Do Data Dependencies in Source Code complement Call Dependencies for Understanding Requirements Traceability?

Hongyu Kuang[1,2], Patrick Mäder[2], Hao Hu[1], Achraf Ghabi[2], LiGuo Huang[3], Lv Jian[1], and Alexander Egyed[2]

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

[2]Institute of Systems Engineering and Automation
Johannes Kepler University
Linz, Austria
firstname.lastname@jku.at

[3]Dept. of Computer Science and Engineering
Southern Methodist University
Dallas, TX 75275, USA
lghuang@lyle.smu.edu

***Abstract*—It is common practice for requirements traceability research to consider method call dependencies within the source code (e.g., fan-in/fan-out analyses). However, current approaches largely ignore the role of data. The question this paper investigates is whether data dependencies have similar relationships to requirements as do 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 several research questions and validated them on three large software systems, covering about 120 KLOC. Our findings are that data relationships are roughly equally relevant to understanding the relationship to requirements traces than calling dependencies. However, most interestingly, our analyses show that data dependencies complement call dependencies. These findings have strong implications on all forms of code understanding, including trace capture, maintenance, and validation techniques (e.g., information retrieval).**

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

I. INTRODUCTION

Requirements traceability refers to the practice of capturing relations between artifacts of a development process as traceability links. These links can support stakeholders in development-related tasks if they are of high quality. Ensuring high quality traceability links is especially difficult for requirements–to-code traces due to typically large numbers of required traces and frequent changes to the traced code.

Requirements traceability research has thus started to focus on control dependencies within source code in order to gain more information on what code elements contribute to the implementation of a requirement and to assess whether existing traceability relations are correct [1]. Among others, researchers use fan-in/fan-out analyses [2], identified typical patterns of requirements implementation, [3] and complement keyword matching techniques on the code with control flow analyses [4,9]. All the work we found is based on either statically or dynamically analyzing sequences of method calls in order to deduct dependencies between methods. These analyses, essentially, investigate the callers and callees of a method in order to assess traceability between a method and a requirement. However, method calls are just one form of communication in source code. Only few current approaches (e.g., [13]) consider the role of data sharing for assessing traceability.

978-1-4673-2312-3/12/$31.00 ©2012 IEEE

This paper investigates whether method data dependencies are as relevant as method call dependencies and, if yes, whether call and data dependencies are complementary for assessing requirements-to-code relationships in alleviating each other's weaknesses. 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 several research questions and validated them on three large software systems, covering about 120 KLOC. The validation is based on the investigation of 4,767 methods in context of 50 requirements, resulting in a total of 86,866 assessments.

To answer the research questions, we considered all methods that were connected by either call or data dependencies and for which we had initial requirements-to-code traces. For any given method we then used the traces of its neighboring methods to derive a trace recommendation for that given method and compared it with the initial trace. These initial traces could have been created manually; they could be old versions of earlier releases, or automatically generated ones. The recommendation is thus never biased by its initial trace but solely determined from the initial traces of its neighbor methods which makes it useful for activities such as trace capture or trace maintenance. These applications are explored in future work.

Our findings are that data relationships have a slightly weaker, but still strong relationship to requirements traceability. But, most interestingly, our analyses show that data dependencies complement call dependencies strongly. If we separate between precision and recall (i.e., wrong trace rate vs. missing trace rate), we find that the combined analysis results in the better for the call/data analyses. This observation is of particular importance because the traceability research community has found that complementary techniques usually benefit either precision or recall but not both. These findings thus have strong implications on all forms of code understanding, including trace capture, maintenance, and validation techniques (e.g., information retrieval).

The remainder of this paper is structured as follows. Section II briefly introduces the concepts of requirements traceability and code dependencies. Section III states our research questions and Section IV introduces the software

systems that we evaluated for answering those questions. Section V discusses the developed framework for capturing and analyzing code dependencies. Section VI reports the results of our experiments and answers the research questions. In Section VII we discuss possible improvements to the applied recommendation algorithm. Section VIII refers to limitations of our work. Section IX discusses related work in the area of requirements traceability, feature location, and program analysis. Finally, Section X concludes and proposes the practical implications of this paper.

II. Traceability and Code Dependencies

A. Requirements to Code Traceability

A *traceability link* captures where in the source code a requirement is implemented. This is similar to feature mapping if we think of requirements as features [4,8,9]. In this paper we focus on requirements to source code mappings at the granularity of methods. However, our observations should apply for traces at other levels of granularity also, i.e., on class or on package level.

A traceability link typically captures the relationship of individual requirements and methods. But, of course, a requirement can be implemented by multiple methods. Thus, multiple trace links (or short traces) may exist for the same requirement where each trace relates to a different method. Furthermore, a method can be implemented by multiple requirements. Accordingly, multiple traces may exist from different requirements to the same method. Trace links are typically captured in the form of a requirements traceability matrix (RTM), which captures in each cell one traceability link. RTMs thus contain n*m cells where n is the number of requirements and m is the number of related code elements (classes or methods).

