TraceRefiner: An Automated Technique for Refining Coarse-Grained Requirement-to-Class Traces

Mouna Hammoudi
Johannes Kepler University
Linz, Austria
mouna.hammoudi@jku.at

Christoph Mayr-Dorn
Johannes Kepler University
Linz, Austria
christoph.mayr-dorn@jku.at

Atif Mashkoor
Johannes Kepler University
Linz, Austria
atif.mashkoor@jku.at

Alexander Egyed
Johannes Kepler University
Linz, Austria
alexander.egyed@jku.at

***Abstract*—Requirement-to-code traces reveal the code location(s) where a requirement is implemented. Traceability is essential for code evolution and understanding. However, creating and maintaining requirement-to-code traces is a tedious and costly process. In this paper, we introduce TraceRefiner, a novel technique for automatically refining coarse-grained requirement-to-class traces to fine-grained requirement-to-method traces. The inputs of TraceRefiner are (1) the set of requirement-to-class traces, which are easier to create as there are far fewer traces to capture, and (2) information about the code structure (i.e., method calls). The output of TraceRefiner is the set of requirement-to-method traces (providing additional, fine-grained information to the developer). We demonstrate the quality of TraceRefiner on four case study systems (7-72KLOC) and evaluated it on over 230,000 requirement-to-method predictions. The evaluation demonstrates TraceRefiner's ability to refine traces even if many requirement-to-class traces are undefined (incomplete input). The obtained results show that the proposed technique is fully automated, tool-supported, and scalable.**

I. Introduction

Traceability specifies which code region implements a requirement. It facilitates change impact analysis and program understanding. It is most beneficial when engineers are not or little familiar with the existing code, a scenario that they typically encounter during software maintenance. It has been shown [1] that engineers making changes to unfamiliar code leads to errors, accelerated software decay, and consequently wasted effort. This is due to the fact that they are more prone to applying changes to incorrect/suboptimal code regions [3], [4]. Keeping track of traceability information between requirements and code counters this problem [3]. Doing so is recommended by software engineering practices and standards (such as CMMI level 3) [5]–[7]. Nevertheless, the benefits of requirement-to-code traces are contingent on their availability, completeness, and correctness. Correct traces assist engineers in applying changes to appropriate code regions. Likewise, complete traces assist engineers in becoming aware of a requirement's implementation in its entirety. Herein lies one of the fundamental problems of traceability: to date, no automated technique exists that is able to generate requirement-to-method traces at an acceptable level of completeness and correctness [8]–[10].

Hence, today, the only truly effective technique for capturing traces is a manual one. However, the main downside of manual trace capture is its high effort. Take, for example, iTrust [11] – one of the case studies later used in this paper. It has 4,913 methods in 718 Java classes distributed between a client and a server. Even though, we merely investigated a subset consisting of 34 of its requirements, engineers need to capture 4,913$\times$34=167,042 requirement-to-method traces to precisely document every Java method's relationship. The manual burden is overwhelming.

Considering trace granularity, code can be divided into coarse-grained regions such as classes, or fine-grained ones such as methods. Coarse-grained traces are high level traces contained in large portions of code (classes) as opposed to fine-grained traces, which are specific to a limited amount of lines of code (methods). Since there are less coarse-grained regions (classes) than fine-grained ones (methods) [8], it is faster, cheaper, and easier to capture coarse-grained requirement-to-code traces than fine-grained ones [12]. However, informing engineers of the coarse-grained region implementing a requirement is not as useful as informing them of the fine-grained region implementing a requirement [12]. Indeed, pinpointing precisely which small code region (i.e., method) implements a requirement is more useful than pinpointing which large code region (i.e., class) implements a requirement [12].

In this paper, we are introducing our technique *TraceRefiner*, which takes coarse-grained requirement-to-class traces and refines them into fine-grained requirement-to-method traces. To the best of our knowledge, this is the first attempt to do so. This allows the engineer to focus on the cheaper task to produce coarse-grained requirement-to-class traces. In the following, the terms coarse-grained traces and requirement-to-class traces are used interchangeably. Likewise, the terms fine-grained requirement-to-code traces and requirement-to-method traces are used interchangeably.

To evaluate TraceRefiner, we compare the fine-grained predictions output by TraceRefiner against a ground truth fine-grained requirement-to-code traces (gold standard) collected from professional software developers [17]. The ground truth consists of four case study systems ranging from 7.2 to 72KLOC in size and covering 81 requirements for which we had available high-quality requirement-to-method traces for a large range of classes and methods. Hence we were able to validate TraceRefiner's ability to handle incompleteness (i.e., undefined requirement-to-class traces), which is common since engineers rarely capture traces completely [8]. Even with

incompleteness, TraceRefiner is successful in refining class to method-level traces with high precision and recall.

The remainder of this paper is organized as follows. Section II introduces TraceRefiner, subsequently evaluated in Section III. Section IV presents related work before Section V concludes the paper.

II. Technique

A. T trace, N trace and U trace

Requirement-to-code traces are typically provided in the form of matrices called requirement-to-code trace matrices (*rtm*). A requirement-to-class rtm_c is the input to TraceRefiner. Table I shows an example of such an $rtm_m[m,r]_c$ for a vehicle management system, which we use as an illustrative example in this paper. The columns of this matrix enumerate the different requirements under consideration and the rows denote the classes of interest. We consider two requirements namely, "Requirement 1: Start Car and GPS" and "Requirement 2: Set Passenger Info and Book Ticket". Each entry within Table I (apart from the headers) denotes the tracing information relative to the given requirement-to-class entry of interest. For simplicity, we use the abbreviations T, N and U to respectively designate T traces, N traces and U traces.

A T trace in an entry implies that the corresponding class implements the requirement. For example, the classes *GPS* and *Car* have T traces to requirement 1. An N trace in an entry implies that the class definitely does not implement the requirement. For example, the class *Passenger* has an N trace to requirement 1. Unique to our problem is a third possible entry state, which we refer to as the U trace. A U trace is an incompleteness and denotes a trace that is unknown to be an N trace or a T trace. For example, it is unknown whether the classes *Seat*, *Vehicle*, and *Train* have T or N traces to requirement 1. U traces are mostly ignored in literature even though it is our observation that, in practice, U traces are the norm and not the exception. Indeed, engineers rarely capture the full requirement-to-code relationships for a given software system [2]. It also motivates TraceRefiner's focus on T and N trace refinement because both convey more knowledge than U traces.

Definitions 1:

- rtm_c*[class,requirement]* returns the trace value of an entry in a requirement-to-class rtm_c. Possible values are either a T trace, an N trace, or a U trace.
- rtm_m*[method,requirement]* returns the trace value of an entry in a requirement-to-method rtm_m. Its possible values are either a T trace, an N trace, or a U trace also.
- rtm_c*[set<classes>,requirement]* returns the set of trace values for the requirement-to-class entries input.
- rtm_m*[set<methods>,requirement]* returns the set of trace values for the requirement-to-method entries input.

