IEICE TRANS. INF. & SYST., VOL.E102–D, NO.9 SEPTEMBER 2019

PAPER

Recovering Transitive Traceability Links among Various Software Artifacts for Developers*

Ryosuke TSUCHIYA†a), **Kazuki NISHIKAWA**††, *Nonmembers*, **Hironori WASHIZAKI**††, **Yoshiaki FUKAZAWA**††, *Members*, **Yuya SHINOHARA**††, **Keishi OSHIMA**†, *and* **Ryota MIBE**†, *Nonmembers*

SUMMARY Traceability links between software artifacts can assist in several software development tasks. There are some automatic traceability recovery methods that help with managing the massive number of software artifacts and their relationships, but they do not work well for software artifacts whose descriptions are different in terms of language or abstraction level. To overcome these weakness, we propose the Connecting Links Method (CLM), which recovers transitive traceability links between two artifacts by intermediating a third artifact. In order to apply CLM for general use without limitation in terms of software artifact type, we have designed a standardized method to calculate the relation score of transitive traceability links using the scores of direct traceability links between three artifacts. Furthermore, we propose an improvement of CLM by considering software version. We evaluated CLM by applying it to three software products and found that it is more effective for software artifacts whose language type or vocabulary are different compared to previous methods using textual similarity.

key words: *traceability link recovery, transitive, connecting link, version*

1. Introduction

Traceability in software development is the ability to trace relationships between software artifacts (e.g., requirements, designs, source code, and test cases). We call these relationships "traceability links." Grasping traceability links is instrumental in several software development tasks, such as impact analysis, program comprehension, and reuse of existing software [2]–[4].

However, most software development projects have difficulty managing traceability links because of the massive number of possible combinations between software artifacts. Therefore, developers are often forced to expend significant effort and cost to perform the aforementioned tasks without traceability links. To solve this problem, various methods to automatically recover traceability links have been developed [13]–[27].

These methods utilize textual similarity among software artifacts to recover the traceability links between them, and they work quite well in recovering links between software artifacts whose language type and vocabulary are the same. However, if there are differences, these methods do not work well. Some assistive technologies have been proposed to improve the accuracy of link recovery in such situations, but they are often limited to certain types of software artifacts or require specific preconditions. Therefore, developers cannot easily apply them for general use.

To overcome these problems, we propose the Connecting Links Method (CLM), which recovers transitive traceability links between two artifacts by intermediating a third artifact. This method is based on the idea that traceability links can be recovered transitively by tracking links among multiple software artifacts. For example, if there is a requirement "Reserve tickets" and a source code file "RSVTCT.java" for implementing the requirement, it is difficult to recover links between the two using only their textual information because the name of the source code file is an abbreviation of the requirement. However, if there is a third artifact, such as a design document that includes information about both the requirement and the source code file, a traceability link can be recovered transitively after passing from the requirement to the design document and then from the design document to the source code file. In this work, we have designed a standardized method to recover links transitively among three sets of software artifacts.

This paper addresses the following research questions.

RQ1 What kind of software artifacts can CLM be applied for effectively?

RQ2 What benefits and drawbacks does the proposed scoring design give for accuracy of transitive links?

RQ3 Can consideration of software version improve the accuracy of CLM?

We conducted three experiments using three software products to investigate these questions and confirmed that CLM is effective for use with software artifacts whose language type or vocabulary are different. We also observed and clarified the relationships between the score of links recovered transitively and the reliability of these links. Finally, we proposed and evaluated an improvement of CLM by considering software version.

The contributions of this study are:

Manuscript received October 1, 2018.
Manuscript revised February 8, 2019.
Manuscript publicized June 7, 2019.

†The authors are with the Research & Development Group, Center for Technology Innovation - Systems Engineering at Hitachi, Ltd., Yokohama-shi, 244–0817 Japan.

††The authors are with the Dept. Computer Science at Waseda University, Tokyo, 169–8555 Japan.

*This paper is an extended version of a paper presented at the 31st IEEE International Conference on Software Maintenance and Evolution [1]. We have added some detailed descriptions that explain how to determine and prepare intermediate artifacts and direct links in Sects. 2 and 3. Moreover, we propose an improvement of CLM by considering software version. We also conducted experiments using two additional software products, including an industrial one.

a) E-mail: ryosuke.tsuchiya.rs@hitachi.com

DOI: 10.1587/transinf.2018EDP7331

- We propose a transitive traceability recovery method, CLM, without the limitation of software artifact types.
- We evaluate the kind of software artifacts to which CLM can be effectively applied, the impact on accuracy by the scoring design of CLM, and the effectiveness of considering software version for CLM.

Section 2 of this paper provides background information. Section 3 describes our approach, and in Sect. 4 we evaluate it experimentally. Section 5 discusses related works. We conclude in Sect. 6 with a brief summary and mention of future work.

2. Background

2.1 Traceability Link Recovery

To improve the efficiency of software development tasks, traceability links between various kinds of software artifacts need to be recovered. Software artifacts consist mainly of two elements: one, a word in natural language, and two, a symbolic token that is part of a programing language. The composition ratio is different depending on the type of artifact. For example, most of the requirement specification documents are written in natural language, while source code files typically consist of symbolic tokens (except for comments). There are some artifacts in which words and tokens coexist: API documents, bug reports, commit logs, and so on. The difference of the composition ratio makes it difficult to recover traceability links between different software artifacts.

Traceability links between software artifacts are divided broadly into two categories: probabilistic links and deterministic links.

Probabilistic links are constructed by scoring the degree of some kind of relationship between software artifacts. The most common degree of relationship is textual similarity, which has been adopted by most of the previous traceability recovery methods [13], [15], [16], [18], [19], [21]–[27]. These methods calculate textual similarity between software artifacts by using Natural Language Processing (NLP) techniques such as the Vector Space Model (VSM) [9], [10], Latent Semantic Indexing (LSI) [11], and word embedding [12]. Therefore, when recovering links between software artifacts that contain a lot of natural language words, the probabilistic links are reliable with high accuracy. However, if language type and vocabulary are different between artifacts, these methods can barely construct the probabilistic links.

Deterministic links are constructed by tracking the references to identifiers. Source code files written in a programming language refer to each other and the reference is clearly described (e.g., call relationships). Other examples contain a reference to source code files or various internal elements (e.g., class, method, and field): commit logs, API documents, bug reports, application execution logs, and so on. Compared to probabilistic links, these reference relationships are clear and deterministic. Therefore, most of the previous studies utilize deterministic links to improve the reliability of probabilistic links [14]–[19], [21]–[24].