TABLE I. Excerpt of the Requirements Traceability Matrix (RTM) of the Video on Demand System

	R0	R2	R10
VODClient.init()	X		X
ListFrame.buttonControl3_actionPerformed()	X	X	X
ListFrame()	X	X	X
Movie.gettitle()	X	X	
ListFrameListener3.actionPerformed()		X	

Table 1 depicts an excerpt of such a RTM for the VoD system, one of the three case study systems we will discuss later (see Section IV). VOD, which stands for Video-on-Demand system [12], supports basic operations such as selecting a movie from a server, playing that movie, pausing it, etc. The VoD system is only about 3.6 KLOC in size and the smallest of the three systems we analyzed. However, being an intuitive system, we use excerpts of VoD throughout this paper as an illustration. The RTM in Table 1 depicts a few methods (rows) and a few 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 methods such as the `ListFrame`’s constructor do. While the method `VODClient.init()` does not contribute to the implementation of R2, it does trace to requirements R0 and R10.

B. Capturing Dependencies Between Traced Methods

There are two general ways in which methods can be dependent upon another: (1) calling each other and (2) sharing data. 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()`.

```
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 through which the data is accessed are the same. This complex 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 `List Frame` 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 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 upon another. Such data dependencies can help to reveal traceability, because data dependencies much like control dependencies help identify related functionality.

III. Research Questions

It has been argued that requirements are typically implemented in methods that directly or indirectly communicate (so-called requirements regions [10]). However, in previous work, only method call dependencies were used to understand communication. The goal of our work is to evaluate the usefulness of method data dependencies for understanding requirement traces. We are interested in their usefulness in general and in relation to method call dependencies. According to that goal, we formulated five research questions:

1) *Are method call dependencies relevant for evaluating requirements traces?*
2) *Are method data dependencies relevant for evaluating requirements traces?*
3) *Are method call dependencies more relevant than method data dependencies for requirements traces?*
4) *Are method call and method data dependencies complementary to each other in evaluating traces?*
5) *Are additional code characteristics relevant for evaluating requirements traces?*

We will investigate these research questions on three different software systems (next section) and will discuss results in Section VI.

TABLE II. Information on the three evaluated systems

	Video on Demand	Gantt Project	JHot-Draw
Version	–	2.0.9	7.2
Programming language	Java	Java	Java
KLOC	3.6	45	72
Executed methods	169	2930	1668
Evaluated requirements	12	17	21
Number of methods implementing a requirement (avg.)	11–152 (46)	79–932 (405)	9–515 (126)
Size of the golden RTM	2028	49810	35028
Requirements traces	560	6892	2424
Random chance of guessing	0.5–7.4%	0.1–1.8%	0.02–1.4%
Method call dependencies	222	5560	3943
Method data dependencies	899	24243	14555

IV. Evaluated Software Systems

Our evaluation is based on three real-world software systems: VideoOnDemand (VoD), GanttProject, and JHotDraw. Table 2 lists basic metrics about the three systems. We chose these systems because of the availability of requirements specifications and, more significantly, high quality requirements-to-code traces. The three open-source projects are of different size and of different application domains. The systems cover about 120 KLOC and we randomly selected 50 requirements from their respective requirements specification 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 individually. 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 0.02-7.4% of the code (measured by the number of methods).

A. Capturing of Requirements Traces

Having available a high quality gold standard for requirements traces is essential for this work because it is the goal to understand the relationship between code dependencies and requirements traceability and we would not expect finding relationships among bad input (garbage-in/garbage-out). In order to capture high quality traces, we asked the original developers of the evaluated systems (in case of the larger systems GanttProject and JHotDraw) or a person who was very familiar with the system (in case of the smaller VoD) to generate Requirements-to-Code Trace Matrices (RTMs). In total, the developers identified 9,876 trace links among the 50 requirements, with an average of 197 traces per requirement. Table 2 provides further details. For example, we see that the number of methods implementing a given requirement ranged from 9 methods (smallest) to 932 methods (largest). Most of the requirements were functional but five of the 50 requirements were non-functional. Examples of requirements are:

