUATING THE PERFORMANCE OF EACH APPROACH FOR ALL NINE EXPERIMENT VARIATIONS

		VSM			LSI			JS		
		AP	***MAP***	***p-value***	***AP***	***MAP***	***p-value***	***AP***	***MAP***	***p-value***
iTrust	IR-ONLY	45.18	58.69	**0.02**	45.47	59.92	**0.03**	42.21	56.90	**< 0.01**
	TRICE	**49.21**	**61.65**	-	**49.12**	**62.70**	-	**49.33**	**62.61**	-
	PageRank	42.42	54.03	0.13	39.07	49.47	0.82	30.76	39.31	**< 0.01**
	O-CSTI	37.24	44.87	0.63	34.17	42.02	0.14	27.24	34.80	**< 0.01**
Gantt Project	IR-ONLY	42.85	49.80	**< 0.01**	**43.94**	51.70	**< 0.01**	36.30	46.77	**0.01**
	TRICE	**46.42**	**54.00**	-	43.56	**52.40**	-	**40.11**	**49.70**	-
	PageRank	43.34	48.84	**< 0.01**	41.18	45.38	**< 0.01**	39.76	43.98	**< 0.01**
	O-CSTI	42.84	47.70	**< 0.01**	38.06	41.18	**< 0.01**	35.80	39.20	**< 0.01**
JHotDraw	IR-ONLY	41.13	50.09	0.35	42.23	49.51	**0.02**	37.06	44.86	**0.01**
	TRICE	**43.24**	**52.05**	-	**44.51**	**51.91**	-	**39.20**	**47.10**	-
	PageRank	26.12	29.24	**< 0.01**	22.48	22.52	**< 0.01**	18.24	18.28	**< 0.01**
	O-CSTI	21.74	23.00	**< 0.01**	18.95	20.23	**< 0.01**	18.57	19.92	**< 0.01**

$$\mathrm{AP} = \frac{\sum_{r=1}^{N}(\mathrm{Precision}(r) \times \mathrm{isRelevant}(r))}{|\mathrm{RelevantDocuments}|} \quad (7)$$

where r is the rank of the target artifact in an ordered list of links, *Precision(r)* denotes its precision value, *isRelevant()* is a binary function assigned 1 if the link is relevant and 0 otherwise, and N is the total number of documents. Meanwhile, MAP is the mean of the AP scores over a set of queries (requirements) and is computed across all queries as follows:

$$\mathrm{MAP} = \frac{\sum_{q=1}^{Q} AP(q)}{q} \quad (8)$$

where q is a single query and Q is the total number of queries. For more comprehensive evaluations we use both AP and MAP to study our research questions.

C. Threshold Calibration

In Section III, we introduced four thresholds: $\mathrm{Threshold}_{idtf}$, $\mathrm{Threshold}_{DC}$, $\mathrm{Threshold}_{CD}$, and the k value for LSI. Based on previous investigations [11] and case study results, we defined a fixed $\mathrm{Threshold}_{idtf}$ (1.4). To calibrate $\mathrm{Threshold}_{DC}$, 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 $\mathrm{Closeness}_{DC}$. We then used min-max normalization to rescale closeness measures into [0, 1]. Filtered abnormally high closeness measures were set to 1 and abnormally low closeness measures were set to 0 in the rescaled range. We defined a fixed threshold for all nine experiment variations to maximize the performance of our approach based on the proposed metrics in Section IV.B. We used the same process to find $\mathrm{Threshold}_{CD}$. This led to a $\mathrm{Threshold}_{DC}$ of 0.7 and a $\mathrm{Threshold}_{CD}$ of 0.9. These values are adaptive thresholds for establishing the initial requirement region for a given requirement only. We still use original closeness measures to re-rank candidate links outside the initial region. For the k value of the LSI method, we found that k = 85 provided the best accuracy for all three systems.

Since we use the same fixed four thresholds for all nine experiment variations, they are not biased by varying thresholds. For new datasets we propose the discussed calibration process to identify thresholds.

