Understanding Where Requirements are Implemented

The Relationship between Requirements Traces and Method Calls in Code

Benedikt Burgstaller
Johannes Kepler University
4040 Linz, Austria
benedikt.burgstaller@jku.at

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

***Abstract*—Trace links between requirements and code reveal where requirements are implemented. Such trace links are essential for code understanding and change management. The lack thereof is often cited as a key reason for software engineering failure. Unfortunately, the creation and maintenance of requirements-to-code traces remains a largely manual and error prone task due to the informal nature of requirements. This paper demonstrates that reasoning about requirements-to-code traces can be done, in part, by considering the calling relationships within the source code (call graph). We observed that requirements-to-code traces form regions along calling dependencies. Better knowledge about these regions has several direct benefits. For example, erroneous traces become detectable if a method inside a region does not trace to a requirement. Or, a missing trace (incompleteness) can be identified. Knowledge of requirement regions can also be used to help guide developers in establishing requirements-to-code traces in a more efficient manner. This paper discusses requirement regions and sketches their benefits.**

Keywords-requirements, call tree, traces, feature location

I. Introduction

Trace links connect requirements, design, code, test cases, and many other artefacts. This paper focuses on requirements-to-code traces (sometimes also called feature location) which reveal where the requirements are implemented in the code. Requirements traces are essential [1] for adapting to requirements changes (i.e., to determine what code is affect by a requirements change), they reveal the rationale for design and coding decisions (what is this class good for?), and convey design decisions (why was this class implemented this way?). Indeed, trace links are an important pre-requisite to model-based software development [2-4] and at the centre of end-to-end integration [5]. It is thus not surprising that traceability is mandated by numerous standards and prescribed in best-practice methods (DOD Std 2167A, IEEE Std. 1219, ISO 15504, and SEI CMM).

Unfortunately, real-world software systems typically lack trace links despite their reported benefits [1]. Although it may appear easier to capture trace links during development, it is rarely done in industrial practice because of the cost of having to maintain them (i.e., trace links deteriorate during software evolution like other software artifacts). The alternative, trace recovery at a later time when the traces are needed, is also problematic because developers no longer remember the many vital details needed for doing so (often the original developers are no longer available).

Unfortunately, there is little automation available for the capture, recovery and subsequent maintenance of requirements-to-code trace links [6] because of the informal nature of requirements (most requirements are captured informally in natural language). The most widely researched technology for capturing requirements-to-code traces to date is information retrieval where trace links are derived through wording similarities between requirements and code [7-11]. To achieve high precision and recall, information retrieval requires rich requirements descriptions and well-documented source code. An alternative focus of researchers has been on the problem of feature interactions – as in crosscutting concerns [12], concern graphs [13] or concept lattices [14]. These technologies do not reason about traces directly but rather about how concerns in general (i.e., requirements could be concerns) interact in the source code which is related to the traceability problem. There, it has been shown that calling relationships within the source code are useful for understanding feature interactions. Indeed, knowledge about calling relationships have even been used in concert with IR technologies to automatically improve its quality [15] which already suggests that there must be some kind of relationship.

This paper thus investigates the relationship between calling relationships and requirements-to-code trace links. The biggest difference to state of the art is that we investigate the calling relationship for each requirement separately. 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.

There are more subtleties to these two interpretations but the fundamental dilemma is that a calling relationship between two methods is not enough to correctly decide on requirements-to-code traces.

Nonetheless, during the examination of requirements-to-code traces for several software systems, we observed that the code that implements any given requirement is usually connected via calling relationships. It is rare for a single requirement to be distributed across many areas of the code. This observation is based on over fifty, mostly functional requirements and does not constitute proof but rather a strong heuristic that should be exploited. This observation also does not stand in contradiction to the two interpretations above. It simply implies that when a method implements a requirement then other methods implementing that requirement will be among the methods called (callees) or methods calling it (callers). If we think of methods and their calling relationships as a call graph (the nodes are the methods and the edges are the calling relationships) then requirements implement connected areas in that graph – we refer to these areas as **requirement regions**. Only methods inside such a region belong to a given requirement (interpretation 1 above) but not necessarily all methods they call belong (interpretation 2 above).

