Finding Similar Defects Using Synonymous Identifier Retrieval

Norihiro Yoshida, Takeshi Hattori, Katsuro Inoue
Graduate School of Information Science and Technorogy, Osaka University
1-5 Yamadaoka, Suita, Osaka, 565-0871, Japan
{n-yosida, inoue}@ist.osaka-u.ac.jp

ABSTRACT

When we encounter a defect in one part of a program, it is very important to find other parts of the program that may contain similar defects. In this paper, we propose a novel system to find similar defects in the large collection of source code. This system takes a code fragment containing a defect as the query input, and returns code fragments containing the same or synonymous identifiers which appear in the input fragment. Case studies with two open source systems and their defect data show the advantages of the proposed retrieval system, compared to the code-clone based retrievals.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging aids and diagnostics*; D.2.4 [**Software Engineering**]: Software/Program Verification—*Statistical methods*

General Terms

Algorithms, Experimentation

Keywords

Code Retrieval, Natural Language Processing, Defect Detection

1. INTRODUCTION

In the early morning of October 12th, 2007, more than 4,000 automatic gate machines at Tokyo's 662 railway stations were unable to start up correctly [20]. Those gate machines, made by a Japanese vendor, failed to download daily data from a central data server during their boot-up process, and this failure caused system crashes. The vendor tried to resolve this failure immediately, but it took more than a half day to locate and fix a single-line defect in the source code, because of its complex logical structure. This failure inconvenienced approximately 2.6 million passengers.

```
ir_debug( Dmsg(10, "ProcWideReq7 start!!\n") );

// buffer overflow check is missing!
buf += HEADER_SIZE;
Request.type7.context = S2TOS(buf);
buf += SIZEOFSHORT;
Request.type7.number = S2TOS(buf);
buf += SIZEOFSHORT;
Request.type7.yomilen =(short)S2TOS(buf);
```

(a) Example of code fragment containing a buffer overflow defect

```
ir_debug( Dmsg(10, "ProcWideReq14 start!!\n") );

// buffer overflow check is missing!
buf += HEADER_SIZE;
Request.type14.mode = L4TOL(buf);
buf += SIZEOFINT;
Request.type14.context = S2TOS(buf);
buf += SIZEOFSHORT;
Request.type14.yomi = (Ushort *)buf;
```

(b) Example of code fragment which is slightly different from Figure 1(a) and also contains a buffer overflow defect

Figure 1: Similar defects in Canna 3.6 [3]

Six days after this incident, a second incidence occurred. Similar boot-up time failure happened at more than 100 fair adjustment machines in Tokyo's 65 stations, and about 400 passengers were troubled. Those machines, also made by the same vendor, share a similar boot-up process as the automatic gate machines, but they used a different data format, causing failure on a different day. Just after the first incident, the vendor investigated other programs which relate to the automatic gate machines, but the vendor could not find the similar defect which would have prevent the second incident.

These incidents indicate the importance and also difficulty of finding similar defects in software product collections. There are various literatures discussing the creation and removal of similar defect code [14, 15, 22]. We are interested in finding similar code fragment in a large collection of source code, to effectively locate similar code defects.

To specify our focus of similar defects, we give an example in Figure 1, which presents two code fragments of an open source Japanese input system Canna Ver. 3.6 [3]. These two fragments involve several buffer input operations (e.g., buf += HEADER_SIZE; ... = S2TOS(buf);), but there are no buffer boundary checks, resulting in implicit buffer overflow defects. Suppose that we first recognize that code fragment Figure 1(a) contains the buffer overflow defect. To prevent the same buffer overflow failures, we need to find and fix all the similar code fragments such as shown in Figure 1(b), which is slightly different from Figure 1(a) but which shares same variables and macros. The research question we are interested in here is how effectively we can locate code fragments like Figure 1(b) in a large-scale source code collection when a reference code fragment like Figure 1(a) is given.

A straightforward approach to this issue, which is widely used in the practice, would be using search tools such as grep [8]. However, it is not easy for the developers to choose an appropriate identifier as the search keyword [21]. In general, there are many candidate identifiers in the reference code fragment and the choice of the identifiers heavily affects the search results. For example, in the case of Figure 1(a), we may choose a variable name buf as the search identifier; however, it will produce many unwanted retrieval results since buf is widely used in many parts of the system. We probably need to combine several identifiers as the search keys, but it is a generally difficult task.

Another approach would be to employ more sophisticated retrieving tools such as code clone detection tool [1, 10, 11, 12]. However, most code clone detection tools permit only minor changes between clones, since they basically depend on string or tree matching algorithms after the input sequence normalization, tokenization, and/or tree construction. In the case of Figure 1, there two fragments are not reported as a code clone pair by a popular code clone tool, CCFinder significant differences in the last line.

