Combining Textual and Structural Analysis of Software Artifacts for Traceability Link Recovery

Collin McMillan, Denys Poshyvanyk, Meghan Revelle

Department of Computer Science
The College of William and Mary
Williamsburg, VA 23185
{cmc, denys, meghan}@cs.wm.edu

Abstract

Existing methods for recovering traceability links among software documentation artifacts analyze textual similarities among these artifacts. It may be the case, however, that related documentation elements share little terminology or phrasing. This paper presents a technique for indirectly recovering these traceability links in requirements documentation by combining textual with structural information as we conjecture that related requirements share related source code elements. A preliminary case study indicates that our combined approach improves the precision and recall of recovering relevant links among documents as compared to stand-alone methods based solely on analyzing textual similarities.

1. Introduction

Existing methods of traceability link recovery rely on the analysis of textual information derived from software artifacts using Information Retrieval (IR) techniques [1, 2, 7, 11, 13, 18, 19, 21]. Feature and class descriptions in external documentation, for instance, are often likely to share some terms in common with their related source code artifacts due to naming conventions and formats that enforce identifier names, making IR techniques effective at discovering these *documentation-to-source* traceability links. However, there is no reason why two related documentation artifacts will use similar terminology since they might describe different tasks. IR methods are unlikely to recover useful traceability links in this situation. Therefore, an analysis of textual similarities among related source code artifacts (e.g., classes) may not yield any pertinent links.

Another approach to traceability link recovery is structural analysis, which alleviates this particular problem by examining control and data flow (i.e., coupling) among source code artifacts.

Taken together, textual and structural analyses provide good methods to find *source-to-source* and *source-to-documentation* links, but not *documentation-to-documentation* links. Structural analysis clearly lends itself to source code analysis, whereas using IR methods for traceability link recovery is a well-studied problem [1, 6-9, 11, 13, 15, 21]. In this work, we combine both textual and structural analysis methods on two types of software artifacts (i.e., source code and documentation) as well as on artifacts of one type to another, to create an indirect traceability link recovery scheme for software documentation.

This paper makes the following contributions:

- A method for indirectly recovering traceability links in requirements documentation using a combination of textual and structural information;
- A comparison of the proposed method with existing IR-based approaches for traceability link recovery;
- An investigation into the impacts of choosing various dimensionality reduction factors on the performance of both combined and stand-alone IR methods.

The paper is organized as follows. Section 2 presents background information on existing approaches for traceability link recovery. Section 3 outlines some key details of our approach. Section 4 presents an initial case study comparing the proposed combination with existing methods. Finally, related work and conclusions are outlined in Sections 6 and 7 respectively.

2. Background

In this section, we provide essential background information on the tools we use for textual and static analysis of software artifacts.

2.1. Textual analysis

Latent Semantic Indexing (LSI) [10] is an IR technique that is used to determine textual similarities among words and documents in large passages of text. For example, LSI can determine how similar the text of

a section of documentation is to the text of a method in source code.

LSI works by extracting all the words from a set of documents and generating a corpus. In our case, the documents are source code methods and sections of external documentation. Common words, known as stop words, are removed from the resulting corpus. Words in the corpus documents are converted to their stems (e.g. "words" becomes "word" and "converted" becomes "convert"). Additionally, compound identifiers, such as "compoundName" and "compound_name," are split based on the observed naming conventions. Next, LSI indexes the corpus and creates a signature for each document. This signature is used to determine similarity values between the documents. If one wishes to find source code methods that pertain to a section of documentation, the cosine similarity values are computed between this section of documentation and all the other source code documents in the corpus.

Often, the number of documents prohibits manual inspection of all the documents in the results, thus several different strategies, such as cut-points, thresholds and combined strategies [1, 7, 21] for retrieving candidate traceability links are used. In this work, we use a strategy based on similarity threshold values. According to this strategy, all documents with a textual similarity value above the threshold are considered as candidate traceability links.

Like the Vector Space Model (VSM) of analysis it is based on, LSI forms a term-by-document matrix in which term frequencies are included for each document; that is, each entry represents how common a given term is in a given document. When working with a natural language, however, related terms will not always overlap. Thus two documents will not be considered related if they discuss related topics but use a different vocabulary. This is known as the synonymy problem in natural language processing.