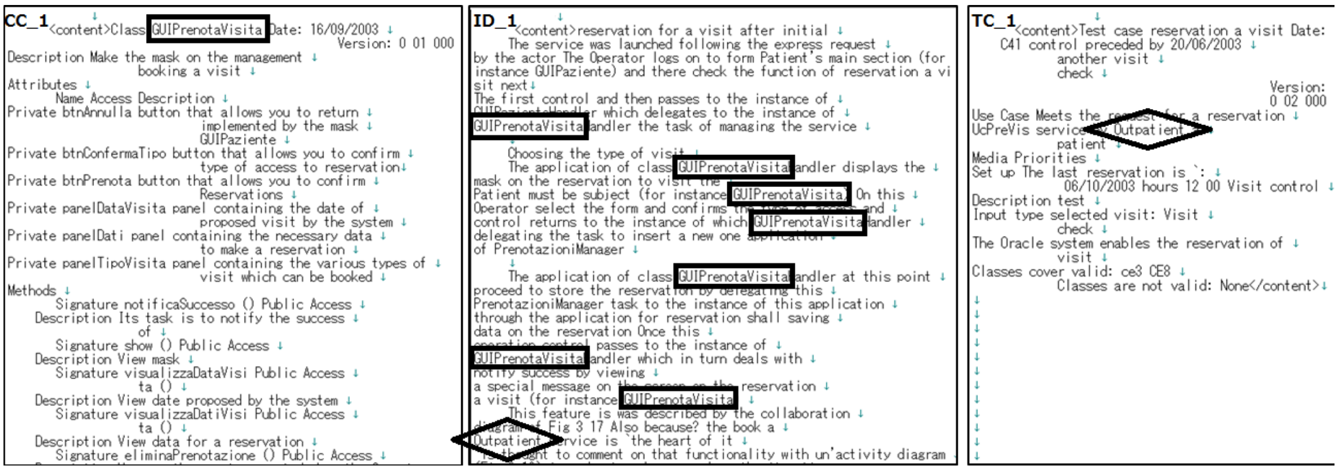
2.2 Problems

As mentioned above, the accuracy of recovering probabilistic links depends on the commonality of language type and vocabulary between artifacts. If developers need to grasp traceability links between artifacts that do not share common words, assistive technologies are required. Previous studies (discussed in more detail in Sect. 5) have proposed several assistive technologies, including structural [15]–[19], [23], repository-based [14], [21]–[24], feedback-based [19], [23], [25], and version-based [18], [20]–[22] approaches. However, developers can apply these technologies only to projects that fulfill their prerequisites for target software artifacts or the way to manage artifacts. Furthermore, even in similar approaches, there are some key differences in the way to utilize the assistive technologies among the previous studies; in other words, the usage is not standardized. This makes it a bit difficult for developers to decide which usage should be adopted for their projects. Therefore, we have aimed to develop a technology that has few limitations and is standardized to minimize the influence from diversity of target software artifacts.

2.3 Motivating Example

As shown in Fig. 1, there are three types of software artifacts in the EasyClinic software product: descriptions of source code classes, descriptions of the interaction diagram, and test cases. CC_1 located on the left is a description of the class "GUIPrenotaVisita," and TC_1 located on the right is a test case for "reservation a visit Date." According to documents that record the traceability links of EasyClinic, CC_1 links to TC_1. However, if developers want to recover the link by using textual similarity, it becomes difficult because they do not share characteristic words (e.g., "GUIPrenotaVisita" emphasized in CC_1 and "Outpatient" emphasized in TC_1). The word "reservation," which is shared between CC_1 and TC_1, is not effective for calculating textual similarity because it is used in EasyClinic universally.

On the other hand, ID_1 (located at the center of Fig. 1) is a description of the interaction diagram of applications including the class "GUIPrenotaVisita." As the emphasized words in Fig. 1 show, ID_1 contains both the characteristic words of CC_1 and TC_1. Therefore, there is a possibility of recovering the links between CC_1 and ID_1, TC_1 and ID_1 by using textual similarity. Developers can guess that there are relations between CC_1 and TC_1 by applying the transitive rule to these two links. This example forms our motivation to study the approach to recover traceability links using the transitive rule.



```
CC_1 ↓
<content>Class GUIPrenotaVisita Date: 16/09/2003 ↓
                                   Version: 0 01 000
Description Make the mask on the management ↓
                   booking a visit ↓
Attributes ↓
       Name Access Description ↓
Private btnAnnulla button that allows you to return ↓
                            implemented by the mask ↓
                            GUIPaziente ↓
Private btnConfermaTipo button that allows you to confirm ↓
                            type of access to reservation↓
Private btnPrenota button that allows you to confirm ↓
                            Reservations ↓
Private panelDataVisita panel containing the date of ↓
                            proposed visit by the system ↓
Private panelDati panel containing the necessary data ↓
                            to make a reservation ↓
Private panelTipoVisita panel containing the various types of ↓
                            visit which can be booked ↓
Methods ↓
       Signature notificaSuccesso () Public Access ↓
    Description Its task is to notify the success ↓
                   of ↓
       Signature show () Public Access ↓
    Description View mask ↓
       Signature visualizzaDataVisi Public Access ↓
                   ta () ↓
    Description View date proposed by the system ↓
       Signature visualizzaDatiVisi Public Access ↓
                   ta () ↓
    Description View data for a reservation ↓
       Signature eliminaPrenotazione () Public Access ↓
```

```
ID_1 ↓
<content>reservation for a visit after initial ↓
     The service was launched following the express request ↓
by the actor The Operator logs on to form Patient's main section (for
instance GUIPaziente) and there check the function of reservation a vi
sit next↓
The first control and then passes to the instance of ↓
GUIPazienteHandler which delegates to the instance of ↓
GUIPrenotaVisitaHandler the task of managing the service ↓
     ↓
     Choosing the type of visit ↓
     The application of class GUIPrenotaVisitaHandler displays the ↓
mask on the reservation to visit the ↓
Patient must be subject (for instance GUIPrenotaVisita) On this ↓
Operator select the form and confirms the type of access and ↓
control returns to the instance of which GUIPrenotaVisitaHandler ↓
delegating the task to insert a new one application ↓
of PrenotazioniManager ↓
     ↓
     The application of class GUIPrenotaVisitaHandler at this point ↓
proceed to store the reservation by delegating this ↓
PrenotazioniManager task to the instance of this application ↓
through the application for reservation shall saving ↓
data on the reservation Once this ↓
operation control passes to the instance of ↓
GUIPrenotaVisitaHandler which in turn deals with ↓
notify success by viewing ↓
a special message on the screen on the reservation ↓
a visit (for instance GUIPrenotaVisita ↓
     This feature is was described by the collaboration ↓
diagram of Fig 3 17 Also because? the book a ↓
Outpatient service is `the heart of it ↓
It is thought to comment on that functionality with un'activity diagram
```

```
TC_1 ↓
<content>Test case reservation a visit Date:
   C41 control preceded by 20/06/2003 ↓
           another visit ↓
           check ↓
                                         Version:
                                         0 02 000
Use Case Meets the request for a reservation ↓
UcPreVis service by Outpatient ↓
           patient ↓
Media Priorities ↓
Set up The last reservation is `: ↓
            06/10/2003 hours 12 00 Visit control ↓
Description test ↓
Input type selected visit: Visit ↓
           check ↓
The Oracle system enables the reservation of ↓
           visit ↓
Classes cover valid: ce3 CE8 ↓
           Classes are not valid: None</content>↓
↓
↓
↓
↓
↓
↓
↓
↓
↓
↓
↓
↓
```

Fig. 1 Motivating example.

