Revisiting Capability of PDG-based Clone Detection

Yoshiki Higo, Hiroaki Murakami, and Shinji Kusumoto

Graduate School of Information Science and Technology,
Osaka University
{higo, h-murakm, kusumoto}@hist.osaka-u.ac.jp

November, 2013

Abstract

Code cloning is an active research topic in the field of software engineering over the last two decades. Recently, many research efforts have been focusing on detecting clones from very large scale source code and detecting clones incrementally from multiple versions of source code. However, PDG(Program Dependency Graph)-based detection techniques have not been researched very actively over the last several years. In this paper, we revisit capability of the PDG-based clone detection and discuss its potential and importance. Besides, we propose new techniques, which increase capability of the PDG-based clone detection. The proposed techniques have already been implemented as a software tool, SCORPIO, *which is open to the public. We evaluated the proposed techniques on four open source software by using the tool. The application result showed that the proposed techniques were able to detect more clones to be detected than conventional techniques without raising the rate of false positives.*

I. Introduction

Until just a decade ago, all the people related to software development (e.g., engineer, researcher) thought all the clones in source code have negative impacts on activities of software development and maintenance. Therefore, many research efforts have been proposed various techniques for detecting, visualizing, refactoring, and modifying code clones [1, 2].

However, in the last decade, many researcher reported that clones do not necessarily make mischief. For example, Kapser and Godfrey reported their empirical study on device drivers [3]. When a programmer develops a new device driver, s/he usually reuses code of existing drivers by copy-and-past operations because most of the functionalities required for the new driver are the same as ones required for the old drivers. In other words, functionalities required for the new drivers have already been implemented in the old drivers. Consequently, there are plenty of clones among device drivers. However, Such reused code rarely cause unintended inconsistency problems because they are very stable. Besides, Kim et al. said that refactoring clones is not necessarily a good way for avoiding unintended inconsistencies if we consider the future evolution of software systems [4]. They found that there are code fragments that are duplicated with other code fragments only a short period of time. After they become unduplicated, they evolve differently. Moreover, there are many empirical studies that compared code in clones with code outside the clones [5, 6, 7, 8]. They revealed that the modification cost on cloned code is not necessarily higher than the one on non-cloned code.

However, there is a fact that some clones make it more difficult to keep source code consistent [9, 10]. It is important to identify such bad clones efficiently. Besides, clone information can be utilized for versatile applications such as promoting code reuse or supporting understanding architectures of software systems [11, 12, 13, 14]. For these reasons, code cloning is an active research topic in the field of software engineering still now. In particular, many research efforts have been conducted on detecting clones from very large scale source code and detecting clones incrementally in the last couple of years. [15, 16, 14, 17, 18, 19, 20].

On the other hand, PDG(Program Dependency Graph)-based detection techniques have not been researched very actively in the last couple of years. Authors think that the result of Bellon's study might be one of the reasons why the PDG-based detection is inactive in research related to clones. In Bellon's study, the PDG-based detection technique scored lower precision and recall than other detection techniques [21]. In the paper, there are descriptions that the settings of the empirical study might work against the PDG-based detection. However, the low precision and recall of the PDG-based detection might be so impressive that negative image of the PDG-based detection spread over researchers[1].

However, the PDG-based detection has some advantages that the other detections do not have. If two or more code fragments are detected as clones by the PDG-based technique, they have the same data and control dependencies. Hence, clones detected by the PDG-based technique are suited for refactoring. Hotta et al. proposed a technique that supports removing clones by applying *Form Template Method* pattern [22], which is a relatively complex procedure among refactoring patterns [23]. Besides, the PDG-based technique can detect intertwined clones (We show an example of intertwined clones in Section III). Intertwined clones can be detected by none of the clone detection techniques except the PDG-based one.

As mentioned above, the PDG-based technique has some strong points. If its weak points are eliminated or released (its precision and recall are improved), we might be able to apply it in various situations instead of other techniques. In this paper, we firstly consider why the PDG-based technique scored lower precision and recall than other techniques in Bellon's study. After that, we propose a new PDG-based detection technique.

There are the following contributions in this paper.

- We revealed why the PDG-based technique scored lower precision and lower recall than other techniques.
- We proposed a new PDG-based detection technique solving the revealed problems.
- We implemented the proposed technique as a software tool. The tool is open to the public in Github [2] and anyone can use it.
- We applied the tool to four open source systems, which were used in Bellon's study. The application result showed that the precision and recall of the proposed technique were better than the conventional technique. Note that, in this empirical study, we used gap information in clones, which was not used in Bellon's study. That mean the comparison in this experiment is more precisely than Bellon's one.

