Recovering Traceability Links between Requirements and Source Code in the Same Series of Software Products

Ryosuke Tsuchiya
Dept. Computer Science
Waseda University
Tokyo, Japan
ryousuke_t@asagi.waseda.jp

Hironori Washizaki
Dept. Computer Science
Waseda University
Tokyo, Japan
washizaki@waseda.jp

Yoshiaki Fukazawa
Dept. Computer Science
Waseda University
Tokyo, Japan
fukazawa@waseda.jp

Tadahisa Kato
Yokohama Research Laboratory
Hitachi, Ltd
Kanagawa, Japan
tadahisa.kato.en
@hitachi.com

Masumi Kawakami
Yokohama Research Laboratory
Hitachi, Ltd
Kanagawa, Japan
masumi.kawakami.ch
@hitachi.com

Kentaro Yoshimura
Hitachi Research Laboratory
Hitachi, Ltd
Ibaraki, Japan
kentaro.yoshimura.jr
@hitachi.com

ABSTRACT

If traceability links between requirements and source code are not clarified when conducting maintenance and enhancements for the same series of software products, engineers cannot immediately find the correction location in the source code for requirement changes. 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 or specific to the products. 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 will be improved, and software product line introduction will be facilitated.

Categories and Subject Descriptors

D.2.7 [**Distribution, Maintenance, and Enhancement**]: Extensibility

General Terms

Management

Keywords

Traceability Recovery, Configuration Management Log,

Commonality and Variability Analysis

1. INTRODUCTION

Traceability in software development is the ability to trace the relationship between artifacts. This relationship is called traceability link. Traceability links are formed between the following pairs: documents of requirements specification 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 [13] (the target of our evaluation experiments in this paper), the requirement "Running tests in Automated mode" links with the file *Automated.c* that implements the requirement.

If traceability links between requirements and source code are ambiguous, development efficiency is significantly reduced because engineers cannot immediately modify the source code when there are change requests of requirements.

In some cases, traceability decreases through software maintenance and incremental development. Documents are not managed properly because of the short interval of the software release. For example, if documents are not updated when deleting existing features or adding new features, a mismatch arises between documents and source code. This may cause a decrease in software maintainability.

As described above, traceability is important in development. Therefore, it is ideal that clear traceability links are formed between requirements and source code. However, traceability links in actual products are typically not managed because engineers tend to not recognize the benefits of traceability links and worry about management costs [1].

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

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 configuration management log is large, so we refine the links by conducting the commonality and variability analysis.

Our proposed method is semi-automatic: with regard to recovered links 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.

The following are the Research Questions addressed in this study.

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

RQ2 How many non-explicit traceability links can we manually refine from candidate links?

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

In order to evaluate the validity our framework, 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, we found that our framework is effective in the recovery of traceability links regardless of the size of products and 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 software products within a reasonable amount of time.
- We have proposed a framework including the process to recover traceability links using the tool mentioned above.
- We have applied the framework to actual products that have more than 60KLOC, and have confirmed its validity.

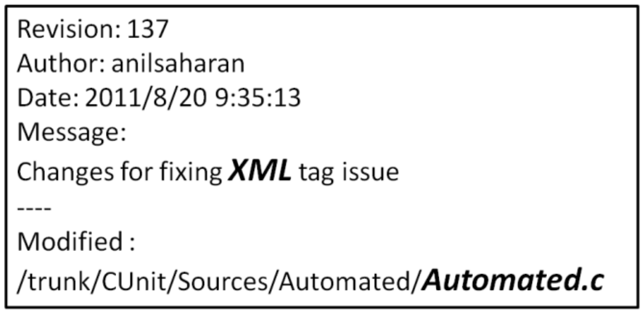
Our framework classifies requirements and source code as common to some products or as specific to a product, and recovers links between these elements. Therefore, our framework may support extraction of core assets with high reusability. As a result, software product line introduction will be facilitated.

