PAPER

Recovering Traceability Links between Requirements and Source Code using the Configuration Management Log*

Ryosuke TSUCHIYA[†], ***Nonmember,*** **Hironori WASHIZAKI**[†], **Yoshiaki FUKAZAWA**[†], ***Members,***
Tadahisa KATO[††], **Masumi KAWAKAMI**[††] ***and*** **Kentaro YOSHIMURA**[†††], ***Nonmembers***

SUMMARY Traceability links between requirements and source code are helpful in software reuse and maintenance tasks. However, manually recovering links in a large group of products requires significant costs and some links may be overlooked. Here, we propose a semi-automatic method to recover traceability links between requirements and source code in the same series of large software products. In order to support differences in representation between requirements and source code, we recover links by using the configuration management log as an intermediary. We refine the links by classifying requirements and code elements in terms of whether they are common to multiple products or specific to one. As a result of applying our method to real products that have 60KLOC, we have recovered valid traceability links within a reasonable amount of time. Automatic parts have taken 13 minutes 36 seconds, and non-automatic parts have taken about 3 hours, with a recall of 76.2% and a precision of 94.1%. Moreover, we recovered some links that were unknown to engineers. By recovering traceability links, software reusability and maintainability will be improved.

key words: *traceability recovery, configuration management log, commonality and variability analysis, software product line*

1. Introduction

Traceability in software development is the ability to trace the relationships between artifacts. These relationships are called traceability links. Traceability links are formed between the following pairs of artifacts: the requirements specification document and source code that implements the requirements, design documents and test cases, requirements and design, etc. In this paper, we focus on links between requirements and code elements (e.g., function, class, file). For example, in CUnit [18] (the target of our evaluation experiments), the requirement "Running tests in Automated mode" links with the file *Automated.c* implements the requirement.

Manuscript received January xx, 20xx.
Manuscript revised March xx, 20xx.
†The authors are with Dept. Computer Science of Waseda University, Tokyo, 169-8555, Japan.
††The authors are with Yokohama Research Laboratory of Hitachi, Ltd., Kanagawa, 244-0817, Japan.
†††The author is with Hitachi Research Laboratory of Hitachi, Ltd., Ibaraki, 319-1292, Japan.
*This paper is an extended version of a paper presented at the 17th International Software Product Line Conference [1]. We have added some descriptions that explain how our framework supports software reuse and maintenance in the sections 2.1 and 3.9. Moreover, we have added some discussions to the section of evaluation.

Traceability links can be helpful in several software engineering tasks such as maintenance and reuse [12]. If engineers can grasp the relationships between requirements and source code, they can effortlessly identify the code elements implementing the requirements that they want to reuse or maintain (e.g., bug fix, modifications for change request [4]).

It is not practical from the viewpoint of cost that engineers manually recover all traceability links of large products. Moreover, there are links that are difficult to find manually, for example, if there is no apparent similarity in notation between the requirements and code, or if there is no description of the relationship in the documents. We call these "non-explicit traceability links."

We propose a framework to recover traceability links between requirements and source code in the same series of large software products. In order to support differences in representation between requirements and code elements (e.g., notation, language), we recover links by applying natural language processing and document retrieval to the configuration management log. However, the granularity of links recovered from the log is large, so we refine the links by conducting the commonality and variability analysis.

Our proposed method is semi-automatic; if any of the recovered links were unknown to engineers, engineers must manually judge whether they are non-explicit traceability links or false positives. If the accuracy of the recovery method is poor, or support information is missing, the decisions take significant costs. Our framework enables engineers to judge the validity of links with practical costs by reliable accuracy and support information.

The following are the Research Questions addressed in this study.

RQ1 How accurately can we recover candidate traceability links semi-automatically?

RQ2 Can non-explicit traceability links be manually recovered from candidate links suggested by our method?

RQ3 Can we recover traceability links within a reasonable amount of time?

In order to evaluate, we applied the framework to two products: open source software CUnit and a network

control system developed by a company. CUnit has more than 7KLOC, and the network control system has more than 60KLOC. In CUnit, we recovered traceability links with a recall of 76.0% and a precision of 70.4%. In the network control system, we recovered traceability links with a recall of 76.2% and a precision of 94.1%. Therefore, our framework is effective in the recovery of traceability links regardless of the size of the product or the development organization. The following are our contributions.

- We have proposed a method to semi-automatically recover traceability links using the configuration management log.
- We have proposed a method to refine traceability links by conducting the commonality and variability analysis.
- We have developed a tool that can recover links in large products within a reasonable amount of time.
- We have proposed a framework including the process to recover traceability links using the tool.
- We have applied the framework to actual products that have over 60KLOC, and have confirmed its validity.

By recovering links between requirements and source code using our framework, software reusability and maintainability is improved with practical costs.

The remainder of the paper is organized as follows. First, we provide some background information (Section 2). Then, we describe our framework to recover traceability links (Section 3). In Section 4, we present our evaluation of the framework by conducting experiments on two targets. In Section 5, we discuss related works. Finally, we provide a conclusion and future works (Section 6).

2. Background

2.1 Software Reuse and Maintenance with Traceability

Traceability links between requirements and source code facilitate the identification of code elements for reuse or modification especially when engineers do not have advanced knowledge of the previous product. In fact, in order to handle frequent software change requests, engineers are recommended to reduce the cost of identifying code elements impacted by change requests using traceability links.

Empirical studies have verified the effectiveness of traceability links in software reuse [3] and maintenance tasks [4]. Although these studies are not industrial records, both explicitly show traceability benefits by conducting experiments that conform to actual reuse and maintenance tasks.

Software Product Line Engineering (SPLE) has been widely recognized as an efficient method for software reuse. SPLE aids software development by using reusable core assets (e.g., feature, architecture, and code elements) [5]. In the extractive approach to develop core assets, we need to analyze the commonality and variability of existing products. Furthermore, SPLE requires relationships between those core assets (e.g., traceability links between features and code elements) to reuse them efficiently [6][7][8].

