Can Method Data Dependencies Support the Assessment of Traceability between Requirements and Source Code?

Journal:	*Journal of Software: Evolution and Process*
Manuscript ID:	Draft
Wiley - Manuscript type:	Research Article
Date Submitted by the Author:	n/a
Complete List of Authors:	Kuang, Hongyu; State Key Laboratory for Novel Software Technology, Nanjing University Mäder, Patrick; Fakultät für Informatik und Automatisierung, Technische Universität Ilmenau Hu, Hao; State Key Laboratory for Novel Software Technology, Nanjing University Huang, LiGuo; Dept. of Computer Science and Engineering, Southern Methodist University, Lv, Jian; State Key Laboratory for Novel Software Technology, Nanjing University Egyed, Alexander; Institute for Software Systems Engineering, Johannes Kepler University Linz,
Keywords:	requirements traceability, feature location, source code dependencies, program analysis, method call dependencies, method data dependencies

SCHOLARONE™
Manuscripts

Can Method Data Dependencies Support the Assessment of Traceability between Requirements and Source Code?

Abstract

Requirements traceability benefits many software engineering activities, such as change impact analysis and risk assessment. However, capturing and maintaining the required complete and correct traceability links is not trivial, making traceability assessment an important field of study. In recent years, requirements traceability research has focused on call dependencies within source code to understand how code properties contribute to the implementation of a requirement and to assess whether traceability links are correct and complete. These approaches largely ignore the role of existing data dependencies within the source code. Methods may never call each other, but may still depend upon another by sharing data. We identified five research questions and validated them on five software systems, covering 4 to 72 KLOC. We found that data dependencies are as relevant as call dependencies for assessing requirements traceability. Even more interesting, our analyses show that data dependencies complement call dependencies in the assessment. These findings have strong implications on code understanding, including trace capture, maintenance, and validation techniques.

Keywords- requirements traceability; software traceability; feature location; source code dependencies; program analysis; method call dependencies; method data dependencies;

I. Introduction

Requirements-to-code traceability refers to the practice of capturing links between requirements and source code – usually suggesting whether some code implements a given requirement. These links can support stakeholders in development-related tasks. In a recent study [1] we discovered that requirements-to-code traceability strongly benefits developers in performing software maintenance tasks. We found that subjects with traceability performed on average 24% faster on a maintenance task and created on average 50% more correct solutions as compared to maintenance task where traceability was not available. These findings were based on correct and complete traceability. Unfortunately, literature suggests that high quality traceability links are difficult to obtain [12, 25, 26], especially so for requirements–to-code traces due to the typically large numbers of required traces, frequent changes to the traced code, and informal nature of the relationships.

In recent years, requirements traceability research has started to focus increasingly on control dependencies within source code in order to gain more information on how code properties contribute to the implementation of a requirement and to assess whether existing traceability relations are correct [2, 12]. Among others, researchers use fan-in/fan-out analyses [3], identified typical patterns of requirements implementation [4], and complement keyword matching techniques on the code with

For Peer Review

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60

control flow analyses [5,6]. These works either statically or dynamically analyze the source code to retrieve sequences of method calls, which are then used to deduct dependencies between methods and to imply relationships to requirements. These analyses, essentially, investigate the callers and callees of a method in order to assess traceability between a method and a requirement. For example, it was observed that if a method implements a given requirement then a method it calls is more likely to implement that requirement compared to a random other method [13]. However, method calls are not the only form of dependency among source code. Source code is also dependent upon another through shared data. This paper specifically investigates whether method data dependencies are as relevant as method call dependencies for understanding requirements-to-code traces. This paper further investigates whether call and data dependencies are complementary in alleviating each other's weaknesses.

We thus formulated five research questions and validated them on five, small to large software systems, covering in total 171 KLOC, 5236 methods, and 91 requirements (four out of the five being open source systems). In this study, we considered all methods that were connected by either call or data dependencies and for which we had available trustworthy requirements-to-code traces. For any given method we then used the traces of its related methods to derive traces for the given method and compared it with the original traces of that method, which were created manually. That means that the classification of whether a method implements a requirement solely determined from the method's call and data dependencies to neighboring methods and the given traces of those neighbor methods.

Our findings are that data relationships, like call dependencies, have a strong relationship to requirements traceability. But, most interestingly, our analyses show that data dependencies complement call dependencies. Measured in precision and recall (i.e., wrong trace rate vs. missing trace rate), we find that call and data dependencies combined resulted in significant improvements in both. The average increase of both precision and recall is 19.80% compared to considering call dependencies only. This observation is of particular importance because many technologies utilized by the traceability research community tend to trade off precision or recall but rarely is there a technology that improves both. Our findings thus have strong implications on all forms of code understanding, including trace capture, maintenance, and validation techniques (e.g., information retrieval). Furthermore, we demonstrated that our findings are robust against trace incompleteness and incorrectness. This particular observation implied that our findings can benefit the real-world traceability scenarios since in the real-world situation traceability techniques often have to deal with imperfect traceability which could be both erroneous and incomplete.

This paper extends our work presented at the 2012 International Conference on Software Maintenance (ICSM 2012) [16] in five major aspects (1) it proposes an integrated tool for capturing both method call and data dependencies. (2) it proposes and evaluates an advanced classifier for analyzing the captured code dependency graph using additional information from data dependencies for analysis (i.e., the number of types and the access type shared in a data dependency), (3) it studies the effect of different trace granularities, (4) it includes results from additional evaluated systems, and (5) it studies the application of the approach in realistic scenarios with imperfect traceability.

The remainder of this paper is structured as follows. Section II briefly introduces the research background. Section III states our five research questions and Section IV discusses the proposed technologies for capturing and analyzing code dependencies. Section V introduces how we set up the experiments based on the five software systems for answering those questions. Section VI reports the results of our experiments and answers the research questions 1-5. Section VII refers to limitations of

For Peer Review

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60

our work. Section VIII discusses several interesting issues related to this paper. Section IX discusses related work in the area of requirements traceability, feature location, and program analysis. Finally, Section X concludes this paper.

II. Background: Traceability and Code Dependencies

In this section, we introduce the concepts of requirements-to-code traceability, code dependencies, and traceability assessment. For illustrating our concepts throughout the paper, we are using the Video on Demand system (VoD) [9]. VoD allows selecting and streaming of movies from a server including basic operations such as stopping or pausing the movie. The VoD system is 3.6 KLOC in size and the smallest of the five systems we analyzed. However, being an intuitive system, we use excerpts of VoD as an illustration for all relevant concepts we are introducing.

For Peer Review

A. Background on Traceability

Software traceability is achieved through the creation and use of trace links, defined by the Center of Excellence for Software and Systems Traceability [30] as “specified associations between pair of artifacts, one comprising the source artifact and one comprising the target artifact”. When correct, traceability demonstrates that a rigorous software development process has been established and systematically followed. Guidelines for the development in safety-critical industries prescribe traceability for two reasons. First, as an indirect measure that good practice has been followed, the general idea being that traceability information serves as an indicator that design and production practices were conducted in a sound fashion. Second, as a more direct measure, to show that requirements are explored and identified and that the system is demonstrably designed and implemented.

B. Requirements-to-Code Traceability

A *traceability link* (or *trace* for short) captures where in the source code a requirement is implemented. This is similar to the concepts of feature mapping, concern mapping, and concept mapping [10, 6, 8]. A requirements-to-code traceability link typically captures the relationship of individual requirements and code elements – such as classes, methods, or any other level of granularity. But, of course, a requirement can be implemented by multiple code elements. Thus, multiple traces may exist for the same requirement where each trace relates to a different code element. Furthermore, a code element can be implemented by multiple requirements. Accordingly, multiple traces may exist from different requirements to the same code element. To support this multiplicity, trace links are commonly captured in form of a requirements traceability matrix (RTM), which captures in each cell a traceability link. RTMs contain n*m cells where n is the number of requirements and m is the number of code elements.

	R0	R2	R5
VODClient.init()	X		
ListFrame.buttonControl3_actionPerformed()	X	X	
ListFrame()	X	X	
ServerReq.getmovie()	X	X	X
ListFrameListener3.actionPerformed()		X	X

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60

Table 1. Excerpt from a requirements traceability matrix (RTM) of the Video on Demand example

Table 1 depicts an excerpt of such a RTM for the VoD system. The RTM in the table depicts selected combinations of methods (rows) and requirements (columns). An 'X' in a cell indicates a trace between the cell's requirement and the cell's method. A blank in the cell indicates a no-trace. For example, R2 is the requirement "Users should be able to display textual information about a selected movie." In Table 1, method `VODClient.init()` does not trace to requirement R2; however the `ListFrame's` constructor does. We define these two kinds of traces as "no-trace" and "trace". While the method `ListFrameListener3.actionPerformed()` has a "no-trace" to R0, it does trace to requirements R2 and R5.

C. Call and Data Dependencies between Methods

In this paper we consider two kinds of method dependencies: (1) calling each other and (2) sharing data with each other. Calling means that the source code of one method contains a call to the other method. Figure 1 shows an excerpt of the VoD source code, covering three Java methods: `VODClient`'s `init()`, the constructor of `ListFrame`, and one of `ListFrame`'s event handler `buttonControl3_actionPerformed()`. In method `init()`, the object server of type `ServerReq` is initialized. Then this object is passed to the constructor of the `ListFrame` class and there assigned to the `ListFrame` field `ser`. Finally, the event handler `buttonControl3_actionPerformed()` accesses the same field `ser`. The fact that `VODClient`'s `init()` instantiates a `ListFrame's` object is essentially a method call onto `ListFrame`'s constructor. However, neither `VODClient.init()` nor `ListFrame`'s constructor call the method `buttonControl3_actionPerformed()`. Hence there are no call dependencies between `buttonControl3_actionPerformed()` and the other methods.

```
class VODClient
 public final void init()
…
   server = new ServerReq( "127.0.0.1", s);
 server.connect();
  listframe = new ListFrame(server, this);

class ListFrame
 public ListFrame(ServerReq serverReq,
    VODClient vODClient)
 …
 ser = serverReq;
 parent = vODClient;
 …

 void buttonControl3_actionPerformed(…)
   …
   String s = listControl1.getSelectedItem();
   if (s != null){
  Movie movie = ser.getmovie(s);
…
```

Figure 1 Code snippets of the Video on Demand system

Sharing data means that two or more methods manipulate or read variables that point to the same data in (physical) memory irrespective as to whether the variables which hold the pointers are the same (i.e., two classes may have differently named variables that point at the same object). This complex

For Peer Review

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60

formulation is necessary as the same underlying data is often accessed through references or even chains of references that in a simple static analysis would appear independent. Figure 1 shows an example that demonstrates such a situation. There is an obvious data dependency between the two `ListFrame` methods because both access the `ser` field even though we do not find method calls between them. This data dependency is easily recognizable. Not easy to recognize is the data dependency between `VODClient`'s `init()` and `ListFrame`'s `buttonControl3_actionPerformed()`. Neither method accesses the same fields. However, the local `server` variable defined in `VODClient`'s `init()` method is eventually passed to `ListFrame`'s constructor as a parameter where it is stored as `ser`. The variables `server` and `ser` thus point to the same data in memory. Thus, all three methods access or manipulate the same underlying data object and this implies that all three methods are data dependent. We demonstrate that such data dependencies can help assess traceability, because data dependencies much like call dependencies help identify related functionality.

