Tracing Requirements to Defect Reports:

An Application of Information Retrieval Techniques

SURESH YADLA

JANE HUFFMAN HAYES

ALEX DEKHTYAR

ABSTRACT

To support debugging, maintenance, verification and validation (V&V) and/or Independent V&V (IV&V), it is necessary to understand the relationship between defect reports and their related artifacts. For example, one cannot correct a code-related defect report wihtout being able to find the code that is affected. Information Retrieval (IR) techniques have been used effectively to trace textual artifacts to each other. This has generally been applied to the problem of dynamically generating a trace between artifacts in the software document hierarchy „after the fact“ (after development has proceeded to at least the next lifecycle phase). The same techniques can also be used to trace textual artifacts of the software engineering lifecycle to defect reports. We have applied the term frequency- inverse document frequency (TF-IDF) technique with relevance feedback, as implemented in our tool RETRO (REquirements TRacing On-target), to the problem of tracing textual requirement elements to related textual defect reports. We have evaluated the technique using a dataset for a NASA scientific instrument. We found that recall of over 85% and precision of 69% and recall of 70% and precision of 99% could be achieved, respectively, on two subsets of the dataset.

1. Introduction

Establishing the relationship between defect reports and software development artifacts is an important task. Imagine a defect report that documents a code fault or defect. A software engineer needs to find the corect locations in the source code requiring repair, based on information in the defect report. After the engineer has corrected the code, the engineer needs to identify the test cases that were executed previously on the modified code. Again, the engineer must „search“ for the related test cases. Each of these „search“ tasks can be made less difficult and more accurate by using information retrieval (IR) techniques [1, 2, 3].

Our current work focuses on tracing unstructured textual artifacts such as requirements or design elements found in requirements or design specifications to defect reports[1]. In prior work, we have studied the problem of tracing textual requirement elements to other textual requirement elements, as well as tracing textual requirement elements to textual design elements. To measure the effectiveness of IR techniques, we developed a tool in C++ and Java called RETRO (REquirements TRacing On-target). The tool implements a number of IR techniques such as term frequency-inverse document frequency (TF-IDF) [4] and latent semantic indexing (LSI) [5] together with standard Rochio relevance feedback [4] to facilitate dialog with the analyst. We evaluate the techniques using recall (coverage measure – were all the relevant elements found?), precision (signal-to-noise ratio – were only relevant items found?), as well as a number of secondary measures that we have developed [6]. We used a dataset for a NASA scientific instrument [7] to validate our techniques. We found that recall exceeds 85% and precision is at 69% for one of the two subsets of the dataset and is at 70% and 99%, respectively, for the other data subset (very good results).

The paper is organized as follows. Section 2 introduces the IR techniques and measures, and the RETRO tool. Section 3 describes our approach as well as related work. Section 4 describes the validation of our work. Section 5 presents conclusions and future work.

2. Information Retrieval Techniques for Tracing

In this section, we discuss the TF-IDF method of computing similarity between documents and queries. TF-IDF is a standard Information Retrieval method. We also describe the standard Rochio relevance feedback method. We define some measures used to evaluate IR techniques for tracing. Finally, we describe our tool, RETRO.

---

[1] The techniques we use are such that one can just as easily trace in the other direction, from defect reports to textual artifacts, by making the defect reports the high level artifact and the unstructured text the low level artifact.

2.1 IR Techniques for Tracing Defect reports

In information retrieval (IR), we have a document collection within which we are searching for some information described in a query, also called an information request. In the common case of web surfing using a search engine, the „document collection" is the set of all webpages indexed by a search engine, and the „query" is the information that one types into the search box. In the case of tracing requirements to defect reports, the query may be the requirement and the document collection may be the set of all defect reports. Unlike web search, the „document collection" and „query" designations in IV&V tracing tasks are reversible: the query can be a given defect report and the document collection can be all the requirement elements.

Before moving into a discussion of specific techniques, let us first examine the overall process of interest. Suppose that one is tracing a requirement to all related defect reports. The general process is as follows. First, the requirement (query) is given to the tool along with the document collection (set of defect reports). Next, an IR method is executed. It generates a list of candidate links or candidate matches along with their percieved relevance scores (the higher the score, the more relevant, in the view of the method, is the connection between the requirement and the defect report). This information is then presented to the analyst for dispensation. At that point, the analyst can decide if matches are correct or not. Or, the analyst could instead examine a subset of the candidate links and provide feedback (specify whether the candidate match is correct or incorrect) to the tool. This feedback can then be used to modify information about the query and/or document and the IR method can be re-executed (thus becoming an iterative process). We refer to this as the analyst feedback loop. At some point of this process, the analyst can finalize the list of matches. Let us first examine the IR method in use, TF-IDF.

2.1.1 TF-IDF

The most commonly used IR technique is the Vector space retrieval with TF-IDF term weighting[4]. In this method, the query and the document are representated in the form of vectors of keyword weights. The keywords, or index terms, are words that occur in the document collection (some words like „the", „where", etc. are called stopwords and

are not considered to be index terms. The lists of stopwords are standardized in IR applications; all words that are not stopwords are considered to be index terms). Given the vectors for a document and a query, their similarity is given by the cosine of the angle between the vectors. More formally, the definition of TF-IDF (vector model) is given as, Given a document vector $dj = (w_1,...,w_N)$ and a query vector $q=(q_1,...,q_N)$, the similarity $sim(dj,q)$ is computed as [4]:

$$sim(dj,q) = \cos(dj,q) = \frac{\sum_{i=1}^{N} w_i \cdot q_i}{\sqrt{\sum_{i=1}^{N} w_i^2 \cdot \sum_{i=1}^{N} q_i^2}}.$$

The weights w_i and q_i are computed as follows: $w_i\ (q_i) = f_i * log(M/m_i)$. Here the term f_i , called term frequency (tf), is the frequency of occurrence of the keyword k_i in dj or q (usually normalized by the maximal term frequency of the document/query). M is the number of documents in the document collection and m_i is the number of documents in which the index term k_i occurs. M/m_i is called the inverse document frequency (idf). Thus, the weight of a keyword in a document (query) is proportional to the frequency of its occurrence in the document (query) and inversely proportional to the logorithm of its frequency of occurrence in the entire document collection. The idf portion of the weight accounts for the fact that rare terms have higher discriminatory power.

2.1.2 Standard Rochio Relevance Feedback

As discussed above, feedback is an iterative process where an analyst vets the results provided by the tool. An analyst may examine the top candidate link for all defect reports, marking those that are relevant, and then start a second iteration. We refer to this as examining the „top 1“ candidate links. An analyst may examine the top 2 candidate links for every other defect report, and then start a third iteration. In general, the analyst has total flexibility on how much feedback to provide.