3. Approach

3.1 Key Ideas

Our basic idea is that traceability links can be recovered transitively by tracking links among multiple software artifacts. We call a link recovered transitively a "transitive link" and a link recovered directly between two artifacts a "direct link." An example of the transitive link is given in Fig. 2. There are three sets of software artifacts. First Target Artifact (FTA) and Second Target Artifact (STA) are sets of artifacts that are targets of transitive link recovery, and Intermediate Artifact (IA) is a set of artifacts that are used as intermediates when recovering transitive links between FTA and STA. FTA, STA, and IA are defined as FTA = {fta_1, fta_2, . . . , fta_l}, STA = {sta_1, sta_2, . . . , sta_m}, and IA = {ia_1, ia_2, . . . , ia_n}. The suffixes l, m, and n are the number of artifacts in each set. There are direct links between fta_3 and ia_3, sta_3 and ia_3, so a transitive link between fta_3 and sta_3 can be identified by tracking the direct links.

If we limit the types to FTA, STA, IA, and the direct links between them, developers cannot utilize this approach to recover transitive links in general use scenarios. Therefore, we only specify that direct links are deterministic links or probabilistic links with a relation score; we do not place any restrictions on how to prepare or recover the direct links. Thus, our approach can be applied for several targets.

When considering transitive traceability recovery, it is necessary to discuss the reliability of transitive links. In particular, when the category of direct links is probabilistic, for both of the direct links between FTA and IA, STA and IA, high reliability (i.e., high relation score) is required.

There is another factor that can affect the reliability of transitive links: the number of transitive paths between FTA and STA. For example, if there are some artifacts of IA that are directly linked with fta_3 and sta_3 the same as ia_3—in other words, if there are multiple transitive paths between fta_3 and sta_3—we can assume that the reliability of the transitive link is higher than that of the single transitive path.

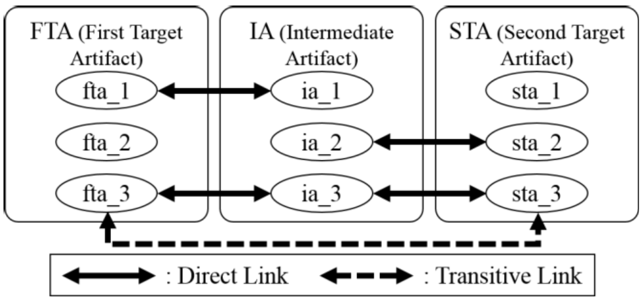

FTA (First Target Artifact)
IA (Intermediate Artifact)
STA (Second Target Artifact)
fta_1
fta_2
fta_3
ia_1
ia_2
ia_3
sta_1
sta_2
sta_3
: Direct Link
: Transitive Link

Fig. 2 Transitive link.

3.2 Connecting Links Method (CLM)

In light of the above considerations, we have designed the Connecting Links Method (CLM) to recover transitive links among various software artifacts. An overview of CLM is provided in Fig. 3. CLM consists of four steps, described in the following.

3.2.1 Step (1): Select FTA and STA

As shown in Fig. 3, this step is performed manually with software artifacts of the development project as input and it outputs FTA and STA as a result. When developers want to recover transitive links, first, they select two sets of software artifacts that are targets of transitive link recovery: FTA and STA.

CLM does not limit which types of FTA and STA can be used, but if the difference of language type and vocabulary between FTA and STA is small, we recommend recovering direct links between them using textual similarity, like the previous methods. If not, CLM can be used to recover links more accurately.

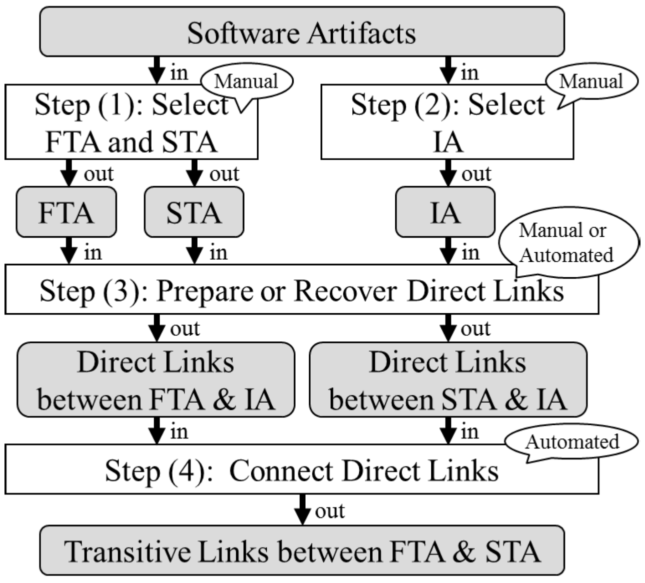

Software Artifacts
in
Manual
Step (1): Select FTA and STA
in
Manual
Step (2): Select IA
out
out
out
FTA
STA
IA
Manual or Automated
in
in
in
Step (3): Prepare or Recover Direct Links
out
out
Direct Links between FTA & IA
Direct Links between STA & IA
in
in
Automated
Step (4): Connect Direct Links
out
Transitive Links between FTA & STA

Fig. 3 Overview of CLM.

3.2.2 Step (2): Select IA

As shown in Fig. 3, this step is performed manually with software artifacts of the development project as input and it outputs IA. After execution of Step (1), the developers select a set of software artifacts that is used as intermediates of transitive recovery: IA.

We recommend choosing IA that has deterministic links with FTA or STA. For example, if FTA or STA are source code files, artifacts that have references to the source code files are suitable for IA (commit logs, API documents, and so on). We also recommend choosing IA that contains characteristic words of FTA and STA.

3.2.3 Step (3): Prepare or Recover Direct Links

As shown in Fig. 3, this step is performed manually or automatically with FTA, STA, and IA as inputs. Then, this step outputs direct links between FTA and IA, STA and IA. After execution of Steps (1) and (2), the developers prepare or recover the direct links.

CLM allows any method of recovering direct links only if the recovered links are deterministic or probabilistic links with a relation score. Examples of methods that can be used include manual recovery, tool-assisted recovery (e.g., TraceLab [5]–[8]), and applying a conventional traceability recovery method.

3.2.4 Step (4): Connect Direct Links

As shown in Fig. 3, this step is performed automatically with two sets of direct links as input and it outputs transitive links between FTA and STA. After execution of Step (3), the transitive links are automatically recovered with a relation score by connecting the direct links as in the following process.