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

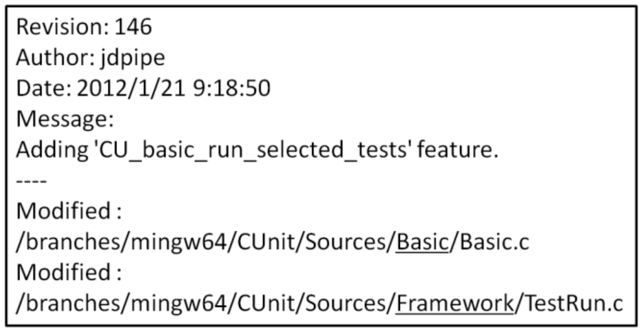
2.1 Configuration Management Log

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 is easy. However,



```
Revision: 137
Author: anilsaharan
Date: 2011/8/20 9:35:13
Message:
Changes for fixing XML tag issue
----
Modified :
/trunk/CUnit/Sources/Automated/Automated.c
```

Figure 1. Revision modifying a single file



```
Revision: 146
Author: jdpipe
Date: 2012/1/21 9:18:50
Message:
Adding 'CU_basic_run_selected_tests' feature.
----
Modified :
/branches/mingw64/CUnit/Sources/Basic/Basic.c
Modified :
/branches/mingw64/CUnit/Sources/Framework/TestRun.c
```

Figure 2. Revision modifying multiple files

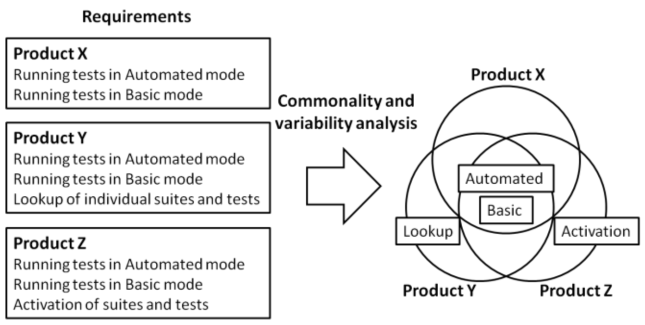

Requirements
Product X
Running tests in Automated mode
Running tests in Basic mode
Product Y
Running tests in Automated mode
Running tests in Basic mode
Lookup of individual suites and tests
Product Z
Running tests in Automated mode
Running tests in Basic mode
Activation of suites and tests
Commonality and variability analysis
Product X
Automated
Basic
Lookup
Activation
Product Y
Product Z

Figure 3. Commonality and variability analysis

it is often the case that the notation and language are different between requirements and source code. 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. In the above cases, it is difficult to recover traceability 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) [14]. Figures 1 and 2 show excerpts from a log of CUnit as specific examples of the revision of SVN. They show that each revision has a message and file paths. 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 links between requirements and source code without depending on the notation.

The configuration management log is recorded whenever the source code is modified, so it may record information on traceability links that cannot be grasped from documents. Therefore, it is possible to recover non-explicit traceability links using the log.

Some revisions simultaneously modify files of multiple domains in the configuration management log. Here, domain is a directory that has files implementing the same feature. For example, in the revision shown in Figure 2, files of the *Basic* and *Framework* domains are modified at the same time. When this kind of revisions is used to recover traceability links, unrelated requirements and files may be linked. However, because requirements related to files of multiple domains exist, these revisions cannot be ignored. Therefore, we classify revisions into different types based on the number of domains they affect.

2.2 Commonality and Variability Analysis

Ttraceability 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.

The CVA is used to analyze to which products elements (e.g., requirements, code elements) belong. The CVA classifies elements as common to some products or as specific to a product. Figure 3 shows a concrete example: the requirements “Running tests in Automated mode” and “Running tests in Basic mode” are common to three products, whereas the requirement “Activation of suites and tests” belongs to the product Z only.

Kumaki et al. have proposed a method that analyzes the commonality and variability of the requirements of legacy software products [2]. The method measures the similarity of sentences using the vector space model, and analyzes whether the requirements are common to multiple products.

In the vector space model proposed by Salton et al. [3], a sentence is represented by one vector that depends on the valid words in the sentence. The contents of the sentences are determined by the direction of the vector.

There is a previous study that measures software similarity and analyzes the commonality and variability of code elements using code clone detection [4]. A code clone is a code fragment that is identical or similar to another in a source code. Software similarity can be measured by detecting code clones that exist in different products.

If a requirement is linked to a file, the result of the CVA of the requirement may coincide with that of the file. If there is a mismatch, however, the information can be used to recover traceability links between requirements and functions. Moreover, it makes the detection of mismatches between documents and source codes possible.

2.3 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 it is not easy to associate them by comparing them. In the network control system used as a target of our evaluation experiments, while the identifier of code elements is written in English, the requirements are written in Japanese.