As described above, traceability links are essential for software reuse and maintenance. However, we must consider the cost of traceability recovery and management. If the recovery and management cost exceeds the benefits of traceability links (i.e., amounts of cost reduction in software reuse and maintenance), it does not meet the needs of engineers. Therefore, automatic support methods are required to minimize the cost of recovering and managing traceability links.

Many previous studies have proposed automatic or semi-automatic methods to recover traceability links (described in section 5.2). However, these methods have a common weakness — they depend on the representation being similar between the requirements and source code (described in section 2.2). We focus on overcoming this weakness in this paper.

2.2 Configuration Management Log

If the identifier of code elements (e.g., file name, function name) and requirements are represented using the same notation and language, automatic recovery of traceability links using previous methods is easy. However, this is often not the case. For instance, while the purpose is described in the requirements, the identifier that signifies the means can be given to the code elements. In another case, the identifier can be the short form of requirements. The most difficult case is when the language is different between the requirements and source code in previous methods.

In the above cases, it is difficult to recover links by comparing the requirements and the identifier of code elements. In order to support differences in expression, an intermediary is required. Here, we focus on the configuration management log that contains information related to requirements and source code. It is composed of revisions that include messages and file paths.

The two targets of our evaluation experiments use the version management system Apache Subversion (SVN) [19]. Figures 1 shows an excerpt from a log of CUnit as specific examples of the revision of SVN. It shows that the revision has a message and a file path. By examining these logs, we have confirmed that words related to requirements appear in the messages of the log. For example, in Figure 1, the word "XML" appears in the message. This word is strongly correlated with the requirement “Running tests in Automated mode” because this functional requirement is the only one that outputs results in XML format. If these words are recorded along with file paths, we can recover traceability links without depending on the notation.

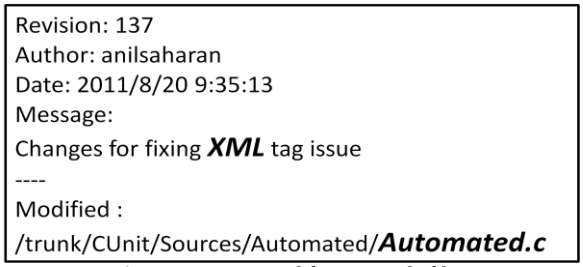


Fig. 1 Revision modifying a single file.

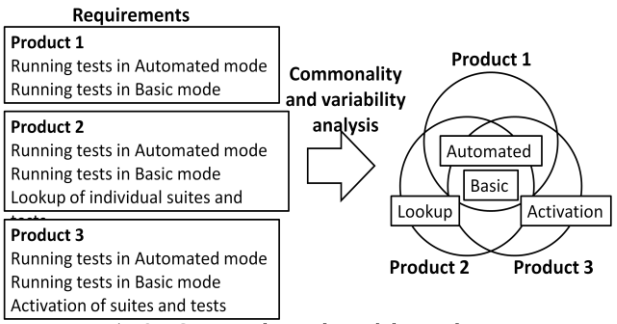


Fig. 2 Commonality and variability analysis.

2.3 Commonality and Variability Analysis

Traceability links between requirements and functions cannot be recovered using file paths in the configuration management log. Therefore, we use the Commonality and Variability Analysis (CVA) on the same series of software products so that we can recover traceability links between requirements and functions. We will explain how to recover links with functions by CVA in section 3.7.

CVA is used to analyze to which products elements (e.g., requirements, code elements) belong. CVA classifies elements as common to some products or as specific to a product. Figure 2 shows a concrete example: the requirements "Running tests in Automated mode" is common to three products, whereas the requirement "Activation of suites and tests" belongs to product Z only.

CVA is used to support the development of core assets in SPLE. There are several methods of CVA in software elements (e.g., requirements, architecture). In the following paragraphs, we describe the methods that we have previously proposed.

We have proposed a method of CVA of the requirements in legacy software products [9]. This method measures the similarity of sentences using the vector space model, and analyzes whether or not the requirements are common to multiple products. In the vector space model proposed by Salton et al. [10], a sentence is represented by one vector that depends on the valid words in the sentence. The contents of the sentence are determined by the direction of the vector.

There is a method of CVA of code elements using code clone detection [11]. A code clone is a code fragment that is identical or similar to another in the code.

2.4 Motivating Example

In many products, the notation and the abstraction level are different between the requirements and the identifier of code elements. In CUnit, the requirement "Lookup of individual suites and tests" links with the functions *CU_get_suite()* and *CU_get_test()* that belong to the file *TestDB.c*. There are some overlapping words between the requirements and the identifiers, but they cannot be easily associated through comparison. In the network control system used as a target of our evaluation experiments, the identifier of code elements is written in English, while the requirements are written in Japanese.

Non-explicit traceability links exist in most products. In CUnit, the user manual describes most traceability links between requirements and code elements. However, information on the relevant requirements of some files (e.g., *MyMem.c*) is not mentioned. This information may be unnecessary if CUnit is used as a testing framework, but it is useful for derived development based on CUnit. In the network control system, there are many traceability links that engineers have not grasped because the number of requirements and files is quite large.

3. Traceability Link Recovery Framework

3.1 Overview

We propose a framework to recover traceability links between requirements and source code in the same series of large software products. Figure 3 shows the overview of our framework. The targets of our framework are series of large software products developed by one organization. As inputs, the assets of requirements and source code are required for each product. Assets of requirements are documents about product requirements written in natural language. In our framework, we focus on requirements that are concrete and objective (i.e., software functional and non-functional requirements). Our framework mainly treats links with component level. However, we can optionally treat links with function level by refining links with components. In this paper, "component" means source code files or classes. And, "function" means a subroutine as part of the component. (e.g., methods of Java classes)

Traceability links can be recovered by finding revisions that contain words related to requirements in the configuration management log. For refining these links, CVA is conducted prior to recovering links. Finally, we refine traceability links using the CVA results in order to enhance accuracy.