For Peer Review

D. Traceability Assessment

A large number of traceability links is required to trace requirements to their implementing source code. These links easily get outdated once the specification and the implementation evolve, making it necessary to constantly assess and update traceability links in order to ensure the required completeness and correctness. Given the large number of trace links in a project, and the cost and effort of assessment, it is appealing to consider techniques for performing that task automatically – even if such techniques can only provide a confidence score for each link on which basis a list of suspect links is being created that humans need to prune. While we would desire a technique that can interpret and assess the semantics of a link, there is no approach in sight that would be able to do that. Instead, researchers go for a more realistic target by analyzing the relation between a traced artifact and other artifacts of the same type. That analysis allows drawing conclusions about the consistency and conclusiveness of existing and missing traces. This compounding evidence can increase confidence in the correctness of a trace link.

Let us discuss how our approach assesses traceability links for a given method based on its related methods traceability assessment on the VOD system we discussed above (i.e., through call or data dependencies). According to Table 1, `ListFrame.buttonControl3_actionPerformed()` has a "trace" to R2 and a "no-trace" to R5. Let us now assess these two traces. Looking at the implementation (code) of the VoD system, we find that the method we are trying to assess (`ListFrame.buttonControl3_actionPerformed()`) is called by `ListFrameListener3.actionPerformed()` and it calls `ServerReq.getmovie()`. Table 1 shows that these two neighboring methods (caller and callee) both trace to R2 and R5. Given that both the caller and callee of the method we are trying to assess also trace to R2, we may be more confident that the trace to R2 is correct. However, we may need to be more doubtful about that method's no-trace to R5 because the caller and callee do trace to R5. This assessment included information about calling relationships only. From Section II.C we know that `buttonControl3_actionPerformed()` also has data dependencies with `VODClient.init()` and the constructor of `ListFrame`. Hence, there are more possible dependencies to consider. From Table 1, we learn that `init()` does not trace to R2 and R5 while `ListFrame`'s constructor traces to R2 but not to R5. So if we consider traces from this more complete set of neighboring methods (call and data dependencies), we find that three out of four

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60

methods trace to R2 and merely half of the neighbor methods trace to R5. This observation increases our confidence that R2 traces to `ListFrame.buttonControl3_actionPerformed()` but we now also have more evidence that it may not trace to R5 after all. The example above shows that a successful assessment technique needs to analyze each cell of a traceability matrix to classify it as trace or no-trace and briefly illustrates our approach for trace assessment.

III. Research Questions

The goal of our work is evaluating the usefulness of method data dependencies in addition to method call dependencies for assessing traceability links. We are interested in their usefulness separately and together in having a potential complementary effect. This led to two research questions:

1) *Are method data dependencies relevant for assessing requirements-to-code traces?*
2) *Are method call and method data dependencies complementary in assessing requirements-to-code traces?*

In practice, requirements-to-code traces occur in various granularities. For example, traces may link requirements to methods in the source code or they may link requirements to classes, packages, or components. As the assessment of a trace considers traces on dependent code entities, trace granularity could potentially influence the relevance of method data dependencies. This consideration led to the third research question:

3) *Is the complementary effect of method call and data dependencies for assessing requirements-to-code traces affected by trace granularities?*

If method data dependencies are indeed relevant for assessing requirements-to-code traces and there is a complementary effect, how can we demonstrate that our findings can benefit the real-world traceability scenarios? Trace assessment is practically relevant for two traceability related tasks: trace generation and trace validation (see Section II.D). Trace generation approaches (e.g., [6] and [11]) often use information retrieval techniques first to generate initial traces between requirements and code elements (such as methods). Then a threshold based on the calculated IR value is set to choose highly relevant traces as the seeds for the subsequent process. The whole set of these seed traces could be viewed as an imperfect RTM which is less erroneous but highly incomplete. On the other hand, trace validation approaches (e.g., [12]) assess RTMs, which can be both erroneous and incomplete in the context of code dependencies to perform trace maintenance tasks. So if we discover a complementary effect between call and data dependencies, it should be robust against the incorrectness and incompleteness of input RTMs if the real-world traceability scenarios can get benefit from it. Therefore we define the following additional research questions:

4) *Is the complementary effect of method call and data dependencies affected by trace incorrectness?*
5) *Is the complementary effect of method call and data dependencies affected by trace incompleteness?*

In investigating research questions 1–5, we will also reevaluate the effect of method call

For Peer Review

dependencies for assessing requirements-to-code traces for two reasons: a) to demonstrate that our case studies and empirical observations are consistent with previous studies and b) to quantify the differences between call and data dependencies, which need to be compared directly. We will investigate our research questions on five case study systems, which are presented in Section V. The empirical evidence we gathered is then used to answer the research questions 1-5 in Section VI. Before discussing the study and its results, we will introduce the research framework that we developed for conducting the study, i.e., the technologies that we used for obtaining call and data dependencies and for understanding their relationship to requirements traceability links. This is done in Section IV.

IV. Developed Research Framework

Figure 2 depicts the framework of our proposed technologies. We analyzed each software system separately. First of all, we explain the reason why we chose dynamic analysis to capture code dependencies instead of static analysis and show the overall structure of our capture tool (Step 0). We initiated our study by capturing method call and method data dependencies during runtime to lay the foundation for investigating the relationship between code dependencies and requirements-to-code traceability (Step 1 and 2). We then built a graph structure called Code Dependency Graph (CDGraph), which combines the captured method call and data dependencies so that we can establish our basic model for correlating call and data dependencies with requirements traces added to the CDGraph. In the basic trace model, each trace is computed based on trace information of neighboring methods. We then introduce a more advanced algorithm called Inverse Data Frequency (idf) and apply this algorithm to the basic trace model to generate the experiment results. This algorithm was needed to better handle data dependencies because they strongly outnumbered call dependencies and contained significant noise (Step 3). Finally, we evaluate the correctness of the computed classifications by comparison with the gold standard RTM based on the basic trace model with the idf algorithm. Based on this data, we answer the aforementioned research questions 1 to 5. These five steps are explained in more details in the following subsections.

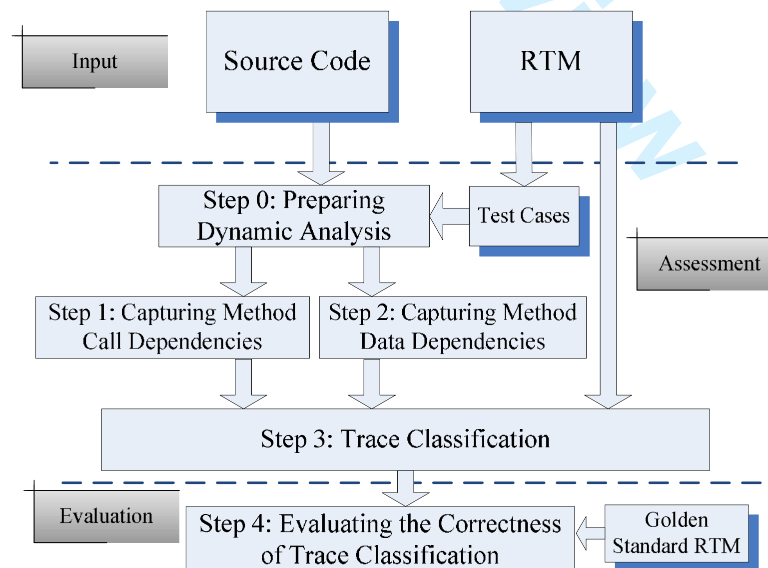


Figure 2. Steps of the proposed research framework and developed technologies.

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60

A. Step 0: Preparing Dynamic Analysis

For our approach to work, we require high quality method call and data dependencies. If the method call and data dependencies were of poor quality, we would not expect to find any relationship between them and requirements traces beyond the random chance of correctness discussed earlier. There could be wrong calling dependencies or missing calling dependencies if the technology were to identify incorrect calls (=wrong) or failed to identify calls (=missing). Likewise, there could be wrong data dependencies and missing data dependencies.

Method call and data dependencies can be captured through static and/or dynamic program analysis. Static analysis does not account for program input and its result must be applicable to all executions of the program. This situation inevitability forces approximations to be made [29]. Sound static analyses to capture code dependencies can make correctness guarantees (there are no missing dependencies) through overapproximation but typically produce a large number of false positives (wrong dependencies) and consume a significant amount of time or resources if the analysis is performed on the whole software system. This is not acceptable for us because we need to capture the actual system behavior to lay a good foundation for our observations. Unsound static analyses make no such guarantees, but can often quickly produce correct or "close enough to be useful" results. The problem with unsound static analysis techniques is that they generally err on both sides. If the call and data dependencies were roughly equally wrong or missing then this might still allow us to investigate research questions; however, there is no guarantee that this is the case. And it would be hard to argue on the effects of wrong and missing dependencies in context of requirements traceability. Indeed, we believe that the static analysis for data dependencies is far less reliable than the static analysis of call dependencies because data dependencies are much harder to detect and track (e.g., points-to analysis [15]). If we were to use static analysis techniques, we thus would require manual investigation to improve the quality of these captured call and data dependencies. We identified 9660 call dependencies (method calls) and 55382 data dependencies across the five systems (shown in Table 4, Section V) and manually validating all of them would have been infeasible.

We thus relied on dynamic analysis, which required us to execute the software system and observe method call dependencies and method data dependencies. Dynamic analysis is guaranteed to neither cause false call dependencies nor false data dependencies because it observes what actually happens in the executing system rather than trying to guess it. However, dynamic analysis does not guarantee complete call and data dependencies because only those code dependencies are observed that were actually triggered during the execution of the system. The degree of completeness is thus a factor of the completeness of the test data. We tested all five systems according to use case descriptions of the gold standard RTMs. But our testing was not limited to the sample requirements nor was it attempted to test the requirements individually to avoid any bias towards particular requirements. We cannot prevent the problem of missing call and data dependencies, but we believe that missing dependencies are not so much a problem for as long as call and data dependencies are missing in a roughly equal ratio. This appears to be true as long as both call and data dependencies are captured within the same tool when we were running the same test cases. We will continue to discuss the effect of incomplete testing in the threats to validity part.