In most products, non-explicit traceability links exist. 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. The 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 a lot of traceability links that engineers have not grasped because the number of requirements and files is quite large.

Traceability links between requirements and functions cannot be recovered by using only the file paths in the configuration management log. As explained above, 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*. If we use only file paths, we recover a link between the requirement and the file, but not the functions. However, we may recover links with the functions using the CVA because the requirement and the functions are specific to version 2.12 whereas the file is common to multiple versions.

3. TRACEABILITY LINKS RECOVERY FRAMEWORK

We propose a framework to recover traceability links between requirements and source code in the same series of large software products. Our framework to recover links does not depend on the similarity of the representation between requirements and code elements. We find source code that implements requirements from the configuration management log using keywords set for each requirement. Moreover, we automatically refine links using CVA results, and also review and refine links manually.

3.1 Overview

Figure 4 shows the overview of our framework. 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 requirements of products written in natural language. For automatic analysis, a document that lists all requirements one line at a time and a summary of each requirement are required. In addition, a configuration management log that supports all targeted products is needed.

Traceability links between requirements and source code can be recovered by finding revisions that contain words related to requirements in the configuration management log. For refining these traceability links, the CVA is conducted prior to recovering traceability links. Our design of the framework is divided into the following seven steps.

Step (1). CVA of Requirements

The CVA of requirements is conducted by entering the assets of requirements for each product. The vector space model is used as the element technology.

Step (2). CVA of Code Elements

The CVA of code elements is conducted by entering the source code for each product. The results of the analysis are outputted at the granularity of both components (e.g., class, file) and functions. Code clone detection is used as the element technology.

Step (3). Keyword Setting

Keywords are set for each requirement to find the revisions that contain words related to the requirement. Then, TF-IDF (Term Frequency and Inverse Document Frequency) supports the keyword setting.

Step (4). Classification of Revisions

Revisions are classified based on the number of domains they affect.

Step (5). Recovery of Traceability Links between Requirements and Components

Traceability links between requirements and components are recovered by finding the relevant revisions using keywords set in Step (3). The results are outputted as a traceability matrix.

Step (6). Auto Refine of Traceability Links

Traceability links between requirements and functions are recovered using the results of the CVA, and mismatches between documents and source code are detected.

Step (7). Manual Refine of Traceability Links

Engineers review the messages of revisions to check for the existence of false positives. If there are a large number of false positives, keywords that cause them are identified and the process goes back to Step (3), where new suitable keywords are set.

The following sections describe each step in detail.

3.2 CVA of Requirements

In this step, the CVA of requirements is conducted. Assets of requirements are required as input. We use a method proposed by Kumaki et al. By measuring the similarity of sentences between the requirements in each product, each requirement is classified as being common to some products or as being specific to one product. If the similarity exceeds a threshold set by the users, requirements are judged to be identical. The result of classification is represented as a subset of a set of all products targeted. For instance, in Figure 3, a 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, the CVA of code elements is conducted. Source code is needed as input. The analysis is conducted at the granularity of both components and functions. In the same way as requirements, code elements are classified as either common or specific.

As with the previous study (mentioned in section 2.2), we use code clone detection to analyze commonality and variability of code elements. The following describes how to measure the similarity of code elements. N is the number of code elements that share code clones. E_i represents the total number of tokens for each code element. T_i represents the number of tokens of code clones for each code element. Then, S (the similarity of the code elements) is defined by the following formula:

$$S = \frac{\sum_{i=1}^{N} T_i}{\sum_{i=1}^{N} E_i}$$

If S exceeds a threshold set by the users, these code elements are judged to be identical. This similarity measurement is conducted for all code elements that share code clones, and each code element is classified by which products it belongs to. The result of classification is represented in the same way as requirements.