By applying Singular Value Decomposition (SVD) [27], LSI is able to somewhat overcome this problem [17]. SVD breaks the term-by-document matrix into singular values and vectors that are truncated such that the most significant portions of the matrix are preserved. These top components are reconstituted into an approximation of the term-by-document matrix. Document similarities are found by comparing the vectors of this approximation, just as is done with the original in VSM. As a result, SVD creates an approximation in which related documents appear to have a higher similarity because they share more related terms [7, 16, 17]. The number of most significant entries is known as the *dimensionality reduction factor* for SVD. Intuitively, since smaller reduction factors mean less space in which to approximate the term-by-document matrix, more documents will appear related as the reduction factor is decreased. In other words, low dimensionality reduction factors exaggerate the effects of SVD approximation. We explore how this behavior affects our approach during the case study.

2.2. Structural analysis

JRipples[1] [3] is a structural analysis tool for tracking incremental changes in Java systems. Taking as input source code and an indicated main method, JRipples builds an *Evolving Interoperation Graph* (EIG) [25] at either class or method granularity. At method granularity, for example, the output of this initial analysis is a graph where all methods and variables are nodes (of type *method* and *field*, respectively) and all relationships among these nodes are directed edges. Methods point to all variables used and other methods called within. As the programmer adds or alters functionality to the system, JRipples provides a list of related code portions for the programmer to step through and mark indicating whether a change needs to occur to that area. We use JRipples in our work to recover *source-to-source* traceability links among source code artifacts (i.e., methods or classes).

3. Approach

The intermediate goal in combining structural and textual analysis is to combine the two sources of information into a common representation. Our solution is based on a *traceability link graph* (TLG) that encodes all recovered textual and structural links as edges between nodes representing either program requirements or source code methods. The following are requisite definitions used in our model:

Definition. A *source artifact node* in a TLG is any node describing a portion of source code (e.g., a method).

Definition. A *documentation artifact node* in a TLG is any node describing a portion of software documentation (e.g., a section in documentation).

Definition. A *document-to-source edge* in a TLG is an edge between a documentation artifact node and a source artifact node describing a traceability link between these nodes' respective software artifacts.

Definition. A *source-to-source edge* in a TLG is an edge between two source artifact nodes describing a traceability link between these nodes' respective software artifacts.

Definition. A *document-to-document edge* in a TLG is an edge between two documentation artifact nodes describing a traceability link between these nodes'

[1] http://jripples.sourceforge.net/ (verified on 1/17/09)

respective software artifacts.

3.1. TLG construction

We collect the elements of a TLG by reading the information sources one at a time so that elements are added in this order:

- All Java methods extracted by JRipples from the source code of the target system are added as *source artifact* nodes. Technically, we do this by stripping the EIG of all the nodes except those for methods;
- All edges between EIG nodes for Java methods given by JRipples are added, creating the *source-to-source* edges (e.g., method invocations). In other words, we use structural analysis implemented by JRipples to find the *source-to-source* edges (i.e., coupling);
- Since we want to find traceability links among requirements documentation, each requirement is added as a *documentation artifact* node;
- The textual similarities among all Java methods and each requirement are obtained from LSI. We apply a threshold to determine the candidate traceability links. These links comprise the *documentation-to-source* edges in the TLG.

3.2. TLG interpretation: finding new links

The *document-to-document* edges of a TLG represent the traceability links among the requirements documentation and are therefore what we attempt to recover. To do this, we interpret the constructed TLG by applying ***two rules*** in order to suggest documentation links. Consider the example graph in Figure 1. Nodes *s[1...4]* are source artifact nodes, and nodes *d[1...3]* are documentation artifact nodes. In the simple case (our first rule), nodes *d2* and *d3* point to *s4*, so we deduce that a *document-to-document* edge should exist between those nodes because they share relevant source elements. That is, they reference (or are implemented by) the same method. Our second rule is a variation on the theme as the first: we suggest a link between *d1* and *d3* because an edge exists from *d1* to *s1*, from *s1* to *s3*, and from *d3* to *s3*. In other words, we recommend linking *d1* to *d3* since a source code element pointed to by *d1* is linked to a source code element pointed to by *d3*.

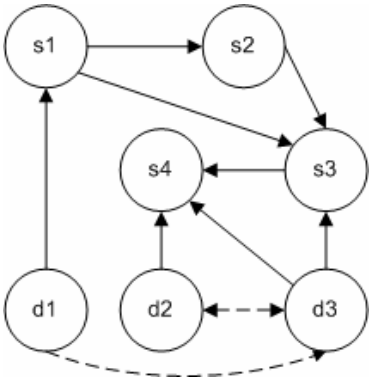

Figure 1: A sample traceability link graph. Solid lines are harvested links; dashed lines are suggested ones

Note that in our current implementation, the edges are not weighted. Having one source element link to another will produce an edge between the associated nodes of those elements; another link in the same direction between the same elements will affect neither the graph nor our analysis. The *document-to-document* edges recovered using the rules described above is our list of suggested candidate links for the example in Figure 2.

