Interactive Recovery of Requirements Traceability Links Using User Feedback and Configuration Management Logs

Ryosuke Tsuchiya[1], Hironori Washizaki[1], Yoshiaki Fukazawa[1], Keishi Oshima[2], and Ryota Mibe[2]

[1] Dept. Computer Science of Waseda University, Tokyo, Japan
ryousuke_t@asagi.waseda.jp, {washizaki, fukazawa}@waseda.jp
[2] Yokohama Research Laboratory of Hitachi, Ltd., Kanagawa, Japan
{keishi.oshima.rj, ryota.mibe.mu}@hitachi.com

Abstract. Traceability links can assist in software maintenance tasks. There are some automatic traceability recovery methods. Most of them are similarity-based methods recovering links by comparing representation similarity between requirements and code. They cannot work well if there are some links independent of the representation similarity. Herein to cover weakness of them and improve the accuracy of recovery, we propose a method that extends the similarity-based method using two elemental techniques: a log-based traceability recovery method using the configuration management log and a link recommendation from user feedback. These techniques are independent of the representation similarity between requirements and code. As a result of applying our method to a large enterprise system, we successfully improved both recall and precision by more than a 20 percent point in comparison with singly applying the similarity-based method (recall: 60.2% to 80.4%, precision: 41.1% to 64.8%).

Keywords: traceability, configuration management log, interactive method

1 Introduction

Traceability of software development is defined as the ability to trace the relationships between software artifacts. We call these relationships “traceability links.” Here we focus on links between requirements and source code files, which are called “requirements traceability links.” For example, if there are the requirement “Recover links automatically” and the source code file “LinkRecover.java” implementing the requirement, a requirements traceability link exists between them.

Grasping requirements traceability links is effective in several software maintenance tasks, especially for improving the modification efficiency for change requests and understanding the source code [1,2,12]. For example, traceability links allow an engineer to effortlessly identify source code files that need to be modified upon receiving a change request.

adfa, p. 1, 2011.
© Springer-Verlag Berlin Heidelberg 2011

Because software must be analyzed to identify and extract traceability links, if the size of the target software is large, it is difficult to recover requirements traceability links manually due to the massive number of combinations between requirements and source code files. Consequently, some methods to automatically recover requirements traceability links have been developed [2,3,4,5,6,17,18,19,20]. Most of them are similarity-based methods recovering links by comparing representation similarity between requirements and source code. They can recover links with high accuracy if target software is within the applicable range. However, they cannot work well if there are some links independent of the representation similarity. To confirm the effectiveness for actual software products, we applied a typical similarity-based method to a large enterprise system developed by a company, and the method recovered links with a recall of 60.2% and a precision of 41.1%. This accuracy is unsuitable for practical use.

To cover weakness of the similarity-based method and improve the application effect, herein we propose a method that extends the similarity-based method using two elemental techniques. The first technique is a log-based traceability recovery method using the configuration management log to compensate for the lack of information about the relationships between the requirements and the source code. The second technique is the "link recommendation" using user feedback which is results of validation for recovered links. This process is not an additional burden for the users because validation of links is an inevitable and ordinary cost. These techniques are independent of the representation similarity between requirements and source code.

We applied our refined method to the abovementioned enterprise system to evaluate the improvement in recall and precision. This system has more than 80KLOC. We recovered traceability links between 192 requirements and 694 source code files. The system has known 726 correct links. In this study, we evaluate recall and precision by comparing the known correct links to the links recovered by our refined method. This study addresses the following Research Questions.

- RQ1: How accurately can we recover links by the similarity-based method?
- RQ2: How much does the addition of the log-based method improve the recovery accuracy?
- RQ3: How much does the addition of link recommendations improve the recovery accuracy?

We answered these questions by conducting evaluation experiments. We recovered links with a recall of 80.4% and a precision of 64.8%, which is more than a 20 percent point improvement in both recall and precision (recall: 60.2% to 80.4%, precision: 41.1% to 64.8%). In this accuracy, users can recover 80% of the correct links if they validate about 1.3 links for each source code file compared to validating over 4 links using only the similarity-based method. Although our method uses user feedback, it will eventually require less effort of the user. The contributions of this study are:

- We propose a traceability recovery method that extends the similarity-based method by incorporating two elemental techniques.
- We develop a prototype interactive tool to implement our refined method.

- We validate our refined method by comparative experiments with the similarity-based method using sufficiently large software.

The remainder of the paper is organized as follows. Section 2 provides background information. Section 3 describes our method, while section 4 evaluates our method. Section 5 discusses related works. Finally, section 6 provides a conclusion and future direction.