How does feedback work? When an analyst „marks“ a candidate link as relevant, the important terms from that document are selected and the weight of the terms in that document are increased. The query weights are then recomputed and the IR algorithm is run again iteratively until the desired results are produced.

Relevance feedback for the Vector space retrieval makes an assumption that relevant documents have similarities between them, i.e., the term weight vectors are similar. Consider a query q and a set of documents D returned by an IR method for q. Let D_r be the set of relevant documents from D as identified by the analyst and D_n be the set of non-relevant documents in D as identified by the analyst (note that neither set have to be complete, but they are disjoint). Let $|D_r|$ and $|D_n|$ be the sizes of the respective sets. RETRO uses the Standard Rochio feedback method for specifying the query vector q_{new} for the next iteration:

$$q_{new} = \alpha q + \left(\frac{\beta}{r} \sum_{d_j \in D_r} d_j \right) - \left(\frac{\gamma}{s} \sum_{d_k \in Dn} d_k \right) \text{[4].}$$

Here, α, β, γ are tuning constants signifying the importance of the old query vector, positive feedback and negative feedback respectively.

2.2 Evaluation Measures

This section presents the measures that we use to evaluate the performance of an IR method, such as TF-IDF. The two most commonly used evaluation measures in IR are *recall* and *precision*. Recall is the *fraction of the relevant documents that have been retrieved* [4]. High recall indicates that most of the relevant items have been retrieved. For recall, we consider results above 80% excellent, above 70% - good, and between 60% and 70% - acceptable. Precision is *the fraction of the retrieved documents that are relevant* [4]. High precision indicates that most of the retrieved items are relevant. For precision, 20-30% is acceptable, 30-50% is good, and 50% and above – excellent. Our preference for higher recall stems from the observation, that it is much easier for an

analyst to exclude an incorrect candidate link from the trace (i.e., overcome low precision) than it is to discover a true link that had not been returned (i.e., overcome low recall). In addition, we note that a change from 10% to 20% in precision means that instead of 1 true link in 10, the trace contains 1 true link in 5, a savings of about 50% in terms of the number of links to verify for the analyst. We also note that in general, we strive to achieve good/excellent performance (as defined above) on both precision and recall, as one can always achieve perfect recall by retrieving the document collection in its entirety for each query at the price of miserable precision.

In addition to these measures, we have introduced a number of secondary measures for examining how well IR methods work in tracing software engineering artifacts to each other*: lag, diffAR*, and *selectivity* [3, 6]. Lag is defined as *the average number of false links* (items retrieved that are not relevant) *that occur in a candidate link list that have higher relevance than a true link* (items retrieved that are relevant). This number should get smaller as iterations proceed.

DiffAR is the *difference between the average relevance of a false positive* (false link) *and a true link* in a given candidate link list. It is also called relevance separation. This number should increase as iterations proceed. Selectivity examines the number of comparisons that an analyst potentially must make to perform the tracing task. If given 200 requirements (N) to trace to 1000 defect reports (M), the analyst potentially must examine N x M or 200,000 pairs. Each requirement needs to be compared to each defect report. The IR method produces a list of candidate links or matches. Selectivity of the method is defined as:

$$Selectivity = \frac{\sum_{i=1}^{M} n_i}{M \cdot N}$$

where n_i is the number of candidate links returned for a given requirement p_i. Selectivity measures the savings incurred by the analyst manually going through the

list generated by an automated method rather than manually comparing each pair of elements [6]. The smaller the selectivity, the better the savings for the analyst.

In Figure 1 the solid represents the set of relevant items and the hashed set represents the set of items retrieved by the tool. In Figure 2, the items with only solid shading are relevant items that are not retrieved, the items common to both the sets are the relevant items that are retrieved. The items with only hashing are irrelevant items retrieved.

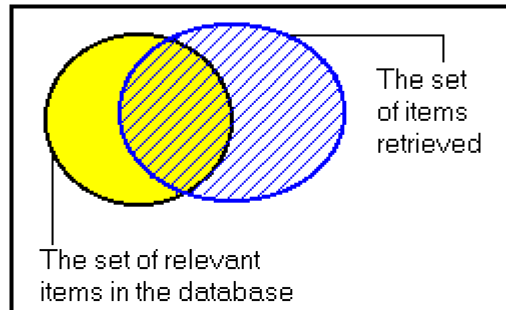

Figure 1 [8]. Items Retrieved.

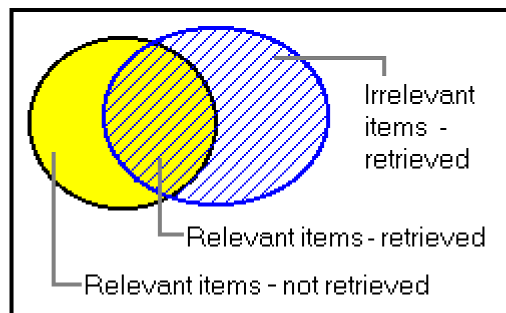

Figure 2 [8]. Relevant Items.

Recall is the measure of the relevant documents retrieved, shown in Figure 3.

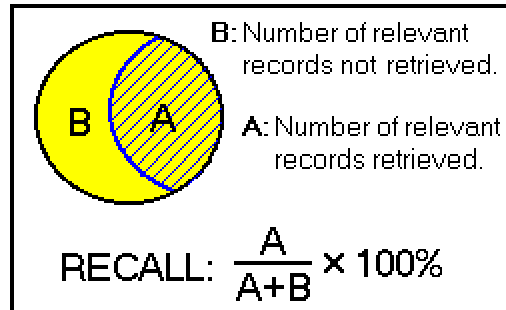

Figure 3 [8]. Recall.

Precision is the measure of the number of irrelevant items retrieved, shown in Figure 4.

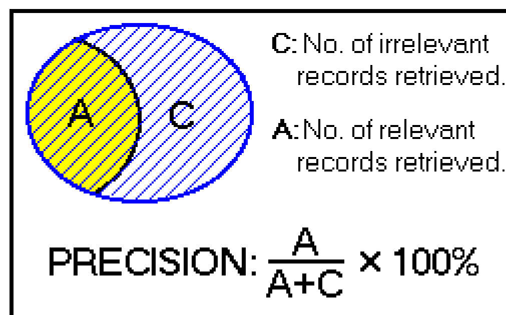

Figure 4 [8]. Precision.