The main aim of this paper is to discuss requirement regions. We believe that requirement regions are a new tool to better understand traceability. To illustrate this, this paper will also sketch applications of requirement regions: 1) guiding users through trace capture by making suggestions; 2) assessing the quality of requirements-to-code traces (i.e., detect errors); and 3) supporting trace maintenance during software evolution to ensure that captured traces remain correct. Requirement region may well be useful beyond traceability but page limits preclude further discussions.

II. PROBLEM

The two main problems of dealing with requirements to code traces are: lack of reliable automation and scalability. The scalability problem is obvious in the fact that for *n* requirements and *m* code pieces (e.g., classes, methods, or lines of code), there are *n*m* potential trace links to investigate. Even if a piece of code (e.g., method or class) does not implement a requirement, this fact must be assessed. Traces are often captured and depicted in form of trace matrices. Table 1 depicts an excerpt of such a matrix for the Chess system – one of four study systems we are basing our observations on. A trace is indicated in form of 'x'. For example, the method *setPiece* traces to the requirement "User should be able to start a new game" (referred to as requirement R0) and some other requirement R1.

To ensure a more general applicability of our findings, we opted to investigate multiple software systems of different application domains. The basic characteristics are depicted in Table 2 (a small excerpt of the trace matrix of the Chess system is depicted in Table 1). For the four study systems we investigated between 8-21 requirements. These requirements covered mostly functional requirements about core features of the system. We obtained the requirements and the requirements traces from the original developers who built the study systems or from people who were familiar with the source code. Thus, the input was complete and as correct as could be. Since trace capture is time consuming and expensive, we focused on a subset of the requirements for each system only. Some sample requirements were:

- Chess: User should be able to start a new game (which is R1 in Table 1)
- VOD: A movie should start playing within 1 second of selection
- Gantt: User should be able to create a new task
- jHotDraw: User should be able to delete connections between figures.

Table 1. Excerpt of Trace Matrix for Chess System

	R0	**R1**	**R2**	**R3**	**...**
setRun			x		
run	x				
getBoard	x				
clone	x	x			
setTimer		x	x		
display				x	
getType		x		x	
doPly	x				
setPiece	x	x			
...					

Table 2. Information on the four Cases Studies

	VOD	Chess	Gantt	jHotDraw
Language	Java	Java	Java	Java
KLOC	3.6	7.2	41	72
# Methods	193	424	13432	21532
# Sample Requirements	14	8	15	21
Size of Method Trace Matrix	2702	3392	201480	452172

In addition, we also captured the calling relationships among the many methods of these systems in form of a call tree. The call trees were the result of exhaustive testing and profiling. Together, this data was a benchmark to assess 1) the existence of the requirement regions and 2) some of its properties.

The quadratic growth of the requirements-to-code trace matrix is evident in the two-dimensional shape of the matrix in Table 1. Table 2 reveals, for example, that the trace matrix for jHotDraw with 21 requirements had over 450,000 cells. Given that we look at a rather small set of 21 sample requirements only, it is clear that trace capture is time consuming. Since trace capture is also a mostly manual and non-trivial process, trace matrices likely exhibit many errors.

For generating and validating trace links between requirements and code, one needs to understand both requirements and code. Much like our requirements, most other requirements are captured informally in natural language. Since natural language processing is still in its infancy, we are far away from automating trace capture and maintenance. This paper however suggests that requirements to code traces can also be understood by investigating the calling relationships.

III. Requirement Regions

This section discusses how calling relationships among methods form regions that relate to single requirements. *Requirement regions occur naturally. They are an observation and not an invention of this work.* In the following, we first discuss our observations on the kinds of rules those requirement regions seem to adhere to. Thereafter, we will discuss how knowledge on requirement regions and their rules can be used to better requirements-to-code traceability. Note that this work investigates each requirement and its region separately (the biggest difference to current state of the art).

