Code Patterns for Automatically Validating Requirements-to-Code Traces

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, traces are known to be error prone. This paper introduces a novel approach for validating requirements-to-code traces through calling relationships within the code. As input, the approach requires an executable system, the corresponding requirements, and the requirements-to-code traces that need validating. As output, the approach identifies likely incorrect or missing traces by investigating patterns of traces with calling relationships. The empirical evaluation of four case study systems covering 150 KLOC and 59 requirements demonstrates that the approach detects most errors with 85-95% precision and 82-96% recall and is able to handle traces of varying levels of correctness and completeness. The approach is fully automated, tool supported, and scalable.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Statistical methods, Validation*

General Terms

Algorithms, Measurement, Reliability, Verification

Keywords

requirements, traceability, feature location, validation

1. INTRODUCTION

Requirements-to-code traces reveal where a requirement is implemented in the code. This is important for program comprehension and understanding change impact. Traces are most useful in cases where developers are not familiar with the source code – a situation that likely occurs during maintenance. It has been shown in several experiments [17, 22] that lack of understanding where a requirement is implemented leads to higher effort and more errors. This is not surprising because in such a case a developer is more likely to perform changes at inappropriate places in the code [8, 13, 19, 21], accelerating code degradation. Capturing and maintaining requirements-to-code traces is therefore mandated by many standards (such as CMMI level 3) and software engineering techniques [1, 3, 18].

Unfortunately, capturing and maintaining traces is an error-prone activity. Some automation exists but many approaches are not applicable to informal/unstructured requirements [10], which are common in practice. The few techniques, that do apply vary widely in quality [6] and often require elaborate descriptions of requirements, requirements dependencies, and/or the code to deliver good results. Even if traces are captured by the original developers and are thus expected to be of good quality, these traces may deteriorate over time as the system evolves. Trace maintenance counters this problem but as Kong et al. [13] have shown, manual trace maintenance is highly error prone and often made the quality of traces worse.

This paper introduces an automated, tool-supported approach for identifying errors among requirements-to-code traces by exploring patterns of traces and calling relationships within the code (i.e., method or function). For example, a given method is likely implementing a given requirement if it is called or calls other methods that also implement the given requirement. Our approach thus computes a trace expectation for a given method by investigating the known traceability of neighboring methods (callers and callees). An error is reported if this expectation differs from the known trace of the given method.

As input, our approach requires 1) existing requirements-to-code traces that need validation and 2) a call graph reflecting method/function calls in the code. The approach does not require special requirements descriptions nor knowledge about requirement dependencies. However, the approach does assume the code to be of good quality (i.e., executable and testable for the given requirements) though it is largely immune to isolated code errors. This assumption is realistic because traceability is typically needed during maintenance where the code has stabilized. To calibrate our approach, we empirically evaluated it on four open-source case study systems – totaling over 150 KLOC and 59 selected requirements. We chose these systems because we had high-quality requirement-to-code trace links, provided by their developers. The requirements were mostly functional and were often inter-dependent (cross-cutting) as is typical in practice. The empirical validation of the calibration showed that our approach applies to about 76-94% of the requirements-to-

ASE '12, September 3-7, 2012, Essen, Germany

code traces (coverage) and most of these traces are validated with >90% correctness. Good performance on calibration data, however, does not necessarily imply our approach's ability to cope with traces of decreasing quality. We thus performed an empirical study to understand the impact of input trace quality on the validation quality by organizing an elaborate experiment involving 85 subjects, who manually generated about 30,000 requirements-to-code traces ([8] discussed the experiment in detail). Since the subjects were predominantly inexperienced students, it can be assumed that this study led to traces of a wide range of qualities – quite possibly the worst quality which is a desired property here because the evaluation showed that our approach was able to validate even these traces with high precision and recall. While the quality of our approach does decrease with decreasing trace input quality and completeness, this decrease is small (>80% correctness even if 50% of the traces of a given requirement were incorrect). The approach thus vastly outperforms the ability of humans (which peaked around 50% correctness [13]). Also, contrary to manual validation, our approach's validation feedback becomes increasingly better with increasing input trace input quality.

This paper is based on a short paper which appeared in ASE 2011 [9] using a similar data set. The short paper demonstrated that code implementing a given requirement typically forms connected regions. This paper builds on this foundation and provides an algorithm for trace validation and an empirical study testing the algorithm's ability to cope with traces of varying levels of correctness and completeness.

The goal of this paper is to describe an algorithm for validating requirements-to-code traces. As input, the algorithm takes calling relationships in the code and a requirements-to-code trace matrix (RTM) were the matrix cells represent the individual relationships between code (methods/functions) and requirements. As a result, the algorithm produces feedback on traces which are presumed incorrect or missing. Since the algorithm proposed in this work is not immune to varying input quality, the goal of this paper is also to demonstrate that the quality of its feedback remains high with diminishing quality/completeness of input RTMs.

2. RELATED WORK

In last two decades, multiple traceability techniques have been presented where different approaches are applied to retrieve and/or maintain trace links between code and other software artifacts. These approaches exploit either textual characteristics [2, 11, 14], static structures [4], or dynamic information [2, 12, 14] of software artifacts. Each technique analysed this information in order to provide a certain understanding to the implementation of different requirements in the system. There are also hybrid approaches that combine multiple technique types in one approach (e.g., CERBERUS [7]). Usually, these approaches deliver a ranked list of potential traces among the input software artifacts where higher-ranked traces are presumed to be more correct traces. Calling relationships in particular have been used to improve the reported ranking of traces in information retrieval approaches [2, 5, 7, 16].

Multiple researchers 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* (ATG). They mainly aim to create or recover trace links from scratch. Different software artifacts, as well as derivatives thereof, are expected as input. The output would be a set of possible recognized trace links. 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. The results of those methods are validated against a benchmark trace link knowledge base (e.g., gold standard RTM) which was initially created manually. If an ATG method would be performed on a system without such a benchmark knowledge base, we would never know how good the produced trace links are.