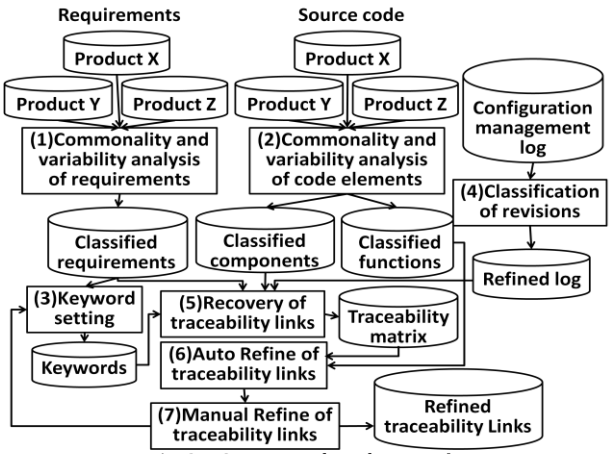


Fig. 3 Overview of our framework.

Our framework is divided into the following seven steps.

Step (1). CVA of Requirements
Step (2). CVA of Code Elements
Step (3). Keyword Setting
Step (4). Classification of Revisions
Step (5). Recovery of Traceability Links
Step (6). Auto Refine of Traceability Links
Step (7). Manual Refine of Traceability Links

The following sections describe each step in detail.

3.2 CVA of Requirements

We use the method mentioned in section 2.3 for CVA of requirements to measure the similarity of requirements.

Each requirement is treated as a sentence in this method, and the vector space model is applied to represent each sentence by a vector determined by the valid words (nouns, verbs, and adjectives) in the sentence. For a sentence S_x containing M valid words, i.e., v_1, v_2, $\cdots$, v_M, $\mathrm{w}(v_p, S_x)$ $(1 \leq \mathrm{p} \leq \mathrm{M})$ is the number of appearance of v_p in S_x. S_x is represented by the M-dimensional vector $\overrightarrow{d_x}$ defined as follows:

$$\overrightarrow{d_x} = \left(\mathrm{w}(v_1, S_x), \mathrm{w}(v_2, S_x), \cdots, \mathrm{w}(v_M, S_x)\right)$$

The similarity between the two sentences S_i and S_j is obtained as the cosine of the angle between the two sentence vectors $\overrightarrow{d_i}$ and $\overrightarrow{d_j}$ (cosine similarity). $SSim(S_i, S_j)$ (Sentence Similarity, $0 \leq SSim \leq 1$) is defined using the cosine similarity as follows:

$$SSim(S_i, S_j) = \frac{\overrightarrow{d_i}\overrightarrow{d_j}}{|\overrightarrow{d_i}||\overrightarrow{d_j}|}$$

By measuring the similarity SSim between the requirements in each product, each requirement is classified as being common to some products or as being specific to one product. If SSim exceeds a threshold set by the users, the corresponding requirements are judged to be identical. The classification result is represented as a subset of the set of all products targeted. For instance, in Figure 2, the set of all products targeted is {X, Y, Z}. The requirement "Running tests in Automated mode" belongs to {X, Y, Z}. On the other hand, the requirement "Lookup of individual suites and tests" belongs to {Y}.

3.3 CVA of Code Elements

In this step, CVA of code elements is conducted at the granularity of both components and functions. Source code is needed as input. In the same way as requirements, code elements are classified as either common or specific. As in a previous study, we use code clone detection for the analysis.

Each code element is composed of tokens (e.g., operation, identifier). $Token(E_x)$ represents the total number of tokens for a code element E_x. $Clone(E_x, E_y)$ represents the number of tokens of code clones between E_x and E_y. The similarity between E_x and E_y is determined by the ratio of the number of code clone tokens to the total number of tokens. $CESim(E_x, E_y)$ (Code Element Similarity, $0 \leq CESim \leq 1$) is defined by the following formula:

$$CESim(E_x, E_y) = \frac{Clone(E_x, E_y) * 2}{Token(E_x) + Token(E_y)}$$

If $CESim(E_x, E_y)$ exceeds a threshold set by the users, E_x and E_y are determined to be identical. This similarity measurement is conducted for all code elements that share code clones, and each code element is classified by the products to which it belongs. The classification result is represented in the same way as requirements.

3.4 Keyword Setting

We utilize words related to requirements appear in the messages of the configuration management log. In this step, keywords that characterize each requirement are set so that they can be used to identify the components related to the requirements in a later.

First, as candidates of keywords, words that have a large TF-IDF value are extracted from the documents that describe the summary of requirements. TF-IDF is a method for word weighting using term frequency and inverse document frequency. In addition, proper nouns, including abbreviations, are extracted as candidates.

Then, engineers set the keywords by adding, deleting, modifying, or combining the candidate words.

3.5 Classification of Revisions

If we use revisions that simultaneously modify components of multiple domains to recover traceability links, unrelated requirements and components may be

linked. Unfortunately, simultaneous revisions often occur. Therefore, in order to extract useful information while avoiding false positives, our framework automatically classifies revisions into the following three types based on the number of domains they affect. Here, domain is a directory that has files implementing the same feature.

Type A. Revisions modifying components of a single domain.
Traceability links recovered from this type are the most reliable. The revision in Figure 1 is classified as this type.

Type B. Revisions modifying components of multiple domains below the threshold number.
Because poorly related features are simultaneously modified in some cases, traceability links recovered from this type of revision should be distinguished from traceability links recovered from Type A revisions.

Type C. Revisions modifying components of multiple domains greater than or equal to the threshold number.
This type of revision causes false positives, so it is removed from targets of search in the latter steps.

The threshold number is set by users. As a guideline, if there are a lot of Type A revisions, users expect Type B revisions the reliability rather than their number, so they should set a low threshold number. Conversely, if there are few Type A revisions, users require a lot of Type B revisions, so they should set a high threshold number.

At the end of this step, a refined log with the revisions classified and Type C revisions removed is outputted. This refined log is used in the following steps.