- VoD R6: The system should have a one second max response time to start playing a movie.
- 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.

While the requirements were very diverse, Table 2 also reveals that their traces are very unlikely to be guessed. If we were to take a method and randomly choose its requirement then we would only be between 0.02–7.4% 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 significantly improve on this random chance.

B. Capturing of Method Call and Data Dependencies

We discussed above that we need high-quality traces to understand the relationship between code dependencies and traces. The same is true for code dependencies. If the method call and data dependencies were of poor quality, we would not expect finding any relationship among these beyond the random chance of correctness identified above. We now focus on how we captured method call dependencies and data dependencies with high quality.

Method call and method data dependencies can be captured through static and/or dynamic program analysis. However, existing state-of-the-art technologies are not without

problems. 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. The problem with state-of-the-art static analysis techniques is that they generally err on all sides. If the call and data dependencies were roughly equally wrong or missing then this might still allows us to investigate our five 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 they are hard to detect and track (e.g., points-to analysis [11]). 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. Table 2 reveals that we identified 9,725 call dependencies (method calls) and 39697 data dependencies across the three systems 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 observing 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 executed 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. To minimize this problem, we performed exhaustive testing on the three evaluated systems. While exhaustive testing can minimize the problem of missing call and data dependencies, it cannot prevent it. However, 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 since incomplete testing does not appear to favor the one over the other. We thus believe that the code dependencies are overall high quality for both calls and data.

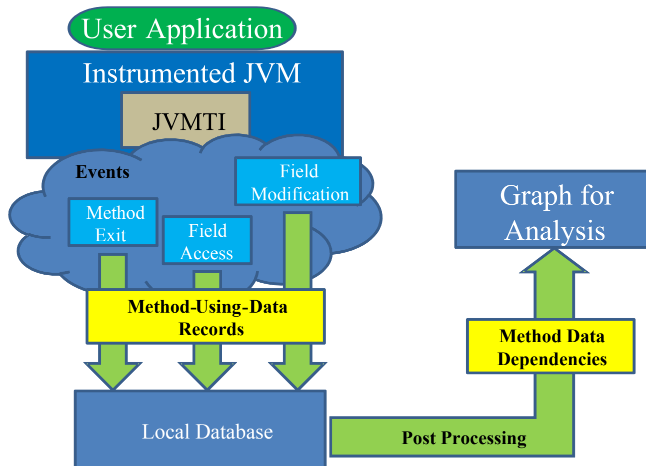

Figure 2. The approach of capturing method data dependencies.

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 on code level such as classes or methods and not on fields or variables and thus technologies for understanding data sharing among variables is not sufficient for our purpose.

We thus developed a prototype, which automatically observed method calls and data sharing among the methods. Currently, our tool is based on Java because of its reliance on the Java JDK, which provides an easy and reliable interface for recoding method calls at runtime. We built our work upon our previous work [3] on capturing method call dependencies at runtime. However, since we were unable to find a tool for capturing data dependencies at the level of detail described above, we extended our approach as illustrated in Fig. 2. We are using JVMTI (Java Virtual Machine Tool Interface), which provides both a way to inspect the state of a system (i.e., its data) and control the execution of a system while it is running in the Java virtual machine (JVM). A client of JVMTI can be registered in interesting occurrences through events that JVM generates. The client can then query and control the target application through JVMTI, either in response to events or independent of them. Naturally, our technology is restricted to Java and hence all three evaluated systems are Java systems. However, our observations should be generalizable to other programming languages because they are based on programming concepts that are similar across most modern programming languages (i.e., Java, .NET, Ada, etc.).

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 allows us to inspect local variables (including parameters and return values) a method created in order to further investigate data dependencies.

For our work, it is important to go beyond shared variables because true data dependencies exist if two methods indeed have access to the very same data, even if the data are referenced by different variables. Two methods thus have a data dependency if both methods access and/or manipulate variables that point to the same data in memory. With the help of JNI (Java Native Interface) and JVMTI, we can locate actual objects in the Java heap that are pointed to by the variable references (we discuss how to handle static data and Java primitive types in Section VIII). 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 method-using-data records for each method. The following examples show four such method-using-data records that we