2 Background

2.1 Similarity-Based Method

To recover links automatically, most of previous methods compare the representation similarity between requirements and source code because related documents often share a lot of same words. We call them "similarity-based method." In the other words, if a requirement has a similar representation as the source code file, they are related. Therefore, we can recover links by calculating similarity between requirements and source code.

Several techniques have been proposed to calculate the representation similarity between documents. A typical example is the vector space model proposed by Salton et al. [13]. In this model, a sentence is represented by one vector that depends on the valid terms in the sentence. The contents of the sentence are determined by the direction of the vector.

When this method compares the representation between requirements and source code files, terms are extracted from each artifact. For the requirements, terms are extracted from the requirement names or the requirement specification documents. For source code files, terms are extracted from the identifiers (e.g., the name of file, class, method, and field) and source code comments. Consequently, the effectiveness of this method depends on these extracted terms, and it performs poorly in some scenarios. For example, in non-English speaking countries, engineers often use their native language in documents and source code comments to facilitate communications. If requirements are written in a non-English language, only the source code comments can be used to compare the representations. Moreover, if there are too few comments, the similarity cannot be calculated. On the other hand, even if requirements are written in English, this method does not work well when the identifier lacks meaningful terms (e.g., using an extremely shortened form).

2.2 Log-Based Method

As mentioned above, the similarity-based method cannot accurately recover links for some software. To recover links in such cases, we previously proposed a log-based traceability recovery method using the configuration management log [11]. The configuration management log contains information about modifications of software artifacts. We mainly considered the log of version control system such as Subversion [14] or Git [15], which is composed of revisions that include messages and file paths

(Figure 1). Hypothesizing that revision messages contain information about requirements, we designed a traceability recovery method using the configuration management log as an intermediary. Because revision messages, requirements and source code comments are often written in an engineer's native language, we can recover the links of software using a non-English language.

Although this method is effective for software using a configuration management log, it cannot be used singly because it cannot recover links with source code files that have no revision histories in the management log. To resolve such weakness, herein we combine the similarity-based method with the log-based method.

2.3 Inevitable Validation Cost of the Candidate Links

Users must validate the recovered candidate links because they may contain incorrect links (false positives) or overlook links (false negatives). This cost is inevitable unless the method recovers links with perfect accuracy. Therefore, this study effectively employs the results of user's validation.

2.4 Call Relationships

A study by Ghabi et al. [10] confirmed that "Code elements sharing call relationships are often linked by the same requirement." The code element represents elements that comprise the source code (e.g., methods, classes and files). For example, in Figure 2, there is a high possibility that the method "ScoreCalculator.calulate()" is linked with the requirement "Recover links" because both the caller method "LinkRecover.recover()" and the callee method "Link.setRelevance()" are linked with the requirement.

In our approach, we use this finding along with user feedback, as described above, in a technique called "Link Recommendation," which is described in detail in section 3.

Revision: 1234
Author: Ryosuke
Date: 2014/11/15 17:35:13
Message: Modified the method recover() reflecting the change request for "Recover links"

Modified: /…/…/…/LinkRecover.java

Fig. 1. Example of a revision in the configuration management log

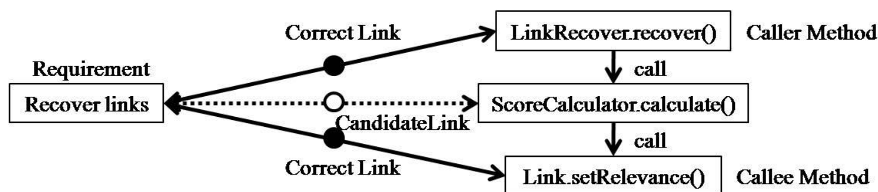

Fig. 2. Traceability links and call relationships

2.5 Motivating Examples

We have applied the typical similarity-based method using the vector space model to a large enterprise system, which was developed by a Japanese company. Hence, the requirement specification documents are written in Japanese. Although the source code comments are also written in Japanese, some source code files lack comments. Consequently, the method recovered links with a recall of 60.2% and a precision of 41.1%. These results motivated us to extend the similarity-based method.

After improving some problems of the log-based method proposed in our previous work, we applied it to the abovementioned enterprise system. As a result, we found that this method is superior to the similarity-based method under certain circumstances. When we limited the target source code files to those with sufficient revision histories in the configuration management log (the number of source code files decreases from 694 to 317), this method recovered links with a recall of 67.6% and a precision of 69.1%. The similarity-based method only recovered links with a recall of 46.3% and a precision of 47.3% in the same situation. The superiority of the log-based method in limited scenarios motivated us to combine the similarity-based method with the log-based method.