B. TraceRefiner's Architecture

Both the program structure and the coarse-grained requirement-to-class traces (rtm_c) are presented as input to TraceRefiner. TraceRefiner requires source code because it uses information about the code's structure (e.g., method calls) to refine requirement-to-class traces into requirement-to-method traces. The code structure is readily computable by parsing the source code. We did so using the open-source library Spoon [13]. Our four-step technique is made of four successive steps, namely (1) N Refinement, (2) N Propagation, (3) T Refinement, and (4) T Propagation. The entries of the initial requirement-to-method matrix rtm_m provided at Step 1 are filled with U traces and each step attempts to replace these U traces with either T or N traces. Furthermore, Steps 1 and 3 take the requirement-to-class rtm_c as an input. The output of TraceRefiner is a requirement-to-method rtm_m that is structurally similar to the requirement-to-class rtm_c except that the rows enumerate methods rather than classes. An entry in a requirement-to-method rtm_m is thus identified by a method and a requirement. Its possible values are either a T, N or U trace.

TABLE I: Excerpt of the Requirement-to-Class T traces for the Vehicle Management System (an illustration)

Class	Requirement 1	Requirement 2
GPS	T	U
Car	T	N
Seat	U	T
Passenger	N	T
Vehicle	U	U
Train	U	T

C. Code Structure

Figure 1 shows an excerpt of the code structure of a vehicle management system. Each class is represented by a rectangle with the top part containing its name and the bottom part enlisting the methods within the class. In Figure 1, we have six classes with eight methods numbered 1 through 8. Calling relationships are represented by full arrows, while interface/implementation relationships are represented by dashed arrows. For instance, method *2-start* calls method *1-startGPS*. We state that *2-start* is the caller of *1-startGPS* and reciprocally *1-startGPS* is the callee of *2-start*. Also, *Vehicle* is an interface that has two implementations (*Car* and *Train*). The implementations *Car* and *Train* both implement method *6-bookTicket* contained in the interface *Vehicle*. TraceRefiner makes predictions for tracing relationships based on method calls parsed from the source code. Nevertheless, we notice that the notion of method calls needs to be extended due to the presence of interface/implementation relationships. Thus, we extend the callers/callees with extra calling relationships that need to be added explicitly. Table II shows the extended callers/callees for each method within Figure 1. The methods shown in bold and italic within Table II represent the additional callers/callees that we obtained after applying our "caller/callee extension". For the sake of simplification, we use the terms callers and callees in this paper to respectively designate the extended callers and callees.

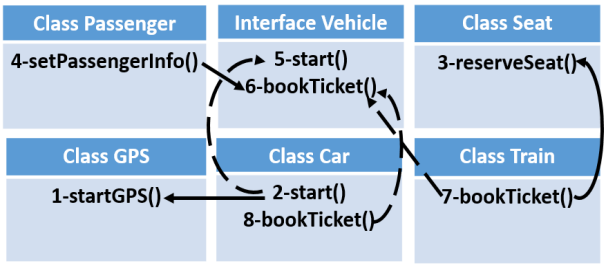

Fig. 1: Program Structure of the Vehicle Management System

D. TraceRefiner: A Four-Step Technique

TraceRefiner initially presumes all requirement-to-method traces to be U traces. It then applies four successive steps outlined above. We apply TraceRefiner on the example shown in Figure 1. Figure 2 shows our rtm_m after applying each step of TraceRefiner on the example shown in Figure 1. For brevity, the method names are omitted in the first column of each rtm_m. As shown in the leftmost rtm_m in Figure 2, TraceRefiner starts out with an rtm_m filled with U traces for each requirement-to-method entry. Each step attempts to replace these U traces with either T or N traces. The highlighted entries with bold predictions in Figure 2 represent the new predictions made at each step of TraceRefiner. Every step of TraceRefiner is represented by an algorithm, we define the keywords used by our algorithms below:

Definitions 2:

- *collectCallers(method)* is used to collect all of the extended callers of a given method.
- *collectCallees(method)* is used to collect all of the extended callees of a given method.
- *allNs($rtm_{c/m}$[set<classes/methods>,requirement])* is used to check if all the requirement-to-method entries (if rtm_m is input) or all the requirement-to-class entries (if rtm_c is input) have N traces to a given requirement.
- *allTs($rtm_{c/m}$[set<classes/methods>,requirement])* is used to check if all the requirement-to-method entries (if rtm_m is input) or all the requirement-to-class entries (if rtm_c is input) have T traces to a given requirement.
- *atLeast1N($rtm_{c/m}$[set<classes/methods>,requirement])* is used to check if at least one requirement-to-method entry (if rtm_m is input) or requirement-to-class entry (if rtm_c is input) has an N trace to a given requirement.
- *atLeast1T($rtm_{c/m}$[set<classes/methods>,requirement])* is used to check if at least one requirement-to-method entry (if rtm_m is input) or requirement-to-class entry (if rtm_c is input) has a T trace to a given requirement.

We will present each of TraceRefiner's steps below.

1) *Step 1- N Refinement (Algorithm 1)*

A class may have a T trace to more than one requirement. Hence, not all methods of a class must have a T trace to the same requirement(s). However, if a class has an N trace to a given requirement then none of its methods will have a T trace to that requirement. Hence, it is easiest to start trace refinement with classes that have N traces to a given requirement. Algorithm 1 depicts the first step of TraceRefiner. It iterates over all methods and requirements. If the method's class has an N trace to the requirement r ($rtm_c[m.class,r]==N$) then we infer that the method has an N trace to that requirement ($rtm_m[m,r]=N$).

Algorithm 1 Step 1-N Refinement

```
1: for all r in Requirements do
2:   for all m in Methods do
3:     if rtm_c[m.class,r]==N then
4:       rtm_m[m,r]=N
5:     end if
6:   end for
7: end for
```

Returning to the illustrative vehicle management system example in Figure 1, we know from its requirement-to-class rtm_c (Table I) that the class *Passenger* has an N trace to requirement 1. Hence, Step 1 infers that all methods of class *Passenger* have N traces to requirement 1. Since method *4-setPassengerInfo* is a method of class *Passenger* (Figure 1), it follows that method *4-setPassengerInfo* has an N trace to requirement 1.

The predictions vary for different requirements. For example, only the class *Car* has an N trace to requirement 2. Hence, Step 1 infers that the class *Car*'s methods *2-start* and *8-bookTicket* have N traces to requirement 2. Do note that TraceRefiner never overwrites a prediction. Hence, we have three predictions made in Step 1 as shown by the highlighted entries in our rtm_m in Figure 2.

