

Observations on the Connectedness between Requirements-to-Code Traces and Calling Relationships for Trace Validation

Achraf Ghabi

Johannes Kepler University
4040 Linz, Austria
achraf.ghabi@jku.at

Alexander Egyed

Johannes Kepler University
4040 Linz, Austria
alexander.egyed@jku.at

Abstract— Traces between requirements and code reveal where requirements are implemented. Such traces are essential for code understanding and change management. Unfortunately, the handling of traces is highly error prone, in part due to the informal nature of requirements. This paper discusses observations on the connectedness between requirements-to-code traces and calling relationships within the source code. These observations are based on the empirical evaluation of four case study systems covering 150 KLOC and 59 sample requirements. We found that certain patterns of connectedness have high or low likelihoods of occurring. These patterns can thus be used to confirm or reject existing traceability – hence they are useful for validating requirements-to-code traces.

Keywords-Requirements, Traceability, and Validation

I. INTRODUCTION

Requirements-to-code traces reveal where in the code a requirement is implemented. This is important for many software engineering tasks including code comprehension and understanding the impact of requirement changes. It has been shown in several experiments [1] that lack of understanding where a requirement is implemented leads to higher effort and more errors. This is not surprising because a developer who is not/no longer familiar with the source code is less capable of performing changes than a person familiar with the source code. It has also been shown that developers not familiar with source code are typically only able to locate about half of the code where a requirement is implemented [2, 3]. It is thus not surprising that software engineering suffers from this lack of knowledge - leading to changes at inappropriate places or unnecessary code replication - accelerating a problem known as code degradation. Keeping track where requirements are implemented in the code is thus a fundamental best practice of the software engineering process.

Unfortunately, capturing and maintaining traces is a largely manual activity. Some automation exists but many approaches are not applicable to informal/unstructured requirements [4]. The exception is information retrieval where requirements-to-code traces are derived through wording similarities between requirements and code [5-9]. However, the reported precision and recalls for these approaches vary widely and are typically dependent on the quality and quantity of descriptions that

complement requirements and source code. Whether requirements-to-code traces are captured manually or through the help of such heuristics, the quality of the traces is unpredictable. Even if traces are captured by the original developers and are thus of high quality, these traces may deteriorate over time (with code and requirements changes) unless they are explicitly maintained.

This paper explores the connectivity between requirements-to-code traces in combination with calling relationships in the source code (method calls). The intent is to demonstrate that while requirements may be informal and not easy to parse, the manner in which code implements a given requirement is far from random. We found that if a method has neighboring methods that are calling it or are being called by it and these neighboring methods are known to trace to a given requirement then the method in question is very likely to trace to that requirement also. We refer to this property as “surroundedness” (implying a method being surrounded by similarly tracing methods).

It is intuitive to think that methods that implement a given requirement are likely to call other methods implementing the same requirements (if a requirement is large enough to be implemented in multiple methods). Indeed, we expect some kind of relationship between the different methods sharing the same requirement. To understand this relationship, the goal of this paper is to investigate the clustering and connectedness properties and their usefulness in validating requirements to code traces.

Our observations were empirically evaluated on four case study systems – totaling over 150KLOC in size and covering different application domains. For these systems, over 39,000 requirement-to-code trace links were available which we obtained from the original developers. We observed that not all methods that call a given method or are called by it (caller and callee) implement the same requirement (i.e. a trace); however, that the methods that do implement a given requirement are nearly always chained together (a call chain of methods that all implement the same requirement). We performed a preliminary analysis on whether these call chains can be used to correctly validate requirements to code traces (the presence and absence of said chains) and found that in

average 72-97% of traces and 94-97% of no-traces can be validated correctly.

II. DATA

We are basing our observations on the evaluation of four third-party software systems (between 3-72 KLOC) and 59 randomly chosen requirements. The four open source projects of different sizes and different domains were: VideoOnDemand (VoD), Chess, GanttProject, and jHotDraw (JHD). The basic characteristics of these projects are depicted in Table I. All of them are implemented in Java. These systems were chosen, in part, because of the availability of high quality requirements-to-code traces.

TABLE I. INFORMATION ON THE FOUR STUDY SYSTEMS

	VoD	Chess	Gantt	JHD
Language	Java	Java	Java	Java
KLOC	3.6	7.2	41	72
# Executed Methods	173	416	2603	1768
# Sample Reqs.	14	8	16	21
Gold Standard: Size of RTM	2422	3328	41648	37128

Having high quality traces as gold standards was important to test the clustering and connectedness properties, and ultimately to confirm that certain patterns of traces and calling relationships are indeed useful for trace generation and maintenance.