3.6 Recovery of Traceability Links

3.6.1 Traceability Links Recovery Method

In this method, revisions that have message containing the keywords set in Step (3) (Keyword Setting) are identified to determine the implementation points. The number of keyword appearance must be above the threshold number, which is tuned to the number of words in the revision message. When most of revision messages have the large number of words, we need to set the large number to the threshold of keyword appearance. Then, the requirements connected with the keywords are linked with the modified components written as file paths in the revision. For example, CUnit has the requirement "Running tests in Automated mode." If the word "XML" is set as a keyword of this requirement, the method searches for revisions that have a message containing the word "XML" in the configuration management log to determine the implementation points. One such implementation point would be in the revision in Figure 1. In this revision, the component *Automated.c* is modified. As a result, a traceability link between the requirement "Running tests in Automated mode" and the component *Automated.c* is recovered. The same operation is conducted for all requirements to identify and link the related components.

3.6.2 Types of Traceability Links

For each traceability link recovered, the requirement and the component should belong to the same group of products as classified by CVA. If not, this information can be used to refine traceability links. We classify traceability links into five types using the CVA results. We first define the following terms.

k is the number of targeted products. R_i represents the set of requirements for each product. Then, R (the set of requirements in all targeted products) is defined by the following formula:

$$R = \bigcup_{i=1}^{k} R_i$$

Likewise, C_i represents the set of components for each product. Then, C (the set of components in all targeted products) is defined by the following:

$$C = \bigcup_{i=1}^{k} C_i$$

If $\mathfrak{P}(C)$ represents the power set of C, then, φ (the relationship between R and $\mathfrak{P}(C)$ obtained from the configuration management log) is defined as following:

$$\varphi: R \to \mathfrak{P}(C)$$

Similarly, if $\mathfrak{P}(P)$ represents the power set of the targeted products P, then, I_R (the relationship between requirements and the set of products that have the requirements) and I_C (the relationship between components and the set of products that have the components) are defined by the following:

$$I_R: R \to \mathfrak{P}(P)$$

$$I_C: C \to \mathfrak{P}(P)$$

Finally, c (one of the components $\varphi(r)$ linked to the requirement r) is defined by the following:

$$r \in R \mapsto \varphi(r) \subset C$$
$$c \in \varphi(r)$$

$I_R(r)$ is the set of products that have the requirement r. $I_C(c)$ is the set of products that have the component c. Then, as a result of the comparison between $I_R(r)$ and $I_C(c)$, traceability links between r and c are classified into the following five types. Figure 4 shows examples of when products X, Y and Z are targeted.

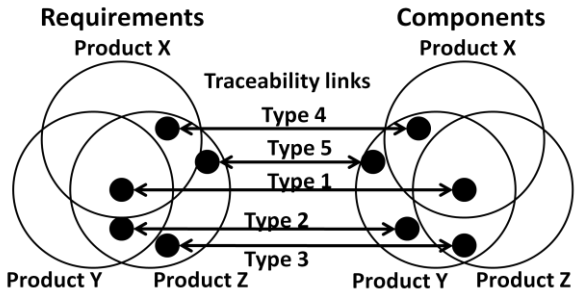


Fig. 4 Types of traceability links.

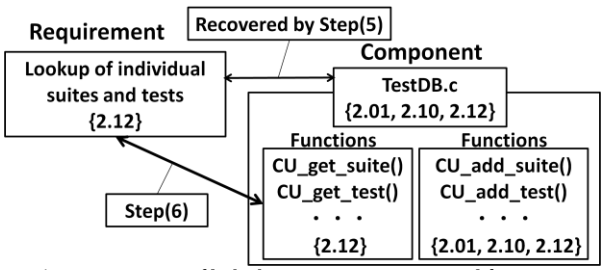


Fig. 5 Recovery of links between requirements and functions.

Type1. $I_R(r) = I_C(c)$

e.g., $I_R(r) = I_C(c) = \{\mathrm{X}, \mathrm{Y}, \mathrm{Z}\}$
The requirement and the component belong to the same group of products.

Type2. $I_R(r) \supset I_C(c)$

e.g., $I_R(r) = \{\mathrm{Y}, \mathrm{Z}\}$, $I_C(c) = \{\mathrm{Y}\}$
The set of products that have the component is a proper subset of the set of products that have the requirement.

Type3. $I_R(r) \subset I_C(c)$

e.g., $I_R(r) = \{\mathrm{Z}\}$, $I_C(c) = \{\mathrm{Y}, \mathrm{Z}\}$
The set of products that have the requirement is a proper subset of the set of products that have the component.

Type4. $\left(I_R(r) \not\subseteq I_C(c)\right) \wedge \left(I_R(r) \not\supseteq I_C(c)\right) \wedge$

$(I_R(r) \cap I_C(c) \neq \emptyset)$

e.g., $I_R(r) = \{\mathrm{X}, \mathrm{Z}\}$, $I_C(c) = \{\mathrm{X}, \mathrm{Y}\}$
Conditions of Type 1, 2 and 3 are not satisfied. However, the sets of products include common products. In the example, X is common product.

Type5. $(I_R(r) \cap I_C(c) = \emptyset)$

e.g., $I_R(r) = \{\mathrm{Z}\}$, $I_C(c) = \{\mathrm{Y}\}$
The sets of products include no common products.

3.7 Auto Refine of Traceability Links

In this step, we refine links using the classification described in the previous step.

1) Recovery of traceability links between requirements and functions
When a traceability link between a requirement and a component is of Type 3, the requirement may link with functions of the component. If the component has functions whose results of the CVA are the same as those of the requirement, these functions may link with the requirement.

Figure 5 shows an example in three products of CUnit {2.01, 2.10, 2.12}. The traceability link between the requirement "Lookup of individual suites and tests," which belongs to the product {2.12}, and the component *TestDB.c* which belongs to the products {2.01, 2.10, 2.12} is recovered by Step (5). These CVA results are different, but the component *TestDB.c* has functions that belong only to the product {2.12}. Some of these functions may link with the requirement "Lookup of individual suites and tests". If links of Type 3 are recovered, functions whose results of the CVA are the same as those of the requirement are demonstrated to users.

2) Suggestion of the presence of sub requirements
When a traceability link between the requirement and the component is of Type 2, the granularity of the requirement may be large. Sub requirements whose results of the CVA are the same as those of the component may exist. However, we only suggest the presence of these because we do not stratify requirements.