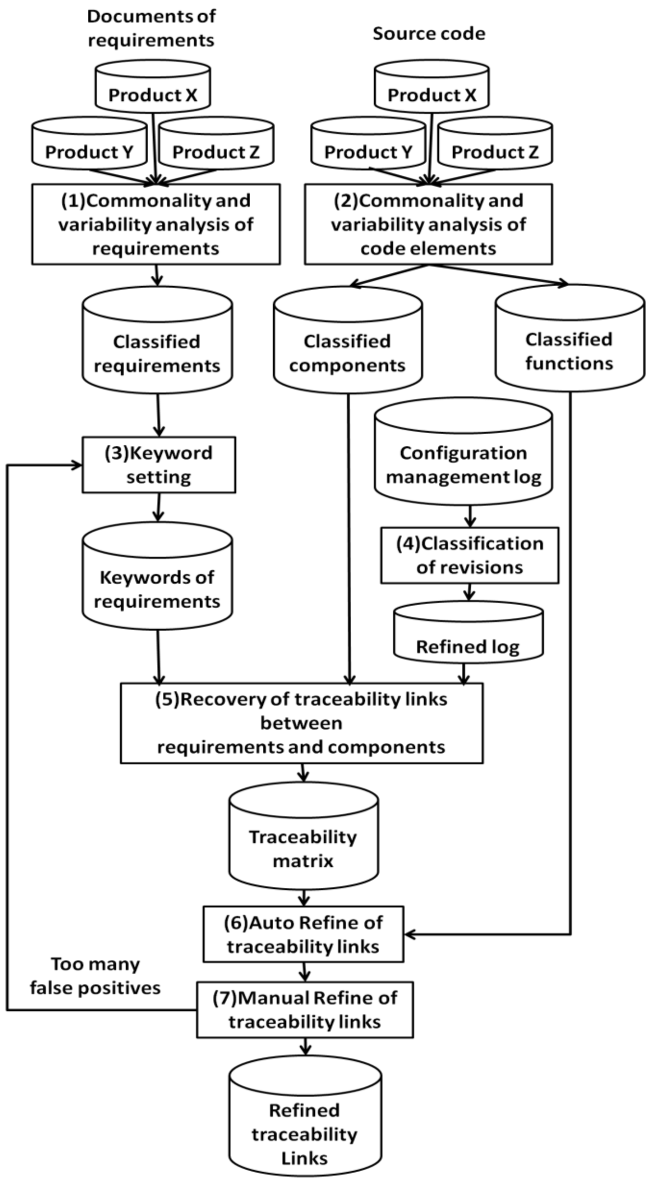

Documents of requirements
Source code
Product X
Product Y
Product Z
Product X
Product Y
Product Z
(1)Commonality and variability analysis of requirements
(2)Commonality and variability analysis of code elements
Classified requirements
Classified components
Classified functions
(3)Keyword setting
Configuration management log
(4)Classification of revisions
Keywords of requirements
Refined log
(5)Recovery of traceability links between requirements and components
Traceability matrix
(6)Auto Refine of traceability links
Too many false positives
(7)Manual Refine of traceability links
Refined traceability Links

Figure 4. Overview of our framework

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. If domain knowledge is not enough, users set the keywords with the following in mind.

- Words included in the requirement itself are preferred.
- General words used extensively are a cause of false positives.
- Keywords should not duplicate in multiple requirements.

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. In order to extract useful information while avoiding false positives, we classify revisions into the following three types based on the number of domains they affect.

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. If the threshold is greater than two, the revision in Figure 2 is classified as this type.

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 guidline, 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 between Requirements and Components

3.6.1 Traceability Links Recovery Method

In this method, revisions that have message containing the keywords set in Step (3) 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. 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". Therefore, the word "XML" is a keyword of this requirement. The method searches for revisions that have a message containing the word "XML" in the configuration management log, such as 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 component should belong to the same group of products as classified by the CVA. If not, this information can be used to refine traceability links. We classify traceability links into five types using the results of the CVA. 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:

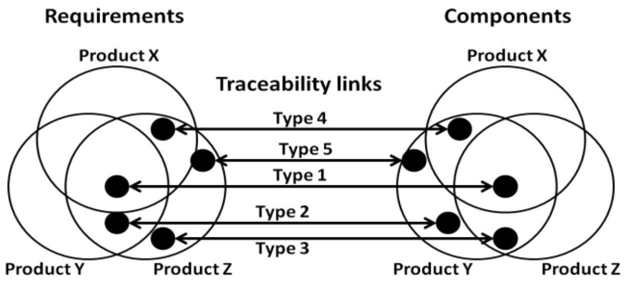


Figure 5. Types of traceability links

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

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:

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

Next, φ (the relationship between R and C obtained from the configuration management log) is defined by the following:

$$\varphi: R \to C$$

$$r \in R \to \varphi(r) \in C$$

Finally, P represents the power set of the targeted products. 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:

$$I_R: R \to P$$

$$I_C: C \to P$$

$I_R(r)$ is the set of products that have the requirement r. $I_C(\varphi(r))$ is the set of products that have the component $\varphi(r)$ linked to the requirement r. Then, as a result of the comparison between $I_R(r)$ and $I_C(\varphi(r))$, traceability links are classified into the following five types. Figure 5 shows examples of when products X, Y and Z are targeted. The usage of these types is described in section 3.7.