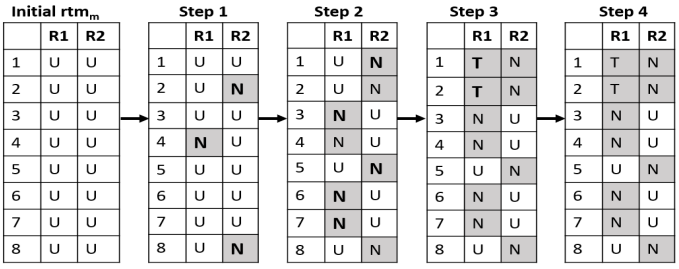

Fig. 2: Our input rtm_m after executing our four-step technique

2) *Step 2- N Propagation (Algorithm 2)*

Predictions made for a method in Step 1 can be propagated to other methods that call this method or are called by it. Since Step 1 predicted N trace information, Step 2 propagates N trace information. The rationale for Step 2 is based on the work of Ghabi and Egyed [1] who observed that trace values are likely similar among a method's callers and callees. To ensure a high likelihood of correct predictions, we only propagate an N trace if and only if all the callers or all the callees of a method have N traces to a given requirement. Hence, any U traces for callers/callees prevent this step from propagating N trace information. This is useful since a U trace could resolve to either a T or an N trace; and a caller/callee method that has a T trace to a given requirement makes the propagation of an

N trace less likely. Algorithm 2 first makes sure that the trace information of a given requirement-to-method entry is still a U trace - line 4 (recall that we do not overwrite previous predictions). The statement *allNs(rtm$_m$[m.collectCallers(),r])* is short for all methods output by *m.collectCallers()* having N traces to requirement r. The function *allNs* is true only if all the callers/callees have N traces as predictions (i.e., a method without a caller or callee cannot satisfy this condition).

Algorithm 2 Step 2-N Propagation

```
1:  do
2:  for all r in requirements do
3:    for all m in methods do
4:      if rtm_m[m,r]==U then
5:        if allNs(rtm_m[m.collectCallers(),r])) then
6:          rtm_m[m,r]=N
7:        else if allNs(rtm_m[m.collectCallees(),r])) then
8:          rtm_m[m,r]=N
9:        end if
10:     end if
11:   end for
12: end for
13:  until no more predictions are made
```

As an example, consider method *6-bookTicket* of class *Vehicle*. This method is called by method *4-setPassengerInfo* (which has an N trace to requirement 1). Since method *4-setPassengerInfo* is the only caller of method *6-bookTicket*, Step 2 infers that method *6-bookTicket* has an N trace to requirement 1. Similarly, Step 2 propagates the N trace from method *1-startGPS* of class *GPS* to method *2-start* of class *Car* for requirement 2. Consequently, we infer:
rtm$_m$[6-bookTicket, requirement 1]=N
rtm$_m$[1-startGPS, requirement 2]=N

Considering a method's callers and callees shown in Table II, further propagations are made in Step 2. For this purpose, Algorithm 2 keeps iterating until no further propagations are made (line 13). For example, the previously propagated N trace from method *4-setPassengerInfo* of class *Passenger* to method *6-bookTicket* of class *Vehicle* is now propagated further to method *7-bookTicket* of class *Train*. The reason for this propagation is that method *4-setPassengerInfo* is a caller of method *7-bookTicket* (see Table II) and it being the only caller implies that *allNs(rtm$_m$[m.collectCallers(),r])* is now true. The algorithm keeps iterating until no more propagations are made (no more changes between two successive iterations). Doing so, we infer additional traces as shown in Figure 2:
rtm$_m$[7-bookTicket, requirement 1]=N
rtm$_m$[3-reserveSeat, requirement 1]=N
rtm$_m$[5-start, requirement 2]=N

*3) **Step 3- T Refinement (Algorithm 3)***

With Step 3, we shift our focus from N traces to T traces. Because our work also considers U traces, we cannot conclude that every method that remains a U trace is a T trace (recall that our work distinguishes three trace values: T trace, N trace, and U trace). However, after we have predicted and propagated N traces, the remaining, still undefined U traces will contain T traces. Step 3 identifies likely candidates and does so by again considering requirement-to-class T traces. This is analogous to Step 1 which did the same for N traces. Algorithm 3 shows Step 3 of TraceRefiner which refines requirement-to-class T traces into requirement-to-method T traces. For this, Algorithm 3 iterates over each requirement-to-method entry (lines 1,2) and makes a T trace prediction if one of the following four scenarios applies. Again, these scenarios are heuristical observations and were derived from empirical observation [1], the evaluation demonstrates their high likelihoods of correctness:

TABLE II: Extended Callers and Callees for the Methods of the Vehicle Management System in Figure 1

Method#	collectCallers()	collectCallees()
1-startGPS	2-start ***5-start***	Empty
2-start	Empty	1-startGPS
3-reserveSeat	***6-bookTicket*** 7-bookTicket	Empty
4-setPassengerInfo	Empty	6-bookTicket ***7-bookTicket*** ***8-bookTicket***
5-start	Empty	***1-startGPS***
6-bookTicket	4-setPassengerInfo	***3-reserveSeat***
7-bookTicket	***4-setPassengerInfo***	3-reserveSeat
8-bookTicket	***4-setPassengerInfo***	Empty

Scenario 1 (lines 4-5): A given method is predicted to have a T trace to a given requirement if it has callers and callees with at least one class that has a T trace to the given requirement. Furthermore, none of the method's callers and callees should have an N trace to the given requirement (lines 4-5). This scenario is very conservative and derives its inspiration from researchers [1] who found a high likelihood for a method to have a T trace to a given requirement if all its callers/callees have T traces to that requirement and none have N traces. However, since we do not have any method-level T traces after Steps 1 and 2, our algorithm uses the T trace information of the classes. Basically, this scenario checks if a given method is called (*atLeast1T*) and calls (*atLeast1T*) at least one class that has a T trace to the given requirement.
Scenario 2 (lines 6-7): Since researchers observed that typically around 50% of all methods do not have callees [1], scenario 1 often does not apply. We provide a second scenario (lines 6-7) for leaf methods - meaning methods that only have callers and do not have callees. Scenario 2 is equivalent to scenario 1 except that we only consider callers and we have empty callees.