4. Case study

We implemented the steps defined in Section 3.1 as an ***I**ndirect Traceability **L**ink **R**ecovery **E**ngine* (*ILRE*), as presented in Figure 2. Taking as input a similarity matrix from LSI, an EIG from JRipples, and a threshold for the similarities, the engine creates a TLG, applies the two rules from Section 3.2, and outputs new *document-to-document* traceability links.

For comparison, we augmented the *ILRE* with another step to perform IR-only textual analysis on the requirements to suggest links in a direct, more classic approach. For this purpose, the *ILRE* takes a second threshold and uses the similarity matrix previously provided, though this last step does not affect and is not affected by the TLG or the links suggested by the

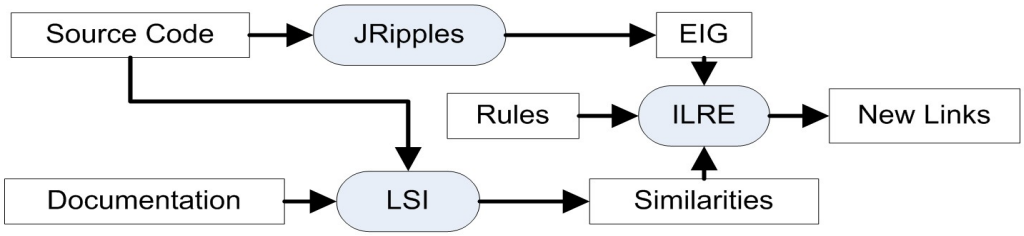

Figure 2: An overview of our approach

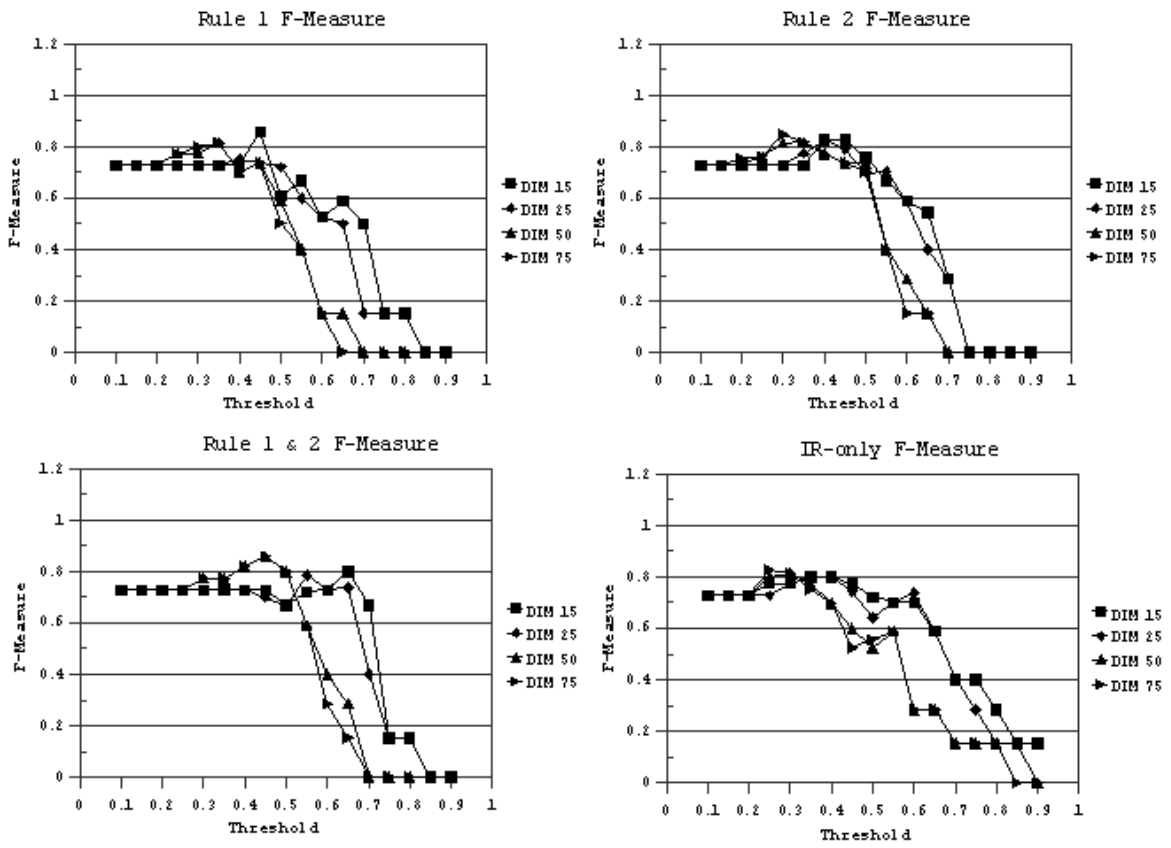

Figure 3: Comparison of F-measures for different configurations of the combined (rule 1, 2, and 1&2) and IR-based approaches

indirect method. In other words, the LSI tool runs once for the entire corpus (all requirements and all Java methods), and that information is used twice: one time combined with the EIG for our indirect approach and again alone for the direct, IR-only method.