Table 1: Information on the Four Study Systems

	VoD	**Chess**	**Gantt**	**JHD**
Language	Java	Java	Java	Java
KLOC	3.6	7.2	41	72
# Executed Methods	169	405	2591	1763
# Sample Reqs.	13	8	17	21
Size of Gold Standard	2197	3240	44047	37023

Indeed, the strongest motivation for our work comes from a study by Kong et al. [13] who conducted an experiment about manual trace link maintenance. As could be expected, the effectiveness of trace maintenance is the quality of the final RTM is influenced by the initial RTM. They made a very interesting observation: Participants who validated RTMs of poor quality tended to improve the RTM. Participants validating good quality RTMs, however, made more errors than correct decisions during the maintenance of RTMs making the accuracy of the input RTM worse. 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 [7, 16, 20], they have never been exploited together with potentially erroneous traces for validation. It is also noteworthy that our approach investigates each requirement's relationship to the source code independently of other requirements (i.e., unlike feature interactions technologies which are based on cross-cutting concerns [15], concern graphs [23] or concept lattices [24]).

3. CASE STUDIES

To validate our approach, we investigated four software systems between 3-72 KLOC in size and 59 randomly chosen requirements (covering functional, non functional, and cross-cutting requirements). The systems were VideoOnDemand (VoD), Chess, GanttProject, and jHotDraw (JHD) and their basic characteristics are depicted in Table 1. We chose them because of the availability of high-quality requirements-to-code traces – a gold standard provided by original developers (in case of the larger systems GanttProject and JHD) or developers familiar with the code (in case of the smaller systems VoD and Chess). We used the gold standard to gain confidence that the patterns we identified were indeed likely or unlikely (important for confirming or rejecting traces as will be discussed later). We also used the gold standard for evaluating the quality of the reported errors generated while validating subject RTMs to assess our approach's ability to cope with decreasing quality of the input RTM.

As is common, we represent traces in form of *Requirements Trace Matrices* (RTM). Each RTM contains $m \times n$

Table 2: Excerpt of RTM from Chess System

Method \ Requirement	R1: Play Move	R2: Show Score
addPlay()	X	X
afterPlay()		
doPlay()	X	
dropPiece()		
getPiece()		
getIndex()	X	X
initPiece()		
initPos()		
setColor()		
setPiece()	X	

cells where m is the number of requirements and n is the number of code elements (methods in this case). Table 2 depicts an excerpt of such a RTM for the Chess system. A cell is either a trace ('X') or a no-trace (blank). A trace in a cell implies that the given code element (row) implements the given requirement (column). A no-trace in a cell implies that the code element is not implementing the requirement. We refer to a trace also by the abbreviation 't' and to a no-trace by the abbreviation 'n'. Note that a RTM may also be incomplete if rows (code) or columns (requirements) are missing.

As was mentioned in the introduction, our approach uses calling relationships among code elements to help validate the correctness of the given requirements-to-code traces. There are standard tools available for recording execution logs during program execution. For example, the Java JDK provides an easy and reliable interface for recording method calls at runtime. We used this interface to record the calling relationships while each system was being tested. Each system was tested according to use case descriptions of the gold standards. But the testing was not limited to the sample requirements nor was it attempted to test the requirements individually to avoid any bias towards particular requirements. The use of the Java JDK ensures that there are no false calling relationships; However, incomplete testing may result in missing calling relationships. The implications of this will be discussed later in more detail.

The nodes in the resulting *call graph* represent individual classes, methods, or functions, depending on the level of granularity chosen. In this work, we focus on methods only but our approach should be applicable to non-OO languages (e.g., functions in C) as well as coarser granularity (classes or packages). Figure 1 depicts a small excerpt of the call graph for the Chess System. The nodes are labeled with method names and for brevity the figure depicts those listed in the RTM in Table 2 only. The edges in the call graph represent the method calls. These edges are directed arrows to distinguish the *caller* and the *callee*. For example, since method `getPiece` calls method `initPiece`, the node corresponding to `getPiece` has a directed arrow to the node corresponding to `initPiece`. We say that `getPiece` is a caller of `initPiece`, and, in reverse, that `initPiece` is a callee of `getPiece`. Note that there is exactly one node for every method. We thus use the terms node and method synonymously in the paper.

4. PATTERNS AND ALGORITHMS

We found that methods implementing a given requirement are very likely to call one another (88-99%) [9], forming *regions* of connected methods. These regions may vary in size and may also overlap if their corresponding requirements are

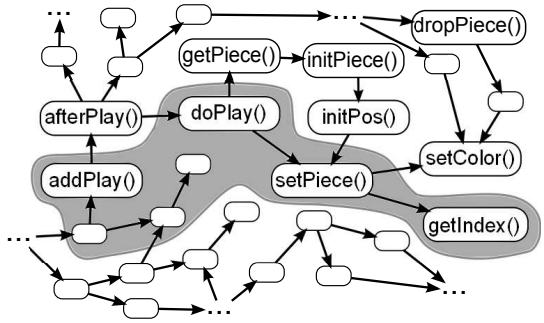

Figure 1: Modified Excerpt of Call Graph from the Chess System (R1 Region Highlighted in Gray)

implemented in close proximity or depend on one another. Our approach to trace validation exploits these regions.

4.1 Principles

Figure 1 shows an example of a region (gray area). The region identifies all methods (nodes) that trace to requirement R1 of the illustrative Chess system (note that the methods inside the gray area are defined to trace to R1 in the RTM in Table 2). The surrounding area represents the methods that do not trace to that requirement. We see that the nodes in the region are generally connected directly or indirectly; though it is not uncommon to have multiple regions also.

To understand the principles of trace validation, let us now assume the existence of an incorrect trace. Obviously, an incorrect trace (=false 't') may only exist in an area outside the region (all methods inside the region are correct 't's). If that false 't' exists far off the region then it is fully surrounded by methods that all do not trace to the given requirement. A method that is labeled as tracing to a given requirement in the input RTM but is being surrounded by methods that do not trace to that requirement thus indicates an erroneous trace with a high likelihood of correctness. Take for example the method `dropPiece`. This method does not trace to R1. However, if it were falsely labeled as tracing to R1 in the input RTM then it would form its own mini region and be surrounded by methods that do not trace to R1. Our approach would identify this method as an erroneous trace because it would be disconnected from a larger requirements region which is counter to the observation in [9]. Let us next assume the existence a missing trace (a false 'n'). A missing trace is a method that is currently marked as not tracing to a given requirement but should be tracing. A false 'n' may only exist inside the region (all methods outside the region are correct 'n's). If that false 'n' is surrounded by methods that trace to the given requirement then this indicates a missing trace with a high likelihood of correctness.