Returning to our illustrative example, method *1-startGPS* is a leaf method (it does not call any methods but is called only) – recall Figure 1. Method *1-startGPS* is still a U trace to requirement 1. However, method *1-startGPS* is called by methods of classes *Car* and *Vehicle*; and class *Car* has a T trace to requirement 1 (see Table I). Hence, there is at least one T trace among the classes that call method *1-startGPS* (*m.collectCallers().classes*). Moreover, neither methods *2-start*

nor *5-start* is currently predicted to have an N trace to requirement 1 (i.e., *rtm_m[2-start, requirement 1]==U and rtm_m[5-start, requirement 1]==U).* The rationale is that some owner classes of the callers of *1-startGPS* must have a T trace to requirement 1 and there are no indications that *1-startGPS*'s callers have N traces. Hence, it is likely that *1-startGPS* has a T trace to requirement 1. Hence, we infer:
rtm_m[1-startGPS, requirement 1]=T

Scenarios 3 and 4 (lines 8-11): Scenario 1 required at least one method's callers' class and at least one method's callees' class to have a T trace to a given requirement. Let us now consider scenario 3 (lines 8-9), the latter applies to a method's callers' classes only but requires all of those caller classes to have a T trace to the given requirement. A given method is thus predicted to have a T trace to a given requirement when all of the classes of that method's callers have a T trace to the given requirement. Moreover, neither of the caller and callee methods should have N traces to the given requirement. Scenario 3 is thus a simplification such that caller classes are considered but a restriction such that all caller classes must have a T trace. However, it does not make assumptions about its callees. Scenario 4 (lines 10-11) is the callee counterpart of Scenario 3. It applies when all of the callees' classes have a T trace to a given requirement and none of the callers/callees have N traces to the given requirement. Method *2-start* illustrates Scenario 4. Method *2-start*'s trace relationship to requirement 1 still has a U trace (*rtm_m[2-start, requirement 1]==U*). Method *2-start* belongs to class *Car* and class *Car* has a T trace to requirement 1. Moreover, method *2-start* calls a single method only (method *1-startGPS*) and this method's class also has a T trace to requirement 1. Since none of this method's callers and callees have N traces to requirement 1, Step 3 infers a T trace. Similar reasoning would apply to method *5-start* of class *Vehicle*. However, since we have a U trace for class *Vehicle* with requirement 1, Step 3 does not apply. The rationale for Scenarios 3 and 4 is similar to Scenarios 1 and 2 and omitted for brevity. Again, we constructed these scenarios based on empirical observations about their high likelihoods of correctness. We conclude that: *rtm_m[2-start, requirement 1]=T* as shown in Figure 2.

Algorithm 3 Step 3-T Refinement

```
1:  for all r in requirements do
2:    for all m in methods do
3:      if rtm_m[m,r]==U and rtm_c[m.class,r]==T then
4:        if atLeast1T(rtm_c[m.collectCallers().classes,r])
          and atLeast1T(rtm_c[m.collectCallees().classes,r])
          and zeroNs(rtm_m[m.collectCallers(),r])
          and zeroNs(rtm_m[m.collectCallees(),r])
          then
5:          rtm_m[m,r]=T
6:        else if isEmpty(m.callees)
          and atLeast1T(rtm_c[m.collectCallers().classes,r])
          and zeroNs(rtm_m[m.collectCallers(), r])
          then
7:          rtm_m[m,r]=T
8:        else if allTs(rtm_c[m.collectCallers().classes,r])
          and zeroNs(rtm_m[m.collectCallers(), r])
          and zeroNs(rtm_m[m.collectCallees(), r]) then
9:          rtm_m[m,r]=T
10:       else if allTs(rtm_c[m.collectCallees().classes,r])
          and zeroNs(rtm_m[m.collectCallers(), r])
          and zeroNs(rtm_m[m.collectCallees(), r]) then
11:         rtm[m,r]=T
12:       end if
13:     end if
14:   end for
15: end for
```

*4) **Step 4- T Propagation (Algorithm 4)***

Now that some T traces have been predicted in Step 3, Step 4 propagates these T traces among callers and callees much like it was done in Step 2 for N traces. Similarly to Step 2, Step 4 keeps iterating until no more predictions are made (line 15). Algorithm 4 is divided into three scenarios where each scenario applies to already predicted requirement-to-method T traces from Step 3. The first scenario (lines 5-6) of Algorithm 4 is specific to leaf methods: a T trace prediction is made if a method's callees are empty and if there is at least one T trace among the method's callers. Scenarios 2 (lines 7-8) and 3 (lines 9-10) are the exact counterparts of Step 2. T traces are propagated if all callers/callees of a method have T traces to a given requirement. In our motivating example, none of the rules of scenario 4 applies. Figure 1 does not show any additional predictions for Step 4. Hence, this is an example in which no further predictions are made by TraceRefiner when moving from Step 3 to Step 4 as shown in Figure 2.

Algorithm 4 Step 4-T Propagation

```
1:  do
2:  for all r in requirements do
3:    for all m in methods do
4:      if rtm_m[m,r]==U then
5:        if isEmpty(m.callees)
          and atLeast1T(rtm_m[m.collectCallers(),r]) then
6:          rtm_m[m,r]=T
7:        else if allTs(rtm_m[m.collectCallers(),r])) then
8:          rtm_m[m,r]=T
9:        else if allTs(rtm_m[m.collectCallees(),r])) then
10:         rtm_m[m,r]=T
11:       end if
12:     end if
13:   end for
14: end for
15:  until no more predictions are made
```

III. Evaluation

A. Research Questions

In order to evaluate the correctness, completeness, and applicability of TraceRefiner, the following research questions are addressed:

TABLE III: Information on the Four Study Systems

	Chess	Gantt	iTrust	JHotDraw
Language	Java	Java	Java	Java
KLOC	7.2	41	43	72
#Methods	752	5013	4913	6520
#Interfaces	23	209	5	99
#Classes	104	666	718	663
#Superclasses	18	180	135	296
#Method Calls	1042	7578	12093	11413
#Sample Reqs	8	18	34	21
***rtm$_m$* Size**	6016	90234	167042	136920

- **R1:** How correct (measured in precision and recall) are the predicted requirement-to-method traces compared to the gold standard?
- **R2:** How does TraceRefiner compare to other techniques?

B. Case Studies

In order to answer these questions, we applied TraceRefiner on four case studies. All of the systems are open source and are written in Java. These systems are Chess, Gantt, iTrust, and JHotDraw (Table III). Chess [14] is an implementation of a chess game where two opponents play on a 2D board. Gantt [15] is a scheduling application for time management allowing one to manage resources and calendars. iTrust [11] is an application that allows patients to keep track of their medical history. JHotDraw [16] is a 2D graphics framework that enables its user to draw graph-like structures, such as architecture and design models.

We chose these systems because they are nontrivial in terms of their code sizes (between 7 and 72 KLOC in size); the high number of lines of code (LOC) reflects the complexity of software developed in industry. Also, we had available 81 functional requirements that were listed as the key features of our case studies by the key developers of the four systems [17]. These 81 requirements were accompanied by the corresponding requirement-to-code traces. In the following, we refer to these ground-truth traces as our gold standard for comparison with the predictions made by TraceRefiner. The dataset for these systems as well as the parsed information (method calls, data dependencies, etc) were previously published as a data showcase study paper [17]. Also, all the trace data we use in this paper is open source and made available online [18].