2.3 REquirements TRacing On-target (RETRO)

RETRO has been designed exclusively for requirements tracing. It can be used as a standalone tool to discover traceability matrices. It can also be used in conjunction with other project management software: the requirements tracing information is exported in a simple, easy-to-parse eXtensible Markup Language (XML) form. The overall look of the RETRO GUI (WinXP port) is shown in Figure 5 [3].

At the heart of RETRO lies the IR toolbox (C++): a collection of implementations of IR methods, adapted for the purposes of the requirements tracing task. Methods from this toolbox are accessed from the GUI (Java) to parse and analyze the incoming requirements documents and construct relevance judgments. The Filtering/Analysis component (C++) of RETRO takes in the list of candidate links constructed by any of the

toolbox methods and prepares a list to be shown to the analyst. This preparation may involve the application of some cleaning, filtering, and other techniques. The GUI of RETRO guides the entire requirements tracing process, from setting up a specific project, to going through the candidate link lists. At the top of the screen, the analyst sees the list of high level requirements (left) and the list of current candidate links for the selected requirement, with relevance judgments (right). In the middle part of the interface, the text of the current pair of high and low level items is displayed. At the bottom, there are controls allowing the analyst to make a decision on whether the candidate link under consideration is, indeed, a true link. This information is accumulated and, upon analyst request, is fed into the feedback processing module (C++). The module takes the results of analyst decisions and updates the discovery process consistent with the changes. If needed, the IR method is re-run and the requirements tracing process proceeds into the next iteration [3].

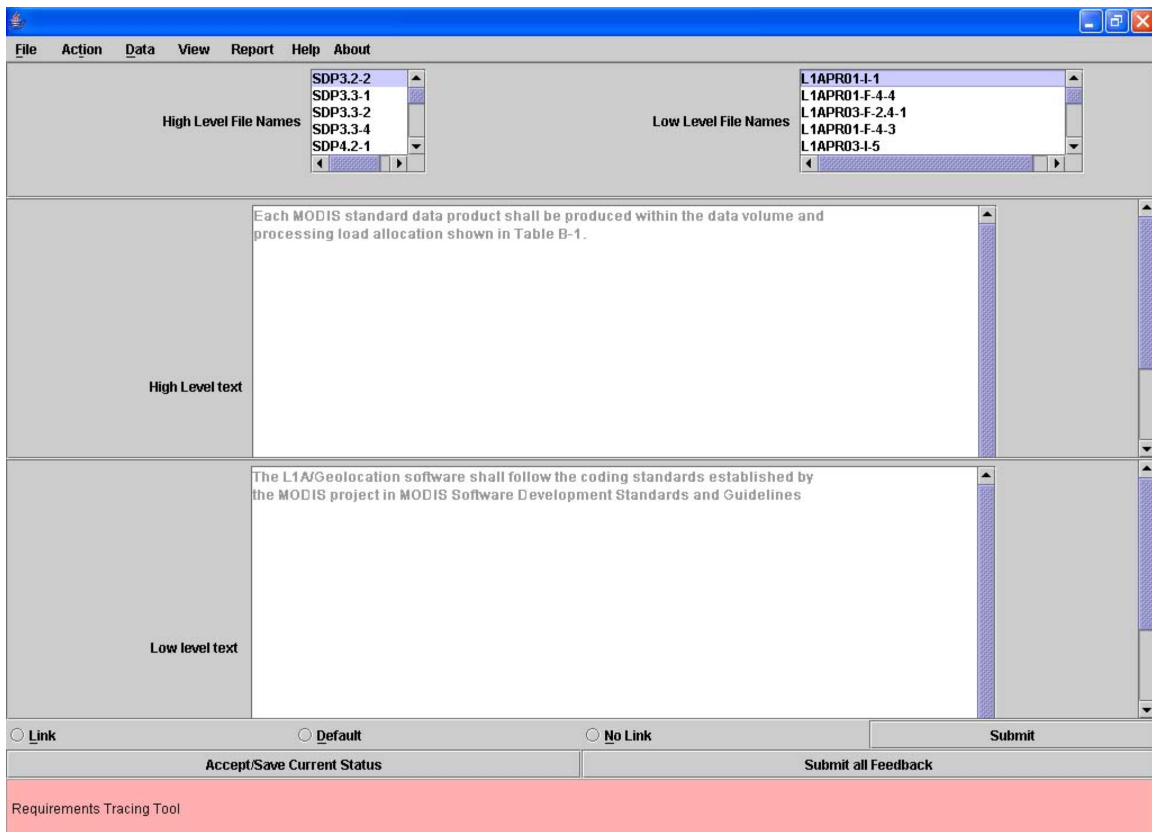

Figure 5. Screenshot of RETRO tool.

3. Tracing Requirements to Defect Reports

In this section, we describe our approach for tracing requirements to defect reports as well as related work.

3.2 Approach to Tracing

We use the RETRO tool to trace requirements to defect reports. First, we get the requirements and defect reports into the format required by RETRO. Each requirement is stored in a separate text file. The same is true for defect reports. In general, it has been our experience that requirements can be found in spreadsheets or in specifications identified with unique identifiers that lend themselves easily to parsing. We have been able to extract these items using simple macros in commercial word processing or spreadsheet applications. For defect reports, we wrote a small, special-purpose parser to extract the individual defect reports from a larger file of reports.

We assume that a defect report is of the following form:

ECR ID: *15-03691-rtos-1*
Title: *GHS linker 1.8.9 incompat with 1.8.7C*
Other Ref #: *
Closed: *Y*
Type (Bug/Chg/Enh): *Bug*
Priority (H/M/L): *L*
Estimated Time:
Actual Time:

Date Reported: *Thu Jul 13 10:59:50 CDT 2000*
Reported By: *
Time Spent:
Found in Release: *1.8.9*
Problem:

Version 1.8.9 lx creates a different symbol format for globaly declared

variables. When 1.8.9 compiled code was loaded on the scu simulator running GHS 1.8.7C compiled vxWorks build, the vxWorks Loader, (ld) failed to dynamically resolve references to some global variables resulting in Data Storage Interrupt exceptions.

Suspected Cause:

ELF file format must be different between the two versions of GHS lx. Re-compiling under 1.8.7C fixed the problem.