D. Research Question

In this paper, we aim to study if combined IR-based approaches for traceability recovery can be improved by considering additional code dependencies. Therefore, we formulated the following research question:

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

To study RQ, we compared the results of our approach with three baseline approaches: a pure IR-based approach (namely IR-ONLY), the pioneer of combined IR-based approaches *Optimistic Combination of Structural and Textual Information* (*O-CSTI* [14]), and the most recent one using *PageRank* [17]. We name our proposed approach as *TRICE (**T**raceability **R**ecovery based on **I**nformation retrieval and **C**los**E**ness analysis)*.

Besides the proposed metrics in Section IV.B, we used a statistical significance test to verify that the performance of TRICE is significantly better than the performance of the baseline approaches. By consulting the significance test used in [29], 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 TRICE improves both precision and recall. The F-measure is computed as:

$$\mathrm{F} = \frac{2\mathrm{P}\times\mathrm{R}}{\mathrm{P}+\mathrm{R}} \quad (9)$$

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. Because the F-measure is the same for each approach at the same level of recall (i.e., the data were paired), we decided to use the Wilcoxon rank sum test [26] to test the following null hypothesis:

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

We use α = 0.05 to accept or refute the null hypothesis.

V. RESULTS AND DISCUSSION

Table VI shows the results of the three evaluated systems (rows). For each system and each IR technique (columns), we compared the performance of the three baseline approaches with TRICE (sub rows). Therefore, 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. The p-value results indicate that TRICE

TABLE VII. IMPROVEMENT OF PRECISION AND REDUCTION OF NUMBER OF FALSE POSITIVES (IN BOLD) AT DIFFERENT LEVELS OF RECALL (COMPARISON BETWEEN TRICE AND IR-ONLY)

		Recall (20%)		Recall (40%)		Recall (60%)		Recall (80%)		Recall (100%)	
		Precision	*FP*	*Precision*	*FP*	*Precision*	*FP*	*Precision*	*FP*	*Precision*	*FP*
VSM	VSM	**+8.64%**	**-6**	**+5.12%**	**-13**	**+3.00%**	**-89**	**+2.74%**	**-280**	**+0.43%**	**-275**
	JS	**+18.58%**	**-13**	**+21.37%**	**-64**	**+5.76%**	**-196**	**+2.57%**	**-445**	0.00%	0
	LSI	**+1.46%**	**-1**	**+4.52%**	**-12**	**+3.40%**	**-86**	**+4.39%**	**-425**	**+0.33%**	**-234**
Gantt Project	VSM	**+7.86%**	**-14**	**+6.03%**	**-36**	**+3.81%**	**-52**	**+0.62%**	**-28**	0.00%	+1
	JS	**+1.75%**	**-6**	**+5.33%**	**-37**	**+1.11%**	**-15**	-0.46%	+13	0.00%	0
	LSI	-1.94%	+3	**+2.92%**	**-20**	**+3.24%**	**-45**	**+2.63%**	**-83**	**+0.28%**	**-30**
jHotDraw	VSM	**+1.95%**	**-2**	**+4.16%**	**-20**	**+0.91%**	**-14**	-3.81%	+185	0.00%	0
	JS	-0.68%	+1	**+6.97%**	**-31**	**+2.54%**	**-42**	**+2.52%**	**-142**	**+0.06%**	**-13**
	LSI	-2.20%	+2	**+4.66%**	**-21**	**+3.54%**	**-51**	**+0.23%**	**-12**	**+0.15%**	**-34**