The traces were available in the form of *Requirements Trace Matrices* (RTM). Each RTM contains $n \times m$ cells where n is the number of requirements and m is the number of code elements (classes or methods). Table II depicts a tiny excerpt of such a RTM for the Chess system. Each cell indicates either a *trace* ('t') or *no-trace* ('n') as was discussed above. A trace in a cell implies that the given method (row) implements the given requirement (column). Vice-versa, a no-trace in a cell implies that the method (code element) is not implementing the requirement.

TABLE II. EXCERPT OF RTM FROM CHESS SYSTEM

	R1: Play Move	R2: Show Score
Board.getBoardAfterPly()	n	n
Game.addRegularPly()	t	t
Game.run()	t	t
Pawn.Clone()	n	n
Piece.init()	n	n
Player.doPly()	t	n

As was mentioned in the introduction, this paper investigates requirements-to-code traces in context of calling relationships among code elements. These calling relationships were observed dynamically by executing the system in order to isolate unused code. The Java JDK provides an easy and reliable interface for recording method calls at runtime. No special prerequisites were necessary for the execution of the systems. We simply tested the systems as exhaustively as could be. The testing was not limited to the sample requirements nor was it attempted to test the requirements individually. This was done to avoid any bias towards particular requirements or

usage scenarios (though we believe that structuring the tests towards individual requirements could be useful as well but this is left to future work). The result of the testing were *call graphs* (see Figure 1) where methods are represented by nodes and method calls by directed edges. We identify caller methods by incoming edges and callee methods by outgoing edges.

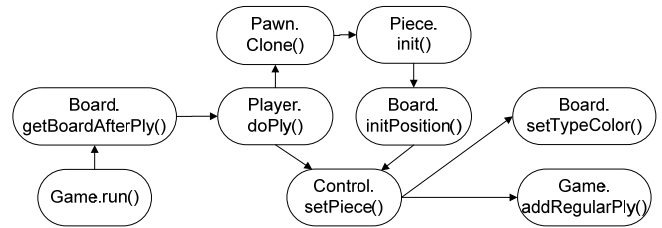


Figure 1. Excerpt of Call Graph for Chess System

III. OBSERVATION

In context of the call graph, we would expect that the caller and callees of a given method are likely related to the same requirements. The clustering and connectedness properties explore this relationship.

A. Clustering

On first inspection, the relationship between any given requirement and code is not straightforward. If some method A implements a requirement R and method A calls method B then one of the following two situations applies:

1. method B implements a service required by method A and, by implication, implements requirement R also
2. method B implements another requirement that is meant to coincide when method A occurs. By implication, method B then does not implement requirement R .

Clustering investigates whether all callers and callees of a method implement the same requirements. Thus, clustering investigates whether situation 1) is more likely than situation 2). Both situations above appear reasonable and probably both should occur. We investigated clustering in context of the four case study systems and their respective gold standard RTM. Table III depicts the empirical results. We see significant differences between the various systems but also between methods that trace to a given requirement and those that do not. If a method traces to a given requirement then, depending on the system, it is 34-76% likely that its callee also traces to that requirement, and 31-74% likely that its caller traces to that requirement. For example, in case of Gantt, 45% of the callees trace to the same requirement the caller is tracing to. Notice that the likelihoods change drastically for 'no-trace' observations. If a method does not trace to a given requirement then it is 79-94% likely that its callee also does not trace to that requirement. The higher percentage is explained in the fact that traces are rare compared to no-traces (at least for the 59 requirements we randomly selected). Much of the code does not trace to individual requirements. Therefore, some random code that is known to not trace to a requirement is likely surrounded by other code that does not trace to that requirement. The Chess system is the only exception where the

requirements covered larger areas of the code and correspondingly we see likelihoods that are more balanced.

TABLE III. LIKELIHOOD OF CLUSTERING

	T Clustering		NT Clustering	
	Caller='t'	Callee='t'	Caller='n'	Callee='n'
VoD	54.33 %	57.31 %	91.04 %	90.00 %
Chess	74.24 %	76.39 %	81.09 %	79.26 %
Gantt	54.49 %	45.21 %	88.82 %	92.02 %
JHD	31.96 %	34.75 %	95.03 %	94.40 %

B. Connectedness

While not all callers and callees of a method seem to relate to the same requirements (implied through clustering above), it seems quite likely that at least one caller or callee relates to the same requirement. The connectedness metric represents the percentage of methods that are directly connected to at least one other method implementing the same requirement. Table IV depicts the connectedness of trace methods on the four systems. Here, we distinguish between *inner nodes* and *leaves*. Methods are so-called inner-nodes when they have at least one caller and one callee. The connectedness of inner nodes can be evaluated on both the caller and callee side. A method is a leaf node if it does not have callee methods and thus, the connectedness can be evaluated on the caller side only. There are also root nodes but these are extremely rare (often there is only one root node which is the `main()` method in Java and this paper ignores root nodes for brevity). For inner-nodes, we have a very high connectedness of 88-99%. Leaves do not perform as well as inner-nodes but are still >59% connected. Furthermore, we see that no-trace methods show higher connectedness values which could be again explained by the high number of no-trace methods compared to trace ones. Higher connectedness implies that a tracing method to a requirement R is very likely to have another neighboring method also tracing to R.