For Peer Review

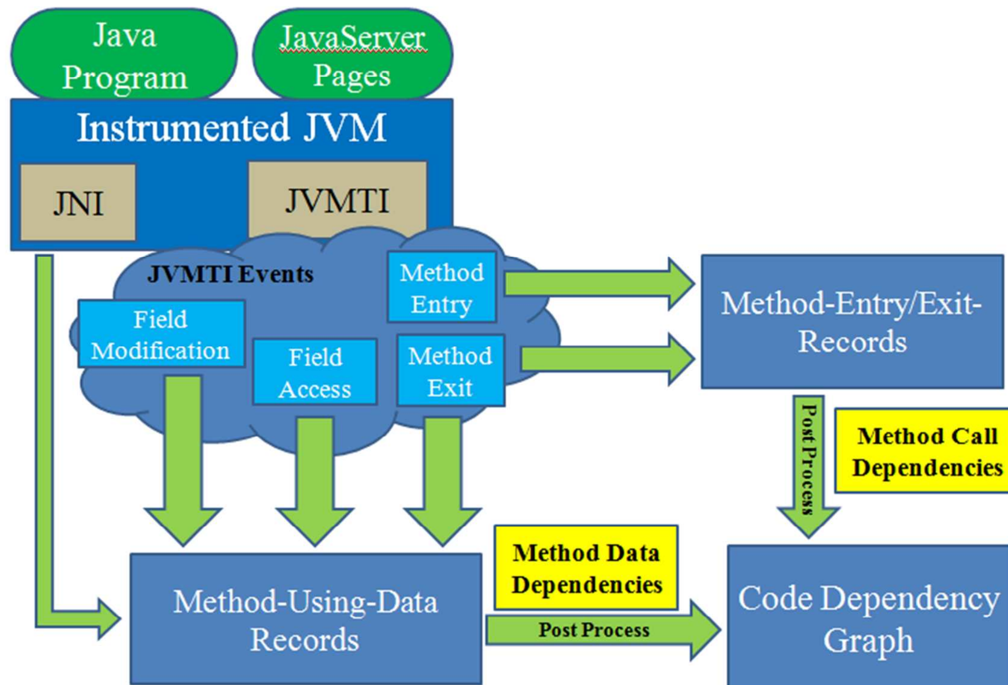


Figure 3. The approach of capturing method call and data dependencies

There are ample technologies for observing method calls at runtime. For example, in Java any runtime profiler or debugger could do this job (e.g., TPTP). However, existing technologies for dynamic analysis do not focus on data sharing between methods. The focus on methods is important here because traceability is typically provided for code elements (such as classes or methods) and not fields or variables. Therefore technologies for understanding data sharing among variables are not sufficient for our purpose.

Since we were unable to find a tool for capturing both call and data dependencies at the level of detail described above, we developed one by ourselves (Figure 3). Our tool was built for Java and relies on the Java JDK. In particular, we are using JVMTI (Java Virtual Machine Tool Interface), which provides a way to control the execution of a system when it is running in the Java virtual machine (JVM) while at the same time inspecting the state of that system (i.e., its data). JVMTI can be used to register for specific, interesting events (such as "method exit" and "field access") that JVM generates. JVMTI then helps query and control the system, either in response to events or independent of them. We also used JNI (Java Native Interface) together with JVMTI to help us locate the actual object in memory during runtime to establish data dependencies. Our capture tool is offline since we use local databases to record information about the entry and exit sequence of methods (method-entry/exit-records) and data accesses triggered by methods (method-using-data records). We then extract method call and data dependencies based on these generated records. The processes of records generation and method call/data dependencies extraction are demonstrated in Section V.B (Step 1) and Section V.C (Step 2), respectively. Furthermore, we extended our tool to capture call and data dependencies in JavaServer Pages by monitoring the corresponding Java Servlets, which are automatically translated by web servers (e.g., Apache Tomcat) to support our experiments on J2EE systems such as iTrust. Our technology is currently restricted to Java. However, we believe that our observations should be generalizable to other

programming languages because they are based on programming concepts that are similar across modern programming languages.

B. Step 1: Capturing Method Call Dependencies

For capturing method call dependencies, we were interested in two JVMTI events particularly: method entry and method exit. These two events allowed us to inspect and record the sequence of method entries and exits by callback functions registered to these events during runtime. To obtain the correct calling sequence and avoid data race, we used two functions provided by JVMTI, called `RawMonitorEnter()` and `RawMonitorExit()`, to synchronize our callback functions. Figure 4 shows eight such method-entry/exit-records that we captured for the Video on Demand system:

- `ListFrameListener3.actionPerformed()` entered, thread ID 19362639.
- `ListFrame.buttonControl3_actionPerformed()` entered, thread ID 19362639.
- `ServerReq.getmovie()` entered, thread ID 19362639.
- `ServerReq.getmovie()` exited, thread ID 19362639.
- `Detail.setmovie()` entered, thread ID 19362639.
- `Detail.setmovie()` exited, thread ID 19362639.
- `ListFrame.buttonControl3_actionPerformed()` exited, thread ID 19362639.
- `ListFrameListener3.actionPerformed()` exited, thread ID 19362639.

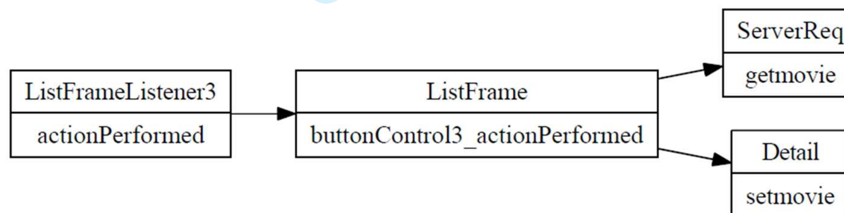


Figure 4. Examples of method-entry/exit-records (top) and a derived call tree (bottom)

By traversing these method-entry/exit-records, we first establish call dependencies among methods that have the same thread ID because each thread executes separately and hence follows its own calling hierarchy (which we captured in form of call trees). A call tree is a hierarchy where the caller is above in the hierarchy. The root element is thus where the execution started (e.g., the function main). A separate call tree was computed for each thread ID. The algorithm is simple and uses a stack structure. If a method-entry record appears, we push this method into the stack; if a method-exit record appears and this method is the same as the one at the top of the stack, we pop the stack and establish a call dependency between the popped method and the method which is right now at the top of the stack. We used the following algorithm to generate call trees (“`stackVar.pop()`” means pop the stack and return the top element; “`stackVar.peek()`” means return the top element only):

```
Stack stackVar
foreach record with same thread ID in database {
  if (record.type == “entry”)
     stackVar.push(record.method);
  else{
     if(!stackVar.isEmpty())
        Method poppedMethod = stackVar.pop();
     if(stackVar.isEmpty() || poppedMethod != record.method){
        display(“Inconsistency” + record);
     }
     else
```

```
        establishCallDependency(stackVar.peek(),poppedMethod);
}
```

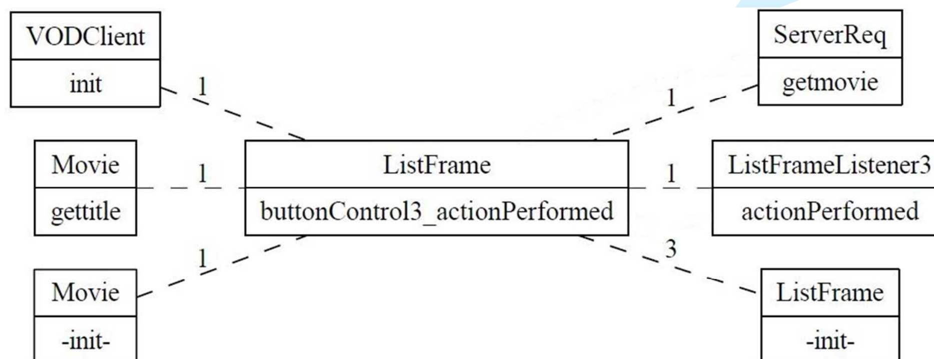
The three call dependencies derived from the listed eight method-entry/exit-records are shown in Figure 4. Each node is labeled with class and method name and represents exactly one method. Method call dependencies are shown as solid arcs with arrows. We combine the call trees of different threads to generate the full set of call dependencies.

C. Step 2: Capturing Data Dependencies

To capture method data dependencies, we were interested in three JVMTI events particularly: field access, field modification, and method exit. In Java, variables can only be created as fields inside a class or as local variables inside a method. The field access and field modification events tell us when a field is accessed or modified by a method at runtime. The method exit event (which is the same event we used in capturing call dependencies) allows us to inspect the full list of created local variables (including parameters and return values) before the end of a method call.

For our work, it is important to go beyond shared variables because true data dependencies exist if two methods have access to the same data even if the data are referenced by different variables. Two methods thus have a data dependency if both methods access or manipulate variables that point to the same data in memory. With the help of JNI and JVMTI, we can locate actual objects in the Java heap that are pointed to by the variable's references (we will discuss how to handle static data and Java primitive types later). JVMTI also provides a key function for our approach called `GetObjectHashCode()` which retrieves a unique[1] identifier of a Java object in memory. Accordingly, we compute separate so-called method-using-data records for each method. Figure 5 shows four of the sixteen method-using-data records that we captured for the Video on Demand system ("-init-" refers to the constructor of a Java class):

- `VODClient.init()` accesses a field in the `VODClient` class named `server`, which is of type `ServerReq` and is uniquely identified by the hash code *13986615*
- `ListFrame.-init-()` declares one of its parameters to be `ListFrame.-init-().serverReq`, which is of type `ServerReq` and is uniquely identified by the hash code *13986615*
- `ListFrame.-init-()` modifies the field `NewValue` of type `ServerReq` in the object *server* of type `ListFrame`, which is uniquely identified by the hash code *13986615*
- `ListFrame.buttonControl3_actionPerformed()` accesses the field `ListFrame.ser`, which is of type `ServerReq` and uniquely identified by the hash code *13986615*



For Peer Review

[1] According to the source code of JVMTI, the hash code is first allocated using os.random() by default, meaning the chance of two hash codes having the same value is $1/2^{32}$ or $1/2^{64}$. By additional type checking we are confident that the hash codes which `GetObjectHashCode()` returns are unique.

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60

Figure 5. Examples of method-using-data records (top) and derived data dependencies (bottom)

By comparing the hash code used in the four listed method-using-data records, we identified two data dependencies among the three methods: `ListFrame.-init-()`, `ListFrame.buttonControl3_actionPerformed()`, and `VODClient.init()`. The complete six data dependencies between `buttonControl3_actionPerformed()` and the other six methods are shown in Figure 5. Nodes are labeled with class and method names. Data dependencies are shown as dashed edges without arrows (see lower part of the figure). These edges are annotated with the number of data types shared by each two methods. Experiments showed that the number of shared objects among methods did not convey more information in terms of trace assessments than the number of shared data types. Our trace classification will utilize this number of shared data types among methods.

D. Step 3: Trace Classification

We merge all method call dependencies and all method data dependencies into a single graph structure, called the Code Dependency Graph (CDGraph). In this graph, one node represents exactly one method. We then annotate each node with the gold-standard traceability information as a reference. Figure 6 shows an excerpt of the CDGraph for the Video on Demand system with traces from the gold standard RTM annotated to each node (listed at the bottom). For example, `ListFrame.buttonControl3_actionPerformed()` is known to contribute to the implementation of the requirements R0, R2, R10, and R12; it is called by method `ListFrameListener3.actionPerformed()`; and it shares data with `VODClient.init()`.