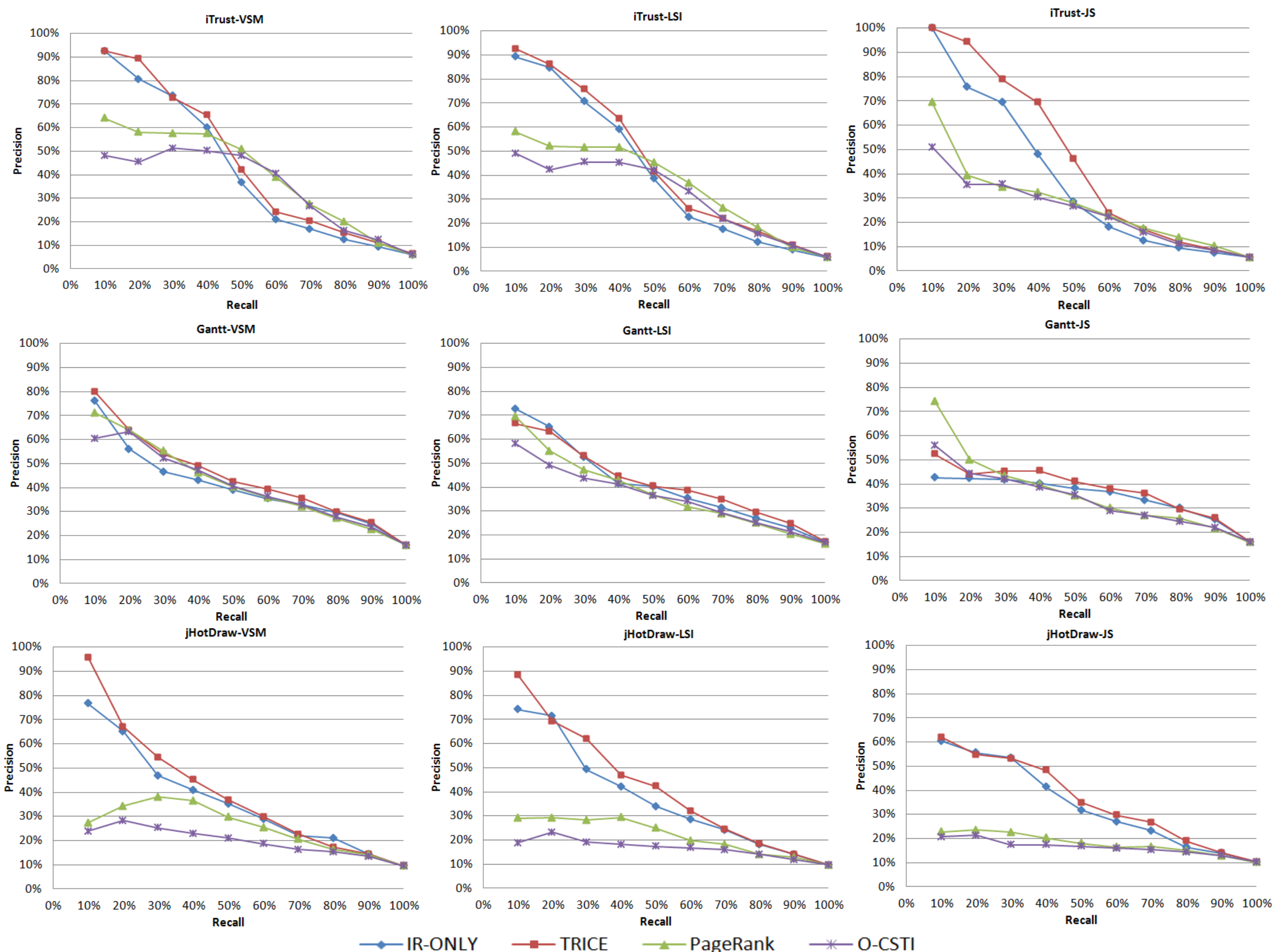

Fig. 4. Precision/Recall curves for all nine experiment variances grouped by evaluated systems (iTrust, Gantt, jHotDraw) and IR models (VSM, LSI, JS)

outperforms the baseline approaches in most cases. In 22 of 27 cases the F-measure for results of TRICE is significantly higher than the F-measure of the compared baseline approach (p-value < 0.05). Moreover, TRICE generally outperforms all three baseline approaches in both AP and MAP. The only exception is comparing with IR-ONLY on Gantt-LSI where TRICE is slightly worse in AP (less than 0.5). Figure 4 compares the precision/recall curves for all nine experiments, illustrating the performance of the four approaches. The results are grouped by evaluated system and IR model.