3 Approach

3.1 Overview

We propose a method to recover requirements traceability links. This method extends the similarity-based method using two elemental techniques. In this study, we calculate the similarity between documents by using the vector space model. Figure 3 shows the overview of our method. Our method requires three artifacts as inputs.

1. Requirements
   A list of requirement names is essential. In addition, we also use the requirement specification documents written in a natural language. In this study, we focus on requirements that are concrete and objective (i.e., software functional and non-functional requirements).
2. Source code files
   Because our method applies natural language processing and analyzes call relationships, we can apply it to source code languages that the above techniques are applicable for. The prototype tool for our method currently supports Java [16] (partly C and C++).
3. Revisions of the configuration management log
   We require the revision histories of the source code files. Our method mainly focuses on the log of the version control system. Our tool currently supports the logs of Subversion and Git. Prior to employing our method, unnecessary revisions, which indicate modification histories other than the source code files, are excluded. Moreover, revisions including simultaneous modification of too many source code files are excluded; that is, the tool excludes revisions involving over 10 source code files.

First, we create a document-term matrix (DTM) using the three input artifacts. Next, two kinds of relevance scores are calculated for each candidate link. The first score denotes the similarity score calculated by the similarity-based method. The second is the relevance score calculated by the log-based method. In this study, candidate links indicate all relationships between the target elements. For example, if the target system has 100 requirements, there are 100 candidate links for each source code file. However, the reliability as a score in the candidate links differs.

After calculating the score, users specify a target (requirement or source code file) that they want to recover links. Then our tool displays candidate links of the specified target after arranging the candidate links according to the sorting algorithm of our method. This algorithm sorts candidate links using two kinds of scores and user feedback. As shown in Figure 4, for example, if users specify the requirement "Recover links" as the target of the recovering links, the tool displays the sorted candidate links.

Users then validate candidate links presented by our tool starting from the top. They judge the correctness of the candidate links, and each time, the result is provided as feedback to the tool. Then the tool sorts the presentation order of the remaining candidate links according to the user feedback. In Figure 4, by validating the correctness of the first presented link, the presentation order is re-sorted.

Finally, after users validate the candidate links and identify some correct links, users can determine the requirements traceability matrix at an arbitrary time. The matrix shows the relationships between the requirements and source code files. Below our method is described in detail.

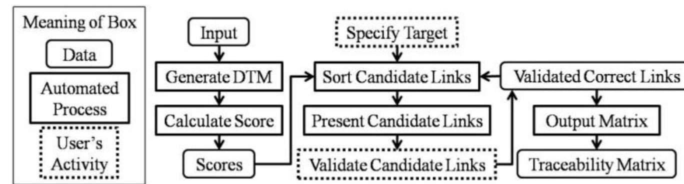

Fig. 3. Overview of our method

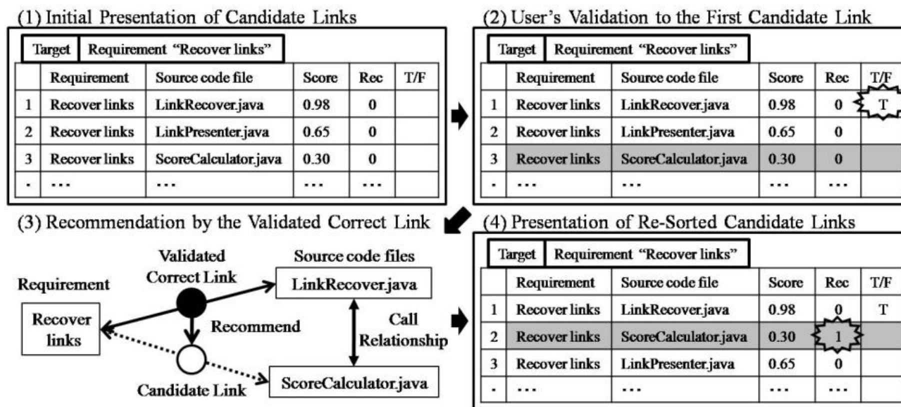

Fig. 4. Presentation and validation of sorted candidate links of a specified requirement

3.2 Document-Term Matrix Generation

Requirements, source code files and revisions are treated as documents in this approach. In the vector space model, each document is represented as a vector determined by valid terms (nouns, verbs, and adjectives) in the document. Terms of the requirement are extracted from the requirement name and the requirement specification document. Terms of the source code file are extracted from source code comments and identifiers. Then, if the identifier is represented as the connected term (e.g., LinkRecover, recover_Links()), the identifier is decomposed into individual terms. However, if requirements and source code comments are written in a non-English language, terms are not extracted from the identifiers. Terms of revisions are extracted from revision messages.