Date: *Wed Aug 29 14:36:10 CDT 2001*
Comment By: *
Time Spent:
Comment:

*Spoke with GHS customer support. They stated that the version of Tornado (1.0.1) used with * is built with the GNU tools that include a linker whose output is compatible with GHS (lx) linker. The output format is the standard ELF format. Although WRS recommends 1.8.8 GHS compiler, GHS assured me that 1.8.9 will work correctly.*

*Another conflict that falls into this category is characterized by using the -g option with *. If the -g option is used, the symbol table of the stand alone vxWorks build become corrupted with the extra symbolic information inserted for the debugger.*

Fix Date:
Fixed By:
Time Spent:
Cause:

Solution:

> ** * DPU has been successfully using GHS 1.8.9 toolchain since August 2000, with no evident problems. It is no longer important to maintain binary backward compatibility with *.*

Fixed In Release:
Docs Affected:

Files Affected:

Approval Date:
Approved By:
Comment:

The tool searches for the ECR ID of a defect report. The ECR ID is kept as the name of the file and only the problem field is written into the file. This process repeats until all defect reports have been written to separate files (in the directory that the user specifies). Figure 6 shows a screenshot of the tool prompting the user for the input file name and the destination directory.

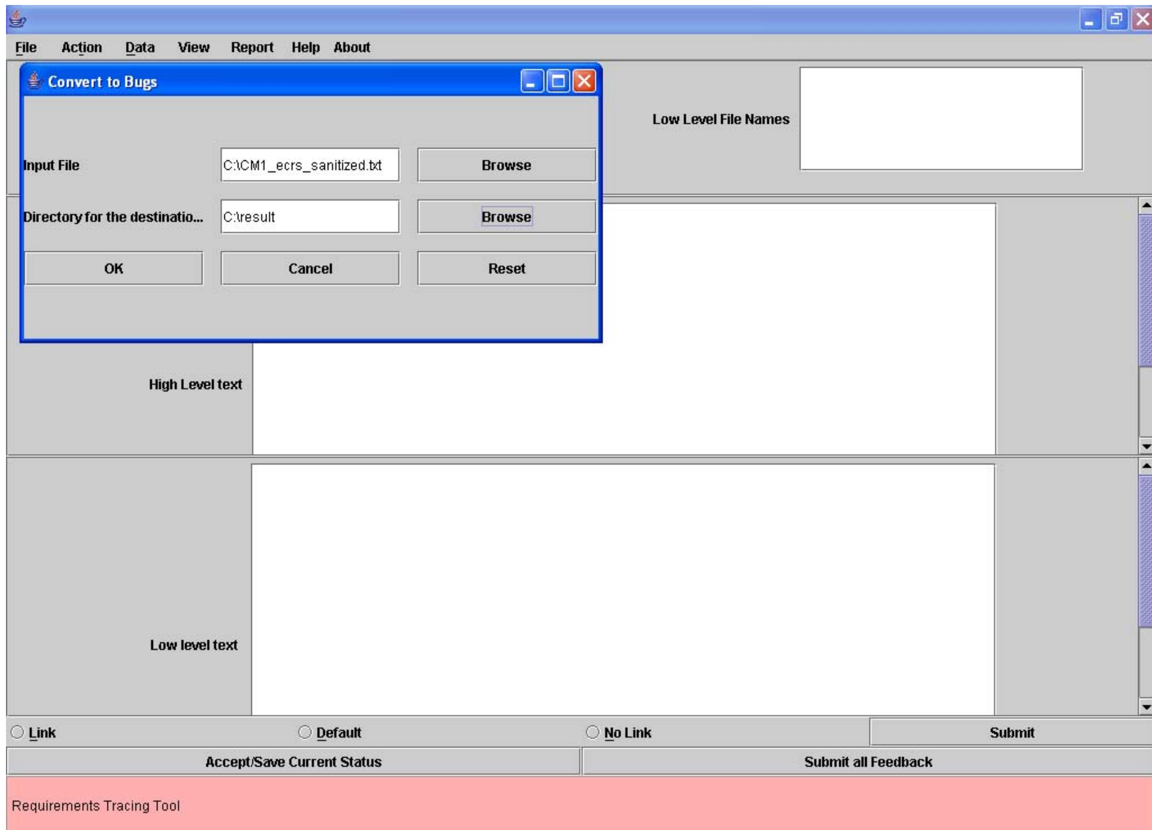

Figure 6. Extracting defect reports for RETRO.

After the user extracts the defect reports (and requirements), RETRO can be used for tracing. The user selects the high level „document collection" from which to trace (in this case, requirements) and then the low level „document collection" (defect reports, in this case) to which to trace. The user may then select an IR method (such as TF-IDF) and whether or not to use a feedback method. Next, RETRO executes the selected method on the two document levels and returns the results. The requirements are shown in the upper left listbox (see Figure 5). When one of the requirements is clicked, then the following two actions occur:

The text of the requirement is shown in the upper TextArea; and

Related defect reports are shown in the upper right listbox.

If a defect report is clicked, then the report text is shown in the lower TextArea.

3.2 Related Work

Recently, a number of researchers investigated the use of IR methods for tracing. Antoniol, Canfora, De Lucia and Merlo [9] considered two IR methods, probabilistic IR and vector retrieval (tf-idf) in studying the traceability of requirements to code for two datasets. They traced C++ source code onto manual pages and traced Java code to functional requirements. They examined the effect of requiring 100% recall and found that the probabilistic model achieves the highest recall values, less than 100 percent, with a smaller number of documents retrieved and then performs better when 100% recall is required. Following them, Marcus and Maletic [10] applied latent semantic indexing (LSI) technique to the same problem. They have shown that LSI methods show consistent improvement in precision and recall and were able to achieve combinations of 93.5% recall and 54% precision for one of the datasets.

While those papers studied requirements-to-code traceability, in [2] we have addressed the problem of tracing requirements between different documents in the project document hierarchy [3]. Examples of such „different documents“ are requirements specifications being traced to design specifications. For example, we were able to achieve 90.2% recall with 77.1% precision (with 0.15 filtering and Top 2 feedback) for a small NASA dataset tracing high level requirements to lower level requirements. We also achieved 61.2% recall and 40.9% precision (same filtering and feedback as above) for the complete CM-1 dataset (described in Section 4), tracing textual requirements to textual design [11].

There has been a large amount of research on the classification of defect reports or problem reports [12 - 19]. In [12] and [17], a taxonomy of defect types is developed for each phase of the lifecycle. Also, the concept of a defect trigger is introdouced. This is a condition that allows a defect to surface, such as “bug fix.” In [13], a taxonomy of defects is developed along with a set of causes for the defects (root causes). The work in [14] develops a taxonomy of requirement faults as well as a method for tailoring a generic taxonomy to a given class of projects or to a specific project. Specification-based testing and its ability to detect certain fault clases is examined in [15]. Von Mayrhauser et al., in [16], examine defect histories to derive fault architectures. In [18], a classification scheme is presented for program faults and associated root causes for

requirements errors in safety critical, embedded systems. In [19], Munson and Nikora examine what constitutes a fault. In general, [12 – 19] have focused on developing taxonomies of fault types and automatically or semi-automatically categorizing defect reports. Our work differs from this in that we are tracing requirements to defect reports (we can also perform the reverse, tracing from defect reports to requirements).