Figure 1 depicts a part of the **call graph** of the Chess system. Each node of the call graph represents a method (for brevity, only the method name is given, ignoring class and package names). Each node implements one or more requirements. For example, method *run* implements Chess requirement R0 or method *setPiece* implements requirement R0 and R1. Table 1 above provided this traceability information. Each method should implement at least one requirement – or else the traceability is incomplete or incorrect. Often methods implement single requirements only, however, requirements also overlap – they share data and they share functionality. Some methods thus implement multiple requirements.

A property of a call graph is that it depicts all possible ways how methods can call one another. For example, *doPly* calls *setPiece* and *setPiece* calls *clone*, *init*, and *setEngine*. This structure is a graph since methods can get called by multiple methods and method calls may even be cyclic. It is not possible to infer from a call graph whether a method is always or only sometimes called. For example, *setPiece* may or may not call *clone* always – though it calls it at least once.

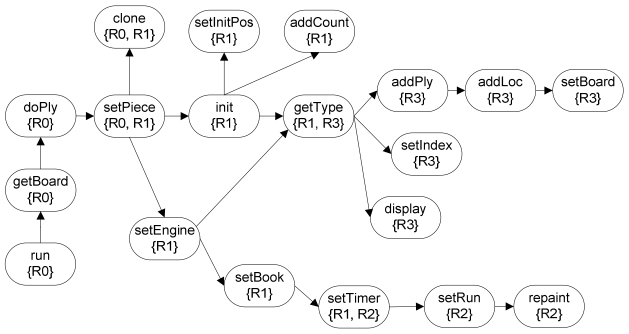

Figure 1. Excerpt Chess Call Graph and Requirements Traces

On first glance, the assignment of requirements to methods in the call graph in Figure 1 seems random. However, most requirements we studied formed regions in the call graph: the methods (nodes) belonging to the same requirement were connected from within such regions via calling relationships (edges).

Figure 2 depicts one such region for requirement R1 in context of the call graph depicted in Figure 1. Highlighted in Figure 2 are thus all methods belonging to requirement R1 (middle part, surrounded by a solid line). Not belonging to R1 are the remaining methods (left and right parts). Since requirements are crosscutting, not all methods that are called by *setPiece* must belong to R1 (called directly such as *setEngine* or indirectly such as *setBlock*). We observe this in the example through method *getType* which belongs to *R1* although it calls methods such as *addPly* that do not.

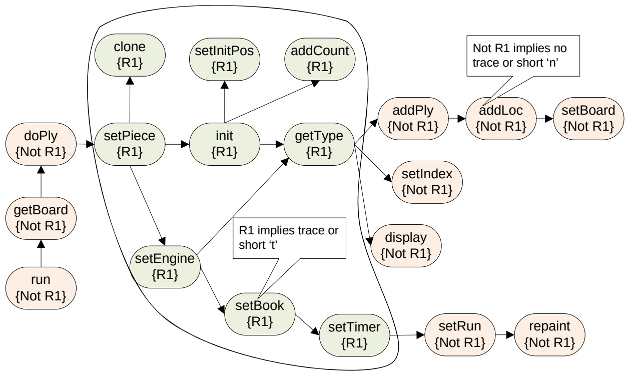

Figure 2. Requirement R1 forms a region

IV. Requirements Patterns

Patterns can demonstrate the existence of the requirement region. The simplest calling pattern is the call from method A (the caller) to method B (the callee). This pattern applies to every call in the call graph. Since each code element in this pattern may or may not implement a given requirement R, there are four possible scenarios. We denote 't' as the method tracing to a given requirement and 'n' as not tracing. The '›' implies that the method on the left calls the one on the right. For example, ' t›t' implies that method A calls B and both trace to a given requirement.