Type1. $I_R(r) = I_C(\varphi(r))$
e.g., $I_R(r) = I_C(\varphi(r)) = \{X, Y, Z\}$

Type2. $I_R(r) \supset I_C(\varphi(r))$
e.g., $I_R(r) = \{Y, Z\}$, $I_C(\varphi(r)) = \{Y\}$

Type3. $I_R(r) \subset I_C(\varphi(r))$
e.g., $I_R(r) = \{Z\}$, $I_C(\varphi(r)) = \{Y, Z\}$

Type4. $\left(I_R(r) \not\subseteq I_C(\varphi(r))\right) \wedge \left(I_R(r) \not\supseteq I_C(\varphi(r))\right) \wedge \left(I_R(r) \cap I_C(\varphi(r)) \neq \emptyset\right)$
e.g., $I_R(r) = \{X, Z\}$, $I_C(\varphi(r)) = \{X, Y\}$

Type5. $\left(I_R(r) \not\subseteq I_C(\varphi(r))\right) \wedge \left(I_R(r) \not\supseteq I_C(\varphi(r))\right) \wedge \left(I_R(r) \cap I_C(\varphi(r)) = \emptyset\right)$
e.g., $I_R(r) = \{Z\}$, $I_C(\varphi(r)) = \{Y\}$

	Activation of suites and tests	Adding suites to the registry	Adding tests to suites	Behavior upon framework errors	Cleanup the test registry	Error handling	Getting test results	Initialization the test registry	Lookup of individual suites and tests	Modifying general runtime behavior	Modifying other attributes of suites and tests	Running tests in automated mode	Running tests in basic mode	Running tests in interactive console mode	Running tests in interactive curses mode
/Automated/Automated.c	3B	1B	1B						3B		3B	1A			
/Basic/Basic.c	3B		1B				1B			3B	3B	1B	1A	1B	1B
/Console/Console.c [2.1-2]	1B		2B								1B			▨	
/Console/Console.c	5B		2B								5B			▨	
/Curses/Curses.c	3B		1B								3B				▨
/Framework/CUError.c				▨		▨									
/Framework/MyMem.c	3B	1A	1A		1A	1A		1A			3B	1B			
/Framework/TestDB.c	3A	1A	1A		1A	1A		1A	3B		3A				
/Framework/TestRun.c	3A	1A	1A		1A	1A	1A	1A	3B	3A	3A	1A	1A	1A	1A
/Framework/Util.c	3B		1B								3B				

Figure 6. Traceability matrix

3.6.3 Output Format

The result of the recovery of traceability links is outputted in matrix form as shown in Figure 6. It shows the presence of the relationship (namely the traceability link) between each requirement and each component. Each traceability link has the following two pieces of information.

- Type of traceability link
- Type of revision from which the link is recovered

For instance, in Figure 6, the traceability link between the requirement "Running tests in automated mode" and the component *Automated.c* is expressed as "1A". This means that the link is recovered as Type 1 from the revision of Type A. The revision number is also written for each link, but is omitted in Figure 6.

3.7 Auto Refine of Traceability Links

In this step, we refine traceability links using the classification in Step (5) (section 3.6.2).

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 7 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 traceability links of Type 3 are recovered, functions of the component whose results of the CVA are the same as those of the requirement are demonstrated to the 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) Detection of mismatches between documents and source code
Traceability links of Types 2 and 3 sometimes suggest mismatches between documents and source code.
For example, in products {X, Y, Z}, if a requirement that belongs to products {X, Y, Z} links with a component that belongs to products {Y, Z}, the product X is contradictory. It could be that in the document of the product X, an unimplemented feature is written, or a feature that has been removed from the source code remains undeleted. The detection of these mismatches is important for the management of traceability.

4) 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 traceability 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.

4. EVALUATION

We carried out experiments targeting two groups of products, which are different in terms of their size and development team. We used our tool implemented in Java and the code clone detection tool CCFinderX [15]. The following sections describe the experiments for each target.