[1] Claimed by JVMTI, however, possibly incorrect which is a technical problem we need to address in future work.

captured for the Video on Demand method discussed earlier ("-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*

By comparing the hash code, data dependencies among the three methods: `ListFrame.-init-()`, `ListFrame.buttonControl3_actionPerformed()`, and `VODClient.init()` can be identified. In order to make sure that captured method data dependencies are correct, we manually inspected the source code of numerous methods and improved the developed technology accordingly. In result, we are confident that our technology captures high quality method call dependencies and method data dependencies. The overhead of capturing call and data dependencies by running test cases for each case study system is a one-time cost and was not unreasonable (20 mins for VoD, 1.5 hours for JHotDraw, 3 hours for Gantt).

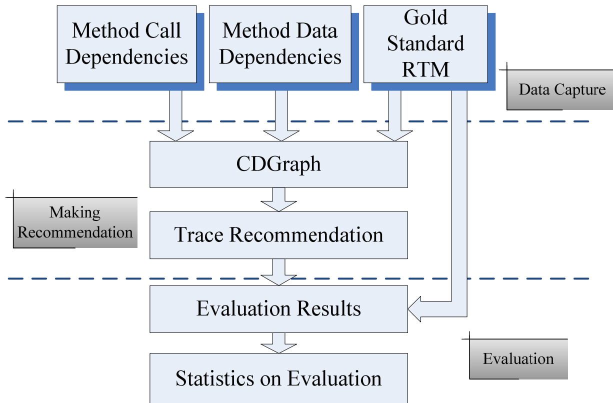

Figure 3. Overview of the proposed framework.

V. PROPOSED FRAMEWORK

The captured method dependencies can now be used to assess the quality of requirements traces on methods. Figure 3 depicts the proposed framework for the evaluation and recommendation process. We analyzed each software system separately. First, we built a graph structure called CDGraph (Call-Data Dependency Graph). This graph structure combines the captured method call dependencies, the method data dependencies, and the requirements-to-code traces from the gold standard RTM. Second, we explored various algorithms for computing trace recommendations based on the CDGraph. These recommendations are computed based on trace information of neighboring methods. Finally, we evaluated the correctness of our recommendations by comparison with the gold standard RTM. Based on this data, we answer our research questions. These three steps are explained in more details in the following subsections.

A. Step 1: Composing the CDGraph

We combine method call dependencies, method data dependencies, and traceability information into a single graph structure, called the CDGraph. In this graph, one node represents exactly one method. The edges of this graph represent captured code dependencies. Figure 4 shows an excerpt of the CDGraph for the Video on Demand system. Method call dependencies are annotated as solid arcs with arrows, while method data dependencies are annotated as dashed arcs without arrows (these edges are also annotated with the number of data types any two methods are sharing which will be discussed in Section VII). Nodes are labeled with class and method names and annotated with the gold-standard traceability information from the gold standard (listed at the bottom). For example, `ListFrame.buttonControl3_actionPerformed()` contributes to the implementation of the requirements R0, R2, R10 and R12; it is called by method `ListFameListener3.actionPerformed()`; and it shares data with `VODClient.init()`.

Figure 4 shows that method call dependencies and method data dependencies appear complementary but also overlapping. For example, there is only one method call edge between `ListFrame.buttonControl3_actionPerformed()` and `Detail.setmovie()`. Furthermore, there is only one method data edge between `buttonControl3_actionPerformed()` and `Movie.gettitle()`. However, between `buttonControl3_actionPerformed()` and `ServerReq.getmovie()` there are both, a method call edge and a method data edge.

B. Step 2: Trace Recommendations

We used the call-data dependency graph (there is one for each evaluated system) to find out whether the method call dependencies and/or the method data dependencies of any given method node in the graph correlate with requirements traces pointing to that node. In order to assess that question, we implemented several traces recommenders that assess requirements traces for each node in the captured graph based on its dependencies to other nodes of the graph and their related requirements. The following introduces one such algorithm. We will later describe others.

An intuitive and also simple algorithm 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 recommends 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 evaluate traceability between each

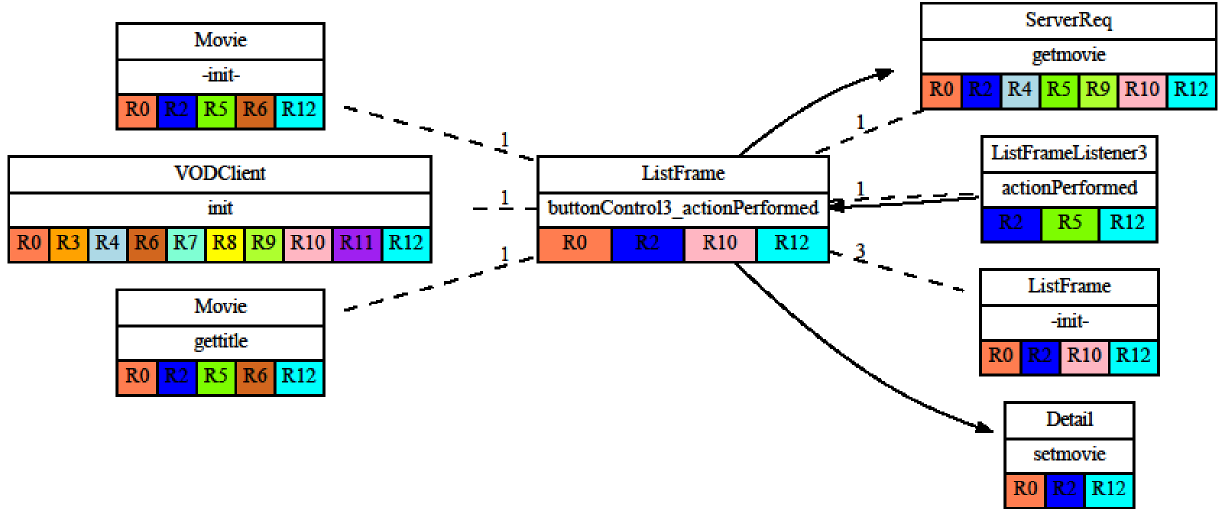

Figure 4. Example of a Call-Data Dependency Graph showing one node and its neighbors related by method call relationships (solid arcs with arrows) and method data relationships (dashed arcs without arrows). Each node identifies the requirements it traces to (labels Rx).

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)
          recommendation(n,r) = 'trace';
       else
          recommendation(n,r) = 'no-trace';
   }
}
```