Listing 1: Trace Validation

```
validateCell( RTMCell c )
  expect = getExpectation(c.method, c.req)
  if (c.value = expect.value)
    return <CORRECT>
  else
    return <ERROR>
```

Algorithm 1 depicts the simple principle behind trace validation. Each cell of the RTM is investigated separately where each cell represents the traceability between exactly one method and one requirement. The algorithm then computes an expected trace for that cell which is based on the

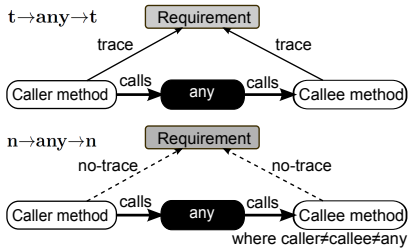

Figure 2: "t→any→t" and "n→any→n" Patterns

traceability of the neighboring methods. While the neighboring methods are the same for all cells in a row, their expectations may differ because they are investigated in context of different requirements. The complexity of trace validation is thus in computing an expectation which is discussed next.

4.2 T and N-Surrounding Patterns

The notion of "surroundedness" forms the basis for understanding expectation. There are two primary patterns: The t→any→t pattern (depicted in Figure 2-top) applies to *any* method that has at least one caller method (the *any* method is called by the caller method) and at least one callee method (the *any* method calls this callee method) where both the caller and callee trace 't' to a same, given requirement. Note that the *any* method represents an arbitrary method.

The n→any→n pattern (depicted in Figure 2-bottom) is simply the negation in terms of traceability. It applies to any method that has at least one caller method and at least one callee method that both do not trace to a given requirement.

Figure 3 depicts an excerpt of the Chess system introduced earlier that combines the call graph with the input RTM. For example, we know from the call graph that `afterPlay` calls `doPlay` and we know from the input RTM that `afterPlay` does not implement requirement R1 while `doPlay` does. This is depicted in the graph where thick arrows denote calling relationships, thin-solid arrows denote trace relationships and thin-dashed arrows denote no-trace relationships. We can identify a t→any→t pattern for method `setPiece` validated on requirement R1 because it is called by `doPlay` and calls `getIndex` which both trace to requirement R1. Similarly, we can identify a n→any→n pattern for method `doPlay` validated on requirement R2 because `afterPlay` and `setPiece` do not trace to R2.

We thus expect that `setPiece` traces to requirement R1 and we expect that `doPlay` does not trace to requirement R2. For trace validation we simply compare these expected values with the their corresponding cell values in the input RTM. Consulting Table 2, we find that our expectations are consistent with the actual cell values in both cases, and thus, our approach would confirm the correctness of these two input RTM cells. Note that neither the t→any→t pattern nor the n→any→n pattern used the trace information of the *any* method. The patterns merely used the trace information of the surrounding methods. The expected value computed through these patterns is thus derived without knowledge of the traceability of the cell being validated.

The t→any→t pattern expects a likely trace whereas the n→any→n pattern expects an unlikely trace (or a likely no-trace). This is useful for trace validation to confirm or reject a trace. However, as is common in traceability research, there are exceptions to their validity and below we will discuss some of these cases. It should be noted that we investigated more elaborate patterns (such as the *any* method

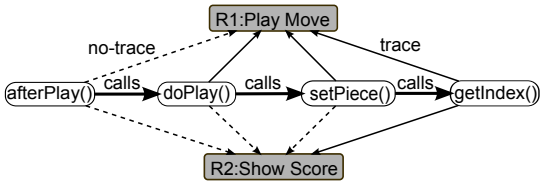

Figure 3: Surrounding Patterns in Chess

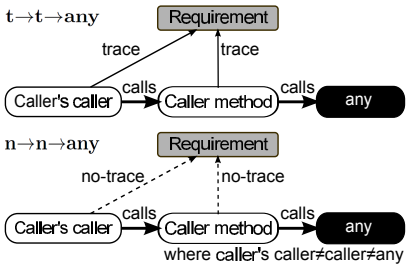

Figure 4: "t→t→any" and "n→n→any" Patterns

having at least two callers and callees) and such patterns improve the quality. However, we also found that more elaborate patterns are much less likely to be applicable (there are comparatively few methods that have two callers and callees with the same trace than compared to single callers and callees). More elaborate patterns are thus little helpful in improving the overall quality. We will see later that a far greater effect can be achieved by combining the simple t→any→t and n→any→n patterns into more elaborate structures.

4.3 Inner, Leaf, and Root Methods

The t→any→t and n→any→n patterns are useful only for methods that have both caller(s) and callee(s). We refer to such methods as inner methods (inner nodes in the call graph). Unfortunately, many methods do not call other methods – so called leaf methods which are leaf nodes in the call graph. In Figure 1, an example of such a leaf method is `setColor`. In context of the four case study systems, roughly half the methods are leaf methods. In the absence of a callee method, we define two additional patterns, depicted in Figure 4. The t→t→any pattern implies a likely trace if the given method has a caller that traces to a given requirement and this caller has in turn a caller that also traces to the given requirement. The n→n→any pattern is the opposite and implies an unlikely trace (or a likely no-trace).

A call graph usually has root methods also but they are rare. Indeed, all four case studies had few root methods each – the Java `main()` method is always one of them, often it is the only one. Other roots are typically methods starting separate threads (e.g., `Thread.start()`). Root patterns and their treatment are analogous to leaf patterns. Since root methods are rare they are of little significance and will be ignored in the remainder of this work.

4.4 Boundary Patterns

The inner method and leaf method patterns (t→any→t, n→any→n, t→t→any, and n→n→any) are not applicable to all methods. As was discussed earlier, requirement regions have boundaries where a method not implementing a given requirement calls a method implementing it or vice versa. We find such boundary cases in Figure 3 where, for example, method `doPlay` implements requirement R1 and calls `setPiece` which implements the same requirement, but it is

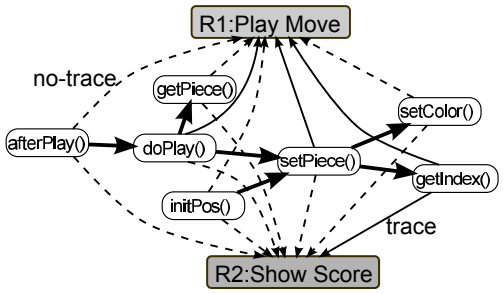

Figure 5: Mixed and Pure Patterns in Chess

called by method `afterPlay` which does not implement R1. We can think of such methods as entry and/or exit points to/from regions. We speak of a boundary pattern for:

- inner methods having a caller that traces to a given requirement while none of the callees do (t→any→n) or, in reverse, having a callee that traces to a given requirements while none of the callers do (n→any→t).