4.1 CUnit

4.1.1 Experimental Overview

CUnit is a testing framework of C. It is open source software. We experimented with three versions of CUnit. Table 1 shows SLOC, the number of requirements and the release date of each version. The requirements are extracted from the user manual. We used the log of SVN as the configuration management log. The number of components is 9. The number of revisions in this log is 156. In addition, we evaluated the validity of our results using the traceability links mentioned in the user manual.

First, we recovered traceability links between 15 requirements and 9 components by conducting Steps (1) ~ (5). To study the impact of the keyword setting on the accuracy of traceability links, we conducted the following two patterns for Step (3):

Pattern 1. The five words with the highest TF-IDF values are set as the keywords of each requirement.

Pattern 2. Some words out of the twenty words with the highest TF-IDF values are chosen as the keywords of each requirement.

Pattern 1 is fully automatic, whereas Pattern 2 is semi automatic like our framework. We chose two to seven words as the keywords of each requirement in Pattern 2.

The threshold number of keyword appearance was set to 1 because few words are contained in revision messages.

Next, we confirmed the traceability links between requirements and functions by conducting Step (6).

Finally, we looked for the traceability links that are not mentioned in the user manual by conducting Step (7). We conducted the review on behalf of engineers.

4.1.2 Experimental Result

Table 3 shows the result of the recovery of traceability links between requirements and components. The first column, *Pat (Pattern)*, contains the pattern of the keyword setting as described in section 4.1.1. The second column, *Rev (Revision)*, contains the types of revisions used. The third column, *Rel (Relevant)*, contains the number of traceability links mentioned in the user manual. The fourth column, *Ret (Retrieved)*, contains the number of traceability links retrieved by Step (5). The fifth column, *Rel ∩ Ret (Relevant∩Retrieved)*, gives the number of traceability links that are mentioned in the user manual and retrieved by Step (5). *Rec (Recall)*, *Pre (Precision)*, and *F-m (F-measure)* are defined by the following formulas:

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

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

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

Figure 6 shows the traceability matrix obtained by the experiment using Pattern 2. The cells representing links mentioned in the user manual are shaded in gray. *Console.c* of version 2.12 is classified as being different from that of the other versions because their similarity is lower than the threshold set in Step (2).

The recovery result using the keyword setting of Pattern 2 and revisions of Type A produced the highest value of *F-measure*. Therefore, we conducted Steps (6) and (7) using this result.

3 of the 27 traceability links retrieved 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 traceability links retrieved by Step (5) were not mentioned in the user manual. By reviewing revision messages for these links, we determined that at least 5 of the links were valid. These links were concerned with the component *MyMem.c*, which

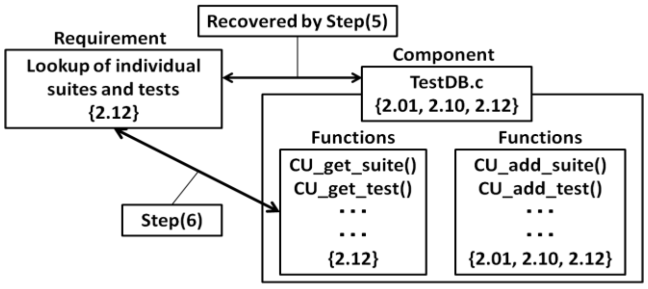

Recovered by Step(5)
Requirement
Lookup of individual suites and tests {2.12}
Component
TestDB.c {2.01, 2.10, 2.12}
Functions
CU_get_suite() CU_get_test() . . . . . . {2.12}
Functions
CU_add_suite() CU_add_test() . . . . . . {2.01, 2.10, 2.12}
Step(6)

Figure 7. Recovery of traceability links between requirements and functions

Table 1. CUnit

Version	SLOC	Requirements	Release Date
2.01	5931	11	September, 2004
2.10	6225	11	March, 2006
2.12	7760	15	October, 2010

Table 2. The network control system

Version	SLOC	Requirements
3.01	54579	41
3.02	55281	48
3.03	62448	49

Table 3. Recall and Precision on CUnit

Pat	Rev	Rel	Ret	Rel ∩ Ret	Rec	Prec	F-m
1	A, B	20	55	14	70.0%	25.5%	0.373
	A	20	16	5	25.0%	31.3%	0.278
2	A, B	20	48	15	75.0%	31.3%	0.441
	A	20	27	14	70.0%	51.9%	0.596

Table 4. Recall and Precision on the network control system

Thres	Rel	Ret	Rel ∩ Ret	Rec	Prec	F-m
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

manages the memory. Therefore, *MyMem.c* links with requirements regarding adding, deleting, and initializing tests and suites. 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 traceability 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 are Step (3) and (7). These steps took 30 minutes each.