Table 3. Possible Calling Patterns involving a single Call

A calls B (denoted **A›B**)	code element B implements requirement R	code element B does not implement requirement R
code element A implements requirement R	**t›t**	**t›n**
code element A does not implement requirement R	**n›t**	**n›n**

We thus studied how often these patterns occurred in the call graphs of the four case study systems (Table 4). For example, the call graph for Chess had 4904 calls, of which 1749 calls were such that both the caller and callee methods implemented the same requirement.

Table 4. Occurrences of Two-Method Patterns

	VOD	**Chess**	**Gantt**	**JHotDraw**
#calls	952	4904	82008	89082
t›t	258	1749	4099	2520
t›n	278	367	5818	2947
n›t	241	313	5058	3519
n›n	175	2475	67033	80096

We can now ask a simple question: If we know that code A implements requirement R then how likely does code element B implement R if we assume that A calls B? We denote this question as the pattern <t›?> (what is the likelihood of the callee if the caller traces to R). From Table 5, we know that the Chess call graph has 1749 occurrences where both the caller and the callee implemented requirement R (t›t) and 367 occurrences where the caller implemented R but the callee did not (t›n). It follows that of these total 2116 occurrences, the callee implemented the same requirement as the caller in 83% of all occurrences. We thus see that the callee is likely to not trace to a requirement if the caller does not trace to that requirement. Likewise, we can now investigate the likelihoods of other patterns. Table 5 shows the likelihoods for all four two-method patterns where we see that the t›? and ?›t patterns favor traces whereas the n›? and ?›n patterns do not.

Table 5. Likelihoods of Two-Method Patterns

	VOD	**Chess**	**Gantt**	**jHotDraw**
t›? (?=t)	48%	83%	41%	46%
?›t (?=t)	52%	85%	45%	42%
n›? (?=n)	96%	89%	93%	96%
?›n (?=n)	82%	87%	93%	97%

It is important to note that the 41-83% likelihoods of ?=t in <t›?> are a strong support for a trace even though one might be mislead into believing they are close to random (50/50). Of the 50+ requirements across the 4 systems we studied, traces were rare because requirements implemented in average in only about 5-12% of their respective code elements. Thus, if someone establishes a trace for a random code element then that person would have a 5-12% chance to be correct. However, using the very simple pattern discussed above, this person now has a 41-83% of correctness if the pattern t-? or ?-t is found. This is a large improvement. Thus, any cell in Table 5 that favors ?=t to a greater percentage than 5-12% is an improvement over random. So, given the small chance to correctly guess a trace link, we have here several patterns where the chances of guessing a correct trace link are significantly stronger than a random guess. Indeed, we already investigated more elaborate patterns and found that the percentages increase considerably with just slightly larger patterns. For example, if a code element is surrounded by two callers/callees that trace to a given requirement then the chances for the method to also trace increases to 73-97% for traces and 95-98% for no traces. This data is strong support to the existence of requirement regions because it implies that the traces of a requirement must be in close proximity and cannot be spread across a system. Future work will explore more complex patterns.

V. APPLICATIONS TO TRACEABILITY

The main contribution of this paper is requirement regions. However, knowledge about these regions is only useful in context of applications that lead to improvements in software engineering. This section sketches a few applications where requirement regions lead to better traceability. Particularly, during maintenance, it is often criticized that developers make suboptimial decisions – that is they change the code in places that is not ideal – leading to code degradation and, in the worst case, to unmaintainable code. Traceability alone may not prevent code degradation, however, complete and correct traceability between requirements and code is a pre-requisite for better code maintenance.

Auto Validation of Requirements-to-Code Traces

We observed two interesting heuristics of requirement regions that are beneficial for trace validation:

- if two methods in close proximity are outside a region then the method(s) in between ought to be outside also
- if two methods in close proximity are inside a region then the method(s) in between ought to be inside also

It is possible to automatically asses the validity of existing trace matrices based on these heuristics. The only exception is requirements that are implemented in few methods only because the heuristics requires a critical mass of methods to surround other methods. However, we believe that such trivial requirements are also more easily understood.