First, the relation score of the direct link between artifacts a_1 and a_2 is normalized and defined as $\{\mathrm{DScore}(a_1, a_2) \in \mathbb{R} \mid 0 \leq \mathrm{DScore} \leq 1\}$. If the category of the direct links is deterministic, the DScore is specified as 1. If the category is probabilistic, the original relation score $\mathrm{OScore}(a_1, a_2)$ between artifacts a_1 and a_2 is normalized to DScore, as

$$\mathrm{DScore}(a_1, a_2) = \frac{\mathrm{OScore}(a_1, a_2) - \mathrm{OScore}_{min}}{\mathrm{OScore}_{max} - \mathrm{OScore}_{min}}, \tag{1}$$

where OScore_{max} and OScore_{min} are the maximum and minimum values of OScore, respectively. The original relation score OScore is respectively defined and calculated by each traceability recovery method adopted in Step (3). For example, if developers recover direct links using VSM, $\mathrm{OScore}(a_1, a_2)$ between artifacts a_1 and a_2 is calculated using the cosine similarity. The similarity is obtained as the cosine of the angle between the two document vectors. Therefore, $\mathrm{OScore}(a_1, a_2)$ is calculated as

$$\mathrm{OScore}(a_1, a_2) = \frac{\overrightarrow{v_1}\,\overrightarrow{v_2}}{\left|\overrightarrow{v_1}\right|\left|\overrightarrow{v_2}\right|}, \tag{2}$$

where $\overrightarrow{v_1}$ is the document vector of a_1 and $\overrightarrow{v_2}$ is the document vector of a_2. Although the document vector can be derived in various ways, we explain the most basic way, called Bag-of-Words (BoW), as follows. Here, D represents a set of documents and T presents a set of terms. For a document $d_x (\in D)$ containing S valid terms [i.e., $t_1, t_2, \cdots, t_S (\in T)$], $w(t_p, d_x)$ $(0 \leq p \leq \mathrm{S})$ is the number of appearances of t_p in d_x. Consequently, d_x can be represented by S-dimensional vector $\overrightarrow{v_x}$ as

$$\overrightarrow{v_x} = (w(t_1, d_x), w(t_2, d_x), \cdots, w(t_S, d_x)). \tag{3}$$

Then, the relation score of the transitive link between artifacts a_1 and a_2 is defined as $\{\mathrm{TScore}(a_1, a_2) \in \mathbb{R} \mid 0 \leq T\mathrm{Score} \leq 1\}$. When recovering transitive links between fta_i and sta_j ($fta_i \in \mathrm{FTA}, sta_j \in \mathrm{STA}$), $\mathrm{TScore}(fta_i, sta_j)$ is calculated as

$$\mathrm{TScore}(fta_i, sta_j) = \sum_{k=1}^{n} \Big(\mathrm{DScore}(fta_i, ia_k) * \mathrm{DScore}(sta_j, ia_k)\Big) \tag{4}$$

where ia_k is an intermediate artifact ($ia_k \in \mathrm{IA}$) and n is the number of intermediate artifacts. As mentioned in Sect. 3.1, the reliability of transitive links depends on the reliability of both direct links. Therefore, our design stipulates that TScore be calculated by multiplying each DScore. Furthermore, the multiplied score is summed to reflect the number of transitive paths.

This simple scoring design makes it possible for developers to calculate the relation score of the transitive link in a universal way for any software artifacts. In other words, the usage of CLM is standardized except for free choice of IA and calculation methods of OScore. For example, if there are three sets of software artifacts for which published traceability recovery tools using VSM (e.g., TraceLab) are applicable, developers can recover transitive links between them easily by executing the tools and tracing the above formulas with OScore output by the tools.

3.3 Improvement by Considering Software Version

CLM has the potential to be improved by cooperating with various assistive technologies (described further in Sect. 5). The version-based approach is particularly effective in compensating for the main weakness of CLM, which is the explosive increase in the number of combinations of software artifacts. When recovering direct links between FTA and STA, the number of combinations that are evaluated is l*m, while when recovering transitive links, the number is l*n*m. Too many combinations causes noise that interferes with the score evaluation and link recovery.

Therefore, if FTA, STA, and IA are separately managed for each software version, transitive links should be recovered between the artifacts that belong to the same software version. This reduces the number of combinations and improves the accuracy of CLM. The CLM improved by considering software version is named "verCLM."

4. Evaluation

4.1 Evaluation Purposes

In Sect. 3.2.1, we suggested considering the difference of language type and vocabulary when developers decide whether to adopt CLM. Therefore, we have to evaluate and clarify whether these characteristics of software artifacts affect the applicability of CLM and the previous methods using text similarity.

In Sect. 3.2.4, we described how to calculate the relation score of transitive links considering the reliability of both direct links and the number of transitive paths. Therefore, we have to evaluate what benefits and drawbacks the scoring design gives for the accuracy of transitive links.

In Sect. 3.3, we proposed verCLM, which improves CLM by considering software version. Therefore, we have to evaluate whether verCLM actually improves CLM.

On the basis of the above, we set the following research questions.

RQ1 What kind of software artifacts can CLM be applied for effectively?

RQ2 What benefits and drawbacks does the proposed scoring design give for accuracy of transitive links?

RQ3 Can consideration of software version improve the accuracy of CLM?

4.2 Experimental Setup

We carried out three experiments using three software applications to answer the research questions. In these experiments, precision, recall, and F-measure, which take values from 0 to 1, are used as the metrics to determine the accuracy of recovering links. They are defined as

$$\text{precision} = \frac{\text{extracted corret links}}{\text{all extracted links}}, \tag{5}$$

$$\text{recall} = \frac{\text{extracted corret links}}{\text{all correct links}}, \tag{6}$$

$$\text{F} - \text{measure} = 2 * \frac{\text{precision} * \text{recall}}{\text{precision} + \text{recall}}. \tag{7}$$

4.2.1 Experiment (1): PAT

The first target software is PAT, which is a program analysis tool developed by a Japanese company. It contains two requirements written in Japanese, 251 source code files written in Java®[†], and 97 test cases written in Japanese. There are direct links prepared between the requirements and the test cases and between the source code files and the test cases. The direct links between the requirements and the test cases were recovered by developers manually. The direct links between the source code files and the test cases were recovered by referring to execution logs that record source code modules executed in test cases. Therefore, both of the direct links are deterministic.

Developers want to recover links between requirements and source code files. Therefore, we recovered direct links by applying a traceability recovery approach using textual similarity and recovered transitive links by CLM.

We adopted VSM as the NLP technique to calculate textual similarity because it is the most common approach adopted as basic technology in many previous methods. Moreover, there is a traceability recovery tool called TraceLab using VSM, which is published and available to anyone. In CLM, we selected the requirements as FTA, the source code files as STA, and the test cases as IA. Then, we use direct links between FTA and IA, STA and IA, which are mentioned above.

In this experiment, we compared the accuracy of two approaches to determine whether the difference of language type between software artifacts affects applicability of the approaches. The total number of correct links, prepared in advance by developers, is 65.

4.2.2 Experiment (2): EasyClinic

The second target software is EasyClinic, which is an open source software designed to manage a medical practitioner's office. It contains 30 use cases, 20 descriptions of interaction diagrams, 47 descriptions of source code classes, and

[†]Java® is a registered trademark of Oracle and/or its affiliates.

63 test cases. The Center of Excellence for Software & Systems Traceability (CoEST) [29], which is a "community of researchers and practitioners working together since 2002 to achieve scalable, effective software and systems traceability solutions", provided correct links between the four artifacts.

In this experiment, we investigated whether CLM can recover links more accurately for software artifacts for which the direct traceability recovery approaches using textual similarity do not work well. Therefore, we recovered direct links between the four artifacts using VSM (TraceLab) and then recovered transitive links for all combinations of the artifacts using the direct links. Then, we compared the accuracy of the direct and transitive links.