Figure 6 shows that method call dependencies and method data dependencies appear complementary but also overlapping. For example, there is a method call dependency between `ListFrame.buttonControl3_actionPerformed()` and `Detail.setmovie()` and a method data dependency between `buttonControl3_actionPerformed()` and `Movie.gettitle()`. However, `buttonControl3_actionPerformed()` and `ServerReq.getmovie()` are related by both a method call dependency and a method data dependency.

For Peer Review

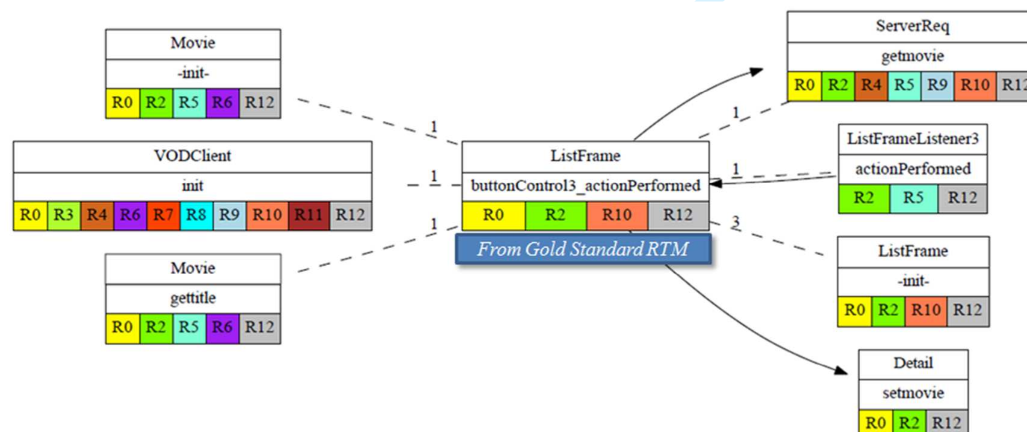


Figure 6. Example of a Call-Data Dependency Graph showing one node and its neighbors related by method call relationships (solid arcs with arrows) and method data relationships (dashed arcs without arrows). Each node identifies the requirements it traces to (labels Rx).

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60

We use a system's annotated CDGraph to find out whether the method call dependencies and the method data dependencies of any given method node in the graph correlate with requirements-to-code traces implementing that node. We implemented several algorithms that can automatically classify requirements traces as trace or no-trace for each node in the captured graph based on its dependencies to other nodes of the graph and their related requirements. We name this process as "trace classification". The reason we implemented several such algorithms was to ensure that the algorithm used does not bias our observations regarding the research questions. The following subsection introduces the most basic classifier we implemented for trace classification. We will describe another more advanced classifier based on this model in the second part of this section.

D.1. Basic Classifier

The basic classifier, which is also an intuitive and simple algorithm, solely counts the neighbors of a node in the graph (neighbors are nodes that are either reachable by call dependencies or data dependencies). Some of those neighboring nodes may relate to a certain requirement and there are those that do not relate to that requirement. The algorithm then classifies requirements traces for the evaluated node based on the ratio of neighbors that relate vs. do not relate to that requirement. We are using the following algorithm to classify the traceability between each requirement in the specification of a system and each executed method in its source code (i.e., each node):

```
foreach n in graph {
  neighbors = countNeighbors(n);
  foreach r in requirementsSpecification {
   tracingNeighbors =
             countTracingNeighbors(n, r);

   if (tracingNeighbors/neighbors > 0.5)
        prediction(n,r) = 'trace';
   else
        prediction(n,r) = 'no-trace';
  }
}
```

We are using a 50% threshold for the basic classifier to decide between a trace and a non-trace. That means that if more than 50%[2] of a node's neighbors (aka dependent methods) are tracing to a certain requirement then we consider it very likely that the node itself is also part of the requirements implementation. We are using the example graph shown in Fig. 6 to demonstrate this classification process. The figure shows the method `buttonControl3_actionPerformed()` of class `ListFrame` in the center and its neighbors around. In order to compute classifications for this node, we would iterate through each of the 12 requirements in the VoD specification. For requirement R0, we discover that 86% of the node's neighbors (six out of seven) trace to R0 and this value is well above the threshold of 50%. That means that we would predict a trace to R0 for the evaluated method. This prediction is correct as the evaluated method is truly related to R0. This is evident in the node for `buttonControl3_actionPerformed()` which also lists R0 as one of its requirements. We want to emphasize that we base our classification on the known traces of the neighboring nodes and not on the gold standard traces of the node under investigation. Thus, we are assessing whether the traceability of neighboring nodes (nodes with call or data dependencies) have a correlation to the traceability of the node itself. Proceeding with this process, we would eventually predict traces to R0, R2, R5, and R12. A

[2] We used different thresholds and found that they trade-off precision and recall while the observations shown in that paper do not differ otherwise.

For Peer Review

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60

comparison with the gold standard traces in Figure 4 shows that a trace to R5 would be classified which is currently missing in node `ListFrame.buttonControl3_actionPerformed()`, while the trace to R10 would be identified as incorrect though it is currently present in the node. We refer to these two situations as wrong trace classifications and missing trace classifications.

D.2. An Advanced Classifier using the Inverse Data Frequency Algorithm

The results of our previous trace classification performed on `ListFrame.buttonControl3_actionPerformed()` are not perfect. We classified it falsely to trace to R5 and we falsely classified it to not trace to R10. As we discussed in Section V.C, a data dependency is composed by data types, which are shared by two methods. Are all these data types helpful for understanding requirements-to-code traceability?

For Peer Review

#	Data Type	Occurrence	idf Value
1	`MPEGDecoder.ServerReq`	15	1.7805
2	`java.awt.event.ActionEvent`	8	2.0536
3	`java.awt.list`	10	1.9566
4	`java.lang.String`	127	0.8528
5	`MPEGDecoder.Detail`	6	2.1785

Table 2. Data types shared in Figure 6

Table 3 shows all five data types shared by all data dependencies in Figure 6 (numbered from 1 to 5). The data dependencies that connect `Movie.-init-()`, `Movie.gettitle()`, and `ServerReq.getmovie()` to the center method are using data type 4; the data dependencies that connect `VODClient.init()` to the center method are using data type 1; the data dependencies that connect `ListFrameListener3.actionPerformed()` to the center method are using data type 2; and the data dependencies that connect `ListFrame.-init-()` to the center method are using data type 1, 3, and 5. The "Occurrence" column in the table shows how often a data type occurred in all data dependencies of a CDGraph. The maximum possible occurrence is the number of all data dependencies of a CDGraph (for VoD the number of all data dependencies is 905) meaning that a data type would be shared by every two pair of methods in the graph. The minimum occurrence is one; meaning only two methods in the graph share this data type. Not surprisingly, we found that data type `java.lang.String` is occuring much more often than other data types. A reasonable guess is that `java.lang.String` is a commonly shared data type to pass string information among many methods in the VoD system. This kind of data types is thus too general for understanding requirements-to-code traces. We define these data types as "commonly shared data types". The goal is to exclude them when making trace classifications. But we want to do so automatically, which is discussed next.

Next, we introduce a new classifier, which can automatically exclude commonly shared data types when making trace classifications based on our basic classifier. We call this classifier the Inverse Data Frequency (idf) classifier. This name is similar to the well-know numerical statistic called Inverse Document Frequency, which is often used as a weighing factor in information retrieval (for details refer to [17]). The idea of Inverse Document Frequency is that the more a word occurs in a collection of documents, which cannot reflect the uniqueness of each document, the less important this word is for matching relevant documents. This idea is very similar to our "commonly shared data types" if we suppose our data dependencies to be documents and data types to be words. So we define our Inverse-Data-Frequency to weigh the importance of each data type for trace classification:

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60

$$\mathrm{idf_d} = \log\frac{\mathrm{N}}{\mathrm{n_d}} \quad (1)$$

where N is the total number of data dependencies in the CDGraph and $\mathrm{n_d}$ is the occurrence of a given data type in all data dependencies. The calculated idf values for each data type in the VoD example are shown in Table 3 (for the CDGraph of VoD, the number of N is 905).

With the idf values for each data type in the CDGraph, we now introduce our Inverse Data Frequency classsifier. The idf value in the algorithm plays two roles. First, we set up a 0.9 threshold (according to the previous investigations and case study results) meaning if a data type has an idf value lower than 0.9 then we exclude this data type when making trace classifications. If all data types from one data dependency are excluded, the neighbor method that is only connected by this data dependency without any call dependency attached to the target method will be ignored when counting neighbors. Second, for data types that are not excluded we keep the idf values as extra weights when counting neighbors (for both trace and no-trace). If a data dependency is composed by multiple data types that are not excluded by the threshold of the idf value, the extra weight for the neighbor method connected by this data dependency will be the sum of all idf values. So the algorithm we use for trace classification is as follows (we still keep the 50% threshold for counting neighbors and their weights):

```
foreach n in graph {
   traceVote = noTraceVote = 0;
   foreach r in requirementsSpecification{
     foreach neighbor in n.neighborCollection{
       neighbor.idfExtraWeight = 0;
         //Remains 0 if only call dependencies exist between two methods
       foreach dataDependency between n and neighbor{
         dataTypeNum = countDataTypes(dataDependency);
         if (dataType.idfValue < 0.9)
           dataTypeNum = dataTypeNum - 1;
         else
           neighbor.idfExtraWeight += idfValue;
       }
       if (dataTypeNum == 0 && !isConnectedByCallDependency(n,neighbor))
         continue;
       if (isTracedTo(neighbor,r))
         traceVote += 1 + neighbor.idfExtraWeight;
       else
         noTraceVote += 1 + neighbor.idfExtraWeight;
     }
     if (traceVote/(traceVote + noTraceVote) > 0.5)
       prediction(n,r) = 'trace';
     else
       prediction(n,r) = 'no-trace';
   }
}
```

If we use our idf algorithm with a 0.9 threshold in Figure 6 to classify traces for `ListFrame.buttonControl3_actionPerformed()`, we would reduce the CDGraph to the one shown in Figure 7. Figure 7 just illustrates how our idf algorithm works. In the figure, two neighbors `Movie.-init-()` and `Movie.gettitle()` are ignored because the data dependencies that connect them to `buttonControl3_actionPerformed()` are composed entirely by data types that have idf values lower than 0.9 (i.e., `java.lang.String`) and there are no call dependencies that connect them to the center method (unlike method `ServerReq.getmovie()`). Now we compute the trace classifications with the calculated idf values as the extra weights for each neighbor methods. For requirement R10, we find that three out of five existing neighbor methods trace to R10 and with the extra weight for each existing data dependency, the value would become:

For Peer Review

$$\frac{(1 + 1.7805) + 1 + (1 + 1.7805 + 1.9566 + 2.1785)}{(1 + 1.7805) + 1 + (1 + 1.7805 + 1.9566 + 2.1785) + 1 + (1 + 2.0536)} = 0.7252$$

and this value is well above the threshold of 50%. Proceeding with this process, we would eventually predict traces to R0, R2, R10, and R12 and this time we make neither wrong traces classifications nor missing traces classifications.