As shown in both Table VI and Figure 4, our experiments did not provide any evidence that O-CSTI and PageRank can significantly outperform the IR-ONLY approach. Thus, we focus on comparing TRICE with IR-ONLY. Table VII shows the differences between the precision values and the differences of the number of false positives achieved with

TRICE and IR-ONLY at different levels of recall for nine experiment variations. From Figure 4 and Table VII we observe an improvement of precision up to 21.37% (on iTrust-JS at 40% recall) and an improvement of reduced false positives up to 445 (on iTrust-JS at 80% recall). We also observed that TRICE barely introduced extra false positives compared to IR-ONLY at each recall level except for one place where extra 185 false positives were introduced: jHotDraw-VSM at 80% recall. These findings are very beneficial because by adding our closeness metric (closeness measures and re-rank algorithms) to IR-based traceability recovery, the performance of the combined approach can be effectively improved with few extra false positives introduced.

We made two additional observations. First, TRICE makes fewer improvements at 100% recall compared to other recall levels. This observation is similar with those from other case studies (such as [9, 13]) and confirms that there is an upper bound to the performance improvements when aiming at recovering all correct links, which is difficult to overcome. The other observation is that TRICE can make bigger improvements if it is based on IR candidate lists with higher quality, (e.g., iTrust-JS against jHotDraw-VSM), implying that TRICE would further benefit from improvements for IR-based traceability recovery and even collaborate with those.

Overall, the results indicate that our proposed closeness analysis is useful to improve IR-based traceability recovery. Such an improvement is particularly evident when the recall is between 20% and 80%

VI. Threats to Validity

A possible threat to validity of our results is the incompleteness of code dependencies because of missed call and data dependencies due to the incompleteness nature of dynamic analysis. However, we consider this incompleteness not as a serious threat regarding our experiment results. Missing dependencies could have negatively impacted TRICE leading to too pessimistic observations instead.

In Section III.A we mentioned that our approach treats class call dependencies, class usage and class inheritance as one kind of code dependency (direct code dependency) since they are structurally similar to each other. Furthermore, we also found that these three kinds of code dependencies largely overlap with each other based on our investigations to all evaluated systems. First of all, since class inheritance is retrieved through method call dependencies, i.e., the constructor of a derived class calling its base class's constructor, it is totally overlapped with class call dependency. Second, although class usage is abstracted from method data dependencies, in iTrust 113 class usages (119 in total) overlap with class call dependencies (268 in total); in jHotDraw 105 class usages (105 in total) overlap with class call dependencies (691 in total); and in GanttProject 85 class usages (165 in total) overlap with class call dependencies (372 in total). This observation is reasonable because if a class is using another class as its field then the former generally calls the methods of the latter unless the fields of the latter are directly accessible to the former, which is not common in practice (GanttProject is a little different since its source code contains several inner classes). In contrast, in iTrust 140 class data dependencies (4844 in total) overlap with direct code dependencies (274 in total); in jHotDraw 415 class data dependencies (1815 in total) overlap with direct code dependencies (691 in total); and in GanttProject 296 class data dependencies (1788 in total) overlap with direct code dependencies (452 in total). This investigation provides support for our approach to treat direct code dependencies and class data dependencies separately.

Another possible threat to validity of our results is the selection of evaluated systems. It is difficult to find software systems that are executable, contain rich requirements for IR techniques to work, that have existing traceability links, and are freely accessible. For example, we cannot use the well-researched data sets provided by CoEST [1] (such as EasyClinic and eTour) to evaluate our approach because they do not contain executable versions of their systems and we cannot capture high-quality code dependencies from these systems. We cannot claim generalizability of results to different kinds of systems based on the three medium-sized systems. However, we consider our findings still relevant since we evaluated three real-world systems from different domains (iTrust: J2EE medical care system, GanttProject: project planning, and jHotDraw: drawing tool). Furthermore, we combined the evaluated systems with three well-known yet distinct IR models (i.e., VSM, LSI, and JS) to generated nine experiment variations in total (e.g., iTrust-JS and jHotDraw-VSM). We then compare our approach with baseline approaches based on these experiment variations.

VII. Conclusions and Future Work

Dependencies in source code (e.g., method calls) have been repeatedly used in solutions to improve IR-based traceability recovery. However, these approaches focused on direct code dependencies for traceability recovery and treated each code dependency equally. In this paper, we considered additionally class data dependencies and proposed a closeness measure to quantify the degree of interaction between two classes. We further proposed an approach to improve IR-based traceability recovery by incorporating the closeness measure. We evaluated our approach on three software systems and found that it outperforms three baseline approaches significantly.