In this paper, we will explore a novel approach which is simpler and more effective in locating similar fault code fragments than existing methods and tools. We propose a synonymous identifier retrieval method. This approach has the following characteristics.

- A whole code fragment is used as the input to the query. This allows developers to perform retrieval without careful choice of the query identifiers.
- We employ a code fragment retrieval method based on the occurrence of similar identifiers in the query and target code fragments. This approach is more robust to syntactic differences of code fragments than sequential or structural matching methods such as the token-based code clone detection tools.
- The retrieval is made using extracted identifiers. A pair of identifiers frequently co-existing in close proximity are considered as the same identifier, and are treated as if they where synonyms of each other. The retrieval is made using extracted identifiers and their synonyms.

We have developed SC-Retriever (Synonymous Code Retriever) implementing the proposed method. SC-Retriever has been applied to find defects in two software systems, Canna and SPARS-J [19]. We have confirmed the advantages of our approach, compared with CCFinder [11].

In Section 2 we present our retrieval method. Section 3 gives results derived from using this method. In Section 4, several related works and discussions will be presented. Finally, we will conclude our paper with a few remarks in Section 5.

2. PROPOSED METHOD

As shown in Figure 2, the proposed method accepts a code fragment as a query, and then identifies similar code fragments in a target source files. The process comprises three steps as follows.

Identifier Extraction We extract identifiers from both query code fragment and target source files, and apply several normalization rules to extracted identifiers. We may call these normalized identifiers simply *identifiers* if there is no confusion.

Synonymous Identifier Determination Synonyms of each identifier are determined by a method commonly used in natural language processing [4].

Retrieval with Query Code Fragment Code fragments are extracted as similar code fragments from the target source files, if those code fragments have the same synonymous identifiers as the query identifiers.

We will explain each step in detail through following subsections.

2.1 Identifier Extraction

At first, identifiers are extracted from both query code fragment and target source files. To each extracted identifier, several normalization rules are applied (e.g. dividing at underscore, number elimination), producing a set of normalized identifiers.

Next, we create a matrix named *identifier matrix*, which represents the occurrence of identifiers in each *structural unit* of the target source files. A structural unit is a source code fragment obtained by a systematic partition of the target source files. A module, a function, or a structural block would be examples of structural units. We employ functions as the structural unit hereafter in this paper.

Figure 3 illustrates an example of functions involving identifiers. There are three functions f_0, f_1 and f_2 and they involve five identifiers i_a, i_b, i_c, i_d and i_e. In the identifier extraction, those five identifiers are extracted from three functions as shown in Figure 3.

Figure 4 is the identifier matrix created from the functions in Figure 3. The matrix represents the occurrences of identifiers i_a, i_b, i_c, i_d and i_e in functions f_0, f_1 and f_2.

2.2 Synonymous Identifier Determination

In the synonymous identifier determination, we perform clustering of the identifiers based on Dagan's model [4]. The clustering procedure is as follows.

Step1: Co-occurrence Matrix Creation.

First, a *co-occurrence matrix* is created from the identifier matrix. Let N denote the number of different identifiers. The co-occurrence matrix is represented as an $N \times N$ symmetrical matrix. Each element (i_x, i_y) is the number of functions (i.e, structural units) in which i_x and i_y co-occur. Figure 5 is the co-occurrence matrix created from the identifier matrix in Figure 4. Since i_c and i_e occur twice in functions f_0 and f_2, then (i_c, i_e) and (i_e, i_c) become 2. Diagonal elements are not used in our method, so they are filled with symbol "$-$".

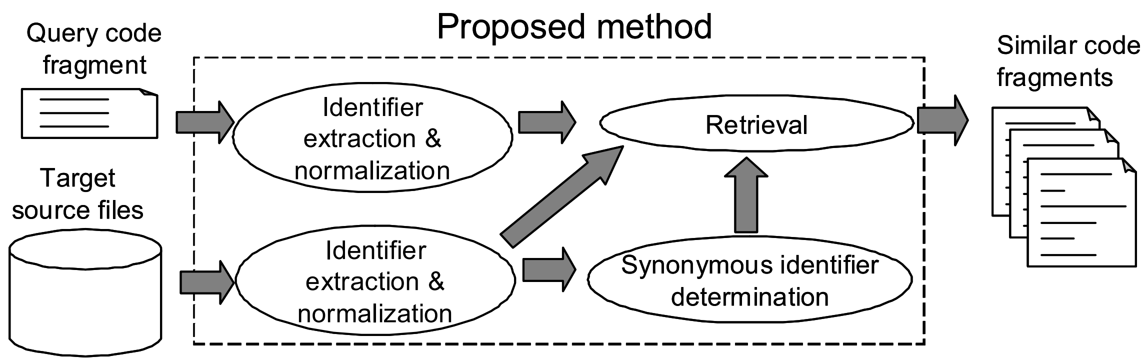
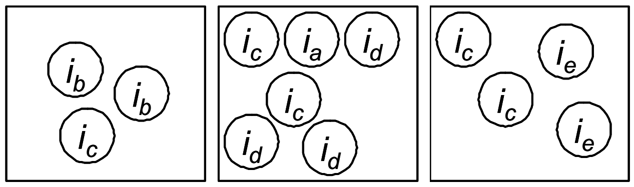