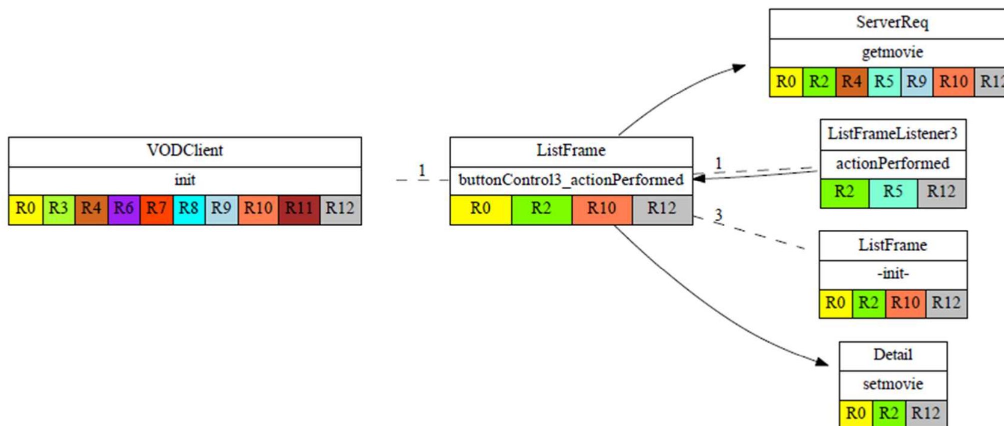


Figure 7. Sample of a Call-Data Dependency Graph to illustrate the Inverse Data Frequency algorithm. The figure shows the pruned graph after the idf algorithm was applied.

Peer Review

E. Step 4: Evaluating the Correctness of Classifications

In this section we discuss how to assess the correctness of trace classifications for each method-requirement combination in an evaluated system. In order to evaluate the correctness of a classification, the result is compared with the golden standard requirements traceability matrix. Table 4 shows the four possible combinations of classification and golden standard value. While TP (True Positive) and TN (True Negative) refer to correctly classified traces and non-traces, a FN (False Negative) refers to a missing trace and a FP (False Positive) refers to a wrong trace classification. For each evaluated system, we count how often each of the four validation results occurs. The reason for splitting classifications of traced and non-traced nodes is that for a typical system the RTM is very sparse and the number of traces compared to no-traces is very low. The splitting allows understanding where false classifications are made and to compare our results to related work (see Section III).

Classification	Golden Standard RTM	Validation Result	Correctness
Trace	Trace	TP	Correct
No-Trace		FN	Incorrect
Trace	No-Trace	FP	Incorrect
No-Trace		TN	Correct

Table 3. Result types for the validation of trace classification

The overall incorrectness (combining “trace” and “no-trace”) is defined as formula (2) and shows the percentage of correct classifications in relation to all given classifications. A value of 0% means that only correct classifications are given and a value of 100% means that only incorrect classifications are computed.

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60

$$\text{Incorrectness} = \frac{\text{FP+FN}}{(\text{TP+FN+FP+TN})} \quad (2)$$

The recall is defined as formula (3) and shows the percentage of proposed traces (ignoring "no-trace") in relation to all classifications on traced nodes. A value of 100% means that trace classifications are complete (none are missing) as compared to the golden standard RTM.

$$\text{Recall} = \frac{\text{TP}}{(\text{TP+FN})} = 1 - \text{Missing Trace Rate} \quad (3)$$

Finally, the precision is defined as formula (4) and shows the percentage of correctly proposed traces (ignoring "no-trace") in relation to all classifications on trace nodes. A value of 100% means that trace classifications are correct (none are wrong) compared to the golden standard RTM.

$$\text{Precision} = \frac{\text{TP}}{(\text{TP+FP})} = 1 - \text{Wrong Trace Rate} \quad (4)$$

For Peer Review

V. Experimental Setup

In order to answer our research questions (see Section III), we performed three experiments per each evaluated system. First, we computed classifications on a graph by considering only method call dependencies (Call Graph). This is the CDGraph minus all data dependencies. Second, we computed classifications on a graph by considering only method data dependencies (Data Graph). This is the CDGraph minus all call dependencies. Finally, we computed classifications on a complete CDGraph that contained both method call and method data dependencies (Call + Data Graph). For experiments based on a graph with data dependencies, we applied the proposed idf algorithm.

Our evaluation is based on five small to large software systems: VideoOnDemand (VoD), Chess, GanttProject [31], jHotDraw [32], and iTrust [14]. Table 2 lists basic metrics about the five systems. We chose these systems because of the availability of requirements specifications and, more significantly, "gold standard" requirements-to-code traces made available by their original developers or developers familiar with the code. The five projects are of different size and of different application domains. The systems cover about 171 KLOC and the gold standard RTMs covered in total 91 requirements with their respective requirement and use case specifications for our evaluation purposes. Our focus on a subset of the requirements does not affect the validity of the findings discussed later because each requirement was evaluated separately. Even though many methods implemented multiple requirements, it is possible to investigate each requirement separately. As can be seen from Table 2, the requirements were diverse in size, being implemented in between 0.003-13% of the code (measured by the number of methods).

	Video on Demand	Chess	Gantt Project[31]	jHotDraw [32]	iTrust [14]
Version	–	0.1.0	2.0.9	7.2	13.0
Programming language	Java	Java	Java	Java	Java
KLoC	3.6	7.2	45	72	43
Executed methods	165	317	2741	1755	258
Evaluated requirements	12	7	17	21	34
Average number of methods implementing a requirement	9-148 (45)	23-288 (173)	78-815 (387)	1-555 (121)	1-33 (12)
Size of the golden RTM	1980	2219	46597	36855	8772
Requirements traces	534	1211	6584	2547	398
Average code coverage per requirement	0.5–7.5%	1–13%	0.2–1.7%	0.003–1.5%	0.01–0.4%
Method call dependencies	210	439	4830	3848	333
Method data dependencies	905	976	30452	17316	5733

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60

Table 4. Overview of the five evaluated systems

The availability of a high quality gold standard for requirements traces was essential for this work. Since the goal is to understand the relationship between code dependencies and requirements traceability, we used the gold standard for assessing whether the various code dependencies indeed are beneficial to requirements traceability. The iTrust case study already had available high quality requirements-to-code traces which we could utilize [14]. For the other four case study systems, we had to obtain them first. We did so by recruiting 1) the original developers in case of the two larger systems: GanttProject and jHotDraw and 2) persons who were very familiar with a given system (in case of the smaller VoD and Chess). Consequently, for all five case study systems we had available high-quality Requirements-to-Code Trace Matrices (RTMs). The RTMs of four of the systems (not iTrust) were at the granularity of classes (requirements to class traces). In order to discover the correlation between requirements traceability and the communications between methods based on calling and data sharing, we propagate traces from one class to all the methods that belong to the class. This means that if a class is traced to a requirement according to the RTM, then all the methods inside this class also traced to that requirement. Meanwhile, we were also able to access a method-level RTM for the iTrust System which has been widely investigated in recent requirements traceability research. We will revisit research question 4 in section VI to find out whether different trace granularities affect our approach. Summarizing all five RTMs, we had available 11274 trace links among 91 requirements, with an average of 124 traces per requirement. Table 2 provides further details. For example, we see that the number of methods implementing a given requirement ranged from 1 method (smallest) to 815 methods (largest). Most of the requirements were functional but five of the 91 requirements were non-functional. Examples of the evaluated requirements are:

- VoD R6: The system should have a one second max response time to start playing a movie.
- Chess R5: User can select and move figure
- GanttProject R04: The user should be allowed to add or remove a task as a subtask to an existing task.
- jHotDraw R11: The user may group shapes into more complex shapes. Grouped shapes should be allowed to be ungrouped.
- iTrust UC1: Create and disable patients.

While the requirements were very diverse, Table 2 also reveals that their traces were very unlikely to be guessed. If we were to take a method and randomly choose its requirements then we would only be between 0.003–13% likely to correctly guess the requirement the method implements (number of requirements traces divided by the size of the RTM). For any automation to be useful, it would have to make significant improvement based on this random chance.

To guarantee the quality of captured code dependencies, we randomly inspected 15% of the executed methods in all five evaluated systems. We found that all the captured method call and method data dependencies for those evaluated methods were correct and we had collected enough dependencies to represent the behavior of each system for understanding requirements-to-code traces in the gold standard RTMs. The overhead of capturing both call and data dependencies by running test cases for each case study system is a one-time cost and was reasonable (20 minutes for VoD, 1 hour for Chess, 1.5 hours for jHotDraw, 3 hours for Gantt, and 1.5 hours for iTrust).

For Peer Review

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60

VI. Results and Discussion

By analyzing the five selected software systems, we computed roughly 96,000 trace classifications. Table 5 shows the results in form of the introduced metrics: incorrectness, recall, and precision (columns) for all five systems (major rows) and for call and data dependencies separately and call/data dependencies combined (minor rows repeated for each system). The results shown in the table are used to answer our research questions (see Section III).

		Correct		Incorrect		Incorrectness	Recall	Precision
		TP	TN	FN	FP			
VoD	Call	349	1249	185	197	19.29%	65.36%	63.92%
	Data	449	1395	85	51	6.87%	84.08%	89.80%
	Call + Data	454	1400	80	46	6.36%	85.02%	90.80%
Chess	Call	1118	950	93	58	6.81%	92.32%	95.07%
	Data	959	956	251	53	13.70%	79.26%	94.76%
	Call + Data	1144	949	67	59	5.59%	94.47%	95.10%
Gantt Project	Call	3926	38286	2658	1727	9.41%	59.63%	69.45%
	Data	3292	39298	3292	715	8.60%	50.00%	82.16%
	Call + Data	4391	38939	2193	1074	7.01%	66.69%	80.35%
jHotDraw	Call	1291	33739	1256	569	4.95%	50.69%	69.41%
	Data	1261	34074	1286	234	4.12%	49.51%	84.35%
	Call + Data	1606	33994	941	314	3.41%	63.05%	83.65%
iTrust	Call	240	8262	158	112	3.08%	60.30%	68.18%
	Data	139	8337	259	37	3.37%	34.92%	78.98%
	Call + Data	233	8299	165	75	2.74%	58.54%	75.65%

Table 5. Number of correct/incorrect classifications and aggregated metrics assessing the computed classifications for the evaluated systems and for the three method dependency combinations (Call, Data, and Call+Data)

Previous finding: Method call dependencies are relevant for evaluating requirements-to-code traces.

We found that by evaluating traces through method call dependencies then merely 3.08% (iTrust) to 19.29% (VoD) of the classifications were incorrect. In contrast, a randomly chosen result for a requirement would be 87-99.997% incorrect (see Table 4) because it would depend on the average code coverage of the requirement. The computed results are clearly far from random guessing and consequently the result shows a strong relationship between method call dependencies and requirements-to-code traces. We conclude that method call dependencies are relevant for evaluating requirements traces. This observation is consistent with related work (see Section IX).

A. Research Question 1: Are method data dependencies relevant for assessing requirements-to-code traces?

Looking at Table 5, we found that by purely evaluating method data dependencies 3.37% (iTrust) to 13.70% (Chess) of the classifications were incorrect. This result shows that also a strong relation exists between method data dependencies and requirements-to-code traces (compared to random guessing). In result, method data relations are relevant for understanding requirements-to-code traces (RQ1) and suggests that method data dependencies are a viable alternative to method call dependencies.