In this algorithm, we are using a 50% threshold. That means that if more than 50% of a neighbor's nodes (aka dependent methods) are tracing to a certain requirement then it is very likely that the node itself is also part of the requirements implementation. We are using the example graph shown in Fig. 4 to demonstrate this recommendation process. The figure shows the method `buttonControl3_action Performed()` of class `ListFrame` in the center and its neighbors around. In order to compute recommendations for this node, we would iterate through each of the 12 requirements in the specification. For the first requirement R0, we find 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 recommend a trace to R0 for the evaluated method. This recommendation is correct as the evaluated method is truly related to R0. This is evident in the node for `buttonControl3_actionPerformed()` to also list R0 as one of its requirements. Do note that we base our recommendation on the known traces of the neighboring nodes and not on the node under investigation. Thus, we are assessing whether the traceability of nodes with call and/or data dependencies (neighboring nodes) have a correlation to the traceability of the node itself. Proceeding with this process, we would eventually recommend traces to R0, R2, R5, and R12. A comparison with the actually existing traces in Figure 4 shows that a trace to R5 would be recommended which is currently missing in node `ListFrame.button-Control3_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 traces and missing traces.

C. *Step 3: Evaluating the Correctness of Recommendations*

In the following section we will discuss the quality of trace recommendations computed for the three evaluated systems with the algorithm introduced above.

TABLE III. RESULT TYPES FOR THE VALIDATION OF TRACE RECOMMENDATION

Computed Recommendation	**Golden Standard RTM**	**Validation Result**	**Correctness**
Trace	Trace	TP	Correct
No-Trace		FN	Incorrect
Trace	No-Trace	FP	Incorrect
No-Trace		TN	Correct

In order to evaluate the correctness of a recommendation, it is compared with the golden standard. Table 3 shows the four possible combinations of recommendation and golden standard value. While TP (True Positive) and TN (True Negative) refer to correctly recommended traces and non-traces, a FN (False Negative) refers to a missing trace and a FP (False Positive) refers to a wrong trace. For each evaluated system, we count how often each of the four validation results occurs. These figures are provided in the next section. The reason for splitting recommendations of traced and non-traced nodes is that for a typical system the RTM is very sparse and the number of traces compared to non-traces is very low. The splitting allows understanding where false recommendations are made with a particular emphasis on trace recommendations, which is typically the main focus of related work (see Section IX).

The overall incorrectness (combining "trace" and "no-trace") is defined as formula (1) and shows the percentage of correct recommendations in relation to all given recommendations. A value of 0% means that only correct recommendations are given and a value of 100% means that only incorrect recommendations are computed.

$$\text{Incorrectness} = \frac{\text{FP+FN}}{\text{(TP+FN+FP+TN)}} \quad (1)$$

The recall is defined as formula (2) and shows the percentage of proposed traces (ignoring "no-trace") in relation to all recommendations on traced nodes. A value of 100% means that trace recommendations are complete (none are missing) as compared to the golden standard RTM.

$$\text{Recall} = \frac{\text{TP}}{\text{(TP+FN)}} = 1 - \text{Missing Trace Rate} \quad (2)$$

Finally, the precision is defined as formula (3) and shows the percentage of correctly proposed traces (ignoring "no-trace") in relation to all recommendations on trace nodes. A value of 100% means that none of the trace recommendations are wrong as compared to the golden standard RTM.

$$\text{Precision} = \frac{\text{TP}}{\text{(TP+FP)}} = 1 - \text{Wrong Trace Rate} \quad (3)$$

VI. RESULTS (RESEARCH QUESTIONS REVISITED)

The goal of our work is to understand the relationship between method call dependencies, method data dependencies, and requirements traces. In doing so, we primarily focused on those parts of the code that implement the given requirements. This led to roughly 90,000 trace recommendations computed for the three evaluated software systems.

TABLE IV. NUMBER OF CORRECT AND INCORRECT RECOMMENDATIONS FOR THE THREE CASE STUDY SYSTEMS

	Correct		Incorrect	
	TP	**TN**	**FN**	**FP**
Method call dependencies only (Call)				
VoD	357	1270	203	198
Gantt Project	4173	41235	2719	1683
JHotDraw	1220	32099	1204	505
Method data dependencies only (Data)				
VoD	435	1413	125	55
Gantt Project	2375	42326	4517	592
JHotDraw	973	32404	1451	200
Method call and method data dependencies (Call+Data)				
VoD	435	1417	125	51
Gantt Project	4075	42028	2817	890
JHotDraw	1222	32384	1202	220

In order to answer our research questions (see Section III) we performed three experiments per evaluated system. First, we computed recommendations on a graph that contained only method call dependencies (Call Graph). This is the CDGraph minus all data dependencies. Second, we computed recommendations on a graph that contained only method data dependencies (Data Graph). This is the CDGraph minus all call dependencies. Finally, we computed recommendations on a complete CDGraph that contained both method call and method data dependencies (Call + Data Graph). Table 4 shows the results of these experiments as number of correct and incorrect recommendations on nodes with and without requirements traces.