Currently, our approach is analyzing direct code dependencies and class data dependencies first separately and then combined. In future work we want to find out whether certain combinations (or patterns) on different kinds of code dependencies will also be helpful for traceability recovery. Another promising research direction is to combine our closeness analysis with user feedback for IR-based approaches.

Acknowledgment

We are funded by the 973 Program of China grant 2015CB352202 and the National Natural Science Foundation of China (NSFC) grants: 91318301, 61321491, 61100037, 61100038, 61472177; by the German Ministry of Education and Research (BMBF) grants: 16V0116, 01IS14026A; and the Austrian Science fund (FWF) grant: P 23115-N23.

References

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

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

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

[4] 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.

[5] 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.

[6] 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), 28(10), pp. 970-983, 2002.

[7] A. Marcus and J I. Maletic, "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.

[8] 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.

[9] D. Diaz, G. Bavota, A. Marcus, R. Oliveto, S. Takahashi, and A. D. Lucia: "Using code ownership to improve IR-based traceability link recovery," in the 21st International Conference on Program Comprehension (ICPC), 2013, pp. 123-132.

[10] 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.

[11] 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, Volume 27, Issue 11, pp. 838–866.

[12] C. McMillan, M. Grechanik, D. Poshyvanyk, C. Fu, Q. Xie, "Exemplar: A Source Code Search Engine For Finding Highly Relevant Applications," IEEE Transactions on Software Engineering (TSE), 99, 2011.

[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 CSMR, 2013, pp. 199-208

[14] 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.

[15] W. Zhao, L. Zhang, Y. Liu, J. Sun, and F. Yang, "SNIAFL: Towards a Static Noninteractive Approach to Feature Location," ACM Transactions on Software Engineering and Methodology (TOSEM), 15(2), pp. 195-226, 2006.

[16] A. V. Aho, Marc Eaddy, Giuliano Antoniol, Yann-Gaël Guéhéneuc, "CERBERUS: Tracing Requirements to Source Code Using Information Retrieval, Dynamic Analysis, and Program Analysis," in 16th IEEE International Conference on Program Comprehension (ICPC), Amsterdam, The Netherlands, 2008, pp. 53-62.

[17] G. Scanniello, A. Marcus, D. Pascale, "Link analysis algorithms for static concept location: an empirical assessment", in Empirical Software Engineering (EMSE), pp. 1–55, 2014.

[18] E. Hill, L. Pollock, and K. Vijay-Shanker, "Exploring the Neighborhood with Dora to Expedite Software Maintenance", in the 22th IEEE/ACM international conference on Automated software engineering (ASE), Atlanta, Georgia, 2007, pp. 14-23.

[19] H. Kuang, P. Mäder, H. Hu, A. Ghabi, L. Huang, J. Lv, and A. Egyed, "Do data dependencies in source code complement call dependencies for understanding requirements traceability?", in 28th IEEE International Conference on Software Maintenance (ICSM), 2012, pp.181-190.

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

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

[22] A.Ghabi and A. Egyed. "Code patterns for automatically validating requirements-to-code traces", in the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE). New York, NY, USA, 2012; 200–209

[23] M. Fowler, "Refactoring: improving the design of existing code", Pearson Education India, 1999.

[24] G. Bavota, A. De Lucia, A. Marcus, R. Oliveto, "Automating Extract Class Refactoring: an Improved Method and its Evaluation", Empirical Software Engineering (EMSE), 2013, Volume 19, Issue 6, pp. 1617-1664.

[25] G. Bavota, A. De Lucia, and R. Oliveto, "Identifying Extract Class Refactoring Opportunities Using Structural and Semantic Cohesion Measures", The Journal of Systems and Software (JSS), 2011, pp. 397-414.

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

[27] GanttProject: http://www.ganttproject.biz

[28] jHotDraw: http://jhotdraw.org

[29] N. Ali, Z. Sharafi, and Y. Gueheneuc, "An empirical study on the importance of source code entities for requirements traceability", Empirical Software Engineering, 2014, 20(2): 442-478