In order to evaluate our approach, we also needed a set of requirements documentation traceability links recovered manually to serve as an ideal set for quantitative evaluation. We asked three graduate students unfamiliar with the CoffeeMaker code or project to answer the question for each pair of requirements A and B, "*Do you expect the implementation of A to overlap with that of B?"* They then filled in a matrix with a zero, one, or two, for no, some, or much overlap expected between the two respective requirements. We combined their answers into an ideal set by voting.

The subject of our study is a small, education-friendly software system provided by the Repository for Open Software Education[2] called CoffeeMaker. Totaling around 1 KLOC, this project has seven user requirements implemented in 136 Java methods, the matching of which is somewhat intuitive since it emulates the functionality of a typical coffee maker.

We keep two independent variables. First, the threshold for choosing the links given by the IR tool, and second, the dimensionality reduction factors used during the IR-based link recovery process. In brief, we execute the *ILRE* for each of four reduction factors: 15, 25, 50, and 75. For each reduction factor, we calculate the *precision*, *recall*, and *f-measure* for the thresholds from 0.1 to 0.9 in increments of 0.05 against the ideal set derived from the questionnaire.

The performance results of using the *ILRE* and IR-based approaches are presented in Figure 3.

4.1. Discussion

We wish to address the following research questions through our analysis of the results from the case study:

[2] http://agile.csc.ncsu.edu/rose/ (verified on 1/17/09)