3) Elimination of false positives
Traceability links of Types 4 and 5 may be false positives because the products to which the requirement and the component belong are different. Therefore, these links are removed from the results.

3.8 Manual Refine of Traceability Links

To check the validity of the links recovered, engineers review the links as follows.

First, engineers look at the traceability matrix to see if there are any requirements that link with a huge range of components. If they find such a requirement, a keyword for the requirement may be a word that is widely used in the configuration management log. In this case, the engineers must go back to Step (3) to review the keyword setting.

Next, for traceability links whose relationship is hard to understand at a glance, engineers check their validity by reviewing the revision messages from which they were recovered. If their validity is confirmed, the recovery of these non-explicit traceability links is considered a success.

Finally, engineers identify traceability links between requirements and functions using information obtained in Step (6). In Figure 5, functions that belong only to the product {2.12} have been suggested to link

with the requirement "Lookup of individual suites and tests." However, there is a possibility that these functions have been suggested to link with other requirements in the same way. Therefore, engineers have to identify correct links from candidates obtained in Step (6).

3.9 Application of Framework

3.9.1 Usage of Framework

When engineers would like to reduce the cost of maintenance tasks (especially modifications for change request), traceability recovery techniques including our framework can help them. Traceability links are particularly required in case that change requests for software occurs frequently and continuously. If engineers didn't ensure links in the development phase, they have to apply the traceability recovery techniques in the early phase of maintenance in order to ease the later maintenance tasks.

However, it's not sufficient to recover links only once. Engineers have to manage traceability links continuously because status of the links is changing with the passage of time. If they continue to use the first recovered links, those links may cause misleading.

Our framework can apply to the management of traceability links because the configuration management log has records of modification and addition of source code. For example, when new requirements and components are added or existing components are modified after recovering traceability links, the configuration management log is updated. If we recover links again using the latest log, we can reflect the changes and update traceability links.

If engineers introduce SPLE to their product series, CVA of existing assets is required in order to develop core assets. Our framework cannot extract core assets that can be used immediately. However, our framework can support the extraction of reusable assets by CVA and the recovery of traceability links between the assets.

3.9.2 Scope of Framework

Our framework targets the software products using the configuration management log. When engineers would like to recover traceability links, they cannot use methods comparing representation between requirements and source code if documents are written in their native language (not English). However, if they have the configuration management log written in their native language, they can apply our framework to recover links. Industrial products developed by companies often use documents written in their mother tongue in the same manner as the network control system used as a target of our evaluation experiments. Therefore, approaches independent of the representation similarity, including our framework, may contribute to the industry.

Applicable source code languages of our framework conform to those of the code clone detection technique. The current applicable languages of our tool are Java, C and C++. However, by adding features, it's possible to apply the other languages supported by the code clone detection tool (e.g., C#, Visual Basic and Cobol).

Regarding the variation realization techniques, our framework is not applicable to some cases because we conduct CVA of code elements by comparing contents of functions. When the contents are different between two same name functions, our framework determines that they are different functions. However, our framework doesn't consider the difference of parameters and the presence of macro. Therefore, even if parameters are different between functions, we cannot distinguish between those functions that have similar contents.

4. Evaluation

4.1 Overview

We carried out experiments targeting two groups of products, which are different in terms of their size and development team. One is open source testing framework CUnit, and the other is the network control system developed by a company. Both targets are implemented in the C language. We experimented with three versions of each target. Table 1 shows the SLOC and number of requirements of each version.

For CUnit, we extracted the requirements from the user manual and recovered the traceability links between them and the 9 components in CUnit. We evaluated the validity of the results by comparing them with the links mentioned in the user manual.

We extracted the requirements of the network control system from its design documents of features. We targeted 5 modules that cover the basic features of the network control system. A module is a group of components. Engineers previously prepared links between requirements and modules to evaluate the validity of our results. However, their granularities were larger than those of the links recovered by our method. After recovering links between requirements and components, we linked these requirements with the module that contains the corresponding components. This eliminated the difference in granularity. The SLOC in Table 1 represents the size of the five modules. The size of the entire system is 1.4 ~ 1.7 MLOC.

We used the log of SVN in both targets to obtain the revision. CUnit had 156 revisions and the network control system has 5727 revisions.

First, we recovered traceability links between requirements and components (or modules in the network control system) by conducting Steps (1) ~ (5).

Table 1 SLOC and number of requirements of our target systems.

System	Version	SLOC	Requirements
CUnit	2.01	5931	11
	2.10	6225	11
	2.12	7760	15
Network control system	3.01	54579	41
	3.02	55281	48
	3.03	62448	49

Table 2 Recall and precision.

Target	Thres	Rel	Ret	Rel ∩ Ret	Recall	Precision	F-m
CUnit	1	20	27	14	70.0%	51.9%	0.596
Network	1	16	40	13	81.3%	32.5%	0.464
	5	16	17	11	68.8%	64.7%	0.667
	10	16	7	6	37.5%	85.7%	0.522

With regard to the threshold of keyword appearance, for CUnit, the threshold number was set to 1 because few words are contained in revision messages. On the other hand, for the network control system, we set three different threshold numbers to study the relationship between keyword appearance and accuracy of traceability links. The thresholds were 1, 5 and 10.

Next, we confirmed the traceability links between requirements and functions by conducting Step (6). Finally, we looked for the non-explicit traceability links by conducting Step (7). Engineers of the developer team conducted the review for the network control system, but we conducted the review ourselves for CUnit because CUnit is open source software. For the automatic parts, we used our tool implemented in Java and the code clone detection tool CCFinderX [20].