- leaf methods having a caller that traces to a given requirement while none of the caller's caller trace to the given requirement (n→t→any) or as having no caller that trace to the given requirement while at least one caller's caller does (t→n→any).

The expected values of T and N surrounding patterns (t→any→t and n→any→n) are intuitive to understand: the *any* method in a t-surrounding is expected to have a trace 't' link; the *any* method in a n-surrounding is expected to have a no-trace 'n' link. But, this simple intuition does not work for boundary patterns. Indeed, empirical observations (discussed later) show that the likelihoods of *any* boundary method tracing to the requirements vs. not tracing to that requirement is roughly 50% – or random. Since it is not useful to make random guesses, it may appear that boundary methods are failure scenarios where our approach fails to compute a useful expected value. But, even here our approach does not necessarily fail as will be discussed next.

4.5 Combinations: Pure and Mixed Patterns

Thus far, we discussed patterns in isolation. However, these patterns often occur together. Figure 5 illustrates such cases in context of the Chess system. This figure is analogous to Figure 3 but involves more methods. In context of requirement R1, the n→any→n pattern can be found through a pattern involving the `afterPlay`, `doPlay`(*any*), and `getPiece` methods. This pattern expects that the `doPlay` method does not trace to requirement R1. Additionally, the n→any→t boundary pattern can be found through a pattern involving `afterPlay`, `doPlay`(*any*), and `setPiece` methods. This boundary pattern suggests that it is not decidable whether `doPlay` traces to requirement R1. The method `doPlay` is thus involved in a n-surrounding pattern and a border pattern at the same time.

If multiple patterns can be matched on the given method (*any*) then one of two situations applies:

- Pure Patterns: all patterns are of the same kind. For example, in context of requirement R2 the `doPlay` method is involved with two n-surrounding patterns: one is found with `afterPlay` and `setPiece`; and another with `afterPlay` and `getPiece`.

- Mixed Patterns: the patterns are of different kinds. For example, method `doPlay` is involved in a mixed pattern with R1 as was discussed above.

The combined meaning of pure patterns are simple to decide: multiple t→any→t patterns re-confirm a likely trace; multiple n→any→n re-confirm an unlikely trace. Pure patterns are quite advantageous because in combination their likelihoods increase, making them more reliable patterns for trace validation. E.g., the empirical validation discussed later confirmed that if any method has two or more t→any→t patterns applying than the method is more likely to trace than compared to a method with a single t→any→t pattern. Mixed patterns, however, require further elaboration.

4.6 Trace, No-Trace over Boundary Dominance

A boundary pattern implies that it is not possible to decide on a likely or unlikely trace. Therefore, if a boundary pattern is found in combination with a likely pattern (t→any→t or t→t→any) or an unlikely pattern (n→any→n or n→n→any) then the likely/unlikely patterns dominate and we ignore the existence of the boundary pattern. We define *dominate* as meaning that, say, in case the t→any→t pattern applies to the same method (*any*) and requirement, this method is likely to trace to R, ignoring the boundary pattern. This was confirmed through the empirical evaluation of the four case study systems and their gold standards as will be discussed later. Most boundary patterns are ignored in this manner. Only if all patterns applying to a method are boundary patterns, a "pure boundary pattern" so to speak, then we truly cannot decide whether the method is expected to trace or expected to not trace to the given requirement. We will see later that our approach fails in a small percentage of RTM cells only.

4.7 Trace over No-Trace Dominance

But what if a likely trace pattern (t→any→t) occurs together with an unlikely trace pattern (n→any→n)? It is common for the t→any→t and n→any→n patterns to occur together. This happens when on each side (caller and callee) of a method, there are at least one method that traces to a given requirement and another that does not. Figure 5 depicts such an example for `setPiece`. The t→any→t pattern (involving `doPlay` and `getIndex`) suggests that `setPiece` traces to requirement R1. The n→any→n pattern (involving `initPos` and `setColor`) clearly suggests the opposite.

This apparent conflict is the result of cross-cutting requirements and/or feature interactions. In such cases, the code serves a purpose that supports a trace and another purpose that does not. Indeed, trace/no-trace conflicts of this kind are common in all four study systems and, upon empirically evaluating this issue on their respective gold standards, we observed that in such cases the t→any→t pattern dominates the n→any→n pattern.

The rationale for this empirical observation has to be seen in the context of granularity. It is said that if some code traces to a given requirement then this code implements the requirement either in part or full. Traceability is rarely exclusive and it is implicitly understood that the code may also implement other requirements. For example, in the context of Java class, even if only few methods of a class implement a given requirement then we say nonetheless that the class implements that requirement. Hence trace dominates no-trace.

4.8 Algorithm for Computing Expectation

Considering above discussion, a concise mechanism emerges that helps us decide whether *any* method is expected to trace or expected not to trace to a given requirement; or cannot be decided upon (fail). Algorithm 2 depicts the mechanism for computing the expected values of inner methods. We use a counting scheme because it is very efficient to execute compared to pattern matching algorithms. Four variables are used: #callerT identifies the number of callers of the input method that trace to the given requirement R; #callerN identifies the callers that do not trace to R; #calleeT identifies the number of callees that trace to R; and #calleeN identifies the number of callees that do not trace to R. Since this algorithm is called only for inner classes, we know that there must be callers and callees.