TABLE V. AGGREGATED METRICS ASSESSING THE COMPUTED RECOMMENDATIONS FOR THE EVALUATED SYSTEMS (VOD, GANTT, AND JHOTDRAW) AND FOR THE THREE METHOD DEPENDENCY GRAPHS (CALL, DATA, AND CALL+DATA)

		Incor-rectness	**Recall**	**Precision**
VoD	Call	19.77%	63.75%	64.33%
	Data	8.88%	76.78%	88.78%
	Call + Data	8.68%	77.68%	89.51%
Gantt Project	Call	8.84%	60.55%	71.27%
	Data	10.26%	34.46%	75.01%
	Call + Data	7.44%	59.13%	82.08%
JHotDraw	Call	4.88%	50.33%	70.73%
	Data	4.71%	40.14%	76.34%
	Call + Data	4.06%	50.41%	84.75%

Table 5 reports the findings from Table 4 in form of the introduced metrics: incorrectness, recall, and precision for all three systems and the three different graphs. The results shown in these tables are used in the following to answer our research questions (see Section III).

RQ1: Are method call dependencies relevant for evaluating requirements traces?

We found that by purely evaluating method call dependencies 4.88% (JHotDraw) to 19.77% (VoD) of the recommendations were incorrect. Given that there are two possible traceability states (a trace vs. a no-trace), there is 50% chance of randomly guessing correctly. The result thus shows a strong relationship between method call dependencies and requirements regions captured by traceability links, the computed results are far from random guessing. Thus, method call dependencies are relevant for evaluating requirements traces (RQ1). This observation is in line with some related work which has been exploiting method call dependencies (e.g., fan-in/fan-out analysis).

RQ2: Are method data dependencies relevant for evaluating requirements traces?

We found that by purely evaluating method data dependencies 4.71% (JHotDraw) to 10.26% (Gantt) of the recommendations were incorrect. This result shows that also a strong relation exists between method data dependencies and requirements regions. In result, method data relations are relevant for evaluating requirements traces (RQ2) and suggest that method data dependencies are a viable alternative to method call dependencies.

RQ3: Are method call dependencies more relevant than method data dependencies for requirements traces?

For JHotDraw and VoD less incorrect recommendations are computed based on the graph that only contains method data dependencies, while for Gantt less incorrect recommendations are computed based on the Call graph. Our results suggest that method data relations are equally relevant for evaluating requirements traces (RQ3). However, the key question is whether there is a combined benefit (next).

RQ4: Are method call and method data dependencies complementary to each other in evaluating traces?

We found that by combining method call dependencies and method data dependencies into one graph (Call + Data) only 4.06% (JHotDraw) to 8.68% (VoD) of the computed

recommendations were incorrect. These are the best results across all three systems, suggesting that method call and method data dependencies are in fact complementary (RQ4).

If we investigate this further, we notice a strong benefit for both precision and recall. Looking at Table 5, we notice that the precision for Call+Data is always far above the precision for Call or Data individually. The same is true for the recall (except for Gantt which is nearly the same as the recall for Call). This suggests that the trace recommendations 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. It shows: 1) total call edges, 2) total data edges, and 3) total overlaps where call edge and the data edge cover the same two nodes. Compared to the totals, we see that the overlaps are very small.

TABLE VI. OVERLAP BETWEEN CALL AND DATA EDGES IN THE CDGRAPHS FOR THE THREE CASE STUDY SYSTEMS

	Call	Data	Overlap
VoD	222	899	66
GanttProject	5560	24243	1042
JHotDraw	3943	14555	893

RQ5: Are additional code characteristics relevant for evaluating requirements traces?

After we found that we were able to compute the best recommendations based on the combined graph (Call + Data), we started to explore which other factors were influencing the correctness of recommendations. We evaluated several code metrics on method, class, and package level: lines of code per method, number of method per class, number of attributes per class, depth of inheritance tree, afferent coupling, efferent coupling, and McCabe's cyclomatic complexity. However, we did not find a correlation between recommendation mistakes and any of these metrics. On the surface, it appears that typical code smells do not affect the relationship between code dependencies and traces.

TABLE VII. PERCENTAGE OF INCORRECT RECOMMENDATIONS IN RELATION TO PROPERTIES OF EVALUATED METHODS

	Parameters		Return Value		Is Constructor	
	with	*without*	*with*	*without*	*yes*	*no*
VoD	8.99%	9.20%	6.10%	9.51%	9.43%	8.46%
GanttProject	9.53%	6.05%	8.43%	6.71%	5.14%	8.09%
JHotDraw	4.40%	3.89%	4.36%	3.88%	2.91%	4.39%

However, we also evaluated the relation between properties of a method and recommendation mistakes. By properties of a method we refer to whether the method has parameters or not, whether it has a return value or not, and whether it is a constructor of a class or not. Though, we found differences for each studied property (see Table 7), we did not find a consistent correlation across all three evaluated systems. In fact, GanttProject and JHotDraw show the same effects, while Video on Demand shows exactly the opposite effect for all three properties. This finding requires a deeper analysis with additional evaluated systems. Video on Demand is a very small system compared to GanttProject and JHotDraw, suggesting that there might be differences depending on the size of the system.