4.2 Results

Table 2 shows the results of the recovery of traceability links for each target. The second column, *Thres (Threshold)*, contains the threshold numbers of keyword appearance. The third column, *Rel (Relevant)*, contains the number of previously known traceability links that we used to evaluate our method. The fourth column, *Ret (Retrieved)*, contains the number of traceability links retrieved by Step (5). The fifth column, *Rel∩Ret*, gives the number of traceability links that were both previously known and retrieved by Step (5). *Recall*, *Precision*, and *F-m (F-measure)* are defined as follows:

$$\text{Recall} = \frac{\text{Relevant} \cap \text{Retrieved}}{\text{Relevant}}$$

$$\text{Precision} = \frac{\text{Relevant} \cap \text{Retrieved}}{\text{Retrieved}}$$

$$\text{F} - \text{measure} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

We present the accuracy of our method for recovering known links in this section. In next sections, we show results for each target in terms of the recovery of links between requirements and functions, recovery of non-explicit links, and the time taken to recover links. Here, we used the results with the highest *F-measure*. (For the network control system, the threshold number is 5.) However, we can also apply Step (6) and Step (7) to the other cases.

4.2.1 CUnit

3 of the 27 links retrieved for CUnit by Step (5) were of Type 3. These were links between requirements that belong to the product {2.12} and components that belong to the products {2.01, 2.10, 2.12}. We extracted functions that belong to the product {2.12} from these components using our tool, and found that some of these functions were mentioned in the user manual as being related to the corresponding requirements.

13 of the 27 links retrieved by Step (5) were not mentioned in the user manual. By reviewing the revision messages for these links, we determined that 5 of the links were valid. These links were concerned with the component *MyMem.c*, which manages the memory. Therefore, *MyMem.c* links with requirements regarding adding, deleting, and initializing tests. However, the relationship between *MyMem.c* and those requirements were not mentioned in the user manual. When we included these 5 links to *Relevant*, *Recall* became 76.0%, *Precision* 70.4%, and *F-measure* 0.731.

Regarding the time taken to recover links in CUnit, most of our framework is automated, and the running time of our tool was 1 minute 40 seconds. The semi-automated parts of our framework (Steps (3) and (7)) took 30 minutes each.

4.2.2 The Network Control System

3 of the 17 links retrieved by Step (5) were of Type 3. These were links between requirements that belong to the products {3.02, 3.03} and components that belong to the products {3.01, 3.02, 3.03}. We extracted functions that belong to the products {3.02, 3.03} from these components using our tool, and found that the identifiers of some of these functions used the short form of the requirements.

6 of the 17 links retrieved by Step (5) were not mentioned by engineers. By reviewing revision messages, we determined that 5 of the links were valid. When we included these 5 links to *Relevant*, *Recall* became 76.2%, *Precision* 94.1%, and *F-measure* 0.842.

Regarding the time taken to recover links in the network control system, the running time of our tool was 13 minutes 36 seconds. Step (3) took approximately 2 hours, and Step (7) took approximately 1 hour.

4.3 Discussion

4.3.1 Research Questions

RQ1 How accurately can we recover candidate traceability links semi-automatically?

For CUnit, *Recall* was 70.0%, and *Precision* was 51.9%. For the network control system, *Recall* was 68.8%, and *Precision* was 64.7%.

With regard to false negatives, we failed to recover approximately 30% of known links. We have not been able to recover traceability links involving components that have not been modified in the period of the configuration management. For example, if a component that is reused from past assets is not modified, only the record of adding it remains. This will make it difficult for our framework to recover traceability links involving this component. However, traceability links of reusable past assets tend to be known to engineers, so the engineers may recover these links easily.

With regard to *Precision*, it was high enough to judge the validity of remain links (i.e., non-explicit traceability links or false positives).

RQ2 Can non-explicit traceability links be manually recovered from candidate links suggested by our method?

In CUnit, 5 of 13 traceability links that were not mentioned in the user manual were refined as non-explicit traceability links. Consequently, *Recall* became 76.0%, and *Precision* became 70.4%. In the network control system, 5 of 6 traceability links that were not grasped by engineers were refined as non-explicit traceability links. Consequently, *Recall* became 76.2%, and *Precision* became 94.1%. The results show that non-explicit traceability links can be successfully recovered.

With regard to false positives, when the name of an asset treated by multiple requirements is set as the keyword of these requirements, a revision message containing the keyword will cause the components tied to the revision to be linked with all of these requirements. If the same keyword needs to be used for multiple requirements, the possibility of the number of false positives increasing should be considered.

In both targets, we could recover links between requirements and functions. This shows that using CVA is effective in the recovery of links with functions.

RQ3 Can we recover traceability links within a reasonable amount of time?

In CUnit, the automatic parts took 1 minute 40 seconds, and the non-automatic parts took about 1 hour. In the network control system, the automatic parts took 13 minutes 36 seconds, and the non-automatic parts took about 3 hours. These results show that traceability links can be recovered within a reasonable amount of time. Moreover, when we applied our framework to 35 modules (200 KLOC) of the network control system, the running time of our tool was 58 minutes 12 seconds.

4.3.2 Threshold of Keyword Appearance

For CUnit, we didn't set the threshold number to over 2 because there were not any revisions including 2 or more keywords. In the configuration management log of CUnit, most of revisions have one line message. In the case that revision messages have the small number of words, we have to set the small threshold number.

For the network control system, we set three different thresholds. Table 2 shows the relationship between the threshold and the accuracy. In the case of the small threshold, *Recall* is high. On the other hand, in the case of the large threshold, *Precision* is high. Therefore, users need to adjust the threshold number in accordance with the purpose. If users put emphasis on completeness, they should set a small threshold. If they give priority to correctness, they should use a large threshold.

4.3.3 Scalability of Framework

The number of components is small in our experiments. When we selected targets of experiments, there were some conditions. The target needs to have some versions with the configuration management log (The scope of our framework). And, the information of previously known traceability links is required in order to evaluate the accuracy. Except for CUnit, we could not find products satisfying the conditions from open source software. On the other hand, we prepared the previously known traceability links for only five modules in the network control system because of engineer's time constraints.

If we apply our framework to products including more components, the time cost and the accuracy of recovery are influenced. The time taken in automatic parts of our framework will increase because the code clone detection is conducted as many times as the number of combinations between code elements. And, the manual refine of links becomes difficult with increasing the number of components. Therefore, it increases not only cost but also misjudgment of users. As a result, the accuracy of recovery will decrease.