Listing 2: Get Expectation for a Given Inner Method and Requirement

```
getInnerExpectation(method M, requirement R)
#callerT:=#M. caller tracing to R
#callerN:=#M. caller not tracing to R
#calleeT:=#M. callee tracing to R
#calleeN:=#M. callee not tracing to R
if ( #callerT > 0 & #calleeT > 0)
  if ( #callerN+#calleeN = 0)
    return <TRACE, PURE>
  return <TRACE, MIXED>
else if ( #callerN > 0 & #calleeN > 0)
  if ( #callerT+#calleeT = 0)
    return <NO_TRACE, PURE>
  return <NO_TRACE, MIXED>
return <FAIL, BOUNDARY>
```

The algorithm shows that a given method is expected to trace to a given requirement if at least one caller method traces to the requirement (#callerT>0) and at least one callee method traces to the given requirement (#calleeT>0). If the trace is a pure trace then there also should be no caller and no callee methods that do not trace to the given requirement (#callerN+#calleeN=0). The algorithm thus returns either pure or mixed trace depending on the outcome. Similarly, the algorithm shows that a method is expected to not trace to a given requirement if both #callerN and #calleeN are greater than zero (i.e.., N surrounding). The T-over-N dominance is implied in the order of the ifs (trace checking before no-trace checking). Note that at the time of no-trace checking, it is no longer possible that both callers and callees trace to the given requirement. However, it is still possible that either caller or callee traces to a requirement. This allows us to distinguish a pure no-trace from a mixed no-trace (note that the mixed no-trace is essentially the N-over-boundary dominance). All other situations are failure scenarios (i.e., pure boundary patterns). As mentioned earlier, the algorithm distinguishes pure from mixed patterns and leaf from inner methods because the likelihoods of correctness are different. This is discussed below. Not depicted is the algorithm for computing the expected values for leaf methods because it is essentially identical if we assume the variable #calleeT to stand for callers' callers that trace to the given requirement and #calleeN to stand for callers' callers that do not trace.

4.9 Expectations for Incomplete RTMs

The RTM in Table 2 is complete because it identifies for every cell whether it traces or does not trace to a given requirement. However, we believe that it is useful to distinguish defined traces/no-traces from undefined traces (incompleteness). Since to date very little work is available for maintaining traces, it is not a common practice to provide partially complete RTMs. However, we believe that developers may want to start validating RTMs even at times where the RTMs are not yet fully captured. Thus trace maintenance ought to apply to incomplete RTMs also. Our approach allows for arbitrary cells to be left undefined ('U'). Undefined cells cannot be validated because the expected value cannot contradict or confirm an undefined cell. At best, the expected value could be used for auto completion. However, undefined cells obstruct the validation of other cells. The following defines what happens if a pattern encounters an undefined cell.

Listing 3: Get Expectation for a Given Inner Method and Requirement with Incompleteness

```
getInnerExpectation(method M, requirement R)
...
#callerU:=#M. caller with undefined traces to R
#calleeU:=#M. callee with undefined traces to R
if ( #callerT > 0 & #calleeT > 0)
  if ( #callerN = 0 & #calleeN = 0 &
       #callerU = 0 & #calleeU = 0)
    return <TRACE, PURE>
  return <TRACE, MIXED>
else if ( #callerN > 0 & #calleeN > 0)
  if ( #callerT = 0 & #calleeT = 0 &
       #callerU = 0 & #calleeU = 0)
    return <NO_TRACE, PURE>
  if ( #callerU > 0 & #calleeU > 0)
    return <FAIL, INCOMPLETE>
  return <NO_TRACE, MIXED>
else if ( #callerU > 0 | #calleeU > 0)
  return <FAIL, INCOMPLETE>
return <FAIL, BOUNDARY>
```

Algorithm 3 expands on Algorithm 2. For brevity, we again restrict this discussion to inner methods. As was discussed above, the leaf methods are handled analogously. The algorithm introduces two more variables to reflect the number of undefined callers and callees – #callerU identifies the number of calling methods that have undefined traces and #calleeU identifies the number of callee methods with undefined traces. To understand Algorithm 3, we must understand that an undefined cell is a placeholder for either a trace or a no-trace – we just do not know yet. The implication of an undefined trace is thus explored by considering both possible interpretations. For example, Algorithm 3 has an alternative fail scenario at the end. The last "else if" statement is reached only when all caller methods and all callee methods have undefined traces to a given requirement (pure and mixed scenarios are addressed by earlier statements). The method may thus trace to the requirement (if the undefined traces are eventually changed to traces), the method may not trace (if the undefined traces are eventually changed to no-traces), or we may even fail to compute an expectation for the undefined traces (if the traces resolve to a boundary pattern). Anything is possible. Our approach reports this problem as failure to compute an expectation due to incompleteness. Note that this failure is a factor of the completeness of the input and not a limitation of the approach as in boundary patterns and thus we distinguish boundary from incompleteness failure. A more complete input would fix this failure.

The existence of undefined traces does not always lead to failure to compute an expectation. Let us assume that the RTM cell for `getPiece` and requirement R2 is undefined in Table 2. The cell `getPiece` cannot be validated; however, the cell for its neighboring method `doPlay` can (recall Figure 5 but assume that there exists no "no-trace" edge between `getPiece` and R2). During the validation of method `doPlay` on requirement R2 we find a n→any→n pattern involving methods `afterPlay`, `doPlay`, and `setPiece`. However, no patterns apply to the methods `afterPlay`, `doPlay`, and `getPiece` because the trace for `getPiece` is undefined. Method `getPiece` may either trace to requirement R2 or it may not. If it were to trace then methods `afterPlay`, `doPlay`, and `getPiece` would form a n→any→t boundary pattern. We know that the n→any→n pattern above would dominate this boundary pattern and hence we would expect that `doPlay` does not trace to R2. If method `getPiece` were not to trace to R2 then methods `afterPlay`, `doPlay`, and `getPiece` would form another n→any→n pattern, which would reconfirm the other one and we would also expect that `doPlay` does not trace to R2. Therefore, no matter how the undefined trace is eventually resolved, we expect that `doPlay` does not trace to R2. Knowing about the undefined trace would only influence our certainty about the expectation but not the expectation itself. Reasoning in the presence of incomplete input RTMs is thus possible. Algorithm in Listing 3 thus also expands on the pure/mixed trace/no-trace computation. For example, we see that a no-trace expectation may lead to an incompleteness failure only if both #callerU and #calleeU are greater than zero. In the case of `doPlay`, #calleeU is not zero and hence the algorithm returns a mixed no-trace. Pure trace/no-trace patterns are found only if #callerU and #calleeU are zero.