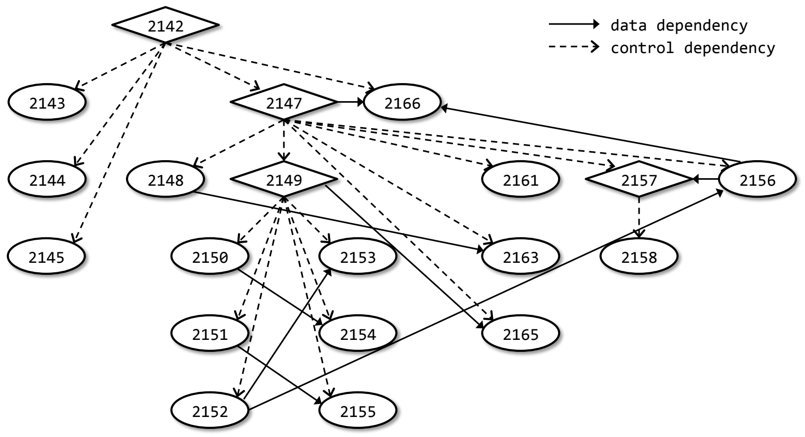
The remainder of this paper is organized as follows: Section II introduces PDGs and PDG-based clone detections; in Section III, we discuss why the PDG-based detection scored lower precision and recall than other detections in Bellon's study; in Section IV, we propose a new PDG-based detection technique that solves the problems shown in Section III; the application result of four open source software systems and its threats to validities are described in Sections V and VI, respectively; lastly, we conclude this paper in Section VIII

[1]At July 2013, the paper [21] of Bellon's study has approximately 250 citations by Google Scholar, which mean the paper has a large impact on subsequent research.

[2]`https://github.com/YoshikiHigo/TinyPDG`

```
2142  public int literalIndexForJavaLangStringBufferDefaultConstructor() {
2143    int index;
2144    int nameAndTypeIndex;
2145    int classIndex;
2146    // Looking into the method ref table
2147    if ((index = wellKnownMethods[STRINGBUFFER_DEFAULT_CONSTR_METHOD]) == 0) {
2148      classIndex = literalIndexForJavaLangStringBuffer();
2149      if ((nameAndTypeIndex = ellKnownMethodNameAndTypes[DEFAULT_CONSTR_METHOD_NAME_AND_TYPE]) == 0) {
2150        int nameIndex = literalIndex(QualifiedNamesConstants.Init);
2151        int typeIndex = literalIndex(QualifiedNamesConstants.DefaultConstructorSignature);
2152        nameAndTypeIndex = wellKnownMethodNameAndTypes[DEFAULT_CONSTR_METHOD_NAME_AND_TYPE] = currentIndex++;
2153        writeU1(NameAndTypeTag);
2154        writeU2(nameIndex);
2155        writeU2(typeIndex); }
2156      index = wellKnownMethods[STRINGBUFFER_DEFAULT_CONSTR_METHOD] = currentIndex++;
2157      if (index > 0xFFFF){
2158        this.classFile.referenceBinding.scope.problemReporter()
              .noMoreAvailableSpaceInConstantPool(this.classFile.referenceBinding.scope.referenceType()); }
2159      // Write the method ref constant into the constant pool
2160      // First add the tag
2161      writeU1(MethodRefTag);
2162      // Then write the class index
2163      writeU2(classIndex);
2164      // The write the nameAndType index
2165      writeU2(nameAndTypeIndex); }
2166    return index; }
```

(a) source code (eclipse-ant/src/zip/ZipEntry.java)



(b) program dependency graph

Figure 1: *Example of program dependency graph (this source code is included in the target of Bellon's experiment [21])*

II. Preliminaries

Here, we explain a definition of PDG and introduce a PDG-based clone detection technique briefly. If readers already have knowledge about them, please skip this section.

I. Program Dependency Graph

A PDG is a directed graph that represents the dependencies among program elements (statements or conditional predicates). A PDG node is a program element, and a PDG edge shows a dependency among two nodes. There are two types of dependencies in PDGs, namely, **control**

dependency and **data dependency** [24].