4.2 Network Control System

4.2.1 Experimental Overview

We experimented with three versions of the network control system developed by a company. We targeted five modules that cover the basic features of the system. A module is a group of components. Table 2 shows SLOC and the number of requirements for each version. The SLOC in Table 2 represents the size of the five modules. The size of the entire system is 1.4 ~ 1.7 MLOC. The requirements were extracted from the design documents of features. We used the log of SVN as the configuration management log. The number of revisions in this log is 5727.

Engineers previously prepared traceability links between requirements and modules, so their granularities were larger than those of links recovered by our method. After recovering traceability links between requirements and components, we linked these requirements with the module that contains the corresponding components. This eliminated the difference in granularity.

First, we recovered traceability links between 49 requirements and 5 modules by conducting Steps (1) ~ (5). We set three different threshold numbers of keyword appearance in Step (5) to study the relationship between keyword appearance and accuracy of traceability links. The thresholds were 1, 5 and 10.

Next, in Step (6), we confirmed the traceability links between requirements and functions.

Finally, in Step (7), we looked for traceability links that were unknown to engineers. The review was conducted by engineers.

4.2.2 Experimental Result

Table 4 shows the result of the recovery of traceability links between requirements and components. The first column, *Thres (Threshold)*, contains the threshold numbers of keyword appearance. The second column, *Rel (Relevant)*, contains the number of traceability links prepared by engineers. The third column, *Ret (Retrieved)*, contains the number of traceability links retrieved by Step (5). The fourth column, *Rel ∩ Ret (Relevant∩Retrieved)*, gives the number of traceability links that are both prepared by engineers and retrieved by Step (5). These results were obtained by using revisions of Types A and B. They were almost the same as the results using only revisions of Type A because there were only a few traceability links recovered from revisions of Type B.

The value of *F-measure* was highest when the threshold number of keyword appearance was 5. Therefore, we used the result from this threshold number for Steps (6) and (7).

3 of the 17 traceability 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 traceability links retrieved by Step (5) were not mentioned by engineers. By reviewing revision messages for these links, we determined that at least 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 traceability links in the network control system, the running time of our tool was 13 minutes 36 seconds. Step (3) took approximately 2 hours. 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%. These were the results with the highest value of *F-measure*.

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 links that were unknown to engineers.

RQ2 How many non-explicit traceability links can we manually refine from candidate links?

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%. These 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, if a revision message contains the keyword, the components tied to the revision will 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.

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

In CUnit, automatic parts took 1 minute 40 seconds, and non automatic parts took about 1 hour. In the network control system, automatic parts took 13 minutes 36 seconds, and 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 Auto Refine of Traceability Links

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

In contrast, with regard to the detection of mismatches between documents and source code, we could not find practical examples. This may indicate that documents and source code are comparatively properly managed in the targeted products. However, it is also possible that our technique is not effective in the detection of mismatches, so we should show practical examples by conducting additional experiments for another target in the future.

4.3.3 Keyword Setting

We have tested two patterns of the keyword setting in CUnit. *Recall* and *Precision* in Table 3 show that the accuracy of traceability links is higher in Pattern 2 than in Pattern 1. TF-IDF has the ability to filter general words, but it is not perfect. Manual adjustment is required because the impact of false positives is large.

4.3.4 Classification of Revisions

From Table 3, we can see that by including revisions of Type B, *Recall* increases while *Precision* decreases. When we use revisions of Type B, we should keep in mind that false positives may increase, but we may be able to recover additional useful links.

4.3.5 Threshold Number of Keyword Appearance

In the network control system, we set three different threshold numbers of keyword appearance. From Table 4, we can see that as the threshold increases, *Recall* decreases while *Precision* increases. In this target, the case in which the threshold is 5 produces the highest value of *F-measure*. However, in CUnit, the threshold is limited to 1 because a threshold of over 2 causes a significant decrease in *Recall*. Therefore, the suitable threshold depends on the target, and users should adjust it accordingly.