Query code fragment
Target source files
Proposed method
Identifier extraction & normalization
Identifier extraction & normalization
Retrieval
Synonymous identifier determination
Similar code fragments

Figure 2: An overview of proposed method



(a) function f_0 (b) function f_1 (c) function f_2

Figure 3: Example of functions involving identifiers

$$\begin{array}{c|ccccc} & i_a & i_b & i_c & i_d & i_e \\ \hline f_0 & 0 & 2 & 1 & 0 & 0 \\ f_1 & 1 & 0 & 2 & 3 & 0 \\ f_2 & 0 & 0 & 2 & 0 & 2 \end{array}$$

Figure 4: Example of identifier matrix (created from Figure 3)

$$\begin{array}{c|ccccc} & i_a & i_b & i_c & i_d & i_e \\ \hline i_a & - & 0 & 1 & 1 & 0 \\ i_b & 0 & - & 1 & 0 & 0 \\ i_c & 1 & 1 & - & 1 & 1 \\ i_d & 1 & 0 & 1 & - & 0 \\ i_e & 0 & 0 & 1 & 0 & - \end{array}$$

Figure 5: Example of co-occurrence matrix (created from Figure 4)

$$\begin{array}{c|ccccc} & i_a & i_b & i_c & i_d & i_e \\ \hline i_a & - & 0.30 & 0.43 & 0 & 0.43 \\ i_b & 0.30 & - & 0.67 & 0.30 & 0.17 \\ i_c & 0.43 & 0.67 & - & 0.43 & 0.30 \\ i_d & 0 & 0.30 & 0.43 & - & 0.67 \\ i_e & 0.43 & 0.17 & 0.30 & 0.67 & - \end{array}$$

Figure 6: Example of distance matrix

Step2: Distance Calculation.

Distance between each identifier pair is computed by using *Jensen-Shannon divergence method* [17]. The computation is based on the measurement of probability distribution of two vectors extracted from the co-occurrence matrix. Figure 6 is an example of distance matrix. Each element is non-negative real number, and it represents the distance of two identifiers, where 0 means no distance.

Jensen-Shannon divergence between two identifiers is intuitively calculated as a difference of two vectors of the co-occurrence matrix. For example, the distance between the identifiers i_d and i_e is represented by the difference between two vectors $[(i_a, i_d), (i_b, i_d), (i_c, i_d)]$ and $[(i_a, i_e), (i_b, i_e), (i_c, i_e)]$ (i.e, the difference between $[1, 0, 1]$ and $[0, 0, 2]$). If such difference is relatively large, the value of the corresponding element in the distance matrix is also relatively large.

Step3: Performing Clustering.

We perform clustering of identifiers based on the distance of identifiers. Initially, a set of identifiers with distances between them is given and a threshold for terminating the clustering is determined by the user. The clustering is performed as the following steps.

(a) Initial clusters are created for each identifier.

(b) For each pair of clusters, a distance value is calculated by *group average method*[1], then the closest clusters (i.e., the cluster pairs which have the minimum distance value) are merged into a single one.

(c) The above step(b) is repeated until the minimum distance value between any two clusters falls larger than the user-determined threshold or all of clusters are merged into a single one.

After the clustering, an identifier involved in a resulting cluster is called a *synonym* of other identifiers in the same cluster.

Figure 7 shows the clustering process for the distance matrix in Figure 6. In this example, we assume that the clustering threshold value is 0.30. At first, the initial clusters are created for each identifier (Figure7(a)). Then, the

[1]When two identifier-clusters are given, the *group average method* obtains the distance value by calculating the average distance between each pairing of identifiers across the two clusters.