4.2.3 Experiment (3): Andlytics

The third target software is Andlytics, which is an Android™† application to collect statistics from the Google Play™†† developer console. We utilized five versions (ver2.1, ver2.2, ver2.3, ver2.4, and ver2.5). All told, ver2.1 includes four requirements, 74 pull requests, and 169 source code files; ver2.2 includes eight requirements, 130 pull requests, and 176 source code files; ver2.3 includes ten requirements, 115 pull requests, and 185 source code files; ver2.4 includes three requirements, 76 pull requests, and 189 source code files; and ver2.5 includes five requirements, 107 pull requests, and 200 source code files.

Direct links were prepared between the requirements and the pull requests and between the source code files and the pull requests. The direct links between the requirements and the pull requests were recovered by TraceLab, so they are probabilistic links with a relation score. The direct links between the source code files and the pull requests were recovered by referring to information of the modified source code files that is recorded in pull requests. Thus, these links are also probabilistic links with a relation score calculated by weighting according to the number of modified lines of code.

In this experiment, we compared the accuracy of four approaches: VSM (an approach that calculates textual similarity by VSM), verVSM (VSM improved by considering software version), CLM, and verCLM by recovering links between requirements and source code files. In CLM, we specified the requirements as FTA, the source code files as STA, and the pull requests as IA. Then, we used direct links between FTA and IA, STA and IA, which are mentioned above. VSM and CLM were applied to artifacts of all software versions in bulk, while verVSM and verCLM were applied to artifacts of each software version separately. The total number of correct links for evaluation is 22. These correct links were prepared between the requirements and the source code files of all software versions manually. All of the links were constructed between the requirements and the source code files, which were contained in same version, because the software artifacts related to each other were updated synchronously in Andlytics.

†Android™ is a trademark of Google Inc.
††Google Play™ is a trademark of Google Inc.

Table 1 Accuracy of traceability recovery in PAT.

Approach	Correct Links	Accuracy when extracting all links with scores				
		Extracted Links	Extracted Correct Links	Precision	Recall	F-measure
VSM	65	38	19	0.50	0.29	0.37
CLM		148	44	0.30	0.68	0.41

Table 2 Accuracy of traceability recovery in EasyClinic.

No.	Target Artifacts	Link Type	IA	Accuracy with highest F-measure		
				Precision	Recall	F-measure
1	UC-ID	direct	–	0.36	0.81	0.50
2		transitive	CC	0.40	0.62	0.48
3		transitive	TC	0.53	0.35	0.42
4	UC-CC	direct	–	0.61	0.59	0.60
5		transitive	ID	0.45	0.72	0.55
6		transitive	TC	0.19	0.34	0.25
7	UC-TC	direct	–	0.47	0.54	0.50
8		transitive	ID	0.25	0.57	0.35
9		transitive	CC	0.39	0.30	0.34
10	ID-CC	direct	–	0.66	0.59	0.63
11		transitive	UC	0.49	0.51	0.50
12		transitive	TC	0.26	0.45	0.33
13	ID-TC	direct	–	0.80	0.61	0.69
14		transitive	UC	0.43	0.55	0.49
15		transitive	CC	0.58	0.73	0.65
16	CC-TC	direct	–	0.40	0.52	0.45
17		transitive	UC	0.20	0.48	0.28
18		transitive	ID	0.42	0.41	0.42

4.3 Results

4.3.1 Experiment (1): PAT

Table 1 lists the accuracy of traceability recovery between requirements and source code files in PAT. VSM in the "Approach" column indicates the traceability recovery approach that calculates textual similarity by VSM. VSM and CLM respectively extracted 38 and 148 links with scores, where a sentence "with scores" means having relation scores greater than 0. Table 1 also shows the values of extracted correct links, precision, recall, and F-measure when extracting all links with scores. CLM could extract about twice as many correct links as VSM.

4.3.2 Experiment (2): EasyClinic

Table 2 lists the accuracy of traceability recovery in EasyClinic, where UC refers to Use Cases, ID to descriptions of Interaction Diagrams, CC to descriptions of source Code Classes, and TC to Test Cases. The column "Target Artifacts" indicates artifacts that are targets of traceability recovery. The column "Link Type" indicates the type of traceability link: direct or transitive. The column "IA" indicates the IA that is used as an intermediary of transitive links. For