Here, D represents a set of documents and T presents a set of terms. For a document d_x ($\in D$) containing N valid terms [i.e., t_1, t_2, $\cdots$, t_N ($\in T$)], $w(t_p, d_x)$ $(1 \leq p \leq N)$ is the number of appearances of t_p in d_x. Consequently, d_x can be represented by N-dimensional vector $\overrightarrow{v_x}$ as

$$\overrightarrow{v_x} = \left(w(t_1, d_x), w(t_2, d_x), \cdots, w(t_N, d_x)\right). \quad (1)$$

In the vector space model, the vector of the document is represented as Formula (1). However, we use the vector weighted by TF-IDF to more accurately represent the document characteristics. TF-IDF indicates the term frequency and inverse document frequency. Frequently used terms in a document have high importance for the document. On the other hand, common terms used in many documents have low importance (e.g., general words). The term frequency value of t_p in d_x is defined as

$$tf(t_p, d_x) = \frac{w(t_p, d_x)}{\sum_N w(t_N, d_x)}. \quad (2)$$

If the number of documents is represented as M and the number of documents containing t_p is represented as $h(t_p)$, the inverse document frequency value of t_p is defined as

$$idf(t_p) = \log_e \frac{M}{h(t_p)}. \quad (3)$$

Therefore, the vector weighted by TF-IDF is defined as

$$\overrightarrow{v'_x} = \left(tf(t_1, d_x) * idf(t_1), tf(t_2, d_x) * idf(t_2), \cdots, tf(t_N, d_x) * idf(t_N)\right). \quad (4)$$

The similarity between two documents d_i and d_j is obtained as the cosine of the angle between the two document vectors $\overrightarrow{v'_i}$ and $\overrightarrow{v'_j}$, and is referred to as the cosine similarity. $DSim(d_i, d_j)$ (Document Similarity, $0 \leq DSim \leq 1.0$) is defined using the cosine similarity as

$$DSim(d_i, d_j) = \frac{\overrightarrow{v'_i}\overrightarrow{v'_j}}{\left|\overrightarrow{v'_i}\right| \left|\overrightarrow{v'_j}\right|}. \quad (5)$$

To calculate the similarity between all documents (containing requirements, source code files and revisions), a document-term matrix $DTM_{M\times N}$ with M rows and N columns is generated. Here M represents the number of documents and N represents the total number of terms in the documents. The matrix is defined as

$$DTM_{M\times N} = \begin{pmatrix} tf(t_1, d_1) * idf(t_1) & \cdots & tf(t_N, d_1) * idf(t_N) \\ \vdots & \ddots & \vdots \\ tf(t_1, d_M) * idf(t_1) & \cdots & tf(t_N, d_M) * idf(t_N) \end{pmatrix}. \quad (6)$$

The row of the matrix indicates the document vector mentioned in Formula (4). We calculate similarities between all documents using the document-term matrix.

3.3 Similarity-Based Score

In this approach, we calculate two kinds of relevance scores for each candidate link. First, we describe the first score in this section. In accordance with similarity-based methods, we directly calculate the similarity score between requirements and source code. We call this first score the "similarity-based score." Basically, $DSim$ is set to the similarity-based score. However, if the source code comments contain a requirement name, we set the maximized score (i.e., 1.0) to the similarity-based score.

Here, R represents a set of requirements and C is a set of source code files ($R, C \subset D$). The similarity-based score $SBScore$ between requirement r_i and source code file c_j ($r_i \in R$, $c_j \in C$) is defined as

$$SBScore(r_i, c_j) = \begin{cases} 1.0 \text{ (if comments of } c_j \text{ contain the name of } r_i) \\ DSim(r_i, c_j) \text{ (in other cases)} \end{cases}. \quad (7)$$

3.4 Calculating the Log-Based Score

Additionally, we calculate the second relevance score using the log-based method. We call the second score "log-based score." Because revisions contain file paths of the modified source code files, we can indirectly associate requirements with source code files by calculating the similarity between requirements and revisions. The log-based score is calculated by two elements: the similarity between requirements and revisions and the weight of the source code files for each revision.