4.10 Algorithm for Validation

Based on the mechanism for computing an expectation for a given method and requirement, trace validation is quite trivial. Algorithm 4 expands on Algorithm 1 we introduced earlier. It iterates over every RTM cell and the validation algorithm is composed of two parts: 1) to compute an expectation via `getExpectation` and 2) to compare the expectation with the actual RTM value via `validateCell`. The `getExpectation` algorithm distinguishes between inner nodes and leaf nodes. Note that the algorithm for computing expectations of incomplete RTM cells (Algorithm 3) subsumes the earlier Algorithm 2. The expectation returned by it is a tuple containing the value (TRACE, NO_TRACE, FAIL) and the category (MISSING, BOUNDARY, INCOMPLETE, PURE, MIXED). The value of the expectation is then compared to the value of the RTM cell. If the expected value matches the current value then the cell is presumed correct. Otherwise, it is tagged as erroneous. If the computation of the expectation fails then this is reported as a warning. A warning implies that the RTM cell could not be validated automatically and requires manual checking. Since we can establish the category of a cell (pure or mixed trace/no-trace), it is possible to order the error feedback based on the different likelihoods associated with the categories discussed next. It is noteworthy that the algorithm is also able to detect missing calls or dead code if the code has neither callers nor callees (see <FAIL, MISSING> in `getExpectation`). Cells that are not called or do not call cannot be validated by our approach. However, this fail scenario is a factor of the input and not a characteristic of the approach. In the most extreme case, no call graph is given as input in which case no method can be validated. We will discuss in threats to validity why smaller numbers of missing calls are unlikely to strongly influence our algorithm.

Listing 4: Validation Algorithm

```
validateCell( RTMCell c )
  expect = getExpectation(c.method, c.req)
  if (expect.value=FAIL or c.value=UNDEFINDED)
    return <WARNING, expect.category>
  else if (c.value = expect.value)
    return <CORRECT, expect.category>
  else
    return <ERROR, expect.category>

getExpectation( Method m, Requirement r )
  if ( isInnerMethod(m) )
    return getInnerExpectation(m, r)
  else if ( isLeafMethod(m) )
    return getLeafExpectation(m, r)

  return <FAIL, MISSING>
```

4.11 Likelihoods on Gold Standard

Section 4.1 claimed in several places that our understanding of the adequacy of the introduced patterns is based on empirical data on four case study systems. We can think of this as a calibration step to understand the performance of our approach under ideal circumstances (ideal because the gold standards are of good quality). This performance is expected to decrease as the quality of the RTM decreases. This is explored in the evaluation section later.

We investigated each category (fail, mixed, and pure for trace and no-trace) in terms of correctness and coverage. Table 3 depicts our findings. The coverage represents the percentage of the cells that fall into a given category. The percentages for coverage add up to 100% for the entire RTM. The correctness refers to the likelihood of our algorithm validating a cell of its respective category correctly (e.g., identifying an incorrect no-trace or trace in the RTM). This data is based on the gold standards of the four cases study systems. As was already suggested earlier, the likelihoods differ depending on category and system. By considering combinations of patterns and dominance, we see that many cells are validated with high likelihoods of correctness. For example, line 4 "n (pure)" in Table 3 depicts the correctness and coverage for "pure no-trace" cells, which are cells where only n→any→n patterns are found. We see that our approach is 96-99% likely to correctly validate such cells and 18-56% of all RTM cells fall into this category. Pure trace cells are also validated with high correctness (line 5). The correctness is a bit lower for mixed traces and mixed no-traces; however, do note that these kinds of cells are encountered less often (only 2-9% of all cells are mixed traces). The lower likelihoods of correctness thus do not affect the overall performance strongly; however, the qualities are still high compared to manual trace validation. As was discussed, the approach fails for pure boundary methods and only 3-8% of inner methods and 1-9% of leaf methods fall into this category. This is a low percentage and implies that our algorithm is able to validate most RTM cells. To summarize this figure, Table 4 provides overall precision and recall numbers:

Table 3: Coverage (Cov.) & Correctness (Cor.)

		Chess		Gantt		JHotDraw		VoD	
		Cor.	Cov.	Cor.	Cov.	Cor.	Cov.	Cor.	Cov.
Inner	fail	N.A	6%	N.A	5%	N.A	3%	N.A	8%
	n (mixed)	74%	3%	79%	6%	85%	6%	76%	4%
	t (mixed)	85%	3%	76%	3%	66%	2%	67%	3%
	n (pure)	98%	19%	96%	36%	99%	57%	96%	28%
	t (pure)	96%	15%	90%	2%	92%	1%	92%	6%
Leaf	fail	N.A	9%	N.A	4%	N.A	1%	N.A	7%
	n (mixed)	75%	2%	92%	3%	90%	2%	79%	1%
	t (mixed)	75%	4%	41%	3%	41%	1%	46%	1%
	n (pure)	90%	23%	95%	33%	98%	26%	91%	30%
	t (pure)	87%	16%	58%	4%	65%	1%	57%	13%

Table 4: Precision & Recall on Gold Standard

	VoD	Chess	Gantt	jHotDraw
Precision	85.3%	94.1%	92.1%	92.8%
Recall	82.4%	84.3%	90.2%	96.1%

$$Precision = \frac{Correctly\,Validated\,Cells}{Validated\,Cells\,(Correctly + Incorrectly)}$$

$$Recall = \frac{Correctly\,Validated\,Cells}{Correctly\,Validated\,Cells + Failed\,Cells}$$

Precision reflects the ratio of correctly validated cells of totally validated cells (including wrong validation). Recall reflects the number of cells that our approach failed to validate compared to the ones it correctly validated (i.e., missing validations). We see that both precision and recall are very high, mainly because pure trace/no-trace situations are very common.

5. EVALUATION

In this Section, we demonstrate the ability of our approach to validate RTMs of arbitrary quality.

5.1 Precision and Recall on Erroneous RTMs

Considering the nature of our patterns, a random error in the RTM is easily identifiable because our algorithm fails only in case of a coordinated set of errors: i.e., to change a likely trace to an unlikely trace for a given method and requirement, requires all caller and callee 't's to be changed to 'n's – an unlikely scenario. However, are RTM errors random or are these errors concentrated in certain areas of the code which increases the chances of a coordinated error? We thus conducted an experiment where 85 subjects were asked to identify the traceability for the larger two systems in our study: Gantt and jHotDraw. The subjects had to inspect the code and then provide traces. Due to the effort involved, each subject typically investigated a part of the original RTM. We then obtained a wide spread of RTM qualities by seeding the gold standard with a variable number of subject traces. The worst RTM we validated included the errors from all subjects combined. This process reflects industrial practice where traces are often captured by multiple developers and are thus expected to have mixed qualities.