TABLE IV. CONNECTEDNESS OF TRACE/NO-TRACE METHODS

	Trace		No-Trace	
	Inner-Nodes	Leaves	Inner-Nodes	Leaves
VoD	88.50%	59.00%	98.50%	93.30%
Chess	99.39%	93.18%	98.98%	92.23%
Gantt	94.31%	76.48%	99.59%	91.99%
JHD	90.15%	72.16%	99.87%	97.77%

C. Call Chains and Requirements Regions

We grouped methods into regions if they trace to the same requirements and are connected via calling relationships. All 59 requirements we studied were implemented in multiple methods and we surprisingly found that all requirements exhibited the same effect: there is usually one big region of connected methods which are implementing a given requirement and several small remaining regions that usually, do not contain more than a single method. Figure 2 shows the distribution of tracing methods over different regions ordered

by size (average group size over all requirements in each system).

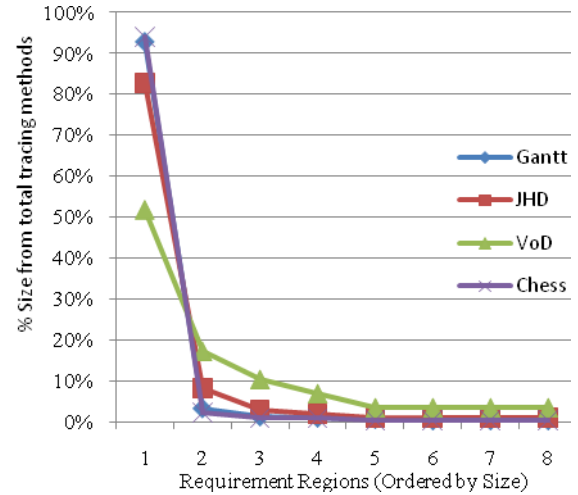


Figure 2. Most Methods Implementing a Given Requirement are connected directly or indirectly through calling relationships.

It is important to note that these regions of methods typically form a long chain of connected methods. These *Call Chains* have different lengths depending on the size of the requirement. The call chain could be “thicker” in some locations (where multiple callers or callees trace to the same requirement). In other places, it is “thin” (where only a single caller and/or a single callee traces to the same requirement).

Clearly, there is a strong indication that calling relationships correlate with traces.

D. Trace Validation

Certain combinations of traces together with calling relationships are very likely whereas other combinations are unlikely. Both situations are useful for validating requirements-to-code traces. Unlikely combinations point to potential traceability errors whereas likely combinations support traceability correctness. Note that method calling relationships are presumed correct because they are observed during execution. Unlikely combinations are thus more likely the result of trace errors rather than method call errors. Currently we identified simple patterns that are already able to validate traces and no-traces with high accuracy. Table V shows that we are able to identify 72-97% of correct traces and 94-98% of correct no-traces among the case study systems. Although, the application of such combinations varies from one system to another, they cover 70-96% of the gold standard RTM. It is future work to combine this basic knowledge on call chains with more advanced patterns and reasoning to further improve the quality of trace validation.

TABLE V. CORRECTNESS VS. COVERAGE OF TRACE VALIDATION

	Trace Correctness	No-Trace Correctness	Coverage
Chess	97.03%	95.87%	70.64%
Gantt	79.09%	96.40%	87.67%
JHD	71.97%	97.79%	95.79%
VoD	82.23%	94.18%	85.88%

Of course, these observations are heuristics and cannot definitely prove correctness or error; However, given that humans are not always able to correctly validate requirements-to-code traces [3], this technology could support developers during trace validation tasks.

IV. RELATED WORK

In the last two decades, there has been considerable progress on traceability issues, especially requirement-to-code traceability. Multiple techniques has been presented where different approaches are applied to retrieve and/or maintain trace links between code and other software artifacts. Revelle et al. [10] classified the state of the art feature location techniques into three main core types: *static*, *dynamic*, and *textual analysis*. These types also apply for traceability techniques. Textual analysis has been mainly exploited by performing Information Retrieval techniques on textual software artifacts (e.g. source code, developer comments, and documentation) [7, 11, 12]. A static analysis method was proposed by Chen and Rajlich [13]. The method should help a user navigating on a static abstract system dependency graph which would help her understand and retrieve implementation locations of different requirements. Other approaches [14] investigated the dynamic information of programs. There are also hybrid methods that combine multiple technique types in one approach. Eaddy et al. [15] presented a very good example of combining the three technique types (static, dynamic, and textual) in order to improve the quality of the trace recovery process in their tool (CERBERUS). Calling relationships in particular have also been used to improve the reported ranking of traces in information retrieval approaches [7, 10, 15] where higher-ranked traces are presumed to be more correct traces.