4. Validation

In order to evaluate the efectiveness of the RETRO tool for the tracing of requirements to defect reports, we undertook an experiment using a NASA dataset, CM-1. CM-1 is a scientific instrument developed by NASA. The Metrics Data Program (MDP) [7] provided the dataset. The many artifacts in the dataset have been sanitized to preserve the anonymity of the instrument. Of interest to us for this study were the complete requirements document and a set of defect reports. A typical requirement is one to two sentences in length. A typical defect report was shown above in Section 3.

The CM-1 dataset provided individual extracted requirements, but we had to use our special-purpose tool to extract the defect reports. Next, *we traced the requirements to the defect reports with the assistance of RETRO and* ***manually verified the traces to build the "ground truth" for the experiment*** – the set of correct links between the requirements and selected defect reports (further called the ***answerset***). We wrote an analysis tool that compares the answerset to the candidate link lists returned by RETRO and calculates primary and secondary measures. We built two datasets from CM-1. We discuss the results for each below.

4.1 Small CM-1 Data Subset

The first, smaller, dataset contains 10 requirements, 58 defect reports, and 14 true links (in the answerset). We used TF-IDF and feedback for eight iterations. On each iteration, we *simulated* the perfect analyst feedback by correctly identifying, for each requirement, the top two *unvisited* items in its candidate link list. We call this strategy

Top 2. Figure 7 plots Recall and Precision for Top 2 feedback with eight iterations. You can see that recall exceeds 85% and precision reaches 69%.

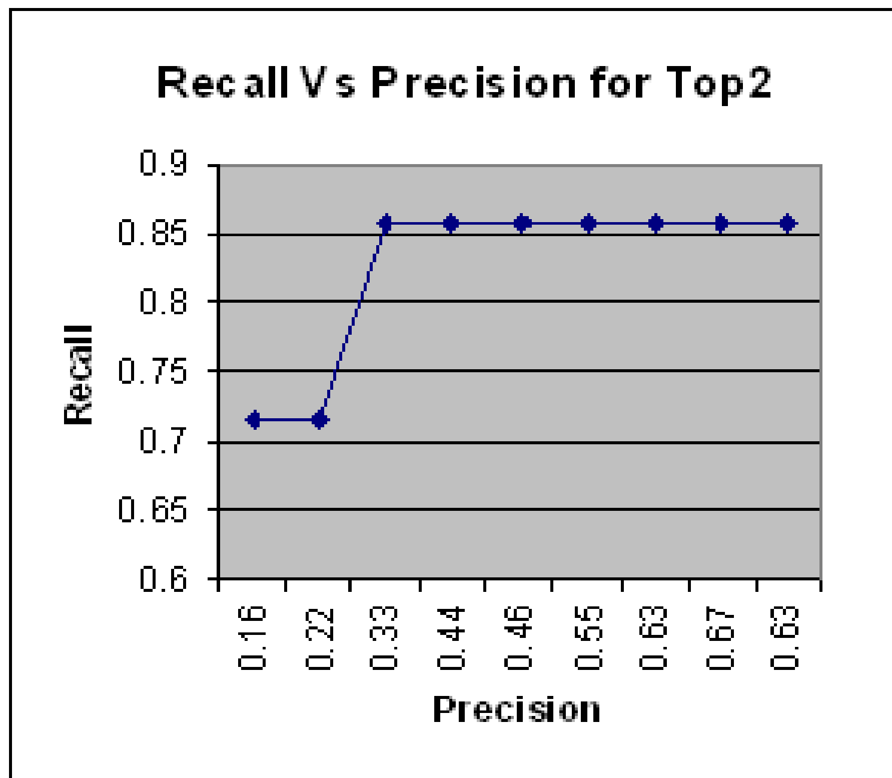

Figure 7. Recall vs. Precision for Top 2 for Smaller CM-1 Subset.

Figure 8 graphs Selectivity against Iteration for top 2 feedback for eight iterations with a filter of 0.1 (any items retrieved with relevance of lower than 0.1 have been ignored in calculations). Selectivity starts at around 0.13, meaning that even on iteration 0, RETRO retrieves only around 13% of all possible candidate links. As iterations proceed, selectivity decreases (as desired), stabilizing at about the 4% level. Figure 9 graphs Lag against Iteration for top 2 with eight iterations and a filter of 0.1. Lag starts relatively low, at about 2.7, and drops immediately to about 1.2 on the first iteration. It continues decreasing as iteration increases (as desired), reaching 0 (meaning there are no false links above true links in the candidate link lists) at iteration 6.

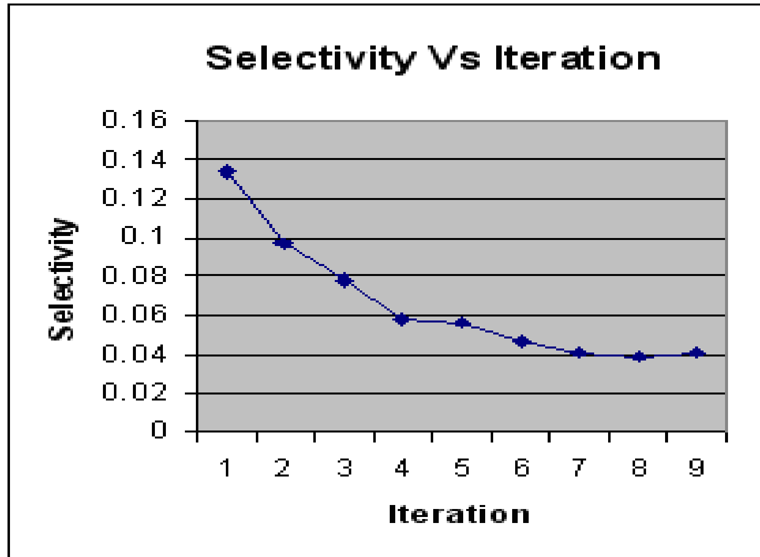

Figure 8. Selectivity vs. Iteration for Top 2 for Smaller CM-1 Subset.