We evaluate TraceRefiner's performance by comparing its output predictions against the gold standard. For Chess, Gantt, and JHotDraw, we contacted the key developers of these systems and we prompted them to enumerate the requirements that implement the code's core functionalities and create traces for these requirements. These developers were given an entire week to produce the requirement-to-method traces (our gold standard) as well as the requirement-to-class traces (TraceRefiner's input). They were also paid for performing these tasks. In case of iTrust, the list of the system's core requirements as well as the list of requirement-to-class traces and requirement-to-method traces were all available on the project's website [19]. These were all made available by the system's original developers. Since iTrust is a commonly used system in traceability experiments, we expect the quality of these traces to be high. For all these case studies, developers did not provide trace information for all requirement-to-method/requirement-to-class entries and left some undefined (U traces). In case of the requirement-to-class traces, trace information was not provided for inner Java classes, interfaces, and abstract classes. Similarly, this was also the case for all the traces related to methods within inner Java classes, interfaces, and abstract classes. More information about our trace data collection process can be found in our previous paper [17].

TABLE IV: Quantifying the requirement-to-class *rtm$_c$* input traces

System	T$_c$(#)	N$_c$(#)	U$_c$(#)	Total	T$_c$(%)	N$_c$(%)	U$_c$(%)
Chess	131	253	448	**832**	15.75	30.41	53.85
Gantt	93	2483	9412	**11988**	0.78	20.71	78.51
iTrust	181	2743	21488	**24412**	0.74	11.24	88.02
JHot.	98	1490	12335	**13923**	0.70	10.70	88.59

TABLE V: Quantifying the requirement-to-method *rtm$_m$* Input Gold Standard

System	T$_m$(#)	N$_m$(#)	U$_m$(#)	Total	T$_m$(%)	N$_m$(%)	U$_m$(%)
Chess	563	2389	3064	**6016**	9.36	39.71	50.93
Gantt	343	23166	66725	**90234**	0.38	25.67	73.95
iTrust	307	7173	159562	**167042**	0.18	4.30	95.52
JHot.	439	12219	124262	**136920**	0.32	8.92	90.76

Table IV provides details on the quantity of requirement-to-class traces available to us (used as input to TraceRefiner) and Table V provides details on the quantity of requirement-to-method traces available to us (gold standard used for comparison with the output of TraceRefiner). As is often the case, the requirement-to-class *rtm$_c$* tends to be incomplete [2]. This also was true for all our case studies. As Table IV shows, we had available only about 11 - 46% of all possible requirement-to-class T and N traces as input ((T+N)/Total). Also, as shown in Table V, we had 4 - 49% of all possible requirement-to-method T and N traces for comparison (correctness assessment). The high proportion of U traces for our systems as shown in Tables IV and V is a normal phenomenon as engineers do not report complete tracing relationships between requirements and code [1]. There might be a high proportion of U traces as is the case for iTrust. Despite the incompleteness, the gold standard provided us with an amount of useful data that is more than sufficient. For example, in the case of JHotDraw's *rtm$_m$*, 12,658 of the 136,920 entries had T/N trace values.

Similarly, we notice that the percentage of T traces is far lower than the one of N traces at the class level. For instance, Gantt has 0.78% of T traces as opposed to 20.71% of N traces within its *rtm$_c$*. The reason for this is that a small area of the code (i.e, one or two classes) implements the requirement and the majority of the remaining classes do not. This explains the higher proportion of N traces as opposed to T traces at the class level (*rtm$_c$*). Likewise, we notice that the amount of T traces is far lower than the amount of N traces at the method level as shown in Table V and the reason for this is similar to the one previously stated at the class level. Indeed, there could be only two methods implementing the requirement and 1,000 other methods not implementing it. This explains the imbalance between the proportion of T and N traces at the method level.

TABLE VI: T trace/N trace Precision and Recall and Completeness of the requirement-to-method traces output by TraceRefiner

Sys.	Step	1- T_p	2- N_p	3- Total	T traces						N traces					
					4- TP_T	5- FP_T	6- FN_T	7- $Prec._T$	8- $Rec._T$	9- $F1_T$	10- TP_N	11- FP_N	12- FN_N	13- $Prec._N$	14- $Rec._N$	15- $F1_N$
Chess	1	0	2063	2063	0	0	0	NA	NA	NA	1612	0	0	100	100	100
	2	0	3144	3144	0	0	46	NA	NA	NA	1822	46	0	**98**	100	99
	3	923	-	4067	417	391	-	52	90	66	-	-	391	-	82	89
	4	1403	-	4547	467	448	-	**51**	**91**	**65**	**-**	**-**	448	-	**80**	**88**
Gantt	1	0	55535	55535	0	0	0	NA	NA	NA	22365	0	0	100	100	100
	2	0	66609	66609	0	0	57	NA	NA	NA	22592	57	0	**100**	100	100
	3	908	-	67517	97	124	-	44	63	52	-	-	124	-	99	100
	4	1527	-	68136	146	184	-	**44**	**72**	**55**	**-**	**-**	184	-	**99**	**99**
iTrust	1	0	17573	17573	0	0	0	NA	NA	NA	6572	0	0	100	100	100
	2	0	28657	28657	0	0	9	NA	NA	NA	6685	9	0	**100**	100	100
	3	290	-	28947	81	27	-	75	90	82	-	-	27	-	100	100
	4	1035	-	29692	93	28	-	**77**	**91**	**83**	**-**	**-**	28	-	**100**	**100**
JHot.	1	0	116787	116787	0	0	0	NA	NA	NA	12013	0	0	100	100	100
	2	0	125748	125748	0	0	86	NA	NA	NA	12066	86	0	**99**	100	100
	3	1738	-	127486	98	34	-	74	53	62	-	-	34	-	100	100
	4	2389	-	128137	132	49	-	**73**	**61**	**66**	**-**	**-**	49	-	**100**	**99**
Avg.	4	1589	56040	57628	210	177	50	**61**	**79**	**67**	10791	50	177	**99**	**95**	**97**

C. Prediction Quality Results

The evaluation of TraceRefiner was done by comparing every entry of the requirement-to-method rtm_m with its gold standard. Note that we only used those rtm_m entries where the gold standard reports a T or an N trace. A U trace in the gold standard would need inspection by a domain expert to decide whether the entry should be a T or an N trace. Hence, we cannot use the gold standard U traces.