4.4 Limitations

4.4.1 Quality 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 Limitation of the CVA

In some cases, elements common to multiple products are classified as different elements. For example, if the contents of source code are significantly modified by refactoring while its feature is not changed, code elements that have the same feature are classified as different elements because their similarity is lower than the threshold. In CUnit, *Console.c* is consistent with the above example. However, if users set a low threshold to avoid the above problem, different elements that have small amounts of code clone may be classified as identical. Therefore, we should consider that the results of the CVA have some errors, and develop countermeasures. For example, we should compare the identifier of code elements in addition to code clone detection.

4.4.3 Threats to Validity

In CUnit, we manually set the keywords for each requirement. We 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 would like to confirm the influence of having multiple people set the keywords on accuracy and costs.

The two targets we used are different in terms of software domain, size, and the development organization. These factors should not significantly affect the validity of our framework. However, these targets are both implemented in the C language. It is a threat to external validity that our evaluation has been limited to a single programming language. Although our framework does not depend on the programming language of targeted products, we should confirm that language does not affect the results by applying our framework to targets implemented in another language.

In both targets, we chose three versions. We have not confirmed the impact of the number of products on the accuracy of the recovery, which is a threat to external validity. Therefore, we should confirm this by applying our framework to more versions.

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 a benchmark in order to evaluate our framework more accurately.

5. RELATED WORK

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 [5]. They compare the identifier in source codes and 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.

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 [6]. 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 traceability links [7][8]. 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 previously proposed methods.

Kaiya et al. have proposed a method to find change impacts on source codes caused by requirements changes [9]. 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.

Settimi et al. have proposed a goal-centric approach to managing the impact of change upon the non-functional requirements (NFRs) of a software system [10]. They retrieve links between classes affected by a functional change and NFRs using a probabilistic network model. In contrast, our framework primarily targets functional requirement. However, if keywords can be set for each NFR, our framework may recover links between NFRs and source code. In the future, we should evaluate our framework using a target that contains NFRs.

Dekhtyar et al. have proposed a method of requirements tracing that uses TF-IDF vector retrieval and latent semantic indexing [11]. The requirements tracing contains tracing between high-level and low-level requirements. We have not made a hierarchy of requirements, so their method may be used to improve Step (6) of our framework (section 3.7).

Ghabi et al. have proposed an approach for validating requirements-to-code traces through calling relationships within the code [12]. In the future, we will utilize calling relationships in order to improve the accuracy of traceability recovery.

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. We have also developed a tool that automates parts of our framework. 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 of the software, thereby facilitating the subsequent derived development and introduction of product line.

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. Furthermore, we are aiming for our framework to be able to support software product line introduction.

7. ACKNOWLEDGMENTS

We thank Kentaro Kumaki for providing a prototype tool that analyzes the commonality and variability of requirements.

8. REFERENCES

[1] G. Spanoudakis and A. Zisman. Software Traceability: A Roadmap, Handbbok of Software Engineering and Knowledge Engineering. World Scientific Publishing, pp.395-428, 2005.

[2] 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.

[3] G. Salton and M. J. McGill. Introduction to Modern Information Retrieval, McGraw-Hill, New York, 1983.

[4] 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.

[5] 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.

[6] 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.

[7] X. Chen, Extraction and Visualization of Traceability Relationships between Documents and Source Code, the IEEE/ACM International Conference on Automated Software Engineering, pp.505–510, 2010.

[8] 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.

[9] H. Kaiya, A. Osada, K. Hara and K. Kaijiri. Design, Implementation and Evaluation of a System for Finfding Change Impacts on Source Codes Caused by Requirements Changes, IEICE Trans D, vol.J93-D, no.10, pp.1822-1835, 2010.

[10] R. Settimi, O. BenKhadra, E. Berezhanskaya and S. Christina, Goal-Centric Traceability for Managing Non-Functional Requirements, the 27th International Conference on Software Engineering (ICSE’05), pp.362-371, 2005.

[11] A. Dekhtyar and S. K. Sundaram, Advancing Candidate Link Generation for Requirements Tracing: The Study of Methods, IEEE Transactions on Software Engineering, vol.32, no.1, pp.4-19, 2006.

[12] A. Ghabi and A. Egyed, Code Patterns for Automatically Validating Requirements-to-Code Traces, the 27th IEEE/ACM International Conference on Automated Software Engineering, pp.200-209, 2012.

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

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

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