Here L represents a set of revisions ($L \subset D$). The number of source code files modified in revision l_k ($l_k \in L$) is represented as $m(l_k)$. Then the weight of source code files c_j in revision l_k is defined as

$$Weight(c_j, l_k) = \begin{cases} \dfrac{1}{m(l_k)} \text{ (if } c_j \text{ is modified in } l_k) \\ 0 \text{ (in other cases)} \end{cases}. \quad (8)$$

Therefore, the log-based score $LBScore$ between requirement r_i and source code file c_j is defined as

$$LBScore(r_i, c_j) = \sum_{k=1}^{G} \left(DSim(r_i, l_k) * Weight(c_j, l_k) \right). \quad (9)$$

G represents the number of revisions. We calculate $DSim$ and $Weight$ for each revision and sum up their multiplied values.

3.5 Sorting Algorithms

After calculating scores, we present candidate links of the target specified by users. Here the target is either the requirement or source code file. Candidate links are sorted by our algorithm. The kind of target determines which algorithm is used.

If the target is a source code file, we selectively use two kinds of score depending on the presence of the revision history. For a file without revision histories in the configuration management log (i.e., all log-based scores are 0), the candidate links are sorted in descending order of the similarity-based score. On the other hand, for file with a revision history, the links are sorted in descending order of the log-based score.

If the target is a requirement, we basically use the similarity-based score. First, candidate links are sorted in descending order of the similarity-based score. However, on the other hand, candidate links get preferential rights if they have the highest log-based score in the group of candidate links when targeting any source code files. Candidate links with preferential rights are prior to links without the rights. (i.e., even if the similarity-based score is low, the link with the preferential right is preferentially presented to users.) Then candidate links with preferential rights are sorted in descending order of the log-based score.

3.6 Link Recommendations

Our tool presents sorted candidate links to users, and then the users validate the links starting from the top. Because their judgments are provided as feedbacks to the tool, our tool focuses on the call relationships of the source code file of the judged link. Here the call relationships of the source code file indicate the relationships of methods in the file. A validated correct link recommends other candidate links based on call relationships. We call this type of recommendation a “Link Recommendation.”

In addition to the two types of relevance scores, candidate links have other two values: “recommendation count by caller” and “recommendation count by callee.” These values increase when recommended by a correct link. For example, in Figure 2, if the link between the requirement “Recover links” and the source code file “LinkRecover.java” is judged as correct, the link recommends a candidate link between the requirement “Recover links” and the source code file “RelevanceCalculator.java” because the method “recover()” in the file “LinkRecover.java” is the caller of the method “calculate()” in the file “RelevanceCalculator.java.” Then the value “recommendation count by caller” of the candidate link increases by one. Likewise, if a link with the file “Link.java” is judged as a correct link, the value “recommendation count by callee” of the candidate link increases by one.

3.7 Sorting Algorithm with User Feedback

Every time the candidate link is judged as correct, the remaining candidate links are sorted by the appropriate algorithm that extends the algorithms described in section 3.5 by two values about the link recommendation.

First, the links are sorted in descending order of the value that multiplies "recommendation count by caller" and "recommendation count by callee." Second, the links with the same multiplied value are arranged in descending order of the value obtained by adding two values about the link recommendation. Third, the links with the same addition are sorted by the algorithm mentioned in the section 3.5.

As mentioned in the section 3.5, for a file with a revision history, the links are sorted in descending order of the log-based score. However, if a candidate link that has not been recommended is judged as incorrect, the kind of relevance scores that is used to sort is changed. This occurs whenever a link is judged as incorrect.

4 Evaluation

4.1 Overview

To validate our method, we carried out experiments targeting an enterprise system developed by a Japanese company. Although this system has a very large scale, its subsystem has 726 known correct links. Hence, the following experiments target this subsystem, which has more than 80KLOC. We recovered traceability links between 192 requirements and 694 source code files where the requirements are extracted from the requirement specification documents. Source code files are implemented by Java. We use 7090 revisions of the Subversion log. Requirements, source code comments and revision messages are written in Japanese.

To evaluate the improvement in recall and precision, we conducted three experiments. First, we recovered links by using only the similarity-based method. Second, we recovered links by using the method combining the similarity-based method and the log-based method. Third, we conducted an experiment evaluating the effectiveness of the link recommendation.

For each experiment, we recovered links by repeating the following cycle.