Table VI shows the results obtained at each step of applying TraceRefiner on our case studies. An entry containing a "-" in Table VI signifies that the value is identical to the one obtained at the previous step. An entry containing "NA" signifies that the value cannot be computed at this step of the technique. Columns 1, 2, and 3 in Table VI respectively show the cumulative amounts of T trace predictions, N trace predictions, and their sum at each step of TraceRefiner. We determine precision and recall for T traces (columns 4 through 9 in Table VI) and N traces (columns 10 through 15 in Table VI) separately. We will explicate our evaluation from the T traces perspective first (columns 4 through 9). In case both our prediction and our gold standard are T traces, then this is a True Positive (TP_T-column 4). If our prediction is a T trace and our gold standard is an N trace, then we have a False Positive (FP_T-column 5). If our prediction is an N trace and our gold standard is a T trace, then we speak of a False Negative (FN_T-column 6).

Considering the N trace perspective, we have a True Positive (TP_N-column 10) when both our prediction and our gold standard are N traces. We have a False Positive (FP_N-column 11) when our prediction is an N trace and our gold standard is a T trace. We have a False Negative (FN_N-column 12) when our prediction is a T trace and our gold standard is an N trace.

We then apply the standard formulas for calculating precision, recall and the F1 measure, once for T traces (columns 7, 8, 9), and once for N traces (columns 13, 14, 15).

1) N trace Precision and Recall

We notice that our N trace precision and recall are high; their average values are 99% and 95%, respectively. Considering the individual values of N trace precision and recall for each of our four case studies, we notice that all these values are above 90% except for the N trace recall for our Chess system which is 80% at the end of the four steps. Here, N trace recall is comparatively lower as we have 448 cases of False Negatives compared to 1,822 True Positives. This means that TraceRefiner makes some incomplete predictions in Step 2 and incorrect N trace predictions in Steps 3 and 4. These incorrect predictions occur for "boundary methods", meaning methods with callers having T traces to a given requirement and callees having N traces to the same requirement, or vice versa. As their name implies, "boundary methods" are located at the boundary of an interconnected region of methods calling each other and all having a T or an N trace to a given requirement. Despite the presence of False Negatives caused by "boundary methods", our N trace recall is still high for Chess (80%) and TraceRefiner still yields a high average precision (99%) and recall (95%) for our case studies.

2) T trace Precision and Recall

We notice in Table VI that at the end of our algorithm (after Step 4), our T trace precision is between 44-77%. This means that slightly more than half of the T trace predictions were made correctly. Note, that this is not equal to a random guess, as the percentage of T traces in the rtm_m is much lower than the N and U traces: below 1% for Gantt, iTrust, and JHotDraw, and less than 10% for Chess (see Table V).

We notice that the T trace recall at the end of Step 4 ranges between 61% to 72% for Gantt and JHotDraw and up to 91% for Chess and iTrust. The lower recall for Gantt and JHotDraw is due to the comparatively high number of False Negatives (FN_T). The False Negatives comprise all gold standard T traces that we incorrectly predicted as N traces. The high amount of False Negatives for Gantt and JHotDraw is due to the high number of "boundary methods" within these two systems.

Similarly to Section III-C1, the low T trace precision is due to the presence of False Positives (FP_T). Again, these False Positives correspond to all situations in which "boundary methods" are encountered. In future work, we plan to devise additional rules specific to "boundary methods" in order to decrease the amounts of False Positives and False Negatives obtained after applying TraceRefiner.

D. Comparison with Other Techniques

To the best of our knowledge, TraceRefiner is the first and only technique that refines coarse-grained requirement-to-class traces into fine-grained requirement-to-method traces. Therefore, it is difficult to perform a comparison of TraceRefiner against other trace refinement techniques since the latter are nonexistent.

Also, predicting N traces is an innovation of TraceRefiner given that existing approaches predict T traces assuming that everything else is an N trace. Therefore, only a high-level comparison of T trace predictions is possible. Hence, the performance of our N trace predictions cannot be compared given that explicit N trace predictions are a novelty of TraceRefiner that no other researchers have considered. As previously mentioned, N traces are important as they allow engineers to discard an entire group of requirement-to-method entries that are guaranteed to not trace to a given requirement.

1) Comparison against State of the Art Techniques

State of the art techniques perform requirement-to-code recovery at the level of classes. These could, in principle, also be applied at the method level. Example state of the art techniques include Trustrace [9], UDCSTI [20], and TLE [21]. Trustrace [9] is an information retrieval technique that mines software repositories to generate requirement-to-class traces. UDCSTI [20] is a human assisted information retrieval technique in which the human validates the IR requirement-to-class traces automatically generated. TLE [21] is a technique that automatically evolves requirement-to-class traces across two consecutive versions of a system.

These techniques report precision and recall at the level of requirement-to-class traces. Pinpointing the exact method of a class that traces to a requirement is a lot more challenging than pinpointing the entire class that traces to the same requirement [1]. Thus, we argue that their precision and recall form the upper boundary of what their techniques are able to achieve.

Both Trustrace and TLE yield a precision of 59% which is similar to the precision obtained after using our refinement technique (61%). However, it is important to note that TraceRefiner performs a more challenging task than Trustrace and TLE. As previously mentioned, recovering requirement-to-method traces is more difficult than recovering requirement-to-class traces. Thus, we conjecture that the precision of both TLE and Trustrace would decrease if we were to readjust these techniques to generate requirement-to-method traces instead of requirement-to-class traces. Similarly, the precision of UDSTI is 50% and we conjecture that this value would further decrease if we were to readjust UDSTI to recover requirement-to-method traces rather than requirement-to-class traces.

The recall of Trustrace, UDCSTI, and TLE is respectively 15%, 55%, and 43%, whereas the recall of TraceRefiner is 79% on average. Thus, TraceRefiner outperforms Trustrace, UDCSTI, and TLE in terms of recall.

Similarly, even though TraceRefiner's T trace precision is only 61%, we conclude that TraceRefiner outperforms others in terms of precision.

E. Discussion

The results in Table VI demonstrate that with respect to N traces, TraceRefiner is producing extremely high quality results. We are making 99% correct N trace predictions and are retrieving on average 95% relevant N trace predictions. Thus, our N trace predictions are useful as they are both precise and relevant. No other researchers have predicted N traces and all of them exclusively focus on predicting T traces. Predicting N traces gives the opportunity for engineers to discard a large amount of entries that are guaranteed to not trace to a given requirement. This constitutes one of the strong innovations offered by TraceRefiner.