We validated the combined RTMs using our algorithm. The validation results were compared to the gold standards to test the level of correctness of the feedback computed by the algorithm. We measured this feedback in terms of precision and recall (introduced above) and Figure 6 depicts the precision and recall in relationship to the percentage of

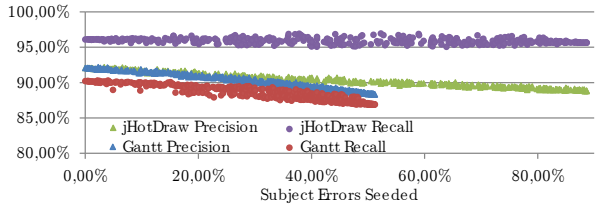

Figure 6: Precision and Recall decreasing slightly with increasing erroneous RTMs

errors introduced by the subjects (relative to the gold standard). Since the traceability literature usually attributes errors relative to the correct number of traces, the x-axis of the Figure 6 reflects erroneous and missing traces relative to the total number of traces. For example, if a requirement traces to 100 methods then 10% error implies that the traces of 10 of these 100 methods were either missing (a no-trace instead of a trace) or wrong (a trace instead of a no-trace). The Figure combines all cells of a RTM and hence combines all requirements tested. There were some variations among the requirements but all requirements we tested behaved similarly.

Our observation on both Gantt and jHotDraw is that precision and recall were slightly affected by subject errors. The quality fell by less than 5% in the worst case. Note that this data is based on the validation of over 80,000 cells with human generated tracing errors among the nearly 30,000 subject traces, which also explains the difference between the maximum number of errors seeded for jHotDraw (about 90%) and the maximum number of errors seeded for Gantt (about 50%). The seeded erroneous traces are arbitrary distributed over both systems. This demonstrates that our approach remains useful in RTMs of varying qualities.

5.2 Precision and Recall on Incomplete RTM

Traditionally, RTMs are presumed to be complete because most applications of traceability require this. Yet, we see trace validation as an ongoing process that must not necessarily start with a complete RTM but could also start at a time where traceability knowledge is only partially available. Our approach works for both complete and incomplete RTMs as was discussed earlier. This section investigates the impact of incompleteness onto trace validation quality.

Figure 7 depicts precision and recall with increasing incompleteness of the input RTM. The incompleteness of the RTM designates commonly the number of cells which are not yet traced. These cells could be randomly distributed in the RTM, aligned in a row when a method has not been traced, or aligned in a column when a requirement has not been traced. Since there are no studies available on how and why traces are incomplete and since the 85 subjects chose not to leave cells incomplete either, we resorted to random seeding to evaluate incompleteness (i.e., removing random cells from gold standard). For example, at 50% incompleteness, half the input RTM cells are undefined (i.e., 'U') and the other half is validated.

The algorithm generating expectations for incomplete input (see Listing 3) isolates the cells which are affected by incompleteness under the category `<FAIL,INCOMPLETE>`. The increasing number of undefined cells will thus increase the total number of cells for which our validation algorithm (see Listing 4) would report a `<WARNING>` and can not validate

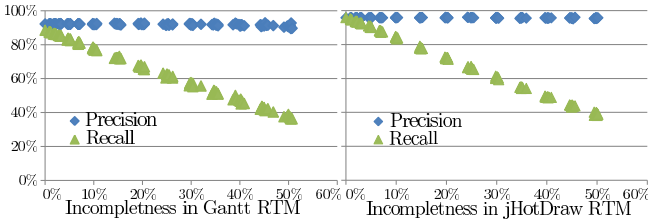

Figure 7: Precision is unaffected by incompleteness. Recall falls linearly only with level of incompleteness

them. Considering the Precision and Recall formulas presented in Section 4.11, we deduce that the increasing `<FAIL>` cells due to incompleteness will only decrease the recall, but do not affect the precision, which is clearly visible in Figure 7 where only the recall is falling linearly but the precision is largely unaffected by the increasing incompleteness. This implies that the correctness of the approach remains high even in the presence of incompleteness.

5.3 Scalability

Our approach is fully automated and tool supported. The computational overhead is linear with the size of the RTM with the largest RTM requiring less than 3 minutes validation time. Most of the runtime was required for parsing the execution log to build the call graph, which is usually performed only once – when the data is loaded the first time. The actual validation algorithm needed less than 100 ms in all cases studies since the validation algorithm is based on a rather simple but effective counting scheme for computing expectations out of the surrounding traces. (recall Section 4.8). Further details about the tool used in this study may be found under: `http://www.sea.uni-linz.ac.at/tools/TraZer`.

5.4 Threats to Validity

We used four systems of different domains (see Table 1) with RTMs of different sizes. Each cell in a RTM was validated separately and this large number of cells makes our findings statistically representative. To assess precision and recall, we relied on a gold standard provided by the original developers of the study systems. The gold standard was probably not perfect. However, given that it requires a coordinated set of errors to fool our patterns and given that 1) so many patterns were validated, 2) the patterns exhibited the roughly same effects on all four study systems, and 3) the study systems were independently developed by different people who did not know each other, leaves us to conclude that the studies are representative.

However, we can see two situations where our algorithm would fail: 1) a requirement being implemented in very few methods only or 2) the methods implementing a requirement being not connected (no calling relationships). As to point 1: a requirement that is implemented in a single method only would appear like an erroneous trace because it would be surrounded by methods that do not trace (n-surrounding). However, it is possible to detect such situations in advance by investigating the input RTM. Our algorithm should not be used on such requirements but we believe such cases to be rare. As to point 2: if the methods implementing a requirement were not connected then they would also be detected as erroneous traces because each method would be n-surrounded. However, connectedness was confirmed on all 59 requirements we investigated, covering four diverse case study systems. Most of these requirements were functional; however, the jHotDraw system also had 5 non-functional requirements. Although we are not claiming that these requirements are of all possible types, we believe that they are representative as they were selected and traced by independent developers of each system. While the issue of connectedness warrants further investigation, at present it appears to be a reasonable assumption for most situations. Like point 1, a problem with the connectedness could be detected by investigating the input RTM.