Finally, we evaluated whether the number of requirements that a method is implementing is related to the number of incorrect recommendations that were computed for that method. Figure 5 shows the results of that analysis. For all three systems we found a strong correlation between both values. The more requirements a method is implementing, the more computed trace recommendations are incorrect. In result, we found that there are additional relevant code characteristics that should be considered when evaluating requirements traces (RQ5). The observation in Figure 5 confirms a known problem that feature interactions are problematic (i.e., the more requirements implement a method, the more pronounced is the feature interaction problem).

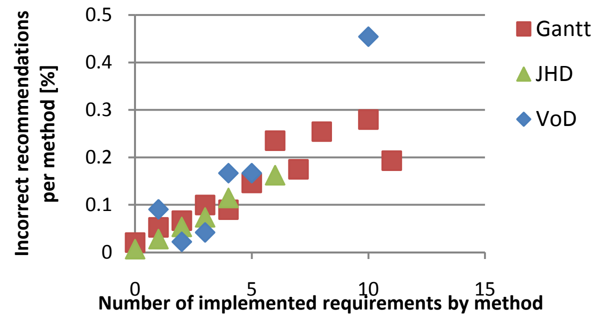

Figure 5. Relation between the number of requirements a method is implementing and its percentage of incorrect trace recommendations.

VII. OTHER TRACE RECOMMENDATION ALGORITHMS

Our focus for this paper was on investigating whether there are relationships between code dependencies and requirements-to-code traceability, but not on finding the ideal algorithm that could exploit this relationship. This will be the focus of future work. For that reason we preferred a simple algorithm to make trace recommendations. However, we did test whether other trace recommendation algorithms would result in the same observations regarding research questions 1-5. We found that the research questions held regardless of trace recommendation algorithm we applied. The following briefly summarizes one additional algorithm that refines the role of data dependencies to illustrate this.

As we discussed in Section V.A, the numbers on data edges in the CDGraph represent the amount of actual data types that two methods share. It is intuitive to think that if two methods share more types of data, they are more likely to cooperate with each other and are also more likely to trace to the same requirement. In an attempt to evaluate that hypothesis, we assigned extra weight to both *tracingNeighbors* and *neighbors* in the recommendation algorithm (see Section V.B) if at least one neighbor method traces to a given requirement and both methods share more than one data types. In fact, if the number of data types that the evaluated method and the neighbor method share is N, then we assigned an extra weight of N-1. For example, in Fig. 4, let's set `ListFrame.buttonControl3_actionPerformed()` to be the evaluated method. Then this method shares three data types with its neighbor method `ListFrame.-init-()`. In giving a trace recommendation for R10, the *tracingNeigh-*

bors will be 5 and the *neighbors* will be 9, so the computed value for R10 is larger than 0.5 (50%) and the trace recommendation for the evaluated method and R10 would be 'trace'. This trace is correct according to the golden standard RTM and improves the earlier problem of a missing trace.

TABLE VIII. NUMBER OF CORRECT AND INCORRECT RECOMMENDATIONS USING TWO DIFFERENT ALGORITHMS BASED ON THE CALL+DATA METHOD DEPENDENCY GRAPH

	Correct		Incorrect	
	TP	**TN**	**FN**	**FP**
Intuitive algorithm in Table 5				
VoD	435	1417	125	51
Gantt Project	4075	42028	2817	890
JHotDraw	1222	32384	1202	220
Algorithm counting data types in data edges				
VoD	455	1417	105	51
Gantt Project	4141	42011	2751	907
JHotDraw	1249	32362	1175	242

TABLE IX. AGGREGATED METRICS ASSESSING THE COMPUTED RECOMMENDATIONS USING TWO DIFFERENT ALGORITHMS BASED ON THE CALL+DATA METHOD DEPENDENCY GRAPH

		Incor-rectness	**Recall**	**Precision**
VoD	Intuitive	8.68%	77.68%	89.51%
	Type Count	7.69%	81.25%	89.93%
Gantt Project	Intuitive	7.44%	59.13%	82.08%
	Type Count	7.34%	60.08%	82.04%
JHotDraw	Intuitive	4.06%	50.41%	84.75%
	Type Count	4.05%	51.53%	83.77%

The computed recommendations for the second algorithm show that the amount of data types on each data edge in the CDGraph can help to provide slightly better trace recommendations. The improvements of the new algorithm are small only. However, as we discussed, it was not the goal of this paper to optimize the recommendation but rather to make sure that different recommendation algorithms replicate our findings discussed above. We thus tried a range of other algorithms also with the same basic observations discussed earlier.