1. Specify a target (requirement or source code file).
2. Validate candidate links of the specified target starting from the top.
3. Validation of the target is complete when the validation count reaches the allowable validation count or all correct links of the target are recovered

Here the allowable validation count indicates how many candidate links users can validate in one cycle. For example, if the allowable validation count is one, users validate only the first presented link. The cycle is repeated as many times as the number of targets. Therefore, we repeated the cycle 192 times when targeting all requirements and also repeated 694 times when targeting all source code files. The recovery targeting requirements is independent of the recovery targeting source code files. Therefore, we can determine the recall and precision for both targeting requirements

and targeting source code files. Recall, precision and F-measure (comprehensive measure of recall and precision) are defined as

$$recall = \frac{validated\ correct\ links}{all\ correct\ links}\ (0 \leq recall \leq 1.0), \tag{10}$$

$$precision = \frac{validated\ correct\ links}{all\ validated\ links}\ (0 \leq precision \leq 1.0)\,, \tag{11}$$

$$F - measure = 2 * \frac{recall * precision}{recall + precision}\ (0 \leq F - measure \leq 1.0)\,. \tag{12}$$

Because the correct links are known, all experiments were fully automated. When targeting requirements, we specified the requirement as the target in alphabetical order of the requirement name. However, the target order affects the accuracy when the targets are source code files and the link recommendation is used. Therefore, when targeting source code files in experiments, we tried using two kinds of orders: the best order (descending order of the highest relevance score of the candidate links) and the worst order (ascending order of the highest relevance score of the candidate links).

Figure 5 shows the experimental results for the recall and precision for each target where the horizontal axis indicates the allowable validation count and the vertical axis indicates the value of recall or precision. Table 1 lists the results with the highest F-measure for each method. In all methods, the highest F-measure occurs when the targets are source code files and the allowable validation count is two.

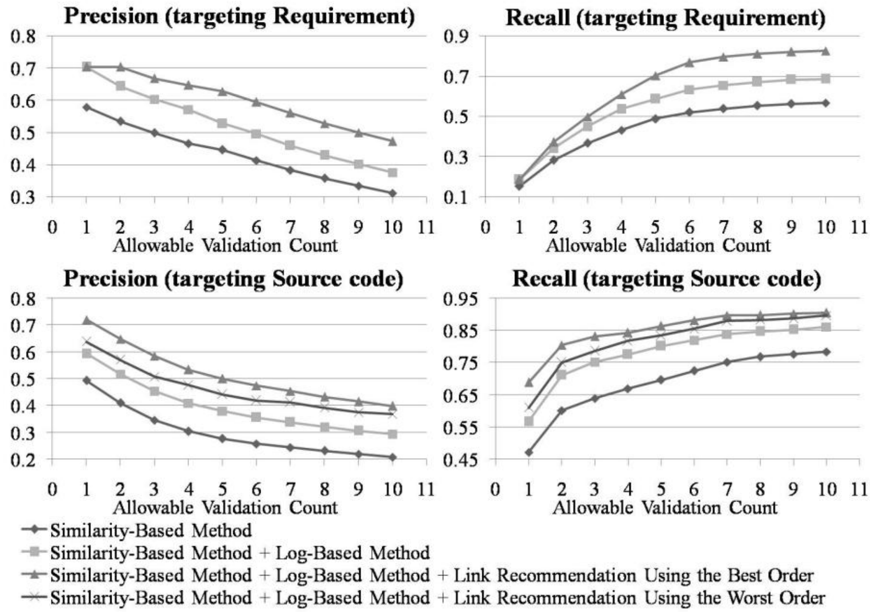

Fig. 5. Recall and precision

Table 1. Recall and precision when F-masure is highest

Method	Recall	Precision	F-measure
Similarity-Based Method	0.602	0.411	0.488
Similarity-Based Method + Log-Based Method	0.712	0.518	0.599
Similarity-Based Method + Log-Based Method + Link Recommendation Using the Best Order	0.804	0.648	0.718
Similarity-Based Method + Log-Based Method + Link Recommendation Using the Worst Order	0.751	0.571	0.648

4.2 First Experiment: Recovering by the Similarity-Based Method

First, we recovered links using only the similarity-based method, which used the similarity-based score to sort candidate links. Based on the results in Table 1, we can answer the first research question.
RQ1: How accurately can we recover links by the similarity-based method?
The similarity-based method recovered links with a recall of 60.2% and a precision of 41.1%. This accuracy is not sufficiently high. However, 42.1% of the recovered correct links are the links with source code files that have no revision histories. Therefore, the similarity-based method can cover the weakness of the log-based method.