Auto-Completion of Requirements-to-Code Traces

Based on the heuristics identified in trace validation, we can also define heuristics for automatically suggesting missing requirements-to-code traces. For example, if some methods inside a region are known than other methods in between them should belong to the region also.

Guiding Trace Capture and Recovery

Improving completeness supports trace capture but it only provides benefits "after the fact" when some traceability information is already available. During trace capture, developers could benefit from guidance. Entry and exit points to regions are key to such guidance. In principle, we only have to find these points because all remaining methods between them then belong to the region.

Maintaining/Evolving Traceability

Today, trace capture is often not done because of the cost of having to maintain traces. When requirements change or the code changes then the requirements-to-code traces between them may change also. Since requirement regions are based on the call graph and the call graph may change in response to code changes, we can understand the impact of requirements and/or code changes by understanding the impact of call graph changes onto requirement regions. Thus developers could benefit from automated maintenance of traceability which is a major benefit because it affects the cost/benefit tradeoff between early trace capture vs. later trace recovery.

VI. RELATED WORK

The recovery of requirements-to-code traceability did receive a fair amount of attention in the research community [8]. However, to date automated approaches are weak because requirements are typically captured informally and cannot easily be reasoned about. Prominent technologies, such as Information Retrieval (IR) [6][8, 9], identify trace links based on naming similarities (synonyms, etc.).

There have been numerous approaches to increase precision and recall of traceability recovery using different methods of gaining information about the application source code. The most relevant approaches are: McMillan et al. [15] who use calling relationships to improve the accuracy of information retrieval approaches and, similarly, the CERBERUS approach by Eaddy et al. [16] who use a three-tiered approach for traceability recovery by combining information retrieval, execution trace analysis, and prune dependency analysis to locate crosscutting concerns in source code. Both approaches laid the ground work in recognizing that there is some relationship between traces and method calls. Yet, our focus was not on how to validate IR technologies and auto-correcting their traces. It was mainly on requirement regions and why more knowledge about them is useful.

The detection and extraction of requirement regions has some commonalities with identifying crosscutting concerns. Marin et al., in [12] use fan-in analysis to count the relevancy of a method for identifying crosscutting concerns. To support the automation of trace recovery, various techniques and heuristics have been developed. For instance feature location techniques [17] or scenario-based techniques [18]. Although advances have been made to automatically recover links, trace capture remains a human-intensive activity with high, initial cost [1, 5, 19, 20]. Unfortunately, most of these techniques cannot be applied to the requirements-to-code traceability problem because of the informal nature of requirements.

VII. Conclusions

This paper discussed that much is to be gained by better understanding requirement regions and their impact onto requirements-to-code traces. We briefly discussed that requirement regions are useful for automatically detecting errors among requirements-to-code trace links and that regions could be used to automatically fill in missing requirements-to-code traces (auto-completion). Current trace capture/recovery is characterized by systematically exploring all methods and requirements (a problem of quadratic complexity). Through the help of requirement regions and their entry/exit points, we could reinvent how requirements-to-code traces should be captured by using regions as guidance (trying to guess entry/exit points). Finally, we believe that requirement regions could also be used to maintain trace links. Currently, the cost of trace maintenance is a major deterrent to early trace capture because trace links may well become obsolete before they are being used. Future work will explore these applications in detail and also investigate further properties of requirement regions. Future work will also investigate other forms of communication that do not involve method calls: middleware, network, data sharing etc. Finally, we plan on investigating how overlaps among requirement regions might yield useful clues about traceability.

We gratefully acknowledge the financial support of the Austrian FWF via grant agreement P21321-N15.

References

[1] B. Ramesh, L. C. Stubbs, and M. Edwards, "Lessons Learned from Implementing Requirements Traceability," *Crosstalk -- Journal of Defense Software Engineering,* vol. 8(4), pp. 11-15, 1995.