If all of the following conditions are satisfied, there is a control dependency from statement s_1 to s_2:

- s_1 is a conditional predicate, and
- the result of executing s_1 determines directly whether s_2 is executed or not.

If all the following conditions are satisfied, there is a data dependency from statement s_3 to s_4 via variable v:

- s_3 defines v,
- s_4 references v, and
- there is at least one execution path from s_3 to s_4 without redefining v.

Figure 1 is an actual Java method and a PDG generated from the source code. Labels attached to the nodes mean the lines where their elements are located in the source code. In this example, there are data dependencies between nodes using variables such as classIndex, and there are also control dependencies between the conditional predicates of the if-statements and their inner-statements. The node labeled 2142 is the enter node of the PDGs. The enter node is regarded as a conditional predicate [25]. We use this example throughout this paper to illustrate differences between conventional and proposed PDGs.

II. Basic Algorithm for PDG-based Clone Detection

We show the basic algorithm we use for detecting clones with PDGs. The algorithm was built based on Komondoor's technique [26].

STEP1: a hash value is calculated from each node in PDGs based on the syntactic contents of their program elements. Nodes having the same hash value are grouped together. Variables and literals in the program elements are converted to special tokens before hashing. Consequently, the same hash value is generated from syntactically identical program elements even if the variables and the literals are different. In Figure 1, for example, two statements located in the lines 2150 and 2151 has the same hash value.

STEP2: every pair of nodes, (r_1, r_2), in all the groups is specified as a start point of a pairwise program slicing in order to detect a pair of similar subgraphs that include r_1 or r_2. Every pair of two slicings are performed in lock step. If predecessors (successors) of both the slicings have the same hash value, they are added to the pair of slices. If any of the following conditions is satisfied, predecessors are not added to the slices, and slicings stop.

- Predecessors (p_1, p_2) have different hash values.
- Predecessors (p_1, p_2) have the same hash value. However, p_1 (p_2) already exists in the slice of r_1 (r_2). This condition is intended to prevent an infinite loop.
- Predecessors (p_1, p_2) have the same hash value. However, p_1 (p_2) already exists in the slice p_2 (p_1). This condition is intended to prevent the two slices from sharing the same node.

Pairs of identified similar subgraphs are clone pairs in the basic algorithm.

STEP3: if a clone pair (s_1, s_2), which is a pair of similar subgraphs identified in STEP2, is subsumed by another clone pair (s'_1, s'_2) ($s_1 \subseteq s'_1 \cap s_2 \subseteq s'_2$), it is removed from the set of detected clone pairs because reporting the subsumed clone pairs is meaningless. The existences of such subsumed clone pairs enlarge the detection results unnecessarily.

III. Research Motivation

Although some PDG-based detection techniques were proposed a decade ago [26, 27], the PDG-based detection is not an active topic the last couple of years. Authors think that, the result of Bellon's study might give a negative image of the PDG-based detection to researchers.

First of all, we describe the overview of Bellon's experiment [21] briefly.

STEP1 Stefan Bellon, who is not a clone detection developer, selected the target software, which is open to the public in his web site[3]. He asked the developers of several detection tools to detect clones and to report the results to him.

STEP2 six developers detected clones from the target using their own tools. Table 1 shows the developers cooperated with Bellon, their tool names, and their detection techniques. The detection results were sent to Bellon in the given format.

STEP3 Bellon selected 2% of the received clone pairs randomly and checked each of the selected pairs to ensure that it was really a clone pair. He built 4,789 clone references from the four Java software and the four C software.

In order to understand why the precision and recall of the PDG-based detection were lower than other detections, we implemented a PDG-based detection tool with the algorithm described in Subsection II. We detected clones from Bellon's target software with the tool, and then we investigated clone references that had not been detected by the tool. As a result, we found that clone references are not likely to be detected by the PDG-based technique if they are contiguous code fragments in the source code. For example, in Figure 1(a), we assume that the code fragment from the line 2148 to the line 2165 is duplicated to a code fragment in another method. The code fragment pair should be able to be detected by token-based or line-based detection because they are contiguous in the source code. On the other hand, the PDG-based technique cannot detect the pair as clones. This is because the path from the node 2149 to the node 2161 must be routed through the node 2147 in Figure 1(b). However, the slicing stops at the node 2147. This node is

[3] `http://www.bauhaus-stuttgart.de/clones/`

Table 1: *Clone detection tools used in Bellon's experiment*

Tool	Comparison	Developer
Dup [28]	Token	Brenda S. Baker
CloneDR [29]	AST	Ira D. Baxter
CCFinder [30]	Token	Toshihiro Kamiya
Duplix [27]	PDG	Jens Krinke
CLAN [31]	Function Metrics	Ettore Merlo
Duploc [32]	Text	Mattias Riegger

not included in the duplicated code. Hence, it is impossible to detect a clone including both the nodes 2149 and 2161 with the PDG-based technique.