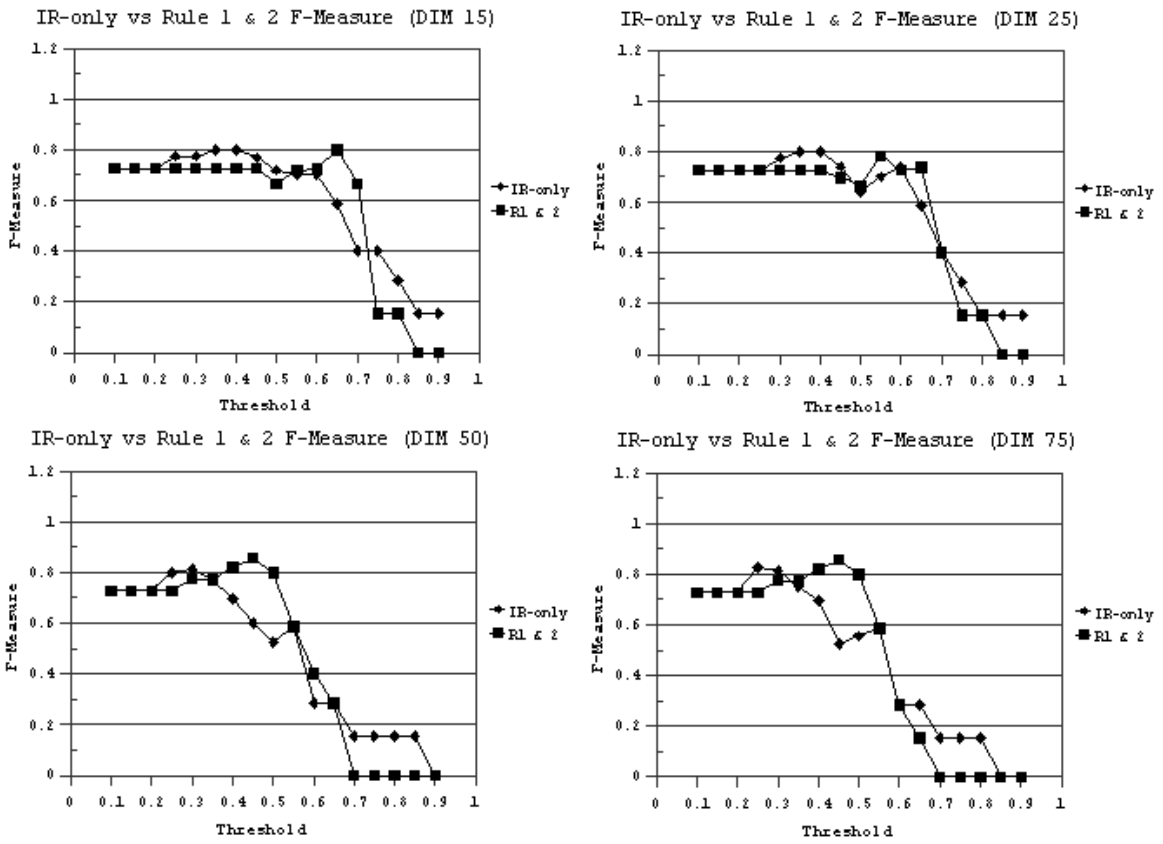

Figure 4: Comparison of F-measures for the combined and IR-based approaches

- **Can combining structural and textual analyses improve the established IR-based approach to *document-to-document* traceability link recovery?** In order to address this research question, we compare the textual-only approach to our *ILRE* method in the contexts in which each of the techniques performs best. In other words, we want to look at the peak performance of IR-only and *ILRE* combined across all thresholds and all dimensionality reduction factors. Figure 4 shows top combined rule performance at a threshold of 0.45 and reduction factor of 75. While the IR-based approach is relatively insensitive to threshold in the lower reduction factors, its peak performance is at a threshold of 0.25 and reduction factor of 75, but still beneath our approach. However, these results are not conclusive. More studies on larger systems are necessary to finalize the conclusions.
- **Do our rules perform better together or individually?** In almost all cases, our rules complemented one another rather than being redundant (see Figure 3 and Figure 4). For example, our peak F-measure was highest when the rules were mixed. We would not see any improvements if they returned the same suggestions.
- **How do our rules achieve higher f-measures at the various dimensionality reduction factors?** As the dimensionality reduction factors decrease for the SVD done during IR, the affects of LSI's solution to the synonymy problem are exaggerated. This is because the space into which the term matrix is being reduced drops in size, meaning that more terms will appear related. That is, more terms will look like synonyms (since they will appear to occur together). Since the number of unique terms in the source code is likely to be much larger than the number in the requirements for our study subject (136 Java methods versus 7 user requirements), this conclusion is not unfair.
- **How are precision and recall affected by the various thresholds?** Figure 5 shows the precision and recall rates for our combined rules as well as the IR-only results. We expect that we will obtain fewer suggested links as we tighten the link selection criterion (that is, raise the IR threshold), and this is what we observe in the results. This conclusion is