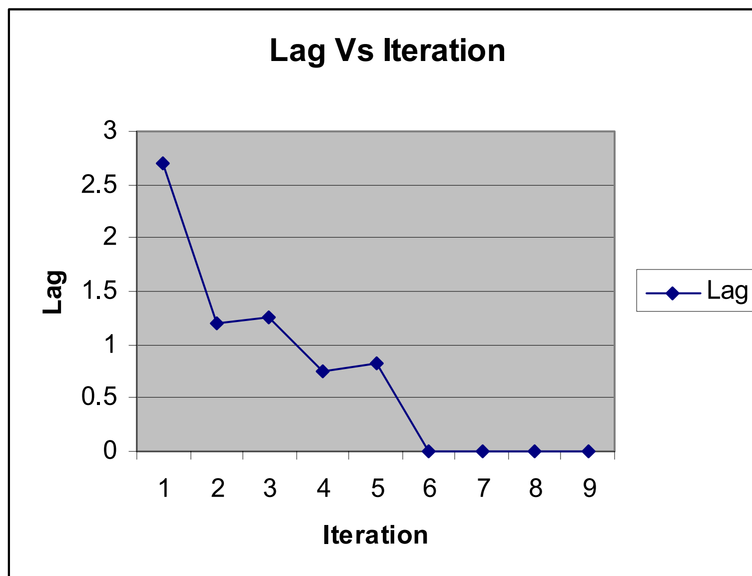

Figure 9. Lag vs. Iteration for Top 2 and Filter of 0.1 for Smaller CM-1 Subset.

Figure 10 graphs DiffAR (Relevance separation) and Iteration for top 2 with eight iterations and a filter of 0.1. Note that DiffAR grows as iteration proceeds, as desired. We note that because of relevance feedback, term weights can become large, and the

relevance can exceed 1. This explains why the value of DiffAR exceeds 1 on later iterations.

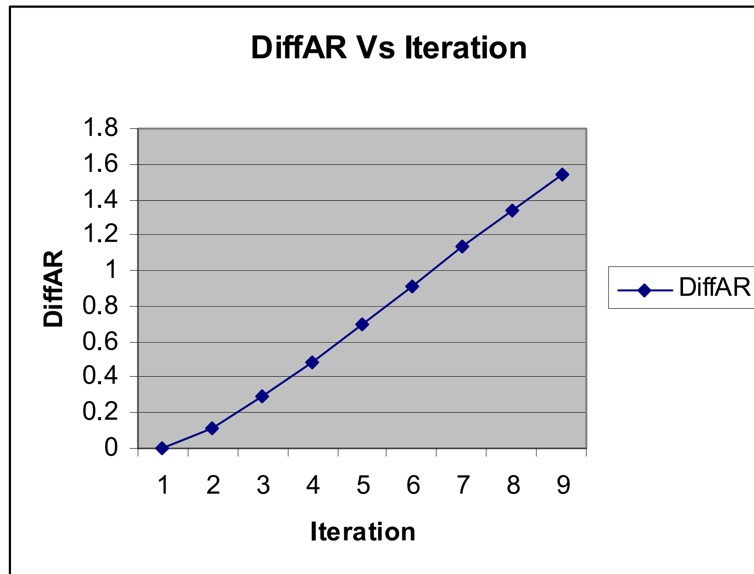

Figure 10. DiffAR vs. Iteration for Top 2 and Filter of 0.1 for Smaller CM-1 Subset.

4.2 Larger CM-1 Data Subset

The second, larger, dataset contains 20 requirements, 98 bug reports, and 44 true links. Figure 11 graphs recall against precision for Top 2 feedback. Note a clear difference between this and the smaller dataset. In the smaller dataset, RETRO retrieved about 71% of correct links on iteration 0 with precision of about 16%. In the larger dataset, initial recall is very poor – only 20%, with precision also 20%. But while the feedback process applied in the first case lead to a significant increase only in precision (recall increased to 85% quickly and stabilized) for the first dataset, for the second dataset **both recall and precision improve drastically**. By iteration 8, recall reaches 70% and precision reaches almost 100%.

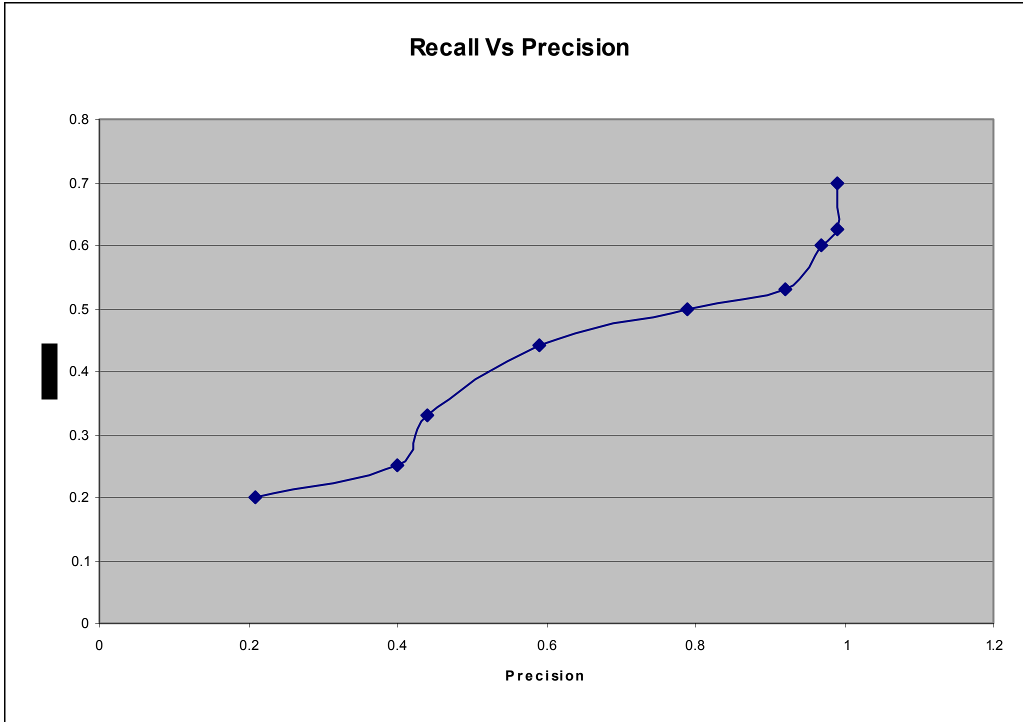

Figure 11. Recall vs. Precision for Top 2 for Larger CM-1 Subset.