4.3 Second Experiment: Recovering by the Combined Method

To confirm the effectiveness of combining the similarity-based method with the log-based method, we recovered links by using the combined method, which used both the similarity-based score and the log-based score. In all graphs of Figure 5, the combined method provides improved results compared to the similarity-based method. Thus, we can answer the second research question based on the results in Table 1.
RQ2: How much does the addition of the log-based method improve the recovery accuracy?
Adding the log-based method improved the recall by an 11.0 percent point (60.2% to 71.2%) and the precision by a 10.7 percent point (41.1% to 51.8%). Then, the F-measure was improved by 0.111 (0.488 to 0.599). Moreover, the average similarity-based score of links recovered newly by the combined method is 0.227, whereas the average score of links recovered by the similarity-based method is 0.635. Therefore, the log-based method can also cover the weakness of the similarity-based method.

4.4 Third Experiment: Effectiveness of Link Recommendations

We recovered links by using the combined method with link recommendations to evaluate the effectiveness. Then we tried using two kinds of orders: the best order and the worst order. In Figure 5, the method with link recommendations is superior to that without link recommendations for all conditions except when the targets are requirements and the allowable validation count is one.

When targeting requirements, the link recommendation becomes effective from the second presented candidate link because the first presented link is not recommended

by any other link. On the other hand, when targeting source code files, the link recommendation is effective from the first presented link excepting for the first targeted source code file, because the first presented link can be recommended by a link that has already been validated when targeting different source code files. Therefore, when targeting source code files, the targeting order affects the accuracy because the presentation order depends on the validation results of other source code files.

Thus, the experiments can answer the third research question.

RQ3: How much does the addition of link recommendations improve the recovery accuracy?

The link recommendation improved the recall by a 9.2 percent point (71.2% to 80.4%) and the precision by a 13.0 percent point (51.8% to 64.8%) when using the best order. Then, the F-measure was improved by 0.119 (0.599 to 0.718). On the other hand, when using the worst order, the effectiveness decreased in comparison with using the best order. Hence, the link recommendation is most effective when the correct link with the high relevance score recommends the correct link with the low relevance score. Therefore, when we put on emphasis on the accuracy, we should preferentially target source code files that have candidate links with the high relevance score. In the experiment, we recovered many additional links that have low relevance scores by applying the link recommendation.

4.5 Threats to Validity

The fact that we validated our method by applying to only one software product is a threat to the external validity. The improvement in accuracy depends on the quality of the revision messages and software structure because our method employs the configuration management log and call relationships. Thus, we should evaluate the relationship between these factors for other software and the effectiveness of our method.

In our evaluation, we independently conducted the recovery targeting requirements and the recovery targeting source code files. However, in an actual application, users randomly specify targets based on their needs. The targeting consistency may affect the accuracy of the recovering links, which is a threat to the internal validity. Therefore, we should conduct an experiment with random targeting in the future.

Additionally, our method uses user feedback to improve the accuracy, which may result in human error. We should conduct experiments by subjects to evaluate the impact of the environment for real applications of our method.

5 Related Work

Arkley et al. conducted a survey of nine software projects using questionnaires and interviews [7], and identified issues of traceability including usability. As these findings suggest, we should improve usability of our tool because validation of the candidate links takes significant costs. For example, supplemental information is necessary to validate links (e.g., rationales of the high relevance score, information about recommenders, etc.).

Mäder et al. conducted a controlled experiment with 52 subjects performing real maintenance tasks on two third-party development projects where half of the tasks were with and the other half were without traceability [1]. They showed that on average subjects with traceability perform 21% faster and create 60% more correct solutions. Their empirical study affirms the usefulness of requirements traceability links.

Some studies have compared the representation between requirements and source code to recover requirements traceability links [2,3,4,5,6,17,18,19,20] using different techniques, such as the vector space model, the probabilistic model, the latent semantic index and keyword matching with a regular expression. Here we propose extended method based on the method using the vector space model.

Chen et al. proposed an approach that combines three supporting techniques, Regular Expression, Key Phrases, and Clustering, with the vector space model to improve the traceability recovery performance [8]. Except for Clustering, their supporting techniques depend on the representation similarity between the requirements and source code files. On the other hand, our elemental techniques are independent of the representation similarity with source code files.

Wang et al. proposed a feature location approach that supports multi-faceted interactive program exploration [9]. Feature Location is a technique similar to recovering requirements traceability links for targeting requirements. Their approach automatically extracts multiple syntactic and semantic facets from candidate program elements. Then users can interactively group, sort, and filter feature location results by facets. Although our method is also an interactive method using user feedback, we require users only to validate correctness of candidate links.