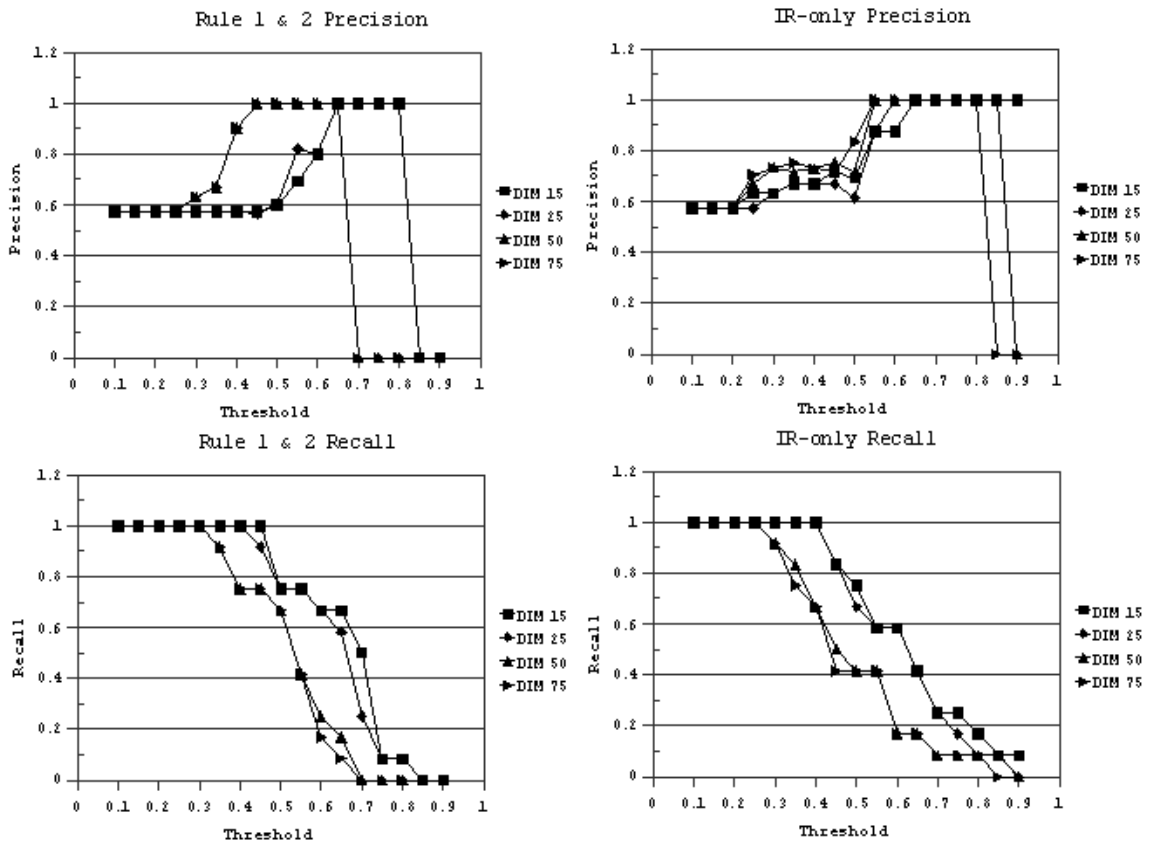

Figure 5: Comparison of Precision and Recall measures for the combined and IR-based approaches

not controversial. As we become more selective about the textual similarities we allow to be declared as links in our graph, we see increased precision in our list of suggestions as well as reduced recall.

4.2. Example

To illustrate the potential advantages of combining structural and textual information, consider the following two requirements from CoffeeMaker.

Requirement A (R_A): Waiting State. When the CoffeeMaker is not in use, it waits for user input. There are six different user input options: 1) add recipe, 2) delete a recipe, 3) edit a recipe, 4) add inventory, 5) check inventory, and 6) purchase beverage.

Requirement B (R_B): Add Inventory. Inventory may be added to the machine at any time. The types of inventory in the Coffee Maker are coffee, milk, sugar, and chocolate. The inventory is measured in integer units. No inventory may be taken away from the CoffeeMaker except by purchasing a beverage. Upon completion, a status message is printed and the CoffeeMaker is returned to the waiting state.

All three respondents to the questionnaire expected the implementation of R_A to have some degree of overlap with that of R_B. Additionally, the Use Case diagram[3] provided by the developers of CoffeeMaker includes a link between these two requirements. During our case study, however, the LSI tool found textual similarities for R_A and R_B between 0.38 and 0.43 for the various dimensionality reduction factors. At thresholds low enough to catch this link, however, precision rates drop noticeably. Our second rule finds this link (and others like it) at much higher IR thresholds, though at lower recall rates. Links between R_A and several others indicated in the Use Case diagram are those detected *first* (that is, at thresholds resulting in high precision and low recall) by Rule 2, but *last* by Rule 1 or the IR-only approach.

5. Threats to Validity

The most important obstacle to generalizing our results is the rather small scope of our input data. CoffeeMaker is an excellent primer for calibrating our approach, however it may or may not be representative of real-world software to provide a basis for widely-applicable conclusions. We need to study our approach on multiple datasets but under similar conditions.

[3] http://open.ncsu.edu/se/tutorials/coffee_maker/ (verified on 1/23/09)

We attempted to limit internal threats to validity. The three requirements readers were not familiar with the code, which could negatively affect the ideal set to which we compared our results. They also had varying levels of understanding about this project as well as related knowledge. The question they were asked was specific to this project rather than derived from use-cases or other documentation artifacts.

6. Related work

Our approach builds on top of two well-studied areas of traceability recovery: textual analysis using IR and structural analyses. We present prior work on traceability link recovery using IR-based, structural, and combined methods.

6.1. Information retrieval methods

Among the earliest attempts to find connections between documentation and source code is work by Antoniol et al [1]. They use a vector space model with reasonable success, which was expanded by Marcus and Maletic [21] using LSI in an attempt to mitigate the effects of synonymy and polysemy. Marcus et al. perform two case studies suggesting that LSI is more suitable to natural language analysis than the vector space model. RETRO [13] is a tool designed for requirements tracing that can employ several IR techniques, including LSI. Advanced summary work by De Lucia et al. [5] adds LSI-based link recovery to ADAMS, a software system artifact management tool. The goal of their modification is to aid software engineers in finding links among several types of software artifacts, rather than only *documentation-to-source*. Additionally, they study ADAMS use on a large group of student users and make several recommendations for IR implementation. Many tweaks to IR have been suggested, such as the *cut-point method*, in which the threshold is picked dynamically based on the IR output.