Figure 12 graphs Selectivity against iteration for Top 2 feedback. At iteration 0, selectivity is just over 2.5%, and it oscillates between 1.5% and 2% throughout the experiment. Figure 13 graphs Lag against iteration for Top 4 behavior. Lag starts very low, at around 0.6, meaning that even with no feedback, RETRO achieves excellent separation: there are very few false positives with higher relevance than true links in the output. By iteration 3, lag drops all the way down to 0. Figure 14 graphs DiffAR against iteration for Top 2 feedback. DiffAR grows steadily over time.

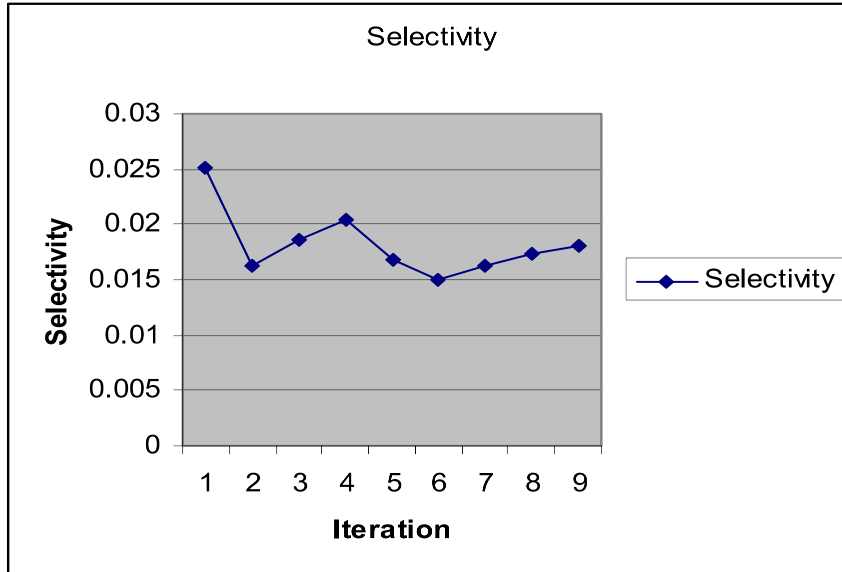

Figure 12. Selectivity vs. Iteration for Top 2 for Larger CM-1 Subset.

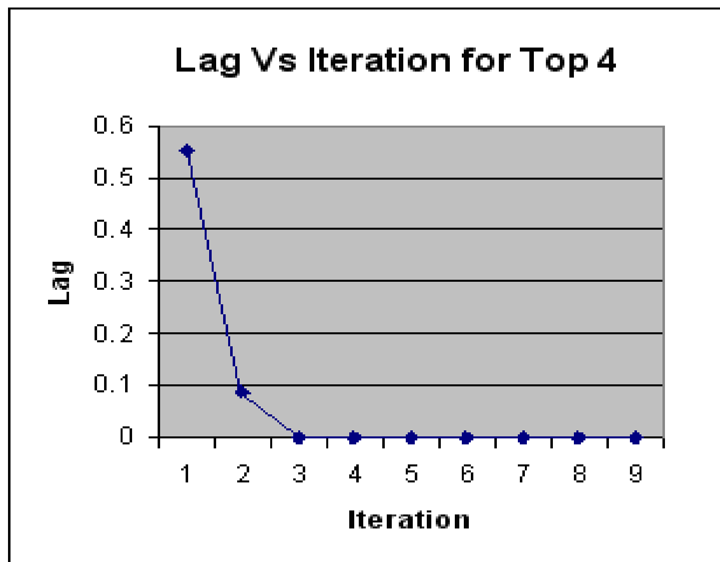

Figure 13.. Lag vs. Iteration for Top 4 for Larger CM-1 Subset.

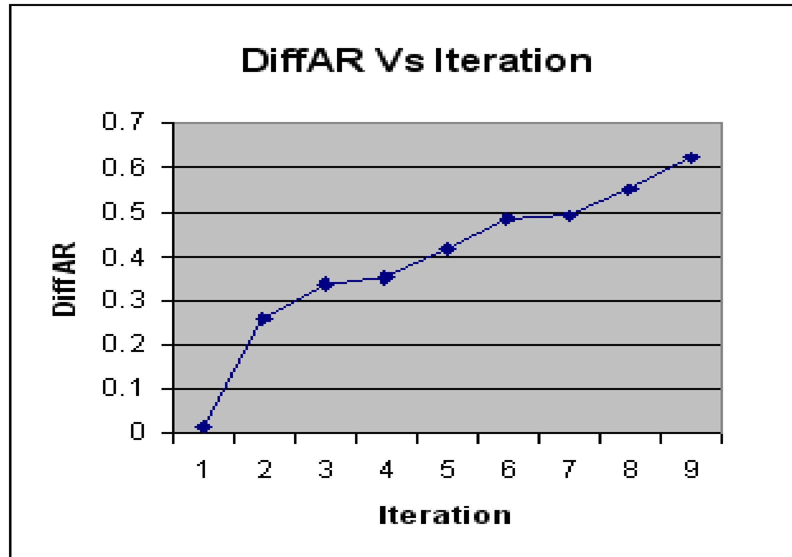

Figure 14. DiffAR vs. Iteration for Top 2 for Larger CM-1 Subset.

4.3 Analysis

Comparing the performance of RETRO on the two datasets we observe the following:

- *Information Retrieval techniques are applicable to tracing requirements to defect reports for different datasets*. In our experiments, the two datasets exhibited distinctly different properties, and RETRO showed different levels of initial success in capturing the trace. However, eventually for both datasets we were able to achieve high recall and high precision.
- *Analyst feedback is essential*. In the second dataset, the majority of correct links were retrieved on subsequent iterations after user feedback had been processed. Also, feedback significantly improved precision in both datasets. This suggests that while there might be defect reports that are not similar textually to the requirements to which they trace, such reports happen to be similar to other defect reports that trace to the same requirement. So, as long as at least one defect report is correctly captured by the IR method outright for a requirement, it appears that it is possible to retrieve many more true links via feedback. Similarly, irrelevant defect reports are efficiently purged by negative feedback.

- *RETRO is capable of assisting analysts with building accurate requirements-to-defect reports traces*. The stress here is on „assisting." RETRO takes care of the tedious part of the tracing task: coming up with the candidate link lists. The results obtained in our experiments (where a „perfect" analyst was simulated) show that the accuracy of candidate link lists improves significantly with the amount of feedback. We also have seen that even if initial results (like for the second dataset) were not encouraging, the feedback process can find the majority of missing links at later iterations. Our secondary measure, lag, shows that from the very beginning, candidate link lists consist of mostly true links with higher relevance than the retrieved false positives, a highly desired feature of candidate link lists.