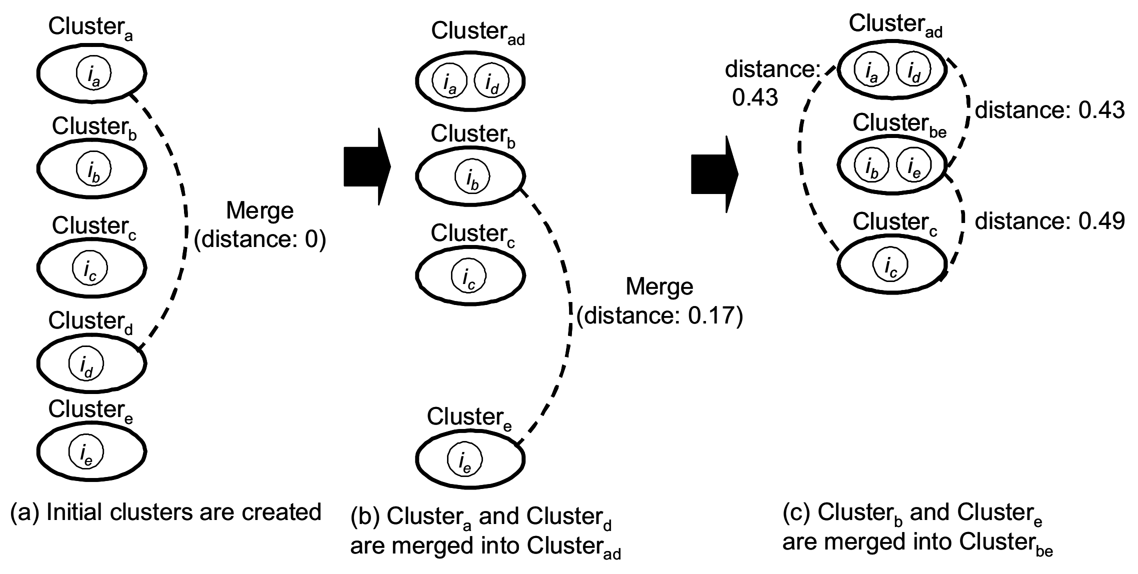


Figure 7: Identifier Clustering

closest pair $(Cluster_a, Cluster_d)$ is merged into $Cluster_{ad}$ (Figure7(b)). Next, the closest pair $(Cluster_b, Cluster_e)$ is merged into $Cluster_{be}$ (Figure7(c)). Here, since the smallest distance between any two clusters is 0.43 and is larger than the clustering threshold 0.30, so the clustering is terminated. As a result, we determine that i_a and i_d are synonyms each other, and also i_b and i_e are synonyms each other.

2.3 Retrieval with Query Code Fragment

In this section, we describe similar code fragment obtained by the retrieval.

First, we introduce correspondence of identifier as follows.

Definition 2.1 (Identifier Correspondence) *When two identifiers i_x and i_y are given, if i_y is identical or synonymous to i_x, we say that i_y corresponds to i_x (and vice versa).*

Now, we define *similar code fragment* as follows.

Definition 2.2 (Similar Code Fragment) *We assume that a query code fragment Q and a target code fragment T involve at least one identifier, respectively. Here, if each identifier in Q corresponds to at least one identifier in T, then T is said similar code fragment to Q.*

Figure 8 illustrates an example of comparing the identifier list $\left[host, alloc, add, host\right]$ derived from a query code fragment and the identifier list $\left[node, \ldots, alloc, add, \ldots, node\right]$ derived from a target code fragment. The edge is drawn when two identifiers correspond, i.e, they are identical or they are in synonym relations. In this case, each identifier in the query code fragment corresponds to at least one synonymous identifier in the target code fragment. Therefore, we say that the target code fragment is a similar code fragment to the query code fragment (and vice versa).

2.4 Implementation of SC-Retriever

We have developed SC-Retriever (Similar Code Retriever) implementing the proposed method.

SC-Retriever performs the detection of C-language function as the code fragment. That is, it retrieves similar functions that are similar to a query code fragment. The advantage of function-wise detection is that developers can easily spot the defective code portion, and also that partitioning the target source files into functions is quite easy.

Also, in order to fit function-wise detection, we employ functions as the structural units described in Section 2.1.

3. CASE STUDIES

We have conducted case studies with two tools: (1) SC-Retriever, (2) a code clone detection tool CCFinder. Using these tools, we have retrieved defective functions in two software systems with their defect data, and efficiency of the retrieval has been compared.

CCFinder is a fast and scalable code clone detection tool [11]. It generally gets a collection of source code as the input and generate the report of the locations of code clone fragments. CCFinder can also report the locations of code fragments for a specific query code fragments [9]. In the case study, we use the latter feature to locate the defective functions.