Multiple researchers [1, 4] investigated also the usefulness of traceability during software engineering. Traceability provides an important support for software engineering activities, such as impact analysis, reverse engineering, and regression testing. But, most proposed methods are related to automatic trace generation. They mainly aim to create or recover trace links from scratch. Although the proposed methods are fully automated and tool supported, a manual check must be performed in order to verify the quality of each technique.

Indeed, the strongest motivation for our work comes from a study by Kong et al. [3] who conducted an experiment about manual trace links validation. They made a very interesting observation: *the quality of the final RTM is influenced by the initial RTM*. Participants validating good quality RTMs did not deliver better quality. Based on that study, we conclude that manually validating traces does not necessarily guarantee a better accuracy in the final RTM.

While the relationships between traces and calling relationships in the code have been explored in the past [10, 15], they have never been exploited together with potentially erroneous traces for validation.

V. CONCLUSION

This paper demonstrated the connectedness property between requirement-to-code traces and method call relationships. We discussed how this connectedness property may be exploited for trace validation and this work thus represents an important step towards more automation for traceability because the state of the art predominantly focuses on creating or recovering requirement-to-code trace links where the correctness of these retrieved traces are typically only verified manually. Yet, manual trace validation was shown to be of poor quality with experiments showing that human subjects are likely decreasing the quality of input traces that exceeded >50% correctness [3].

REFERENCES

- [1] M. P. Robillard, W. Coelho, and G. C. Murphy, "How Effective Developers Investigate Source Code: An Exploratory Study," *IEEE Transactions of Software Engineering (TSE)*, vol. 30, pp. 889-903, 2004.
- [2] A. Eged, P. Grünbacher, and F. Graf, "Effort and Quality of Recovering Requirements-to-Code Traces: Two Exploratory Experiments," in *18th International IEEE Requirements Engineering Conference (RE)*, Sydney, Australia, 2010.
- [3] W. Kong, J. H. Hayes, A. Dekhtyar, and J. Holden, "How Do We Trace Requirements? An Initial Study of Analyst Behavior in Trace Validation Tasks," in *Proceedings of Cooperative and Human Aspects of Software Engineering*, May 2011.
- [4] O. C. Z. Gotel and A. C. W. Finkelstein, "An Analysis of the Requirements Traceability Problem," in *Proceedings of the First International Conference on Requirements Engineering*, 1994, pp. 94-101.
- [5] J. Cleland-Huang, C. K. Chang, and M. Christensen, "Event-Based Traceability for Managing Evolutionary Change," *IEEE Trans. Softw. Eng.*, vol. 29, pp. 796-810, 2003.
- [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* vol. 28, pp. 970-983, 2002.
- [8] J. H. Hayes, A. Dekhtyar, S. K. Sundaram, and S. Howard, "Helping Analysts Trace Requirements: An Objective Look," presented at the 12th IEEE International Requirements Engineering Conference, 2004.
- [9] G. Spanoudakis, A. Zisman, E. Perez-Minana, and P. Krause, "Rule-based generation of requirements traceability relations," *Journal of Systems and Software* vol. 72, pp. 105-127, 2004.
- [10] C. McMillan, D. Poshvanyk, and M. Revelle, "Combining Textual and Structural Analysis of Software Artifacts for Traceability Link Recovery," in *Proceedings of the 2009 ICSE Workshop on Traceability in Emerging Forms of Software Engineering*, Vancouver, Canada, 2009, pp. 41-48.
- [11] J. H. Hayes, A. Dekhtyar, and S. K. Sundaram, "Advancing Candidate Link Generation for Requirements Tracing: The Study of Methods," *IEEE Transactions on Software Engineering (TSE)*, vol. 31, pp. 4-19, 2006.
- [12] A. Marcus and J. I. Maletic, "Recovering documentation-to-source-code traceability links using latent semantic indexing," presented at the Proceedings of the 25th International Conference on Software Engineering, Portland, Oregon, 2003.
- [13] K. Chen and V. Rajlich, "Case Study of Feature Location Using Dependence Graph, after 10 Years," presented at the International Conference on Program Comprehension (ICPC), 2010.
- [14] M. Jiang, M. Groble, S. Simmons, D. Edwards, and N. Wilde, "Software Feature Understanding in an Industrial Setting," in *Proceedings of the International Conference on Software Maintenance (ICSM)* 2006, pp. 66-67.
- [15] 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*, Amsterdam, The Netherlands, 2008, pp. 53-62.