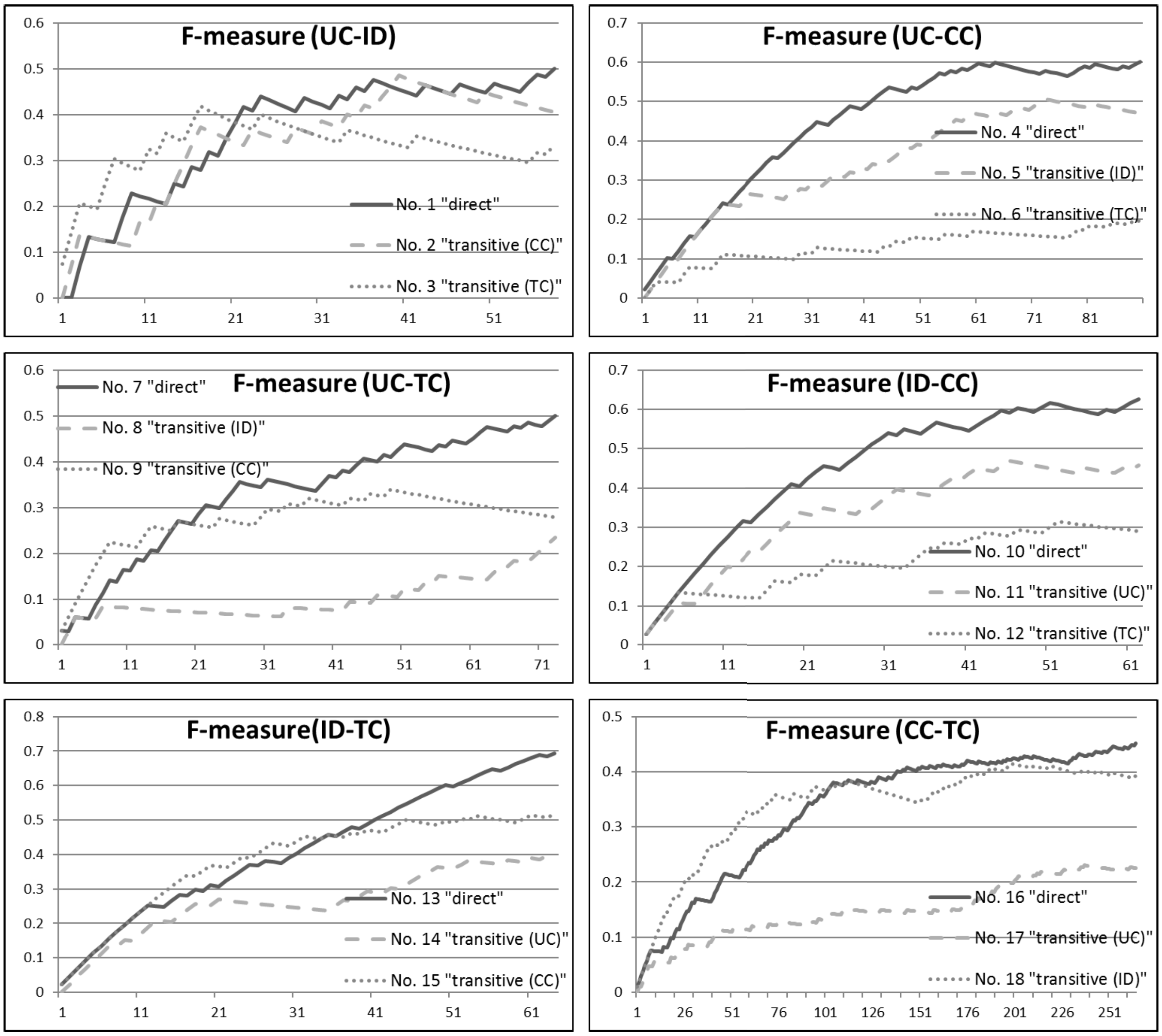


Fig. 4 F-measure of traceability recovery in EasyClinic.

example, line No. 1 indicates the accuracy of direct links between UC and ID, and line No. 2 indicates the accuracy of transitive links between UC and ID recovered by intermediating CC. We show accuracy with highest F-measure in Table 2 to compare the maximum performance among links in terms of the balance between precision and recall. As shown in the table, all direct links exceeded the corresponding transitive links in the terms.

Figure 4 shows six graphs detailing the transition of F-measure according to the number of extracted links in EasyClinic, where vertical axis indicates the value of F-measure and horizontal axis indicates the number of extracted links. Each legend corresponds to the link of each row in Table 2. For example, the top-left graph shows the transition of Nos. 1, 2, and 3 in Table 2. To make the comparison of the early stages easier to see, each graph shows the transition until the F-measure of direct links peaks. After the peak, each F-measure gradually decreased and superiority or inferiority between the direct link and the transitive links did not change. While most direct links exceeded the corresponding transitive links, some transitive links (Nos. 3, 9, 15, and 18) continuously exceeded the corresponding direct links when the number of extracted links was small.

4.3.3 Experiment (3): Andlytics

Figures 5 and 6 show the precision and recall of recovering links between requirements and source code files in Andlytics. Vertical axis indicates the value of precision or recall, and horizontal axis indicates the number of links retrieved by each approach. The links were retrieved in descending order of relation score. The results of applying four approaches are shown: VSM, verVSM, CLM, and verCLM.

CLM and verVSM had higher recall and precision than

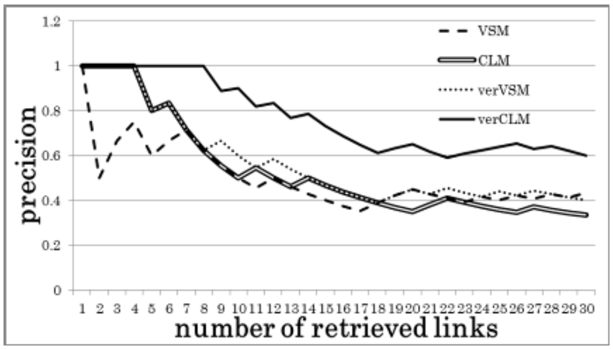


Fig. 5 Precision of traceability recovery in Andlytics.

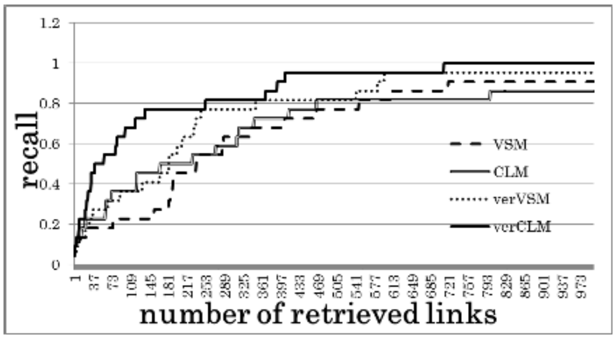


Fig. 6 Recall of traceability recovery in Andlytics.

VSM, especially when the number of retrieved links was small. However, the larger the number of retrieved links, the smaller the difference of accuracy of the three approaches. verCLM had the highest recall and precision of the four approaches regardless of the number of retrieved links.

4.4 Discussion

4.4.1 RQ1: What Kind of Software Artifacts Can CLM be Applied for Effectively?

In experiment (1), VSM extracted only 38 links with scores. In PAT, the requirements are written in Japanese and the source code files are composed of symbolic tokens based on English and a few Japanese comments, which enabled VSM to extract 38 links. However, there are a lot of source code files that contain no Japanese comments at all. In this case, VSM could not extract links for the source code files without Japanese comments. This is why VSM extracted a small number of links compared with the number of target artifacts. In contrast, CLM was not affected by the differences of language type because it utilized direct deterministic links with test cases. Therefore, CLM could extract links for the source code files without Japanese comments. As a result, CLM extracted more correct links than VSM.

In experiment (2), all direct links exceeded corresponding transitive links in accuracy with highest F-measure. We assume this is because all artifacts contain natural language words written in the same language (English), and CLM adopted direct probabilistic links. However, some transitive links (Nos. 3, 9, 15, and 18) exceeded the corresponding direct links when the number of extracted links was small. The common point between these transitive links is to mediate the direct links of which the highest F-measure is over 0.60 (i.e., Nos. 4, 10, and 13). These results show that the accuracy of transitive links depends upon the accuracy of the mediated direct links.

In conclusion, CLM can be effectively applied for recovering links between software artifacts whose language type and vocabulary are different. Furthermore, the existence of IA with deterministic or highly reliable probabilistic direct links is favorable for applying CLM.

4.4.2 RQ2: What Benefits and Drawbacks Does the Proposed Scoring Design Give for Accuracy of Transitive Links?

In multiple cases in experiments (2) and (3), the accuracy of transitive links exceeded that of direct links when the number of retrieved links was small; in other words, when the relation score of links was high. In contrast, transitive links with a low relation score were outperformed by direct links.

We conclude that this is because CLM calculates a score by summing the scores of all transitive paths. If the direct links mediated are probabilistic links, low scores of transitive paths, which are low reliability paths, are also summed. This adversely affects the accuracy of transitive links.

The scoring design that considers the number of transitive paths may preferentially recover reliable transitive links supported by multiple transitive paths with a high score. However, it needs to be improved to reduce the impact of summing low scores of transitive paths.

4.4.3 RQ3: Can Consideration of Software Version Improve the Accuracy of CLM?

In experiment (3), we confirmed that verCLM can recover transitive links more accurately than CLM. Furthermore, verCLM exceeded VSM regardless of the number of retrieved links. As a result of reducing the combinations of artifacts by considering software version, transitive paths with low score are also reduced. This contributes to the improvement of accuracy. However, if the correct links contain links between different versions, it may have affected the results because verCLM and verVSM cannot recover links between different versions. Such links may exist in a project where the update status of each set of software artifacts is not synchronized. Therefore, in practice, we should consider the existence of links between different versions when applying version-based approaches.