[2] N. Aizenbud-Reshef, B. T. Nolan, J. Rubin, and Y. Shaham-Gafni, "Model Traceability," *IBM Systems Journal,* vol. 45(3), pp. 515-526, 2006.

[3] M. Deng, R. E. K. Stirewalt, and B. H. C. Cheng, "Retrieval by Construction: a Traceability Technique to Support Verification and Validation of UML Formalizations," *Intern. Journal of Software Engineering and Knowledge Eng.,* vol. 15(5), pp. 837-872, 2005.

[4] T. Yue, L. C. Briand, and Y. Labiche, "Automated Traceability Analysis for UML Model Refinements," *Journal of Information and Software Technology,* vol. 51 pp. 512-527, 2009.

[5] M. Lindvall, and K. Sandahl, "Practical Implications of Traceability," *Journal on Software - Practice and Experience (SPE),* vol. 26(10), pp. 1161-1180, 1996.

[6] O. C. Z. Gotel, and A. C. W. Finkelstein, "An Analysis of the Requirements Traceability Problem," in Proc. of the 1st Intern. Conference on Requirements Engineering, 1994, pp. 94-101.

[7] J. Cleland-Huang, C. K. Chang, and M. Christensen, "Event-Based Traceability for Managing Evolutionary Change," *IEEE Trans. Softw. Eng.,* vol. 29(9), pp. 796-810, 2003.

[8] J. Cleland-Huang, B. Berenbach, S. Clark, R. Settimi, and E. Romanova, "Best Practices for Automated Traceability," *Computer,* vol. 40(6), pp. 27-35, 2007.

[9] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo, "Recovering Traceability Links between Code and Documentation," *IEEE Transactions on Software Eng.* vol. 28(10), pp.970-983, 2002.

[10] J. H. Hayes, A. Dekhtyar, S. K. Sundaram, and S. Howard, "Helping Analysts Trace Requirements: An Objective Look," in 12th IEEE International Requirements Engineering Conference, 2004.

[11] G. Spanoudakis, A. Zisman, E. Perez-Minana, and P. Krause, "Rule-based generation of requirements traceability relations," *Journal of Systems and Software* vol. 72(2), pp. 105-127, 2004.

[12] M. Marin, A. V. Deursen, and L. Moonen, " Identifying Crosscutting Concerns Using Fan-In Analysis," *ACM Transactions on Software Engineering Methodology,* vol. 17, pp. 1-37, 2007.

[13] M. P. Robillard, and G. Murphy, "Concern Graphs: Finding and Describing Concerns using Structural Program Dependencies," in Proceedings of the 22nd International Conference on Software Engineering (ICSE), Orlando, Florida, 2002, pp. 406-416.

[14] P. Tonella, "Using a Concept Lattice of Decomposition Slices for Program Understanding and Impact Analysis," *IEEE Transactions on Software Engineering* vol. 29(6), pp. 495-509, 2003.

[15] C. McMillan, D. Poshyvanyk, and M. Revelle, "Combining Textual and Structural Analysis of Software Artifacts for Traceability Link Recovery," in Proceedings of Workshop on Traceability in Emerging Forms of Software Engineering, Canada, 2009, pp. 41-48.

[16] 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.

[17] R. Koschke, and J. Quante, "On dynamic feature location," in Proceedings of the 20th International Conference on Automated software engineering, Long Beach, CA, USA, 2005.

[18] A. Egyed, "A Scenario-Driven Approach to Trace Dependency Analysis," *IEEE Transactions on Software Engineering* vol. 29(2), pp. 116-132, 2003.

[19] H. U. Asuncion, F. Francois, and R. N. Taylor, "An end-to-end industrial software traceability tool," in Proceedings of the 6th European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Dubrovnik, Croatia, 2007.

[20] O. Gotel, and A. Finkelstein, "Extended Requirements Traceability: Results of an industrial case study," in Proceedings 3rd International Symposium on Requirements Engineering, 1997, pp. 169-178.