However, the execution of recovering all links is not repeated frequently. Therefore, the automated process of our framework doesn't have to finish recovering links of the large product in a few minutes. Regarding the manual process, in the use case in which users want to recover links of the specific requirements or components, the time cost of our framework may be allowable.

4.3.4 Qualitative Comparison with Previous Methods

Most of previous methods compare the representation between requirements and source code to recover links. Therefore, these methods are not effective for software using a non-English language. If we apply these methods to the network control system (using Japanese), the accuracy of recovery would be inferior to our framework.

On the other hand, in CUnit, there are many words shared between requirements and source code. So, previous methods can recover links with higher accuracy than our framework. However, our framework has the ability recovering non-explicit traceability links which may be overlooked by previous methods.

In countries of non-English speaking, the documents are often written in their native languages. Therefore, our framework is effective for software developed in those countries. On the other hand, many of the open source software projects have the configuration management log including low quality log messages in comparison with commercial products. Our framework is not effective for such software.

Previous methods and our framework have the strengths and weaknesses respectively. Therefore, we should selectively use them depending on the situation. And, we would like to combine our framework and previous methods to improve the accuracy and the applicable scope.

4.4 Limitations

4.4.1 Dependence of Log Messages

Our frame work is highly dependent on the quality of log messages. If engineers do not record detailed information about modifications in log messages, our framework cannot work well. For example, if a revision only contains “Fix” in the log message, our framework cannot use such a revision to recover links. As in Figure 1, at least one meaningful phrase is required for each revision.

4.4.2 Threats to Validity

We manually set the keywords for each requirement and empirically got the trends of unsuitable or effective keywords. This may have affected the accuracy and costs of our evaluation, and is a threat to internal validity. In the future, we should confirm the influence of having multiple people set keywords on accuracy and costs.

The two targets we used are different in terms of software domain and the development organization. These factors should not significantly affect the validity of our framework.

In our evaluation, *Relevant* consisted of links known in advance and correct links recovered by our framework. However, there should be some links that were not known and could not be recovered. Therefore, if we include these links to *Relevant*, *Recall* may become lower. We should conduct experiments using benchmarks in order to evaluate our framework more accurately.

5. Related Work

5.1 Traceability Issues and Benefits

Arkley et al. have conducted a survey of nine software projects using questionnaires and interviews [2]. They have identified three issues related to traceability: the usability of tools and the necessity of additional input data; a lack of understanding on how to employ the traceability information; and a lack of perception of direct benefits to the main development process. Researchers in the traceability field should aim to overcome these issues. Traceability recovery tools, including our tool, have not been able to completely overcome the issue of usability. We should reduce the manual process and additional input data in the future.

Mäder et al. have conducted a controlled experiment with 52 subjects performing real maintenance tasks on two third-party development projects: half of the tasks with and the other half without traceability [4]. Through the experiment, they have shown that subjects with traceability performed on average 21% faster on a task and created on average 60% more correct solutions. This empirical study has affirmed the usefulness of requirements traceability. In order to maximize traceability benefits, the cost of recovering and maintaining traceability links should be reduced. We believe that studies of traceability recovery, including our study, address this important issue.

5.2 Traceability Recovery

Antoniol et al. have proposed a method to recover traceability links between code and documentation using information retrieval technologies, such as the probabilistic model and the vector space model [12]. They compare the identifier in source codes with the words in documents to recover links. In contrast, we recover links using the configuration management log. Our framework can recover links even if the identifier in source codes and the words in documents are different.

Marcus et al. have proposed a method to recover links between documentation and source code using latent semantic indexing (LSI) [13]. They measure the similarity of latent semantic between documentation and source code to recover links, which significantly decreases the dependency on the similarity of representation. However, LSI cannot deal with linguistic differences. They use the comments and identifier names within the source code. Hence, they require that the same language be used in the documentation and source code

in order for their method to work well.

Dagenais et al. have proposed a method to recover traceability links between an API and learning resources by using code-like terms in documents and analyzing their contexts [14]. Our framework does not require code-like terms in documents because it uses the configuration management log to recover links.

There are additional studies that have compared the representation between requirements and source code to recover links [15] [16]. Our framework is intended to cover the weakness of their methods rather than to be upward-compatible with them. Our method does not depend on the representation, but it may be inferior to their methods for targets in which there is little difference in the representation between requirements and code. So the completeness and correctness of the traceability link recovery may be improved by combining our framework with previous methods.

Kaiya et al. have proposed a method to find change impacts on source codes caused by requirements changes [17]. They use documents written in Japanese, and identify requirements from Japanese sentences and implementation points from English sentences. In our method, we use the configuration management log. In the log, requirements and implementation points are distinguished as messages and file paths, so our framework does not depend on the language of targets.

6. Conclusion and Future Work

We have proposed a framework that includes the process to recover traceability links between requirements and source code. We have recovered links using the configuration management log, and have refined the links by applying CVA and having engineers review them. Moreover, we have applied the framework to actual products that have more than 60KLOC, and have confirmed its validity. Our framework enables cost reduction of the recovery of traceability links, and the recovery of non-explicit traceability links. Recovering traceability links may increase the reusability and maintainability of software. For future work, we will consider the hierarchical structure of requirements and code elements, and aim to improve our methods for keyword setting and refining links. And, we should conduct comparison experiments with previous methods in order to argue that our framework can cover the weakness of previous methods.

Acknowledgments

We thank Kentaro Kumaki for providing a prototype tool of the CVA of requirements.

References

[1] 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,” the 17th International Software Product Line Conference (SPLC’13), pp.121-130, 2013.

[2] P. Arkley and S. Riddle, “Overcoming the traceability benefit problem,” the 13th IEEE International Conference on Requirements Engineering (RE’05), pp.385-389, 2005.

[3] R. Pooley and C. Warren, “Reuse through requirements traceability,” the 3rd International Conference on Software Engineering Advances (ICSEA’08), pp.65-70, 2008.

[4] P. Mäder and A. Egyed, “Assessing the effect of requirements traceability for software maintenance,” the 28th IEEE International Conference on Software Maintenance (ICSM’12), pp.171-180, 2012.