4.5 Threats to Validity

We have discussed and evaluated transitive traceability recovery between three sets of software artifacts. However,

Table 3 Prerequisites and weak points of assistive or alternative approaches for textual traceability recovery approaches.

Approach	Prerequisite	Weak Point
Structural	Target artifacts are limited to source code files or the artifacts having structural features equivalent to them.	A different parser or analyzer is required for each programming language to extract structural information.
Repository-based	Target artifacts have to be managed in the repository.	The effectiveness depends on the quality of repository management. If there is too little information in commit messages or the commit granularity is rough, the effectiveness tends to be low.
Version-based	Target artifacts have to be managed for each version.	When the update status of each set of software artifacts is not synchronized, links between different versions may occur. These approaches cannot recover the links.
Feedback-based	At least one developer who can validate the correctness of traceability links is necessary.	Not a few human resources are required for validation of traceability links. Furthermore, human errors that impact the accuracy of traceability recovery may occur.
CLM	The development project has to have at least one set of software artifacts that can provide direct links with both FTA and STA.	If the direct links used as transitive paths are lowly reliable probabilistic links, the reliability of transitive links also tends to be low.

we have not examined situations involving recovery across more than four sets of artifacts. In other words, we have not examined situations where multiple sets of artifacts are used as IA serially or in parallel. This is a threat to external validity because the number of software artifact types varies from software to software and there is not always IA that has direct links with both FTA and STA. Therefore, we need to investigate how the number of serial or parallel transitions affects the accuracy of transitive links in the future.

In PAT and Andlytics, we often manually prepared direct links or correct links for evaluation. This is a threat to internal validity because the accuracy of the manual recovery could affect the evaluation results. Furthermore, in each experiment, we evaluated only one product. This is also a threat to external validity. In the future, we should provide additional evaluation, for example, using a product that contains a combination of other different languages, or a product that provides correct links for evaluation in multiple combinations of software artifacts, or a product that has been developed in parallel by branching to multiple versions.

5. Related Work

Mäder et al. conducted a controlled experiment with 52 participants performing real maintenance tasks on two third-party development projects where half the tasks were with traceability and the other half were without [4]. They found that, on average, participants with traceability performed 21% faster and created 60% more correct solutions. Their empirical study affirms the usefulness of traceability links.

Traceability recovery approaches using textual similarity have been proposed in several previous methods [13], [15], [26], [27]. Antoniol et al. proposed an approach using VSM [13] that formed the basis of later studies. Approaches to improve the accuracy of calculating textual similarity have also been studied. These approaches utilize more advanced NLP technologies considering semantics, such as LSI [15] and word embedding [27]. However, if the language type of software artifacts is different (e.g., natural language vs. programming language, English vs. Japanese), these approaches do not work well. CLM has been proposed as an alternative in these situations.

Various structural approaches [15]–[19], [23] have been proposed to improve the accuracy of the textual approaches mentioned above. They utilize structural information such as call relationships of methods in source code files to filter false positives or suggest additional links. There are also several repository-based approaches [14], [21]–[24], which utilize software repository information to recover links between software artifacts. In our own previous studies [21]–[23], we proposed a method to recover links between requirements and source code files by referring to modification logs in software repositories. Both structural and repository-based approaches consider certain transitive rules. However, they have been designed for specific types of software artifacts. CLM can apply transitive rules to any types of artifacts.

Information about the software version is also utilized to improve accuracy in version-based approaches [18], [20]–[22]. The idea used here is that traceability links between software artifacts belonging to different software versions tend to be false positives. CLM particularly benefits from version-based approaches, as mentioned in Sect. 3.3 and evaluated in Sect. 4.3.3.

Feedback-based approaches [19], [23], [25] have improved traceability recovery by means of user feedback. In our own previous study [23], we proposed a method to recover links between requirements and source code files by utilizing user feedback along with structural information of source code files. Hayes et al. proposed a method to recover links between high and low level requirements by applying relevance feedback analysis to improve the performance of information retrieval algorithms [25]. In this paper, while we have not evaluated the combined use of CLM and feedback-based approaches, we do have some hypotheses about the combination. For example, if a transitive link is validated by the user, the reliability of the direct links included in the transitive path increases. Then, if the score of the direct links is weighted, the accuracy of the transitive recovery may improve.

Finally, we have organized and listed the prerequisites and weak points of assistive or alternative approaches for textual traceability recovery in Table 3, where "prerequisite" means an essential condition to apply the approach and "weak point" indicates a factor that makes developers reluctant to adopt the approach or a situation in which the approach cannot work effectively. Although we have no statistical data to clarify which prerequisites are fulfilled by more development projects, we assume that CLM can be used for more versatile purposes than structural and repository-based approaches because CLM does not limit the type of target or the intermediate software artifacts. However, like the other approaches, there are also situations in which CLM cannot work well. Therefore, we suggest that developers select or combine approaches by comparing the characteristics of their projects with the information listed in Table 3. We have mentioned combinations between CLM and the version-based approach in this paper, between repository-based and version-based approaches in previous studies [21], [22], and between repository-based and feedback-based approaches in a previous study [23].

6. Conclusion and Future Work

We have proposed the Connecting Links Method (CLM) to recover transitive traceability links and evaluated it using three software applications: PAT, EasyClinic, and Andlytics. Results demonstrate that CLM is more effective for recovering traceability links between software artifacts whose language type and vocabulary are different compared to the traceability recovery approaches using textual similarity. Furthermore, we have observed that the accuracy of transitive links with a high score tends to exceed direct links, whereas transitive links with a low score tend to be outperformed by direct links. This knowledge should be useful for improving the scoring design of CLM and for combining CLM with direct traceability recovery approaches in the future. With regard to improving CLM, we have proposed ver-CLM considering software version and evaluated the degree of improvement. In the future, we will investigate combinations of CLM and feedback-based approaches and evaluate the impact of the number of transitions on recovery accuracy.

References

[1] K. Nishikawa, H. Washizaki, Y. Fukazawa, K. Oshima, and R. Mibe, "Recovering Transitive Traceability Links among Software Artifacts," Proceedings of the 31st IEEE International Conference on Software Maintenance and Evolution (ICSME'15), pp.576–580, Oct. 2015.

[2] A. Kannenberg, H. Saiedian, "Why software requirements traceability remains a challenge," The Journal of Defense Software Engineering, pp.14–19, July 2009.

[3] O.C.Z. Gotel and A. Finkelstein, "An analysis of the requirements traceability problem," Proc. 1st International Conference on Requirements Engineering, pp.94–101, April 1994.

[4] P. Mader and A. Egyed, "Assessing the effect of requirements traceability for software maintenance," 2012 28th IEEE International Conference on Software Maintenance (ICSM'12), pp.171–180, Sept. 2012.

[5] J. Cleland-Huang, A. Czauderna, A. Dekhtyar, O. Gotel, J. Huffman Hayes, E. Keenan, G. Leach, J. Maletic, D. Poshyvanyk, Y. Shin, A. Zisman, G. Antoniol, B. Berenbach, A. Egyed, and P. Maeder, "Grand challenges, benchmarks, and tracelab: Developing infrastructure for the software traceability research community," Proceedings of the 6th International Workshop on Traceability in Emerging Forms of Software Engineering, pp.17–23, May 2011.