3.1 Experiment

3.1.1 Measures of Retrieval Efficiency

We used *precision*, *recall* and *F-measure* as the measures of retrieval efficiency. Let D denote the set of defective code fragments determined by the bug records of the software system. Also, R denotes the set of retrieved code fragments by any of the tools. We define the precision, recall and F-

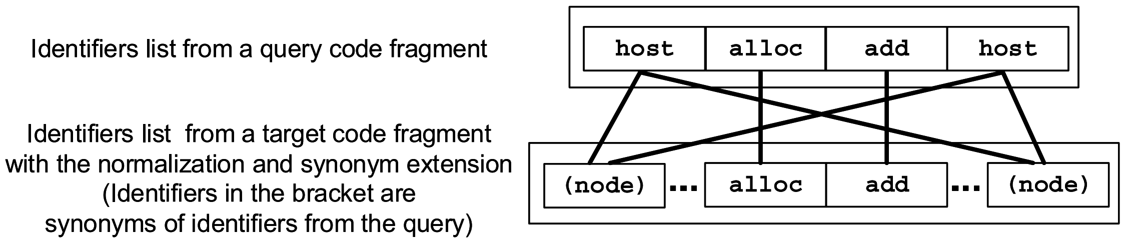

Identifiers list from a query code fragment
host
alloc
add
host
Identifiers list from a target code fragment with the normalization and synonym extension (Identifiers in the bracket are synonyms of identifiers from the query)
(node)
alloc
add
(node)

Figure 8: Comparison between a query code fragment and a target code fragment (The arrow is drawn from each identifier to its corresponding identifiers.)

Table 1: Statistics of target programs

	LOC	**# of functions**	**# of defective functions**	**# of defects**
Canna	90K	2361	18	19
SPARS-J	36K	859	50	75

measure as follows.

$$Precision = \frac{|D \cap R|}{|R|} \quad (1)$$

$$Recall = \frac{|D \cap R|}{|D|} \quad (2)$$

$$F = \frac{2 \cdot Precision \cdot Recall}{Precision + Recall} \quad (3)$$

3.1.2 Experimental Step

Our case studies are composed of the following steps.

1. We choose defective code fragments from target source files, based on the criteria described later.
2. Using SC-Retriever, and CCFinder, we retrieve functions in the target source files. In the case of SC-Retriever, we give the chosen code fragment as the query. Also, in the case of CCFinder, we detect the code clones for the chosen code fragments. The granularity of the code fragments as the output of those retrieval is set to function level here due to the implementation constraint mentioned in Section 2.4.
3. We calculate the precision, recall and F-measure with the retrieved results and the bug records.

3.1.3 Target Software Systems

We applied SC-Retriever, and CCFinder to o two different software systems. Table 1 gives statistics relating to these two systems.

Canna [3] is an open source client-server Japanese character input system. Canna Ver. 3.6 involves 19 buffer overflow defects, and those defects exist in 18 functions.

SPARS-J is Java component retrieval system [19]. The target version of SPARS-J involves 75 defects caused by missing typecast operations[2], and those defects exist in 50 functions.

[2]In the source files implementing the process of software component registration, there are 75 defects caused by lack of typecasting from the internal data types into the types supported by the relational database of SPARS-J.

```
buf += HEADER_SIZE;
Request.type2.context = S2TOS(buf);
```

(a) Query Code fragment C_A (small code fragment)

```
buf += HEADER_SIZE;
Request.type18.context = S2TOS(buf);
buf += SIZEOFSHORT;
Request.type18.data = (char *)buf;
buf += Request.type18.datalen - SIZEOFSHORT * 2;
Request.type18.size = S2TOS(buf);
```

(b) Query Code fragment C_B (large code fragment)

```
buf += SIZEOFINT;
Request.type10.kouho = (short *)buf;
for (i = 0; i < Request.type10.number; i++) {
      Request.type10.kouho[i] = S2TOS(buf);
      buf += SIZEOFSHORT;
      ir_debug(Dmsg(10, "req->kouho =%d\n",
      Request.type10.kouho[i]));
}
```

(c) Query Code fragment C_C (complicated code fragment)

Figure 9: Query code fragments for Canna 3.6

3.1.4 Query Code Fragments

From these defects reported by the bug records, we have selected three types of the query code fragments:

A. Small code fragments (C_A for Canna, and S_A for SPARS-J)

B. Large code fragments (C_B for Canna, and S_B for SPARS-J)

C. Complicated code fragments (C_C for Canna, and S_C for SPARS-J)

Figure 9 and Figure 10 show the selected query code fragments for the case studies of Canna and SPARS-J, respectively.