For Peer Review

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60

B. Research Question 2: Are method call and method data dependencies complementary in assessing requirements-to-code traces?

This is the key question because if method data dependencies are quite similar to method call dependencies then we would not benefit from capturing both. A benefit would exist only if the two where complementary to a large degree. In Table 5 we see that by utilizing both types of dependencies and our advanced classification algorithm only 2.74% (iTrust) to 7.01% (VoD) of the computed classifications were incorrect. These are the best results across all five systems, suggesting that method call and method data dependencies are in fact complementary (RQ2). We see a strong benefit for both precision and recall. Looking at Table 5, we also notice that the recall for Call+Data is almost always far above the recall for either Call or Data individually (except for iTrust which is nearly the same as the recall for Call). The same is true for the precision (except for GanttProject, jHotDraw, and iTrust which are nearly the same as the precision for Data). The average increase of both precision and recall by considering both call and data dependencies is 19.80% compared to considering call dependencies only and 13.10% compared to considering data dependencies only. This suggests that the trace classifications computed for Call+Data leverage from the strengths of both Call and Data individually – i.e., Call and Data observations are complementary. This fact is also observable through Table 6. The table shows the number of call dependencies (left column), the number of data dependencies (middle column), and the overlapping dependencies that are present in the call and the data dependency set (right column) for the five evaluated systems. The table shows a comparably small number of overlapping dependencies.

	Call dependencies	Data dependencies	Overlapping dependencies
VoD	210	905	66
Chess	439	976	121
GanttProject	4830	30452	1120
jHotDraw	3848	17316	981
iTrust	333	5733	65

Table 6. Numbers and overlap of call and data dependencies in the CDGraphs of the five case study systems

C. Research Question 3: Is the complementary effect of method call and data dependencies for assessing requirements-to-code traces affected by trace granularity?

As we discussed in Section V, we had available (gold standard) RTMs with coarser-grained class-level traces for four of the five evaluated systems (VoD, Chess, GanttProject, and jHotDraw). The traces for these systems were automatically propagated from classes to their containing methods for our analyses. Only for the iTrust system we had available a (gold standard) RTM with finer-grained method-level traces. To assess whether method-level traces lead to better classifications compared to

For Peer Review

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60

class-level traces, we created a class-level RTM for iTrust and performed our assessment with those trace granularities. Abstracting traces from method-level to class-level is straightforward (the reverse is not). For example, consider a class C containing two methods: m1 and m2. Method m1 traces to requirement R1 and method m2 traces to requirement R2. Abstracting traces means that class C traces to both R1 and R2. Class level traces thus are the union of their methods' traces.

			Correct		Incorrect		Incorrectness	Recall	Precision
			TP	**TN**	**FN**	**FP**			
iTrust	Class-level RTM	Call	308	7487	660	317	11.14%	31.82%	49.28%
		Data	265	7731	703	73	8.85%	27.38%	78.40%
		Call + Data	358	7648	610	156	8.73%	36.98%	69.65%
	Method-level RTM	Call	240	8262	158	112	3.08%	60.30%	68.18%
		Data	139	8337	259	37	3.37%	34.92%	78.98%
		Call + Data	233	8299	165	75	2.74%	58.54%	75.65%

Table 7. Number of correct/incorrect classifications and aggregated metrics assessing the computed classifications for the iTrust system and for the three method dependency combinations (Call, Data, and Call+Data) based on class-level and method-level RTMs

Table 7 shows results for iTrust's class-level RTM and the original method-level RTM. The table shows that by using the class-level RTM with combined method call and method data dependencies, 8.73% of the computed classifications were incorrect. In contrast, by using the finer-grained (and more precise) method-level RTM, the result improved to 2.74% incorrectness, which is the best value observed across the different evaluation scenarios. However, the observations (i.e., method-level RTMs always yielded better results than class-level RTMS) of RQ1 are still valid if we consider call and data dependencies separately. Also the observations of RQ2 remain valid as we also found a complementary effect of method call and method data dependencies for assessing requirements-to-code traces. This particular complementary effect is not influenced by trace granularity (RQ3). This finding suggests that the trace assessment framework is applicable to software systems with RTMs of different trace granularities. Another finding is that finer trace granularities allow for better assessment results (results based on a method-level RTM are more correct than results based on a class-level RTM), suggesting that the assessment can benefit from the effort of building a more fined-grained RTM (though building a method-level RTM costs more than building a class-level RTM, for details refer to [27]).

D. Research Question 4: Is the complementary effect of method call and data dependencies affected by trace incorrectness?

Thus far, we used the gold standard RTMs as both the input and the oracle. In practice, traceability assessment would be performed based on imperfect RTMs as input – RTMs that are both erroneous and incomplete. An erroneous RTM contains both wrong (wrongly suggested traces) and missing (wrongly suggested no-traces) traces. An incomplete RTM contains undecided cells where no input was provided (it is neither trace nor no-trace). It is important to study the effect that imperfect input traceability can have on the assessment results. Therefore, we wanted to find out whether the complementary effect of method call and method data dependencies is affected by trace incorrectness (RQ4) or incompleteness (RQ5). Considering the type of assessment that we perform, a random error is easily identifiable while a set of coordinated errors could substantially influence the results of the classification based on neighbor counting. For example, to change a likely trace to an unlikely no-trace, the majority of traces

For Peer Review

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60

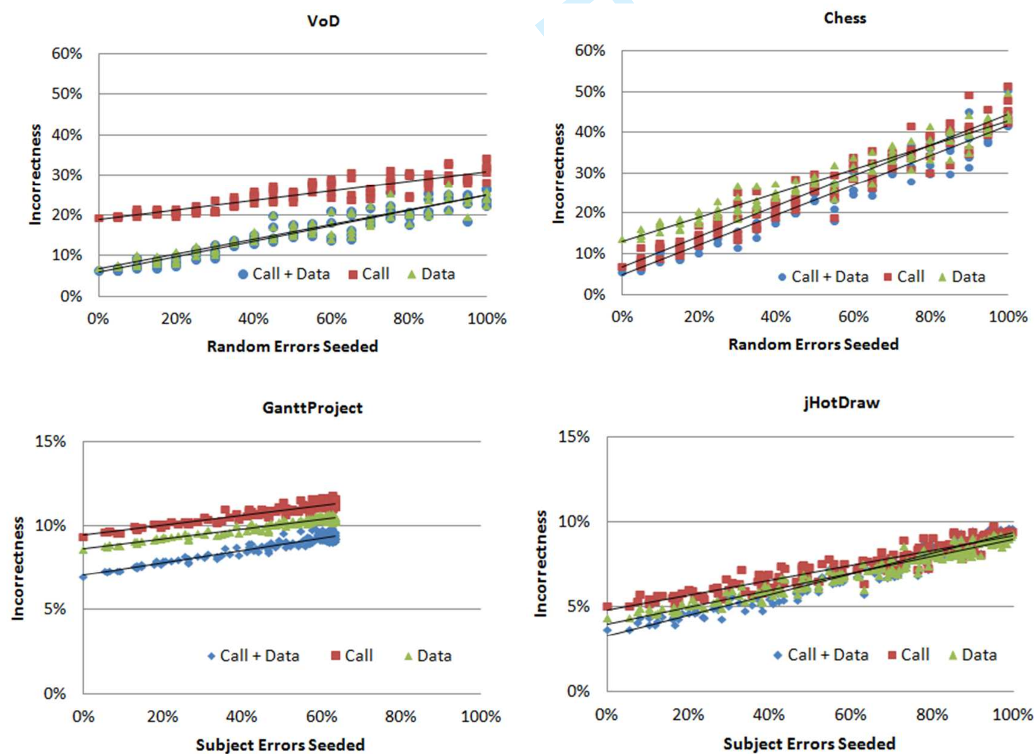
from neighbor methods have to change to no-traces under the 50% threshold, which is an unlikely scenario and vice versa. However, it is not clear whether RTM errors in practice are randomly distributed or concentrated in certain areas of the RTM.

To obtain realistic RTMs, we conducted an experiment with 85 subjects, which had to identify requirements-to-code trace for GanttProject and jHotDraw. The subjects were asked to inspect a subset of the specification and the code for both systems and had to create traces between them. The obtained traces were compared to the original RTM. By merging multiple subset RTMs created by our subjects, we could obtain complete RTMs for both systems. This process fits the industrial practice where traces are often captured by multiple developers. The many combination possibilities allowed us to obtain a wide spread of RTM qualities.

For the other three systems we used random seeding instead, meaning that we randomly picked cells in the gold standard RTM and inverted them.

In both cases, random error seeding was measured as a percentage relative to the total number of traces. For example, if a requirement had 100 traces then 10% errors mean that 10 of these 100 traces are either missing or wrong. In order to balance the effect from both wrong and missing traces, we seeded the same number of wrong and missing traces into the gold standard RTM. So in a 10% erroneous RTM, 5% of the errors are wrong traces and the other 5% are missing traces. We gradually (5%) increased the level of error seeding and obtained randomly 5 erroneous RTMs at each level.

We assessed the obtained RTMs for all five systems using the advanced classifier with the idf algorithm. The classification results were compared with the gold standard RTMs. Similar to the previous experiments, we performed assessments by using only call dependencies, by using only data dependencies, and by using all dependencies. Figure 7 depicts the Incorrectness in relation to the percentage of errors introduced by subjects or random error seeding for five evaluated systems.



For Peer Review

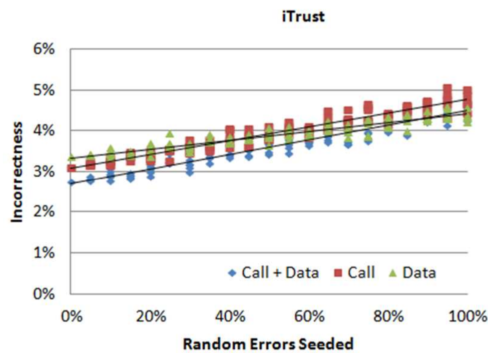


Figure 7. Classification incorrectness in relation to the validated erroneous RTMs for the five evaluated systems and for the three method dependency combinations (Call, Data, and Call + Data)

For Peer Review

Across all evaluated systems, we found that the incorrectness of classifications rises linearly with the amount of seeded errors. At the maximum level of trace errors the incorrectness of assessment results differs quite significantly across the different systems. The highest incorrectness for Chess was 51.53% while the highest incorrectness for iTrust was merely 5.05%. The reason for this variance lies in the fact that the sizes of implementations (the ratio of trace vs. no trace cells in RTMs) differ widely across the five systems. Table 4 shows that there are 1211 traces for Chess in the gold standard RTM containing 2219 cells in total. Seeding 100% errors to this RTM compares to an incorrectness of 54.57% of this RTM. In contrast, iTrust's gold standard RTM contained 398 traces distributed across 8772 cells in total. So the incorrectness of the erroneous RTM with 100% seeded errors is only 4.54%.