5. Conclusions and Future Work

This research combines two recently evolving Software Engineering directions: the push for automating the Software Engineering process, and the use of Data Mining (in the broad sense of this word, which includes IR) techniques for analyzing Software Engineering data. The majority of research on mining in Software Engineering concentrates on prediction: using the data from a completed project, or projects, to predict the behavior of some future projects. On the other hand, our goal in applying IR techniques to tracing here and in our requirements tracing work [1,2,3] has been improvement of the process, automation of the tedious, boring, and time-consuming tasks that are currently performed manually or in a semi-automatic fashion.

In large projects, determination of the origin of a defect often is a non-trivial task. Because defect reports typically come in bulk, the time an analyst can spend on each individual defect report is limited. Our approach: automated tracing of requirements to defect reports allows the analysts: (a) to address this problem in bulk as well, and (b) to spend their time on this problem in a much more productive fashion. At the present time, RETRO is being field-tested by our colleages in industry on a number of large scale projects. While current field and usability tests of RETRO involve requirements tracing in the context of Independent Verification and Validation (IV&V) tasks, we intend to test

its performance on tracing requirements to defect reports (and vice versa) in a similar setting at a later time (a more user-friendly version 2.0 of RETRO is currently under development).

In addition to this, our future work includes tracing between bug reports. This could be useful to detect potential duplicated or overlapping defect reports AS they are being entered into a configuration management or bug tracking system. That is to say, we can take the text of a defect report that is being entered (but has not yet been „stored“) and „trace“ it to the collection of already existing defect reports. We can return potentially relevant defect reports to the user and ask if they still want to continue and add the new defect report to the database (or if they instead want to add detail to an already exisitng report). Also, this capability could assist software engineers who are modifying code to address defect reports. They can retrieve all related defect reports and correct them „en masse.“

Acknowledgements

Our work is funded by NASA under grant NAG5-11732. We thank Stephanie Ferguson and Ken McGill. We thank Mike Chapman and the MDP program. We thank Senthil Sundaram for his assistance with RETRO. We thank Ganapathy Chidambaram for his assistance on LSI. We also thank Sarah Howard and James Osborne who worked on early versions of RETRO.

References

1. Dekhtyar, A., Hayes, J. Huffman, and Menzies, T. Text is Software Too, Proceedings of the International Workshop on Mining of Software Repositories (MSR) 2004, Edinborough, Scotland, May 2004, pp. 22 - 27.

2. Hayes, J. Huffman, Dekhtyar, A., Osbourne, J. Improving Requirements Tracing via Information Retrieval, in Proceedings of the International Conference on Requirements Engineering (RE), Monterey, California, September 2003, pp. 151 - 161.

3. Hayes, J. Huffman, Dekhtyar, A., Sundaram K.S., Howard S., Helping Analysts Trace Requirements: An Objective Look, in Proceedings of the International Conference on Requirements Engineering (RE), Kyoto, Japan, September 2004.

4. Baeza-Yates, R. and Ribeiro-Neto, B. *Modern Information Retrieval,* Addison-Wesley, 1999.

5. Deerwester, S., Dumais, S.T., Furnas, G.W., Landauer, T.K. and Harshman R., Indexing By Latent Semantic Analysis, Journal of the American Society of Information Science, 1990.

6. Hayes, J. Huffman, Dekhtyar, A., and Sundaram, K.S. Measuring the Effectiveness of Retrieval Techniques in Requirements Tracing. 2005. In submission.

7. MDP Website, CM-1 Project, http://mdp.ivv.nasa.gov/mdp_glossary.html#CM1.

8. Measuring Search Effectiveness, http://www.hsl.creighton.edu/hsl/Searching/Recall-Precision.html, last accessed 8 April 2005.

9. Antoniol, G., Canfora, G., Casazza, G., De Lucia, A., and Merlo, E. Recovering Traceability Links between Code and Documentation. IEEE Transactions on Software Engineering, Volume 28, No. 10, October 2002, 970-983.

10. Marcus, A.; Maletic, J. Recovering Documentation-to-Source Code Traceability Links using Latent Semantic Indexing, Proceedings of the Twenty-Fifth International Conference on Software Engineering 2003, Portland, Oregon, 3 – 10 May 2003, pp. 125 – 135.

11. Sundaram, S., Hayes, J. Huffman, and Dekhtyar, A. Baselines in Requirements Tracing. To appear in Proceedings of the International Workshop on Predictor Models in Software Engineering (PROMISE 2005), in conjunction with the International Conference on Softare Engineering, St. Louis, MO, May 2005.

12. Chillarege, R., Kao, W., and Condit, R.G. Defect type and its impact on the growth curve, in Proceedings of the 13th international conference on Software engineering (ICSE 1991), pp. 246—255, Austin, Texas, United States.

13. Leszak, M., Perry, D.E., and Stoll, D. A case study in root cause defect analysis, in Proceedings of the 22nd international conference on Software engineering (ICSE 2000), pp. 428—437, Limerick, Ireland.

14. Hayes, J. Huffman, Building a Requirement Fault Taxonomy: Experiences from a NASA Verification and Validation Research Project, in Proceedings of the IEEE International Symposium on Software Reliability Engineering (ISSRE), Denver, CO, November 2003, pp. 49 - 59.

15. Kuhn, D.R. Fault classes and error detection capability of specification-based testing. ACM Transactions on Software Engineering and Methodology (TOSEM) Volume 8 , Issue 4 (October 1999).

16. von Mayrhauser, A., J. Wang, M.C. Ohlsson and C. Wohlin, Deriving a Fault Architecture from Defect History, Proceedings of the International Symposium on Software Reliability Engineering, ISSRE99, pp. 295-303, November 1999, Boca Raton, Florida, USA.

17. Chillarege, R., Bhandafi, I., Chaar, J., Halliday, M., Moebus, D., Ray, B., and Wong, M. Orthogonal Defect Classification A Concept for In-Process Measurements, IEEE TSE, vol. 18, no. 11 (Nov. 1992), pp. 943-956.

18. Lutz, R. Analyzing Software Requirements Errors in Safety-Critical, Embedded Systems, Proceedings of the First IEEE International Symposium on Requirements Engineering, January 1993, 126-133.

19. Munson, J.C., Nikora, A.P. Toward a quantifiable definition of software faults. 13th International Symposium on Software Reliability Engineering, 2002, p. 388 –395.