Ghabi et al. proposed an approach to validate links through call relationships within the code [10]. They inputted set of candidate links with certain reliability and applied filtering by call relationships all at once, whereas we use only correct links validated by users and interactively apply the link recommendation by call relationships.

6 Conclusion and Future Work

We have proposed a traceability recovery method that extends the similarity-based method using two elemental techniques. The first technique is the log-based method using the configuration management log. The second is link recommendations using user feedback and the call relationships. We applied our method to an actual product and recovered links between 192 requirements and 694 source code files, confirming the effectiveness of applying two elemental techniques simultaneously. In the future, we plan conduct the additional experiments described in section 4.5, and investigate the applicability of other code relationships for link recommendations.

References

1. P. Mäder and A. Egyed. : Assessing the effect of requirements traceability for software maintenance. In: the 28th IEEE International Conference on Software Maintenance (ICSM'12), pp.171-180 (2012)

2. G. Antoniol, G. Canfora, G. Casazza, A. De Lucia and E. Merlo. : Recovering traceability links between code and documentation. In: IEEE Transactions on Software Engineering, vol.28, no.10, pp.970-983 (2002)
3. A. Marcus and J. I. Maletic. : Recovering documentation to source code traceability links using latent semantic indexing. In: the 25th International Conference on Software Engineering (ICSE'03), pp.125–135 (2003)
4. B. Dagenais and M. P. Robillard. : Recovering traceability links between an API and its learning resources. In: the 34th International Conference on Software Engineering (ICSE'12), pp.47-57 (2012)
5. X. Chen. : Extraction and visualization of traceability relationships between documents and source code. In: the 25th IEEE/ACM International Conference on Automated Software Engineering, pp.505–510 (2010)
6. A. De Lucia, R. Oliveto, and G. Tortora. : ADAMS re-trace: traceability link recovery via latent semantic indexing. In: the 30th International Conference on Software Engineering (ICSE'08), pp.839–842 (2008)
7. P. Arkley and S. Riddle. : Overcoming the traceability benefit problem. In: the 13th IEEE International Conference on Requirements Engineering (RE'05), pp.385-389 (2005)
8. X. Chen and J. Grundy. : Improving automated documentation to code traceability by combining retrieval techniques. In: the 26th IEEE/ACM International Conference on Automated Software Engineering, pp.223–232 (2011)
9. J. Wang, X. Peng, Z. Xing and W. Zhao. : Improving feature location practice with multi-faceted interactive exploration. In: the 35th International Conference on Software Engineering (ICSE'13), pp.762-771 (2013)
10. A. Ghabi and A. Egyed. : Code Patterns for Automatically Validating Requirements-to-Code Traces. In: the 27th IEEE/ACM International Conference on Automated Software Engineering, pp.200-209 (2012)
11. R. Tsuchiya, H. Washizaki, Y. Fukazawa, T. Kato, M. Kawakami and K. Yoshimura. : Recovering traceability links between requirements and source code in the same series of software products. In: the 17th International Software Product Line Conference (SPLC'13), pp.121-130 (2013)
12. R. Pooley and C. Warren. : Reuse through requirements traceability. In: the 3rd International Conference on Software Engineering Advances (ICSEA'08), pp.65-70 (2008)
13. G. Salton and M. J. McGill. : Introduction to modern information retrieval. McGraw-Hill, New York (1983)
14. Apache Subversion, https://subversion.apache.org/
15. Git, http://git-scm.com/
16. Java, https://www.java.net/
17. H. Jiang, T. N. Nguyen, I. Chen, H. Jaygarl and C. K. Chang. : Incremental latent semantic indexing for automatic traceability link evolution management. In: the 23rd IEEE/ACM International Conference on Automated Software Engineering, pp.59-68 (2008)
18. A. De Lucia, R. Oliveto, and G. Tortora. : IR-based traceability recovery processes: an empirical comparison of "one-shot" and incremental processes In: the 23rd IEEE/ACM International Conference on Automated Software Engineering, pp.39-48 (2008)
19. C. McMillan, D. Poshyvanyk and M. Revelle. : Combining textual and structural analysis of software artifacts for traceability link recovery. In: the 2009 ICSE Workshop on Traceability in Emerging Forms of Software Engineering, pp.41-48 (2009)
20. R. Settimi, O. BenKhadra, E. Berezhanskaya and S. Christina. : Goal-centric traceability for managing non-functional requirements. In: the 27th International Conference on Software Engineering (ICSE'05), pp.362-371 (2005)