[5] K. C. Kang, J. Lee and P. Donohoe, “Feature-oriented product line engineering,” Software, IEEE, vol.19, no.4, pp.58-65, 2002.

[6] W. Jirapanthong and A. Zisman, “Supporting product line development through traceability,” the 12th Asia-Pacific IEEE Software Engineering Conference (APSEC’05), 2005.

[7] S. A. Ajila and A. B. Kaba, “Using traceability mechanisms to support software product line evolution,” the IEEE International Conference on Information Reuse and Integration (IRI’04), pp.157-162, 2004.

[8] S. Mohalik, S. Ramesh, J. V. Millo, S. N. Krishna and G. K. Narwane, “Tracing SPLs precisely and efficiently,” the 16th International Software Product Line Conference (SPLC’12), vol.2, pp.186-195, 2012.

[9] K. Kumaki, R. Tsuchiya, H. Washizaki and Y. Fukazawa, “Supporting commonality and variability analysis of requirements and structural models,” MAPLE 2012, SPLC’12, vol.2, pp.115-118, 2012.

[10] G. Salton and M. J. McGill, “Introduction to modern information retrieval,” McGraw-Hill, New York, 1983.

[11] K. Yoshimura, D. Ganesan and D. Muthig, “Defining a strategy to introduce a software product line using existing embedded systems,” EMSOFT '06 Proceedings of the 6th ACM & IEEE International conference on Embedded software, pp.63-72, 2006.

[12] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia and E. Merlo, “Recovering traceability links between code and documentation,” IEEE Transactions on Software Engineering, vol.28, no.10, pp.970-983, 2002.

[13] A. Marcus and J. I. Maletic, “Recovering documentation to source code traceability links using latent semantic indexing,” the 25th International Conference on Software Engineering (ICSE’03), pp.125–135, 2003.

[14] B. Dagenais and M. P. Robillard, “Recovering traceability links between an API and its learning resources,” the 34th International Conference on Software Engineering (ICSE’12), pp.47-57, 2012.

[15] X. Chen, “Extraction and visualization of traceability relationships between documents and source code,” the 25th IEEE/ACM International Conference on Automated Software Engineering, pp.505–510, 2010.

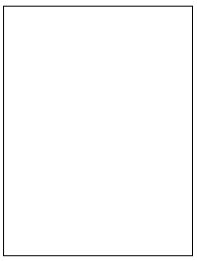
[16] A. De Lucia, R. Oliveto, and G. Tortora, “ADAMS re-trace: traceability link recovery via latent semantic indexing,” the 30th International Conference on Software Engineering (ICSE’08), pp.839–842, 2008.

[17] H. Kaiya, A. Osada, K. Hara and K. Kaijiri, “Design, implementation and evaluation of a system for finding change impacts on source codes caused by requirements changes,” IEICE Trans D, vol.J93-D, no.10, pp.1822-1835, 2010.

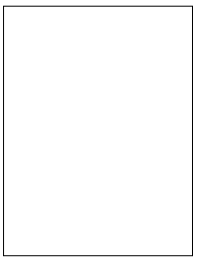
[18] CUnit, http://sourceforge.net/projects/cunit/

[19] Apache Subversion, http://subversion.apache.org/

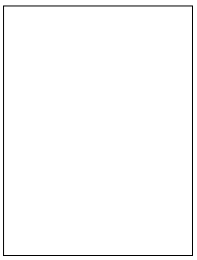
[20] CCFinderX, http://www.ccfinder.net/



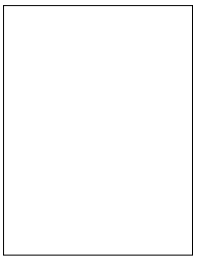
Ryosuke Tsuchiya received the B.E. degree in Information and Computer Science from Waseda University, Tokyo, Japan in 2013. He is now a master course student of Department of Information and Computer Science, Waseda University. His research interests include software engineering especially software traceability.



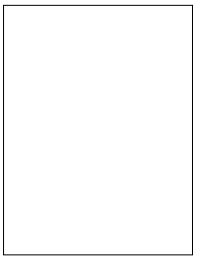
Hironori Washizaki is an associate professor at Waseda University, Tokyo, Japan. He is also a visiting associate professor at National Institute of Informatics, Tokyo, Japan. He obtained his Doctor's degree in Information and Computer Science from Waseda University in 2003. His research interests include software reuse, patterns and quality assurance. He has served as members of program committee for many international conferences including ASE, SEKE, PROFES, APSEC and PLoP. He has also served as members of editorial board for several journals including Journal of Information Processing.



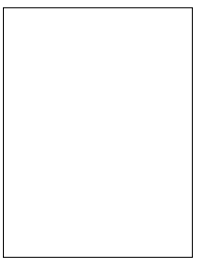
Yoshiaki Fukazawa received the B.E., M.E. and D.E. degrees in electrical engineering from Waseda University, Tokyo, Japan in 1976, 1978 and 1986, respectively. He is now a professor of Department of Information and Computer Science, Waseda University. Also he is Director, Institute of Open Source Software, Waseda University. His research interests include software engineering especially reuse of object-oriented software and agent-based software.



Tadahisa Kato received the B.S. and M.S. degrees in mathematics from Tokyo Institute of Technology, Tokyo, Japan, in 2003, 2005. He is currently a researcher at Yokohama Research Laboratory, Hitachi Ltd, japan. He is working on research of software development methods and applying these methods to actual product development. His research interests include software product line, model-based development and formal method.



Masumi Kawakami received the B.E. and M.E. degrees in knowledge-based information engineering from Toyohashi University of Technology, Aichi, Japan, in 1998. He is currently a senior researcher at Yokohama Research Laboratory, Hiatachi Ltd, Japan. He is working on research of software development method and applying these method to actual product development. His research interests include model-based development, software product line and test automation.



Kentaro Yoshimura is a Senior Researcher of Hitachi Research Laboratory at Hitachi, Ltd. Yoshimura received his Ph.D. degree in Information Science and Technology from Osaka University in 2009. His research interests are centered on software product line engineering and legacy software system analysis.