IR methods for link recovery are useful for matching regions of related documentation and source code, but they are not always suitable on their own for problems such as feature location which requires specific parts of documentation to be matched exactly to their implementation in code. A plethora of applications have been developed for these techniques [12, 22, 24, 28, 29]. Zhao et al. [29] combined knowledge from pseudo-execution traces with candidate links given by the IR vector space model using a branch-reserving call graph to create a static, non-interactive approach. Eaddy et al. [12] also create a hybrid system, combining information retrieval, execution tracing, and prune dependency analysis (PDA). PDA is a source code analysis technique to determine which parts of the source are related to one other section such that if the latter is removed, so can the former. Their tool, Cerberus, uses PDA to find source artifacts related to others found by IR methods. Natural language processing (NLP) has been used in ways similar to IR methods for link recovery. Find-Concept [28] is a tool that uses NLP to locate the source code programmers are looking for during maintenance.

6.2. Structural analysis methods

Our approach uses JRipples to obtain a list of source code links. This tool extracts artifact dependencies in Java for use in impact analysis − that is, what portions of the source code need to be changed alongside others when a developer wants to add new functionality. Impacted artifacts are labeled and returned to the user to facilitate navigation. Instead of this purpose, however, we use the *source-to-source* links in conjunction with *document-to-source* links for indirect link recovery.

Structural analysis can only be applied to source code, not documentation. Several approaches exist that attempt to link source code to a particular concept or feature. Chen and Rajlich [4] presented a study of feature location using dependence graphs by examining data and control flow in software. Robillard [26] developed an approach that automatically proposes and ranks methods that may be of interest to a programmer using fuzzy sets and the topology of a program.

6.3. Combined analysis methods

We are not the first to suggest attaching structural analysis to textual. Maletic and Marcus [20] use semantic and structural information to cluster components for program comprehension. As noted above, Cerberus feeds the results of both analyses to prune dependency analysis for feature location. Another system, Dora [14], uses a program's call graph to isolate a *neighborhood* of methods relevant to a given seed. Eaddy et al. suggest that these tools could be combined to fully automate the process by inputting to Dora the Cerberus' output. That is, a user's input of features would output the neighborhood of related methods. In our prior work [23], we used LSI to determine links in documentation, and coupling measures were used to find links in source code. Strong coupling should also mean high similarities among the corresponding sections of documentation. The approach can be used to find disparities between the two so that documentation can be updated to reflect the code's actual structure.

7. Conclusions

We combine textual and structural analyses of software artifacts, two widely-accepted techniques for recovering traceability links, and conduct a preliminary case study. The results indicate stable but rather minor improvements over a classic, single-pronged approach based on textual analysis only. Our evaluation covers a wide range of IR thresholds and dimensionality reduction factors, with regular and expected overall trends in performance. These results are promising as they imply that we can harvest additional information from the implementation of software requirements about how those requirements are related.

8. Acknowledgements

We thank Andrian Marcus for his valuable comments and discussions on earlier versions of this research. We also acknowledge Maksym Petrenko for his help with JRipples. This research was supported in part by the United States Air Force Office of Scientific Research under grant number FA9550-07-1-0030.

9. References

[1] Antoniol, G., Canfora, G., Casazza, G., De Lucia, A., and Merlo, E., "Recovering Traceability Links between Code and Documentation", *IEEE TSE*, vol. 28/10, pp. 970 - 983.

[2] Antoniol, G., Hayes, J. H., Gueheneuc, Y., and Di Penta, M., "Reuse or rewrite: Combining textual, static, and dynamic analyses to assess the cost of keeping a system up-to-date", in Proc. of IEEE ICSM'08, 2008, pp. 147-156.

[3] Buckner, J., Buchta, J., Petrenko, M., and Rajlich, V., "JRipples: A Tool for Program Comprehension during Incremental Change", in Proc. of IWPC'05, pp. 149-152.

[4] Chen, K. and Rajlich, V., "Case Study of Feature Location Using Dependence Graph", in Proc. IWPC'00, pp. 241-249.

[5] De Lucia, A., Fasano, F., Oliveto, R., and Tortora, G., "ADAMS Re-Trace: a Traceability Recovery Tool", in Proc. of 9th IEEE CSMR'05, 2005, pp. 34-41.

[6] De Lucia, A., Fasano, F., Oliveto, R., and Tortora, G., "Can Information Retrieval Techniques Effectively Support Traceability Link Recovery?" in ICPC'06, pp. 307-316.

[7] De Lucia, A., Fasano, F., Oliveto, R., and Tortora, G., "Recovering Traceability Links in Software Artefact Management Systems", *ACM TOSEM*, vol. 16, no. 4, 2007.

[8] De Lucia, A., Oliveto, R., and Sgueglia, P., "Incremental Approach and User Feedbacks: a Silver Bullet for Traceability Recovery", in Proc. of ICSM'06, pp. 299-309.