This work presumed that the input call graph was reasonably correct and complete. Unfortunately both assumptions might be unsatisfied under certain circumstances. On the one hand, incorrect calling relationships could be caused by undetected bugs in the code. Our assumption is that the code is of reasonably good quality since trace validation is most relevant during software maintenance where, typically, a mature code base exists. Still, isolated bugs may exist. But in order to affect our reasoning, two surrounding calling relationships should be reverted to the wrong kind of traces(i.e. t-surrounding pattern to n-surrounding pattern or vice-versa). In other words, the very nature of our patterns makes it unlikely that isolated code errors significantly influence the quality of the approach. On the other hand, incompleteness in the call graph relationship is more likely when using dynamic call graphs as in our case. Yet, the level of completeness is measurable through metrics that assess the test coverage (e.g., branch coverage tests). Even if the call graph is incomplete, certain patterns would be unaffected. For example, a t-surrounding pattern can never become an n-surrounding pattern with more calling relationships. Static call graph may be an alternative that does not require a running system, but with a down side that we would have to deal with both incomplete (e.g., polymorphism) and incorrect (e.g., dead code) code relationships.

Finally, our work used homogeneous levels of granularity (methods) in call graphs and RTMs. We presume that our approach would perform similarly on other homogeneous levels of granularity (e.g., statements or classes) though the likelihoods would presumably be different. This assumption will be assessed in future work.

In summary, we believe that there is no significant threat to the internal validity and the data/experiments were diverse enough to imply more general applicability – hence we see no major threat to the external validity either. The only uncertainty was our evaluation on incompleteness where we used random seeding. This may not have been appropriate; however, we were unable to find any literature that characterized incompleteness and believe this to be mostly an organizational issue that, from an outsider's perspective, appears random.

6. CONCLUSION

This paper represents an important step towards improving traceability automation because state-of-the-art predominantly focused on recovering requirement-to-code trace links but not maintaining them. Since automated trace recovery or recovery done by subjects not familiar with the code rarely exceeds 50% correctness, we believe that traces should be captured early on by subjects familiar with the system. However, these traces then need to be maintained like other development artifacts with the system. This paper demonstrates a novel approach to maintaining requirements-to-

code traces by validating them in context of code calling relationships. We introduced an algorithm that computes trace expectations based on likely and unlikely patterns and then compares them with the given traces. Errors are detected whenever the given traces differ from the expected traces. Empirical evaluation on over 80,000 of RTM cells have shown that the error feedback produced by our approach is of high precision and recall and the quality of the feedback being lightly affected by the quality of the given traces only. The approach scales, is fully automated, and tool support is provided. The approach applies to functional and non-functional requirements, whether they be independent or cross-cutting.

7. ACKNOWLEDGMENTS

This work was funded by the Austrian Science Fund (FWF) under agreement P23115-N23.

8. REFERENCES

[1] N. Aizenbud-Reshef, B. T. Nolan, J. Rubin, and Y. Shaham-Gafni. Model traceability. *IBM Systems Journal*, 45(3):515–526, 2006.

[2] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo. Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering*, 28(10):970–983, Oct. 2002.

[3] L. C. Briand, Y. Labiche, and T. Yue. Automated traceability analysis for UML model refinements. *Inf. Softw. Technol.*, 51(2):512–527, Feb. 2009.

[4] K. Chen and V. Rajlich. Case study of feature location using dependence graph, after 10 years. In *18th IEEE International Conference on Program Comprehension*, pages 1–3, Braga, Portugal, June 2010.

[5] J. Cleland-Huang, B. Berenbach, S. Clark, R. Settimi, and E. Romanova. Best practices for automated traceability. *Computer*, vol. 40:27–35, 2007.

[6] B. Dit, M. 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)*, accepted for publication, 2011.

[7] M. Eaddy, A. V. Aho, G. Antoniol, and Y. Guéhéneuc. CERBERUS: tracing requirements to source code using information retrieval, dynamic analysis, and program analysis. In *16th IEEE International Conference on Program Comprehension*, pages 53–62, Amsterdam, The Netherlands, June 2008.

[8] A. Egyed, F. Graf, and P. Grunbacher. Effort and quality of recovering Requirements-to-Code traces: Two exploratory experiments. In *18th IEEE International Requirements Engineering Conference*, pages 221–230, Sydney, Australia, Sept. 2010.

[9] A. Ghabi and A. Egyed. Observations on the connectedness between requirements-to-code traces and calling relationships for trace validation. In *26th IEEE/ACM Internation Conference on Automated Software Engineering*, 2011.

[10] O. Gotel and C. Finkelstein. An analysis of the requirements traceability problem. In *Proceedings of IEEE International Conference on Requirements Engineering*, pages 94–101, Colorado Springs, CO, USA, Apr. 1994.

[11] J. Hayes, A. Dekhtyar, and S. Sundaram. Advancing candidate link generation for requirements tracing: the study of methods. *IEEE Transactions on Software Engineering*, 32(1):4–19, Jan. 2006.

[12] M. Jiang, M. Groble, S. Simmons, D. Edwards, and N. Wilde. Software feature understanding in an industrial setting. In *22nd IEEE ICSM*, pages 66–67, Philadelphia, PA, USA, Sept. 2006.

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

[14] A. Marcus and J. I. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *Proceedings of the 25th ICSE*, ICSE '03, pages 125–135, Washington, DC, USA, 2003. IEEE Computer Society.

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

[16] 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*, pages 41–48, Vancouver, BC, Canada, May 2009.

[17] P. Mäder and A. Egyed. Do software engineers benefit from source code navigation with traceability? – an experiment in software change management. In *26th International Conference on Automated Software Engeering*, Kansas, November 2011. IEEE/ACM.

[18] R. E. K. S. Min Deng and B. H. C. Cheng. Retrieval by construction: A traceability technique to support verification and validation of uml formalizations. *International Journal of Software Engineering and Knowledge Engineering (IJSEKE)*, 15:837–872, 2005.

[19] D. L. Parnas. Software aging. In *Proceedings of the 16th international conference on Software engineering*, ICSE '94, pages 279–287, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

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

[21] B. Ramesh, L. C. Stubbs, and M. Edwards. Lessons learned from implementing requirements traceability. *Crosstalk – Journal of Defense Software Engineering*, 8:11–15, 1995.

[22] M. Robillard, W. Coelho, and G. Murphy. How effective developers investigate source code: an exploratory study. *IEEE Transactions on Software Engineering*, 30(12):889–903, Dec. 2004.

[23] M. P. Robillard and G. C. Murphy. Concern graphs: finding and describing concerns using structural program dependencies. In *Proceedings of the 24th ICSE*, page 406, Orlando, Florida, 2002.

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