These code fragments would be considered typical in that they contain sufficient code related to the faulty operations while avoiding inclusion of code not related to the faulty operations. We are interested in the distinction of fragment sizes which might affect the retrieval efficiency, and also we are interested in the effect of the code complexity. We will further discuss the requirement and characteristics of the query code fragment in Section 4.

```
package_name_length = p - class_name;
       //should be: = (int)(p - class_name);
```

(a) Query Code fragment S_A (small code fragment)

```
ret->mm_attachments =
  mime_parsemultipart(bdlc + 1,
  len - (bdlc + 1 - msg),
  //should be: (int) (len - (bdlc + 1 - msg),
  bd, &ret->mm_nattachments,
  &pos, crlfpair);
```

(b) Query Code fragment S_B (large code fragment)

```
pos +=
  (((char *)((i = memchr(map + pos,
// should be: (int) (((char *)((i = memchr(...
  '$', len - pos - 1))
  ? i : map + len))
  - ((char *)map + pos));
```

(c) Query Code fragment S_C (complicated code fragment)

Figure 10: Query code fragments for SPARS-J

```
buf += HEADER_SIZE;
Request.type17.dicname = (char *)buf;
Request.type17.mode =
(char)*(buf + Request.type17.datalen
- SIZEOFCHAR) ;
Request.type17.datalen - SIZEOFCHAR) ;
```

Figure 11: Defective code fragment that SC-Retriever was unable to find in Canna

3.1.5 Clustering Threshold

As described before, the identifier clustering is repeated until the distance of any two clusters becomes greater than the threshold value th_v. Since the distances of two clusters changes case by case, then it is fairly difficult to determine a prefixed th_v used for any purpose. Instead, we introduce the threshold ratio th_r which defines th_v from the maximum distance d_{max} of any two identifiers in the target source files, as follows.

$$th_v = th_r \cdot d_{max} \qquad (4)$$

By introducing the threshold ratio, the effect of the distribution of identification distances is complemented. We will use three threshold ratios $th_r = 0.05$, $th_r = 0.10$, $th_r = 0.15$, and see the distinction of resulting synonyms.

3.2 Experimental Results of Canna

Table 2 shows the result of Canna's case study. We present the result with $th_r = 0.1$ for SC-Retriever here. We would say that SC-Retriever shows better *F-values* than CCFinder for most cases. SC-Retriever has fairly high precision 0.63 for all the cases of C_A, C_B, and C_C, and also has near perfect recall, 0.94 for all the cases. The only defective code fragment that SC-Retriever was unable to find is presented in Figure 11. In this case, *S2TOS* is the key identifier of the mismatch, which occurs in all of C_A, C_B, and C_C, but S2TOS and its synonyms (e.g., L4TOL) do not occur in the fragment of Figure 11 at all.

CCFinder shows very good precision and recall for C_A. This means that all the faulty code fragments contain similar statements to the query fragments. However, it presents very low recalls for C_B and C_C. Since the retrieval results are very sensitive to the query code fragments, we have to choose them very carefully.

Table 2 shows also the effectiveness of the retrieve for three difference threshold ratios. As you see, $th_r = 0.10$ generally shows better F-values than $th_r = 0.05$ and $th_r = 0.15$. If we would choosing $th_r = 0.05$, the F-value for C_A is better, but for cases C_B and C_C, we get very low F-values. Also, we would get very low F-value 0.20 for all the cases with $th_r = 0.15$. Therefore, we think that using $th_r = 0.10$ would be a reasonable choice.

3.3 Experimental Results of SPARS-J

Table 3 shows the results of SPARS-J's case study. We also present the result with $th_r = 0.10$ for SC-Retriever here.

In most cases, the precision and recall values in the results of SPARS-J were worse than those values in the results of Canna. We believe that the primary cause is that the defects (missing typecast operations) spread across various kinds of implementations in the system. In fact, various identifiers are used in the code fragments sharing such defects.

CCFinder shows very good precision and recall for C_A. This means that all the faulty code fragments contain similar statements to the query fragments. However, it presents very low recalls for C_B and C_C. Since the retrieval results are very sensitive to the query code fragments, we have to choose them very carefully.

Table 3 shows also the results of SPARS-J with three threshold ratios. We would think that both threshold ratios $th_r = 0.10$ and $th_r = 0.15$ are promising. If developers emphasize the precision, an appropriate ratio is $th_r = 0.10$. Conversely, if developers emphasize the recall, an appropriate ratio is $th_r = 0.15$.

When we set the threshold ratio to $th_r = 0.05$, the retrieval results include almost only the function involving the query code fragment.

4. DISCUSSION AND RELATED WORKS

4.1 Effectiveness of SC-Retriever

According to Table 2 and Table 3, SC-Retriever shows relatively better F-values than CCFinder for most cases. In the cases of small code fragments C_A and S_A, CCFinder shows better F-values than SC-Retriever. However, it presents very low recalls for both large and complicated code fragments.