Compared to a traditional tracing technique that does not distinguish between T, N, and, U traces, but assumes everything that is not a T trace must be an N trace, having a lower T trace precision is still useful given that we provide high N trace precision. Given that the T trace predictions are a rather small set compared to the N trace predictions, the N trace precision is relevant here (i.e., for an engineer, it would be significantly easier to manually validate up to 2,389 T trace predictions than up to 125,748 N trace predictions in the case of JHotDraw). Hence, engineers could manually examine the few T trace predictions made by TraceRefiner to verify whether they are correct or not. This task would be feasible given that T trace predictions constitute a small portion of the T and N trace predictions made by TraceRefiner (T traces are about 10 times less frequent than N traces [1]). In doing so, engineers could eliminate all of the requirement-to-method entries that were predicted as N traces, given that we demonstrated the correctness and the relevance of our N trace predictions.

For all these reasons, even though TraceRefiner's T trace precision is 61%, we argue that TraceRefiner makes a significant contribution towards the automatic refinement of coarse-grained traces, achieving far superior quality than comparable techniques (especially given the high quality of N trace predictions).

F. Threats to Validity

Internal Validity We counter researcher bias by considering data from various open source systems with traces created by developers rather than by the paper's authors.

External Validity Given the high number of LOC in our case studies and given that we are considering the key requirements of the systems specified by developers, we conjecture that our findings can be generalized to other systems. The four case studies were very diverse and built by different people. One system, iTrust, even included network communication

which obscures method calls (i.e., no observable calls between client and server at source code level). Still, TraceRefiner delivered very good results. However, all case studies had Java in common. Given that very little code structure is used as part of TraceRefiner's reasoning, we believe that TraceRefiner can be applicable to other languages. Methods and method calls are commonly found in non-Java/non-OO languages including C, Python, C#, and many more. Therefore, overall, we see no major limitation in applying TraceRefiner to other systems written in other languages and making use of method calls.

Construct Validity An additional threat to validity is that we relied on a gold standard provided by humans in order to perform our evaluation. It is not only likely but certain that there are errors in this gold standard given that it comprises several hundred thousand entries. We mitigated this risk for our case studies by resorting to the original developers of the systems as a means for creating the gold standard. This is an indicator of the high quality of our gold standard and its minimal amount of error.

IV. Related Work

To the best of our knowledge, no existing work addresses the refinement of coarse-grained traces into fine-grained ones– and in particular not for requirement-to-code traces. However, there are similarities between TraceRefiner and others. Most notably, Ghabi and Egyed [1] developed a technique for predicting likely incorrect traces using information about calling relationships within the code. These researchers rely on calling relationships derived from program execution. TraceRefiner also relies on calling relationships but from parsed source code information. The main difference between TraceRefiner and Ghabi's and Egyed's is that TraceRefiner **refines** coarse-grained requirement-to-code traces into fine-grained ones, while their technique [1] **validates** pre-existing fine-grained traces. TraceRefiner takes as input requirement-to-class traces and generates requirement-to-method traces, while Ghabi's and Egyed's technique takes as input requirement-to-method traces and outputs validated/unvalidated requirement-to-method traces.

It has also been shown that manual trace capture is expensive and time consuming [8], [10]. This has motivated the need for automated trace generation techniques based on information retrieval [9], [22]. However, these techniques lack correctness (precision and recall) given that they are often based on text similarities among requirements and source code [10]. Ali et al. propose Trustrace [9], a technique that combines information retrieval and mining software repositories for generating requirement-to-code traceability links. Again, this technique is limited as its success depends on the quality of the data contained in software repositories [9]. Other researchers [20] have relied on involving engineers in order to generate traceability data derived from information retrieval techniques. All of these techniques focus on establishing traces between requirements and coarse-grained sources of code (i.e, classes). None of these techniques focuses on creating traces between requirements and fine-grained sources of code such as methods [8]–[10], [20], [22]. Also, these techniques do not succeed in producing correct and complete requirement-to-method traces.

Other techniques focus on recovering traces between code and artifacts other than requirements; or between requirements and artifacts other than code. For example, Cleland-Huang et al. suggest a technique based on web mining by using the internet to obtain a relevant set of indicator terms that can be used to produce traces between requirements and regulatory codes [10]. Some techniques automatically recover traces between requirements and architecture [23], [24] while others recover traces between source code and documentation [25]. Guo et al. [26] propose a solution that relies on a tracing network architecture that uses Word Embedding and Recurrent Neural Network (RNN) models to create traces among any class of artifacts (test cases, documentation, source code, etc).

Very different yet related are techniques that rely on haptic feedback – for example, developers' eye gazes like Walters et al. [27] or Sharif et al. [28] who suggest generating traces between bug reports and source code according to developers' eye gazes while performing work in an IDE.

Complementary is also the work on trace evolution. For example, Rahimi and Cleland-Huang [21], [29] developed a technique for evolving traces between requirements and Java classes. Such techniques reason about changes to code and/or requirements and their implications on traces. Their technique produces a good quality of traces – but only if the quality of the traces was good to begin with.

It is also worth noting that many researchers have evaluated the advantages of traceability in the software engineering lifecycle. Research shows that traceability facilitates regression testing, change impact analysis, and reverse engineering [30]. It is also worth mentioning that trace correctness and completeness have received much attention in the community. For example, much of the incentive behind TraceRefiner is based on an empirical study performed by Kong et al. [8], [12] who evaluated the quality of traces obtained after manually evolving traces. Like us, they presume the existence of trace information but measured the implications of trace maintenance on its quality. One of the observations highlighted by Kong et al. [12] is that subjects validating high quality requirement-to-method traces end up deteriorating their quality rather than improving it. Papers like these motivate TraceRefiner because they demonstrate that manual trace capture is limited and should be supported by (semi) automated trace validation and trace refinement whenever feasible.

Note that TraceRefiner focuses on the connection between a given requirement and the source code regardless of the interrelations among requirements. In that regard, TraceRefiner is quite different from feature interaction techniques that take into consideration overlapping concerns [31], concept lattices [32] or concern graphs [33]. Also, researchers have previously explored the connection between calling relationships in the program and traces [22], [34], [35] but they have never considered automatically refining coarse-grained traces into fine-grained ones.

V. Conclusion

We presented a novel technique for refining requirement-to-class traces into requirement-to-method traces. TraceRefiner leverages the code structure to refine traces. To the best of our knowledge, this is the first work that performs trace refinement. Another innovation of TraceRefiner lies in our N traces that are not taken into account by other researchers. TraceRefiner can be used by engineers to automatically recover more useful, fine-grained traces out of easier to produce, coarse-grained traces. This would facilitate and speed up code maintenance activities such as fixing bugs.

Acknowledgements

The research reported in this paper has been funded by Austrian Science Fund (FWF) under the grant numbers P31989, P29415-NBL and I 4744-N, and also by the state of Upper Austria via LIT-2019-8-SEE-118 and the LIT Secure and Correct Systems Lab.

References

[1] A. Ghabi and A. Egyed, "Code patterns for automatically validating requirements-to-code traces," in *2012 Proc. of the 27th IEEE/ACM Intl. Conf. on Automated Software Engineering*, pp. 200–209, 2012.