[9] De Lucia, A., Oliveto, R., and Tortora, G., "IR-Based Traceability Recovery Processes: An Empirical Comparison of "One-Shot" and Incremental Processes", in Proc. of 23rd IEEE/ACM ASE'08, pp. 39-48.

[10] Deerwester, S., Dumais, S. T., Furnas, G. W., Landauer, T. K., and Harshman, R., "Indexing by Latent Semantic Analysis", *Journal of the American Society for Information Science*, vol. 41, 1990, pp. 391-407.

[11] Duan, C. and Cleland-Huang, J., "Clustering Support for Automated Tracing", in Proc. of ASE'07, pp. 244-253.

[12] Eaddy, M., Aho, A. V., Antoniol, G., and Guéhéneuc, Y. G., "CERBERUS: Tracing Requirements to Source Code Using Information Retrieval, Dynamic Analysis, and Program Analysis", in Proc. of ICPC'08, pp. 53-62.

[13] Hayes, J. H., Dekhtyar, A., and Sundaram, S. K., "Advancing candidate link generation for requirements tracing: the study of methods", *IEEE TSE*, 32/1, pp. 4-19.

[14] Hill, E., Pollock, L., and Vijay-Shanker, K., "Exploring the Neighborhood with Dora to Expedite Software Maintenance", in Proc. of ASE'07, pp. 14-23.

[15] Jiang, H., Nguyen, T., Che, I. X., Jaygarl, H., and Chang, C., "Incremental Latent Semantic Indexing for Effective, Automatic Traceability Link Evolution Management", in Proc. of 23rd IEEE/ACM ASE'08, L'Aquila, Italy, 2008.

[16] Kuhn, A., Ducasse, S., and Gîrba, T., "Semantic Clustering: Identifying Topics in Source Code", *Information and Software Technology*, 49/3, March 2007.

[17] Landauer, T. K. and Dumais, S. T., "A Solution to Plato's Problem: The Latent Semantic Analysis Theory of the Acquisition, Induction, and Representation of Knowledge", *Psychological Review*, vol. 104, no. 2, 1997, pp. 211-240.

[18] Lin, J., Lin, C. C., Huang, J. C., Settimi, R., Amaya, J., Bedford, G., Berenbach, B., Khadra, O. B., Duan, C., and Zou, X., "Poirot: A Distributed Tool Supporting Enterprise-Wide Automated Traceability", in Proc. of 14th IEEE RE'06, 2006, pp. 363-364.

[19] Lormans, M. and Van Deursen, A., "Can LSI help Reconstructing Requirements Traceability in Design and Test?" in Proc. of 10th CSMR'06, pp. 47-56.

[20] Maletic, J. I. and Marcus, A., "Supporting Program Comprehension Using Semantic and Structural Information", in Proc. of ICSE'01, pp. 103-112.

[21] Marcus, A., Maletic, J. I., and Sergeyev, A., "Recovery of Traceability Links Between Software Documentation and Source Code", *IJSEKE*, 15/4, pp. 811-836.

[22] Poshyvanyk, D., Guéhéneuc, Y. G., Marcus, A., Antoniol, G., and Rajlich, V., "Feature Location using Probabilistic Ranking of Methods based on Execution Scenarios and IR", *IEEE TSE*, 33/6, June'07, pp. 420-432.

[23] Poshyvanyk, D. and Marcus, A., "Using Traceability Links to Assess and Maintain the Quality of Software Documentation", in Proc. of TEFSE'07, 2007, pp. 27-30.

[24] Poshyvanyk, D. and Marcus, D., "Combining Formal Concept Analysis with Information Retrieval for Concept Location in Source Code", in Proc. of ICPC'07, pp. 37-48.

[25] Rajlich, V. and Gosavi, P., "Incremental Change in OO Programming", in *IEEE Software*, 2004, pp. 2-9.

[26] Robillard, M., "Automatic Generation of Suggestions for Program Investigation", in Proc. of FSE'05, pp. 11 - 20

[27] Salton, G. and McGill, M., *Introduction to Modern Information Retrieval*, McGraw-Hill, 1983.

[28] Shepherd, D., Fry, Z., Gibson, E., Pollock, L., and Vijay-Shanker, K., "Using Natural Language Program Analysis to Locate and Understand Action-Oriented Concerns", in Proc. of AOSD'07, 2007, pp. 212-224.

[29] Zhao, W., Zhang, L., Liu, Y., Sun, J., and Yang, F., "SNIAFL: Towards a Static Non-interactive Approach to Feature Location", *ACM TOSEM*, 15/2, 2006, pp. 195-226.