Current implementation of SC-Retriever is a prototype system, and we did not consider its performance seriously. It takes about 40 minutes on a PC workstation for the whole process of Canna's case. Most part of this execution time is for the clustering process, which can be performed only once before any retrieval. The resulting synonym information can be repeatedly used for later retrievals. We consider that this clustering process would be further optimized and parallelized for the better performance. For the retrieval process, we have employed a simple matching algorithm. Its performance can be also improved greatly if we would use a hash method such as *Locality Sensitive Hashing (LSH)* [5]. These implementation techniques will further strengthen the advantages of our approach.

4.2 Comparison with CCFinder

Table 2: Results of Canna

	SC-Retriever w/$th_r = 0.05$			SC-Retriever w/$th_r = 0.10$			SC-Retriever w/$th_r = 0.15$			CCFinder		
	precision	recall	F-value	precision	recall	F-value	precision	recall	F-value	precision	recall	F-value
C_A	1.00	0.94	0.97	0.63	0.94	0.75	0.11	0.94	0.20	1.00	0.94	0.97
C_B	1.00	0.06	0.11	0.63	0.94	0.75	0.11	0.94	0.20	1.00	0.06	0.11
C_C	1.00	0.06	0.11	0.63	0.94	0.75	0.11	0.94	0.20	1.00	0.06	0.11

Table 3: Results of SPARS-J

	SC-Retriever w/$th_r = 0.05$			SC-Retriever w/$th_r = 0.10$			SC-Retriever w/$th_r = 0.15$			CCFinder		
	precision	recall	F-value	precision	recall	F-value	precision	recall	F-value	precision	recall	F-value
S_A	0.66	0.04	0.08	0.10	0.18	0.13	0.09	0.30	0.14	0.46	0.12	0.19
S_B	1.00	0.02	0.04	0.44	0.08	0.14	0.08	0.38	0.13	1.00	0.02	0.04
S_C	1.00	0.02	0.04	0.66	0.04	0.08	0.50	0.04	0.07	1.00	0.02	0.04

SC-Retriever showed the better efficiency than CCFinder in many cases, but for the small code fragment cases, CCFinder outperforms SC-Retriever. This suggests that when we need to retrieve code fragments with some a specific code pattern, CCFinder is a better choice. On the other hand, if we are unsure of the defective code pattern, SC-Retriever will be the first choice.

As we will discuss in Section 4.4.1, we would use other code clone tools with difference clone detection algorithms. Some of those tools may allow more flexibility of code structure distinction, but such flexibility would not be much as given by SC-Retriever.

4.3 Threshold and Fragment

For the determination of the clustering termination, we have used the threshold value and threshold ratio. We have investigates several threshold ratios and found that threshold ratio 0.10 would be appropriate for our case studies. We have to further investigate the better threshold ratios which can be used for various cases. Also, we would explore other methods to terminate the cluster merging process, such as ones using cluster sizes or so.

Granularity of code fragments in the retrieval is an important factor in our approach. We have used a C-language function as the structured unit of the synonym determination process. We can consider other granularity for this purpose, such as a block. If the single structured unit becomes smaller, the probability of co-occurring two identifiers in the same structured unit generally decreases, generating fewer synonyms for each identifier. This would increase the precision, but reduce the recall greatly. We think that the current choice would be a practical compromise.

Also, we have used a function for the retrieved code fragments as the outputs. This simplified the implementation of SC-Retriever greatly. We would consider other granularities such as a block as the output, which would reduce the overhead of focusing defective code portions in a function. However, without using such fine-grained retrieval methods, we are able to locate the defective portions effectively if the identifiers corresponding to the query identifiers are highlighted in the SC-Retriever output. We will improve SC-Retriever in this way.

In the case studies, we have used three types of query code fragments from the bug records of the systems. Choosing different code fragments for the query would affect the retrieve results. However, as mentioned in Section 3.1.4, we have chosen code fragments to intend that all bug-related statements are covered and non-related statements are excluded as much. We guess that this policy will produce similar results for other query code fragments.

4.4 Related Works

4.4.1 *Code Clone Detection Techniques*

So far, a lot of techniques have been developed on code clone detection [1, 2, 6, 7, 10, 11, 12, 13, 15, 18]. Several of them are token-based detection techniques [1, 6, 11], and they are very scalable in terms of time and space complexity. We have used a token-based detection tool CCFinder [11] for the case studies.

Tree-based and semantic-based techniques have been also devised [2, 10]. Jiang *et al.* have developed an abstract syntax tree (AST) based code clone detection tool DECKARD [10]. It can treat a pair of code fragments as a code clone when their syntax elements are the same (even if their AST structures are slightly different). Semantic-based techniques identify code clones using program dependence graphs [7, 12, 13].

Code clone detection tools find only structurally similar code fragments. On the other hand, our method retrieves similar fragments based on the correspondence of identifiers; thus the retrieved results might not be structurally similar. This would be a major benefit of our approach for the cases of the retrievals for unknown structured patterns, such as the case study for SPARS-J where we sought missing type cast operations.