[6] E. Keenan, A. Czauderna, G. Leach, J. Cleland-Huang, Y. Shin, E. Moritz, M. Gethers, D. Poshyvanyk, J. Maletic, J.H. Hayes, A. Dekhtyar, D. Manukian, S. Hussein, and D. Hearn, "Tracelab: An experimental workbench for equipping researchers to innovate, synthesize, and comparatively evaluate traceability solutions," 2012 34th International Conference on Software Engineering (ICSE), pp.1375–1378, June 2012.

[7] A. Czauderna, M. Gibiec, G. Leach, Y. Li, Y. Shin, E. Keenan, and J. Cleland-Huang, "Traceability challenge 2011: Using tracelab to evaluate the impact of local versus global idf on trace retrieval," Proceedings of the 6th International Workshop on Traceability in Emerging Forms of Software Engineering, pp.75–78, Honolulu, USA, May 2011.

[8] B. Dit, E. Moritz, and D. Poshyvanyk, "A tracelab-based solution for creating, conducting, and sharing feature location experiments," 2012 20th IEEE International Conference on Program Comprehension (ICPC), pp.203–208, Passau, Germany, June 2012.

[9] G. Salton, A. Wong, and C.S. Yang, "A Vector Space Model for Automatic Indexing," Communications of the ACM, vol.18, no.11, pp.613–620, Nov. 1975.

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

[11] S. Deerwester, S.T. Dumais, G.W. Furnas, T.K. Landauer, and R. Harshman, "Indexing by Latent Semantic Analysis," Journal of the American Society for Information Science, vol.41, no.6, pp.391–407, 1990.

[12] T. Mikolov, I. Sutskever, K. Chen, G.S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," In Advances in Neural Information Processing Systems 26, pp.3111–3119, 2013.

[13] 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, Dec. 2002.

[14] H. Kagdi, J.I. Maletic, and B. Sharif, "Mining Software Repositories for Traceability Links," Proc. 15th IEEE Int'l Conf. Program Comprehension, pp.145–154, June 2007.

[15] C. McMillan, D. Poshyvanyk, and M. Revelle, "Combining textual and structural analysis of software artifacts for traceability link recovery," 2009 ICSE Workshop on Traceability in Emerging Forms of Software Engineering, TEFSE'09, pp.41–48, May 2009.

[16] C. McMillan, M. Grechanik, D. Poshyvanyk, C. Fu, and Q. Xie, "Exemplar: A source code search engine for finding highly relevant applications," IEEE Transactions on Software Engineering, vol.38, no.5, pp.1069–1087, Aug. 2011.

[17] A. Ghabi, and A. Egyed, "Code patterns for automatically validating requirements-to-code traces," Proc. 27th IEEE/ACM International Conference on Automated Software Engineering (ASE'12), pp.200–209, Sept. 2012.

[18] H. Eyal-Salman, A.-D. Seriai, and C. Dony, "Feature-to-code traceability in a collection of software variants: combining formal concept analysis and information retrieval," 2013 IEEE 14th IEEE International Conference on Information Reuse and Integration (IRI'13), pp.209–216, Aug. 2013.

[19] A. Panichella, C. McMillan, E. Moritz, D. Palmieri, R. Oliveto, D. Poshyvanyk, A. De Lucia. "When and How Using Structural Information to Improve IR-Based Traceability Recovery," 2013 17th European Conference on Software Maintenance and Reengineering (CSMR), pp.199–208, March 2013.

[20] M. Rahimi, W. Goss, J. Cleland-Huang, "Evolving Requirements-to-Code Trace Links across Versions of a Software System" Proceedings of the 32st IEEE International Conference on Software Maintenance and Evolution (ICSME '16), pp.99–109, Oct. 2016.

[21] R. Tsuchiya, T. Kato, H. Washizaki, M. Kawakami, Y. Fukazawa, and K. Yoshimura, "Recovering Traceability Links between Requirements and Source Code in the Same Series of Software Products," Proceedings of 17th International Software Product Line Conference (SPLC '13), pp.121–130, Aug. 2013.

[22] R. Tsuchiya, H. Washizaki, Y. Fukazawa, T. Kato, M. Kawakami, and K. Yoshimura, "Recovering traceability links between requirements and source code using the configuration management log," IEICE Transactions on Information and Systems, vol.E98-D, no.4, pp.852–862, 2015.

[23] R. Tsuchiya, H. Washizaki, Y. Fukazawa, K. Oshima, and R. Mibe, "Interactive recovery of requirements traceability links using user feedback and configuration management logs," Proceedings of 27th International Conference on Advanced Information Systems Engineering (CAiSE'15), vol.9097, pp.247–262, June 2015.

[24] N. Ali, Y.-G. Gueheneuc, and G. Antoniol, "Trustrace: mining software repositories to improve the accuracy of requirement traceability links," IEEE Transactions on Software Engineering, vol.39, no.5, pp.725–741, Nov. 2013.

[25] J.H. Hayes, 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, Jan. 2006.

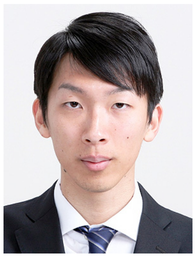
[26] T. Dasgupta, M. Grechanik, E. Moritz, B. Dit, and D. Poshyvanyk, "Enhancing software traceability by automatically expanding corpora with relevant documentation," 2013 IEEE International Conference on Software Maintenance (ICSM'13), pp.320–329, Sept. 2013.

[27] J. Guo, J. Cheng, J. Cleland-Huang, "Semantically enhanced software traceability using deep learning techniques," 2017 IEEE/ACM 39th International Conference on Software Engineering, pp.3–14, May 2017.

[28] Andlytics, https://github.com/AndlyticsProject/andlytics, Jan. 2017.

[29] CoEST, http://www.coest.org/, June. 2015.

Ryosuke Tsuchiya is a Researcher of Systems Innovation Center at Hitachi, Ltd. He received his Master degree in Computer Science and Engineering from Waseda University in 2015. His research interests are centered on legacy system analysis and traceability recovery.



Kazuki Nishikawa received the Master degree in Information and Computer Science from Waseda University, Tokyo, Japan in 2017. His research interests include software engineering especially software traceability.

Hironori Washizaki is a professor at Waseda University, Tokyo, Japan. He is also a visiting 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.

Yuya Shinohara was a junior university student of Department of Information and Computer Science, Waseda University. His research interests include software engineering especially software traceability.

Keishi Oshima is a Senior Researcher of Systems Innovation Center at Hitachi, Ltd. He received his Master degree in Information Science and Technology from Waseda University in 2002. His research interests are centered on legacy system analysis and repository mining.

Ryota Mibe is a Senior Researcher of Systems Innovation Center at Hitachi, Ltd. He received his Master degree in Information Science and Technology from Tokyo Institute of Technology in 1992. His research interests are centered on legacy system analysis and repository mining.