VIII. THREATS TO VALIDITY

A possible threat to validity is the possible incompleteness of code dependencies (as in missing calls and data dependencies). We did aim to cover all code that implemented the requirements we analyzed. However, incompleteness is a likely fact though we believe it is not a serious threat because call and data dependencies would have equally "suffered" from this problem and our goal was the comparison of both.

As discussed in Section V, we first collected method-using-data records at runtime and then capture method data dependencies by comparing the hash code value of objects that are pointed to by the variables in those records. We faced the problem of handling data records without a unique identifier, such as static fields (static variables can only be declared as fields in Java) and local variables of Java primitive types (e.g., int, double, boolean, etc.). 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. So we simply use the type of this field, the name of this field, and the name of the class where this 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, we could not find a unique identifier for local variables with primitive types inside methods.

Although, we were not able to capture all data dependency due to tool limitations, we did capture enough 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), we doubt that it would have changed the primary message about the complementary nature of call and data dependencies.

IX. RELATED WORK

Extracting object-oriented dataflow communication is a research hotspot and lots of work has been done in this field. Milanova et al. [5] extended Andersen's static analysis technology [11] to extract points-to information (this information shows which pointers, or heap references, can point to which variables or storage locations) from Java. In our work, we use the hash code value, which represents a unique id for each memory location, to capture data dependencies among methods. Lienhard et al. [6] 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). We also generate data edges between two method nodes in the CDGraph via method-using-data records, which are collected during runtime. However, all that work focuses on the relationship among objects or classes. Instead, our work is particularly concerned with data dependencies among methods, because we want to compute trace recommendations on the method level of source code.

In the last two decades, plenty of efforts have been done in traceability, especially in requirements-to-code traceability. Information retrieval, to date the most widely researched technology identifies trace links based on naming similarities between source code and software artifacts (including requirements) [7-9]. However, the result of simple keyword matching is rather low in precision so more sophisticated technologies are necessary. Zhao et al. [8] proposed an approach (SNIAFL) using a static representation of the source code to refine trace links achieved by information retrieval. Eaddy et al. [9] presented a framework (CERBERUS) that combines information retrieval, static analysis (similar to the SNIAFL approach), and dynamic analysis. Hill et al. [14] used both lexical analysis and call graph exploration in a tool called Dora to perform software maintenance tasks. This tool then computes a subset of the call graph relevant to the query, called *a relevant neighborhood*. Their work does not rely on data dependencies but it does show that there is a benefit in considering larger neighborhoods which would likely benefit

our approach also. McMillan et al. [13] is the only paper known to us that considers data flow to establish traceability. However, their data flows are approximated and exhibit false positives and false negatives. Their conclusion is that data flows do not appear to have a benefit which is contradictory to our observations. This might be due to the FP/FN problem which needs to be explored in future work. All of these approaches [7-9, 13, 14] use either control flow and/or data flow message to improve the quality of the trace recovery process based on information retrieval and lay the ground of understanding the relationship between traces and method communications such as method calling relationship and method-data-sharing. Yet, our focus was not on automatically identifying new traces. Instead we are focusing on whether call and data dependencies are helpful in assessing requirements-to-code traceability.

In earlier work [10], 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 truly were implemented in connected areas of the source code rather than distributed. In a follow-on publication [3] we introduced a surroundness property to requirements regions. In the publication, we found that a given method typically shares the same traces to requirements as its neighbor methods, identified by method calls. In this work we built upon those previous observations and introduced additional method data dependencies in order to identify an even more relevant set of neighbors per method, compared to only analyzing method call dependencies. We showed that these two kinds of method dependencies are complementary to each other and help to better understand where a requirement is implemented in the source code.

X. CONCLUSIONS

In this paper, we investigated the question of whether method data dependencies are similarly related to requirements as 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 several research questions and validated them on three large software systems, covering about 120 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 method is related to a requirement. These findings have strong implications on all forms of code understanding, including trace capture, maintenance, and validation techniques. For example, we believe that common information retrieval approaches in traceability can and should be augmented with knowledge on call and data dependencies. We also believe that other research directions such as program understanding would 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.

ACKNOWLEDGMENTS

We gratefully acknowledge support from the Joint-Training PhD Program of the Chinese Scholarship Council (CSC): 2011619048, the Austrian Science Fund (FWF) grants P23115-N23 and M1268-N23, the National Natural Science Foundation of China (NSFC) grants 61021062 and 61003019, the 973 Program of China grant 2009CB320702, and the U.S. NSF MRI grant 1126747.

REFERENCES

[1] B. 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

[2] 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

[3] A. Ghabi, A. Egyed. "Observations on the connectedness between requirements-to-code traces and calling relationships for trace validation," *26th International Conference on Automated Software Engineering (ASE),Lawrence, Kansas,* 2011, pp.416-419.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] A. V. A. 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.

[10] 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.

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

[12] 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.

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

[14] E. Hill, L. Pollock, 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.