Despite the variance of incorrectness, the most important observation is that classification results of all five systems based on both call and data dependencies are almost all below those computed based on call dependencies or data dependencies separately, except for some erroneous RTMs of VoD, jHotDraw, and iTrust where the percentage of error seeding reached to 100%. We also found that no matter whether the erroneous RTMs were created by subjects or by random error seeding, the classification results on these RTMs behave similarly. This demonstrates that the complementary effect remains unaffected by trace incorrectness (RQ4).

E. Research Question 5: Is the complementary effect of method call and data dependencies affected by trace incompleteness?

In practice, trace validation can be seen as an ongoing process that must not necessarily start with a complete RTM but often starts with traceability knowledge that is only partially available. This subsection investigates whether and how trace incompleteness is affecting the effect of method data dependencies for trace validation. The incompleteness of a RTM refers to the number of cells that are neither traced nor explicitly not traced. Such cells could be randomly distributed across the RTM, they could be aligned in rows when methods have not been traced at all, or they could be aligned in columns when requirements have not been traced at all.

To study the effect of incompleteness, we selected random seeding that means removing random cells from a gold standard RTM to create an incomplete RTM. We gradually increased (5%) the level of incompleteness seeded to gold standard RTMs and obtained at each level of incompleteness five

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60

randomly generated RTMs for validation.

Similar to the trace classification on erroneous RTMs, we performed the classification first on a call dependencies only (Call), then on data dependencies only (Data), and finally on call + data dependencies (Call + Data). Figure 8 depicts incorrectness of classification results in relation to increasing incompleteness of the validated RTMs.

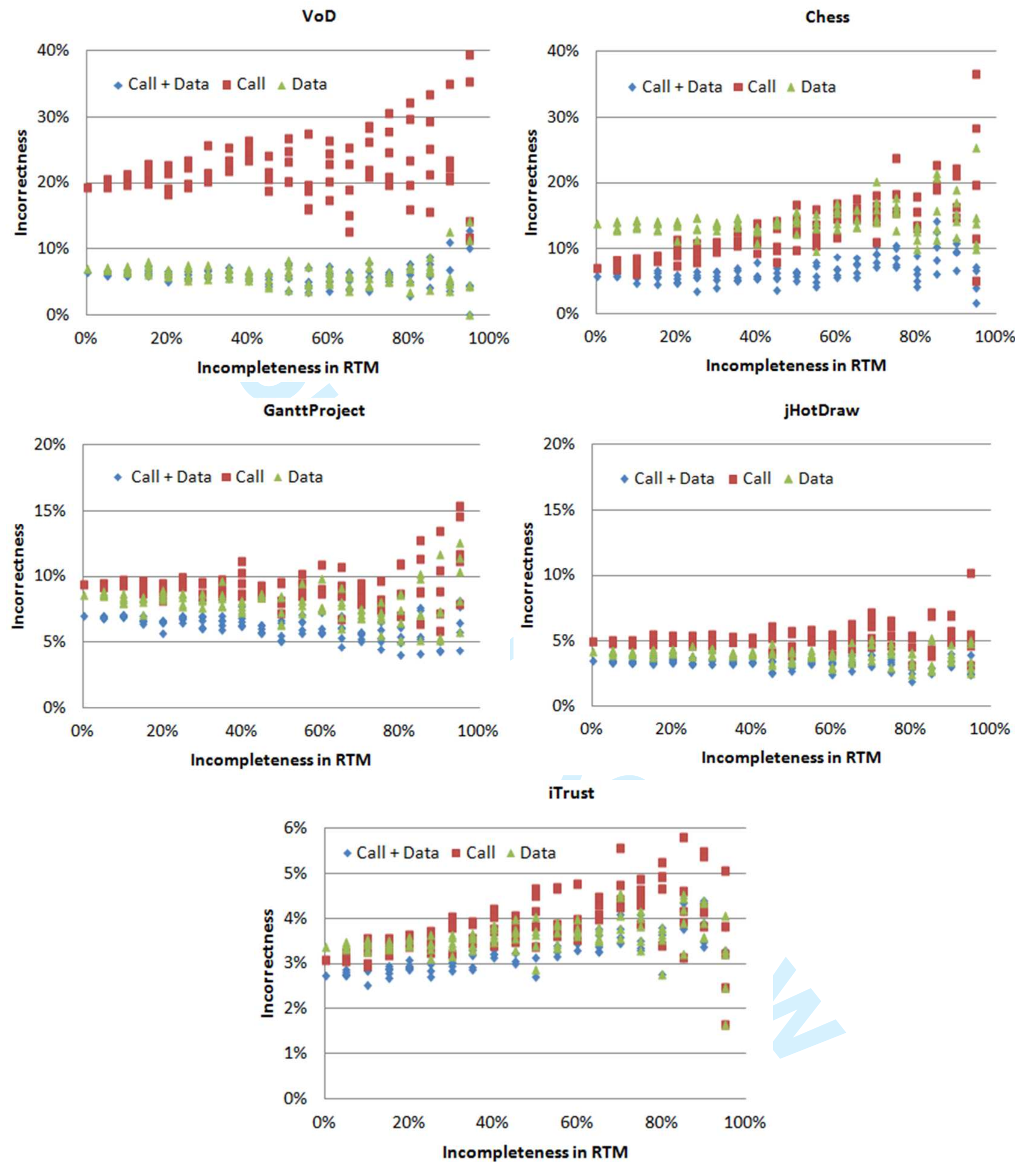


Figure 8. Classification incorrectness in relation to the incompleteness of validated RTMs for the five evaluated systems and for the three method dependency combinations (Call, Data, and Call + Data)

Our observation across all evaluated systems is that the incorrectness of classifications is almost unaffected by incompleteness up to a level of around 75% RTM incompleteness where the incorrectness significantly increases. That is somewhat surprising as it says that the overall level of correctness is not affected unless more than 75% of the input RTM is incomplete. This is very beneficial because it implies that the validation could be used even at the early stages of trace capture and maintenance. We also found that the incorrectness classifications based on both call + data dependencies remains lower than

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60

the results based on call and data dependencies separately. That finding implies that the complementary effect of call and data dependencies is not affected by trace incompleteness (RQ5). This is also beneficial to trace generation techniques as such techniques can take advantage of both call and data dependencies for their analyses starting from highly incomplete sets of automatically retrieved traces.

VII. Limitations and Threats to Validity

1) Quality and Completeness of Gold-Standard RTMs

A possible threat to validity of our results is the incompleteness of code dependencies due to missed call and data dependencies. We did not have available the RTMs for all requirements of the various systems and hence we restricted our analysis to subsets of these systems. However, we consider this incompleteness not as a serious threat regarding the analysis of complementary effect of call and data dependencies , as both should have equally "suffered" from this problem. Regarding our other findings, missing dependencies could have negatively impacted classification results leading to too pessimistic observations.

2) Data Records without Unique Identifier

We faced the problem of handling data records without a unique identifier when capturing method-using-data records, such as static fields and local variables of Java primitive types (e.g., int, double, boolean). Static fields are easy to identify because a static field is initialized only once when its owner class is loaded and this field can be accessed directly by the class name and does not need any object. We use the type of this field, the name of this field, and the name of the class in which the static field is declared to identify a given static field (including static fields with primitive types). For a given non-static field with primitive types, we can first locate the object that owns this field via its hash code value and then identify this field with the type and name of it inside its owner object. Unfortunately, there is no unique identifier for local variables of primitive type declared inside methods and we could not capture data dependencies involving them. Here we made an assumption that developers tend to use more complex data structures (instead of primitive types) to organize meaningful messages that would be shared by methods. Another possible situation is that developers typically use primitive types as flags (type `boolean`) or counters (type `int`) across methods. However, in this situation these primitive types are more likely to be fields inside classes and they are captured by our tool. Our experiments have shown that we did capture enough other data dependencies to demonstrate the strong benefit of combining call and data dependencies. More data dependencies might have tilted the balance even stronger in favor of data (affecting research question 3 mostly) but we doubt that it would have changed the primary message about the complementary nature of call and data dependencies.###

3) Selection of Classification Algorithms

Throughout the research project, we implemented multiple trace classification algorithms. We discussed two of them in our previous work [16]. The previously proposed algorithms perform similarly on the evaluated systems except for iTrust where they resulted in a large quantity of miss-classifications. We analyzed the problem and proposed the idf algorithm to filter out "commonly shared data types". The trace classification results for iTrust by using the idf algorithm then lead to the same principal observations that we discovered for the other four projects. Moreover, the newly proposed idf algorithm improves trace classification results on all five evaluated systems. Since the general observations across all algorithms and all projects remain the same, we consider our observations regarding the research questions not to be biased by the selection of classification algorithms.

For Peer Review

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60

4) Number and Selection of Cases

We used five systems of different domains (shown in Section V) with RTMs of different sizes and trace granularities. Each cell in a RTM was validated separately and this large number of cells (96423 cells in total) makes our findings statistically representative. To assess precision, recall, and incorrectness we relied on a gold standard RTM provided by the original developers of the studied systems. We had no control over the requirements and RTMs we were given. The gold standard RTM was most likely not perfect, but we evaluated it for obvious inconsistencies and did discover none. Furthermore, the trace classification results exhibited roughly the same effects on all five evaluated systems. We made these observations despite the fact that different developers in different domains developed the evaluated systems. This leaves us to the conclusion that our findings are representative.

5) Artificial Incorrectness Seeding on VoD, Chess, and iTrust

For the experiments that aimed to find out whether the discovered complementary effect of call and data dependencies is affected by trace incorrectness (RQ4), we generated erroneous RTMs to the gold standard RTMs by merging subject errors (for Gantt and jHotDraw) and by seeding errors randomly (for VoD, Chess, and iTrust). We found that classification results of RTMs with randomly seeded errors behave similar to those RTMs with human-created errors. We consider the use of random seeding for the three smaller systems as justified.

6) Granularity of Method Data Dependencies

We identify a method data dependency if two methods share the same object in memory. However, for trace classification we only used the data type of shared objects to represent data dependencies. All shared objects of the same type are aggregated. Initial experiments showed that a granularity on the data type level conveyed comparable information as data dependencies on the object level and lead to considerable less computational effort.

7) Semantics of Method Data Dependencies

Previous work showed that the semantic of a discovered call dependency between two methods in relation to a requirement can be manifold [13]. For example, if a method A implements a requirement R1 and method A calls a method B then two possible meanings apply: 1) method B implements a service required by method A and therefore implements R1 as well; or 2) method B implements another requirement R2 that is meant to coincide when A occurs. Trace capture approaches that analyze call dependencies (such as [10] and [12]) handle this ambiguity through majority voting and aggregation. Similarly, if method A and B share a data type T and method A implements requirement R1, then one of two meanings applies: 1) method A is communicating data type T with B to implement requirement R1 and therefore method B also implements requirement R1; or 2) method A passes an common message of type T to method B and therefore method B does not implement requirement R1. Our proposed trace classification methods also apply majority voting. We proposed the basic trace model to handle the ambiguities of both call and data dependencies by neighbor counting. However, if a data type is shared too many times among methods, then this "commonly shared data type" will mislead our basic trace model by considering too many irrelevant methods as neighbors. The idf algorithm excludes these "commonly shared data types".