4.4.2 *Defect Detection Techniques*

Li *et al.* proposed a tool named PR-Miner [16]. It extracts frequently occurring pattern of variable and function names from source code based on frequent item-set mining, then detects the violations of detected patterns. It finally provides code fragments involving such violations as defect candidates. Also, Li *et al.* showed identifier naming inconsistencies among code clones of the Linux kernel as the defect candidates [15].

We believe that by giving SC-Retriever those defect candidates, we can detect more defects that Li's methods cannot detect.

5. CONCLUSION

We have proposed a method for retrieving similar code fragments to a query code fragment. The proposed method can provide not only code fragments which share all identifiers with a query code fragment, but also code fragments which involve all identical or synonymous identifiers.

In the case studies, we have applied SC-Retriever to the source files of Canna and SPARS-J, together with CCFinder for the purpose of comparing the retrieval efficiency. In most cases, the SC-Retriever outperforms CCFinder in the retrieval efficiency.

Acknowledgments

We express our great thanks to Mr. Akihiko Matsuo and Mr. Kenichi Kobayashi of Fujitsu Laboratory for their discussions of SC-Retriever. We also thank Dr. Daniel German of the University of Victoria, Dr. Ian Davis of the University of Waterloo, and Dr. Stan Jarzabek of the National University of Singapore for helpful comments on earlier revisions of this paper.

This work is being conducted as a part of Stage Project. Also, this work is partially supported by JSPS, Grant-in-Aid for Scientific Research (A) (21240002) .

6. REFERENCES

[1] B. S. Baker. Finding clones with Dup: Analysis of an experiment. *IEEE Trans. Softw. Eng.*, 33(9):608–621, 2007.

[2] I. D. Baxter, A. Yahin, L. Moura, M. S. Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proc. of ICSM '98*, pages 368–377, 1998.

[3] Canna. `http://canna.sourceforge.jp`.

[4] I. Dagan, L. Lee, and F. C. N. Pereira. Similarity-based models of word cooccurrence probabilities. *Machine Learning*, 34(1-3):43–69, 1999.

[5] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proc. of SCG 2004*, pages 253–262, 2004.

[6] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *Proc. of ICSM '99*, pages 109–118, 1999.

[7] M. Gabel, L. Jiang, and Z. Su. Scalable detection of semantic clones. In *Proc. of ICSE 2008*, pages 321–330, 2008.

[8] GNU grep. `http://www.gnu.org/software/grep/`.

[9] Y. Higo, Y. Ueda, S. Kusumoto, and K. Inoue. Simultaneous modification support based on code clone analysis. In *Proc. of APSEC 2007*, pages 262–269, 2007.

[10] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. DECKARD: Scalable and accurate tree-based detection of code clones. In *Proc. of ICSE 2007*, pages 96–105, 2007.

[11] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Softw. Eng.*, 28(7):654–670, 2002.

[12] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. In *Proc. of SAS 2001*, pages 40–56, 2001.

[13] J. Krinke. Identifying similar code with program dependence graphs. In *Proc. of WCRE 2001*, pages 301–309, 2001.

[14] B. Laguë, D. Proulx, J. Mayrand, E. M. Merlo, and J. Hudepohl. Assessing the benefits of incorporating function clone detection in a development process. In *Proc. of ICSM '97*, pages 314–321, 1997.

[15] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Trans. Softw. Eng.*, 32(3):176–192, 2006.

[16] Z. Li and Y. Zhou. PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code. In *Proc. of ESEC/FSE 2005*, pages 306–315, 2005.

[17] J. Lin. Divergence measures based on the shannon entropy. *IEEE Trans. Inf. Theory*, 37(1):145–151, 1991.

[18] J. Mayland, C. Leblanc, and E. M. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *Proc. of ICSM '96*, pages 244–253, 1996.

[19] SPARS Project. `http://spars.info/SPARS/`.

[20] M. Williams. Systems glitch hits hundreds of Tokyo stations. *washingtonpost.com*, 2007. `http://www.washingtonpost.com/wp-dyn/content/article/2007/10/12/AR2007101200865_pf.html`.

[21] T. Yamashina, H. Uwano, K. Fushida, Y. Kamei, M. Nagura, S. Kawaguchi, and H. Iida. SHINOBI: A real-time code clone detection tool for software maintenance. Technical Report NAIST-IS-TR2008005, Nara Institute of Science and Technology, 2008. `http://isw3.naist.jp/IS/TechReport/report/2008005.pdf`.

[22] A. Zeller. *Why Programs Fail.* Morgan Kaufmann Pub., 2005.