[2] M. Hammoudi, C. Mayr-Dorn, A. Mashkoor, and A. Egyed, "On the Effect of Incompleteness to Check Requirement-to-Method Traces," in *Proceedings of the 36th Annual ACM Symposium on Applied Computing (SAC '21)*, pp. 1465–1474, 2021.

[3] P. Mäder and A. Egyed, "Do software engineers benefit from source code navigation with traceability? — an experiment in software change management," in *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pp. 444–447, Nov 2011.

[4] M. P. Robillard, W. Coelho, and G. C. Murphy, "How effective developers investigate source code: an exploratory study," *IEEE Transactions on Software Engineering*, vol. 30, pp. 889–903, Dec 2004.

[5] N. Aizenbud-Reshef, B. T. Nolan, J. Rubin, and Y. Shaham-Gafni, "Model traceability," *IBM Syst. J.*, vol. 45, pp. 515–526, July 2006.

[6] L. C. Briand, Y. Labiche, and T. Yue, "Automated traceability analysis for uml model refinements," *Inf. Softw. Technol.*, vol. 51, pp. 512–527, Feb. 2009.

[7] D. L. Parnas, "Software aging," in *Proc. of the 16th Intl. Conf. on Software Engineering*, ICSE '94, (CA, USA), pp. 279–287, IEEE, 1994.

[8] A. Egyed, F. Graf and P. Grünbacher, "Effort and Quality of Recovering Requirements-to-Code Traces: Two Exploratory Experiments," in 18th IEEE International Requirements Engineering Conference, 2010, pp. 221-230.

[9] N. Ali, Y. Guéhéneuc, and G. Antoniol, "Trustrace: Mining software repositories to improve the accuracy of requirement traceability links," *IEEE Trans. on Software Engineering*, vol. 39, pp. 725–741, May 2013.

[10] J. Cleland-Huang, A. Czauderna, M. Gibiec, and J. Emenecker, "A machine learning approach for tracing regulatory codes to product specific requirements," in *2010 ACM/IEEE 32nd International Conference on Software Engineering*, vol. 1, pp. 155–164, May 2010.

[11] "https://sourceforge.net/projects/itrust."

[12] W.-K. Kong, J. Huffman Hayes, A. Dekhtyar, and J. Holden, "How do we trace requirements: An initial study of analyst behavior in trace validation tasks," in *Proceedings of the 4th International Workshop on Cooperative and Human Aspects of Software Engineering*, CHASE '11, (New York, NY, USA), pp. 32–39, ACM, 2011.

[13] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier, "Spoon: A library for implementing analyses and transformations of java source code," *Software: Practice and Experience*, vol. 46, pp. 1155–1179, 2015.

[14] "https://github.com/warpwe/java-chess."

[15] "https://sourceforge.net/projects/ganttproject."

[16] "https://sourceforge.net/projects/jhotdraw."

[17] M. Hammoudi, C. Mayr-Dorn, A. Mashkoor, and A. Egyed, "A traceability dataset for open source systems," in *2021 IEEE/ACM 18th Intl. Conf. on Mining Software Repositories (MSR)*, pp. 555–559, 2021.

[18] https://doi.org/10.5281/zenodo.4453526.

[19] Y. Shin and L. Williams, "Work in progress: Exploring security and privacy concepts through the development and testing of the itrust medical records system," in *Frontiers in Education 36th Annual Conference*, (Los Alamitos, CA, USA), pp. 30–31, IEEE Computer Society, oct 2006.

[20] A. Panichella, C. McMillan, E. Moritz, D. Palmieri, R. Oliveto, D. Poshyvanyk, and A. D. Lucia, "When and how using structural information to improve ir-based traceability recovery," in *2013 17th European Conference on Software Maintenance and Reengineering*, pp. 199–208, March 2013.

[21] M. Rahimi and J. Cleland-Huang, "Evolving software trace links between requirements and source code," *Empirical Software Engineering*, vol. 23, pp. 2198–2231, Aug 2018.

[22] M. Eaddy, A. V. Aho, G. Antoniol, and Y. Guéhéneuc, "Cerberus: Tracing requirements to source code using information retrieval, dynamic analysis, and program analysis," in *2008 16th IEEE International Conference on Program Comprehension*, pp. 53–62, June 2008.

[23] M. Mirakhorli and J. Cleland-Huang, "Detecting, tracing, and monitoring architectural tactics in code," *IEEE Transactions on Software Engineering*, vol. 42, pp. 205–220, March 2016.

[24] A. Goknil, I. Kurtev, and K. van den Berg, "Tool support for generation and validation of traces between requirements and architecture," in *Proceedings of the 6th ECMFA Traceability Workshop*, ECMFA-TW '10, (New York, NY, USA), pp. 39–46, ACM, 2010.

[25] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo, "Recovering traceability links between code and documentation," *IEEE Trans. Softw. Eng.*, vol. 28, pp. 970–983, Oct. 2002.

[26] J. L. Guo, J. Cheng, and J. Cleland-Huang, "Semantically enhanced software traceability using deep learning techniques," 04 2018.

[27] B. Walters, T. Shaffer, B. Sharif, and H. Kagdi, "Capturing software traceability links from developers' eye gazes," in *Proceedings of the 22Nd International Conference on Program Comprehension*, ICPC 2014, (New York, NY, USA), pp. 201–204, ACM, 2014.

[28] B. Sharif, J. Meinken, T. Shaffer, and H. Kagdi, "Eye movements in software traceability link recovery," *Empirical Software Engineering*, vol. 22, pp. 1063–1102, Jun 2017.

[29] M. Rahimi, W. Goss, and J. Cleland-Huang, "Evolving requirements-to-code trace links across versions of a software system," in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 99–109, Oct 2016.

[30] P. Mader and A. Egyed, "Do developers benefit from requirements traceability when evolving and maintaining a software system?," *Empirical Software Engineering*, vol. 20, pp. 413–441, Apr 2015.

[31] M. Marin, A. V. Deursen, and L. Moonen, "Identifying crosscutting concerns using fan-in analysis," *ACM Trans. Softw. Eng. Methodol.*, vol. 17, pp. 3:1–3:37, Dec. 2007.

[32] P. Tonella, "Using a concept lattice of decomposition slices for program understanding and impact analysis," *IEEE Trans. Softw. Eng.*, vol. 29, pp. 495–509, June 2003.

[33] M. P. Robillard and G. C. Murphy, "Concern graphs: Finding and describing concerns using structural program dependencies," in *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, (New York, NY, USA), pp. 406–416, ACM, 2002.

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

[35] D. Poshyvanyk and A. Marcus, "Using traceability links to assess and maintain the quality of software documentation," in *TEFSE'07*, pp. 27–30, 2007.