8) Threshold Determination for the idf Algorithm

We propose the idf algorithm, which excludes data types that have idf values lower than a threshold when using the basic trace model to make trace classifications. We set this threshold to 0.9 according to our manual investigation and case study results. We actually tried several ways to determine this threshold automatically such as excluding data types with the top five lowest idf values or with the

For Peer Review

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60

bottom 5% idf values. These efforts were not helping and 0.9 was the only threshold that can balance the performance of classification results for all evaluated systems on all three metrics. Since the main focus of this paper was to find out whether method data dependencies are helpful in understanding requirements-to-code traceability instead of optimizing the performance of our proposed approach, seeking a mechanism for optimizing the threshold of the idf algorithm will remain a future effort.

VIII. Related Work

The method data dependencies that our study focused on exist in the memory of an application and facilitate data sharing among methods. Our work is particularly concerned with dataflow among methods because we assess traceability links on the method level of source code. Extracting dataflow from source code is a research hotspot and considerable work has been done in this field. A specific topic in this field is program slicing. Program slicing techniques [18, 19] aim to obtain a reduced, executable program from a given program by removing statements so that this generated program replicates part of the behavior of the initial program. Slicing techniques can provide dataflow communications among statements and thus have been widely used in the area of software engineering, for example, for program understanding and impact analysis (e.g., [20]). Points-to analysis [15] is another popular technique, which aims to identify all pointers or heap references that can point to certain variables or storage locations of an application. Similarly to points-to analysis, we also capture method data dependencies based on whether two methods share the same area in memory. Milanova et al. [21] extended Andersen's static analysis technology [15] to extract points-to information from Java. All five systems evaluated for our study are also written in Java. However, we use the hash code of in-memory objects, which is collected during runtime and represents a unique id for each memory location, to capture data dependencies among methods. Lienhard et al. [22] analyzed execution traces and extracted an Object Flow Graph (OFG) in which edges represent objects, and nodes represent code structures (either classes or groups of classes). For our CDGraph, we also generate data edges via method-using-data records representing data types shared by two methods. The main difference between points-to analysis and our tool for capturing data dependencies is that points-to analysis focuses on the relationship among objects or classes while our work focuses on data dependencies among methods. Neither slicing nor points-to analysis provide method data dependencies as we analyzed them for this paper. Therefore, we built a new prototype runtime capture tool to recover both method call and method data dependencies based on JVMTI events.

For Peer Review

Requirements-to-code traceability and the activity of retrieving them, often referred to as feature location [10], also received considerable attention by the scientific community. Information retrieval (IR), perhaps to date the most widely applied and studied technology in the traceability community, identifies traces based on naming similarities between source code and other software artifacts, like requirements [23-26]. However, an important issue hindering the performance of IR techniques when applied to the recovery of traceability is the problem of vocabulary mismatch between source and target artifacts (like requirements and code). This problem is still the focus of ongoing research in the field with various ideas being pursued. For example, Marcus et al. [24] applied latent semantic indexing (LSI) to locate domain concepts for a given system and to preclude term combinations that are less frequently occurring in a given document. Cleland-Huang et al. [25] presented three strategies to enhance the matching results generated by their IR model based on probabilistic networks. The key idea of those works is introducing extra information when matching requirements (such as the section name of a given

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60

requirement) and code elements (such as the package name of a given class or method) and to exclude keywords promoting wrongly retrieved traces. Gethers et al. [26] proposed an IR-based approach that integrates orthogonal information generated by relational topic modeling (RTM), which defines a comprehensive method for modeling interconnected networks of documents in order to achieve a complementary effect for improving traceability recovery. This approach has similarities to our work because we use a graph of methods that are connected by call and data dependencies and we also found a complementary effect by considering two kinds of method dependencies for computing our trace assessments. However, all discussed approaches require rich requirements descriptions and well-documented code. It is important to note that these approaches did not consider the role of code dependencies, let alone demonstrate a benefit in combining call dependencies and data dependencies to better understand requirements-to-code traceability. .

There is previous work that incorporates the analysis of calling relationships between methods to get better understanding of code and their traceability to requirements. Biggerstaff et al. [8] built a prototype to help people assign human concepts back to the code including an editable call graph. Zhao et al. [10] proposed an approach (SNIAFL) using a call graph with additional branch information to refine traces achieved by information retrieval. Hill et al. [11] used lexical analysis and call graph exploration in a tool called Dora to perform software maintenance tasks. Ghabi and Egyed [12,13] propose an approach to maintaining requirements-to-code traces by validating them in context of code calling relationships. The authors argue that requirements are typically implemented in methods that directly or indirectly communicate; the concept is called requirements regions.

Calling relationships are not the only means of method communication. McMillan et al. [7] created an approach called Exemplar to find highly relevant software projects from large archives of applications based on natural-language query by considering the description of applications, the API calls used in the applications, and the data flow among these API calls. They studied data flow in a similar context of requirements-to-code traceability as our study did. However, their data flows are approximated and exhibit false positives and false negatives. The authors conclude that data flows do not appear to benefit requirements-to-code traceability. This finding is contradictory to our observations. A possible explanation is a potential high number of false and missing data flow dependencies in their study (which is mentioned by the authors in the paper). Furthermore, the ambiguity of method data dependencies (see Section VII, part 7) may have also prevented them from finding all data flows, since they treated all data types equally.

All discussed work on code dependencies [7, 8, 10-13] focused either on control flow or on data flow to improve the quality of the trace recovery process based on information retrieval. Yet, our focus was not on finding a well performing algorithm and on automatically identifying or validating traces. Instead we were focusing on whether call and data dependencies are useful, separately and combined, in assessing requirements-to-code traceability.

In earlier work [13], we focused only on calling dependencies between methods in order to identify regions in the source code that implement a given requirement. We found that requirements in fact were implemented in connected areas of the source code rather than randomly distributed. In a follow-on publication [4] we introduced a surroundedness property to requirements regions. There, we found that a given method typically shares the same traces to requirements as the methods it calls or are called by it (i.e., its neighbor methods). In this work we built upon those observations and investigated method data dependencies in order to augment the meaning of a “neighbor method” – i.e., a method that may be related by call or data dependencies. We showed that these two kinds of method dependencies are

For Peer Review

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60

complementary to each other and together they help to better understand where a requirement is implemented in the source code.

IX. Conclusions

In this paper, we investigated the question whether method data dependencies are related to requirements in a manner that is similar to method call dependencies. For example, if two methods do not call one another, but do have access to the same data then is this information relevant? We formulated five research questions and validated them on five software systems, covering about 171 KLOC. Our findings are that method data dependencies are equally related to requirements as method call dependencies. But, most interestingly, our analyses show that method data dependencies complement method call dependencies. That means by evaluating both we reached the best understanding of how a set of methods is related to a requirement and the improvements were considerable. Furthermore, we demonstrated that the complementary effect is not affected by trace incompleteness and incorrectness. Our findings benefit requirements traceability practice and research in several ways. It could be used to improve trace capture, trace maintenance, and trace validation (as was demonstrated in this paper). Furthermore, other research directions such as program understanding can benefit from the combined knowledge of call and data dependencies. This work thus benefits the research community to encourage further research in combining call and data dependencies. The tool for capturing data dependencies is available at http://www.sea.jku.at/tools.

For Peer Review

Acknowledgements

We would like to thank all participants of our user studies. 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; by the U.S. National Science Foundation (NSF) CNS Award: 1126747 and the Oversea Scholar Fund of State Key Laboratory for Novel Software Technology at Nanjing University; and the Austrian Science fund (FWF) grant: P 23115-N23.

References

[1] 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

[2] Dit, Revelle, M. Gethers, and D. Poshyvanyk, “Feature Location in Source Code: A Taxonomy and Survey”, *Journal of Software Maintenance and Evolution: Research and Practice (JSME),* 2011

[3] M. Marin, A. V. Deursen, and L. Moonen. Identifying crosscutting concerns using Fan-In analysis. *ACM Transactions on Software Engineering and Methodology (TOSEM),* 17(1), pp. 3:1-3:37, 2007

[4] A. Ghabi and A. Egyed. "Observations on the connectedness between requirements-to-code traces and calling relationships for trace validation," *in 26th International Conference on Automated*

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60

Software Engineering (ASE), Lawrence, Kansas, 2011, pp.416-419.

[5] C. McMillan, D. Poshyvanyk, and M. Revelle, "Combining Textual and Structural Analysis of Software Artifacts for Traceability Link Recovery," in *ICSE Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE), Vancouver, Canada,* 2009, pp. 41-48.

[6] 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.

[7] 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

[8] T. J. Biggerstaff, B. G. Mitbander, and D. Webster, "The concept assignment problem in program understanding," in *International Conference on Software Engineering (ICSE 1993)*, Baltimore, Maryland, USA, pp. 482–498, 1993.

[9] D. Kim and J. Kim, "Design and implementation of a Java-based MPEG-1 video decoder", IEEE Transactions on Consumer Electronics, 45(4), pp. 1176-1182, 1999.

[10] 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.

[11] 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.

[12] 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, pp. 200-209.

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

[14] iTrust System: http://agile.csc.ncsu.edu/iTrust/wiki/doku.php?id=requirements

[15] L. O. Andersen, "Program Analysis and Specialization for the C Programming Language". PhD thesis, DIKU, University of Copenhagen, 1994.

[16] 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.

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

[18] M. Weiser. "Program slicing". *IEEE Transactions on Software Engineering*, 10(4):352-357, July 1984.

[19] B. Korel and J. Laski, "Dynamic slicing of computer programs". *Journal of Systems and Software*, 1990, 13(3): 187-195.

[20] P. Tonella, "Using a concept lattice of decomposition slices for program understanding and impact analysis". *IEEE Transactions on Software Engineering*, 2003, 29(6): 495-509.

[21] A. Milanova, A. Rountev, and B. G. Ryder. "Parameterized Object Sensitivity for Points-To Analysis for Java". *ACM Transactions on Software Engineering and Methodology*, 14(1), pp. 1-41, 2005.

For Peer Review

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60

[22] A. Lienhard, S. Ducasse, and T. Gîrba. "Taking an object-centric view on dynamic information with object flow analysis". *Journal of Computer Languages, Systems and Structures (COMLAN)*, 35(1), pp.63-79, 2009.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] A. Egyed, P. Grünbacher, M. Heindl, et al. "Value-based requirements traceability: Lessons learned", in *Design Requirements Engineering: A Ten-Year Perspective*. Springer Berlin Heidelberg, 2009: 240-257.

[28] W. Kong, J. Hayes, A. Dekhtyar, and J. Holden. "How do we trace requirements? an initial study of analyst behavior in trace validation tasks." *In Fourth International Workshop on Cooperative and Human Aspects of Software Engineering*, May 2011.

[29] D. Binkley. "Source Code Analysis: A Road Map". In *Future of Software Engineering (FOSE '07),* 2007, pp. 104-119.

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

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

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

For Peer Review