Fortunately, after we had found this problem by the investigation, we came up an idea: if there is another link between the nodes 2149 and 2161, they should be able to be detected as a clone pair. Consequently, in this paper, we investigate the following research question.

RQ1 is detection capability improved by adding an edge for each pair of two nodes whose elements are consecutive in the source code?

Besides, the clone format used in Bellon's study seems not to have been fit for clones detected by the PDG-based technique. Precision and recall of the PDG-based technique might not be so accurate in the experiment. As described in Subsection II, clones are detected by following control and data dependencies in PDGs. Thus, in detection results of the PDG-based technique, there are many clones whose elements are not consecutive in the source code. However, the format used in Bellon's study consists of a triplet (f, s, e). f is the absolute path of the file including a given clone, s and e are the start line and the end line of the clone, respectively. Hence, there is no information about locations of gaps even if a given clone is non-contiguous one. If we use this format for non-contiguous clones, lines not included in a given non-contiguous clone are regarded as being included in it. Authors actually checked how many non-contiguous clones are included in the PDG-based detection by using the tool. Table 2 shows the number of non-contiguous clones and their rate in the detection results. Most of the detected clones are non-contiguous ones. However, there is no space to store locations of gaps in Bellon's format. This fact means that matching between clone references and detected clones might not be incorrect in the case of non-contiguous clones. Consequently, in this paper, we investigate the following research question.

RQ2 is the number of clone references detected with gap information different from the number of ones without it?

A big advantage of the PDG-based technique is its capability to detect intertwined clones. Intertwined clones are ones that are tangled with their correspondent clones in the source code [26]. However, there have not been investigations on intertwined clones yet. Consequently, in this paper, we investigate the following research questions.

RQ3 How many intertwined clones are included in detection results of the PDG-based technique?

IV. Proposed Technique

Here, we propose some techniques to improve detection capability of the PDG-based detection. Subsections I, II, and III explain proposed techniques and then Subsection IV introduce the tool, **SCORPIO**, which has been implemented with the proposed techniques.

Table 2: *Ratio of non-contiguous clones detected by a PDG-based tool*

Software	# of clones	# of non-contiguous clones	ratio
netbeans-javadoc	21	19	90%
eclipse-ant	59	54	92%
eclipse-jdtcore	3,835	3,669	96%
j2sdk1.4.0-javax-swing	1,723	1,581	92%

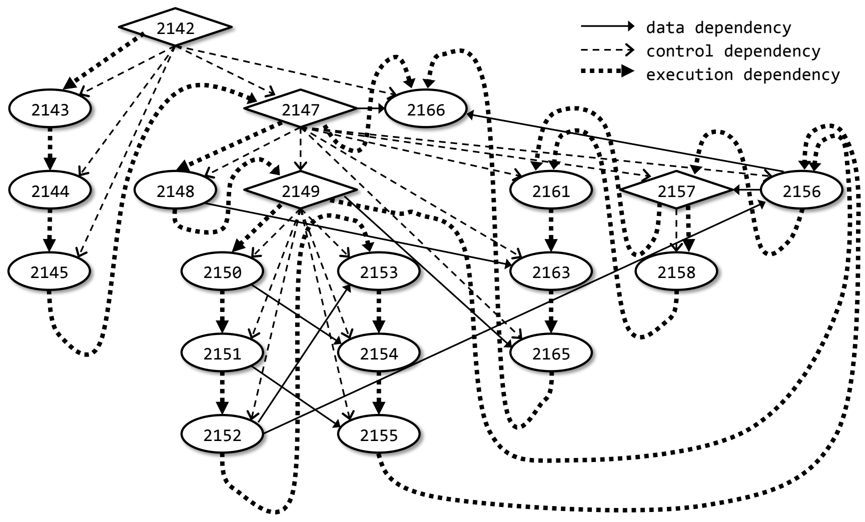


Figure 2: *PDG with execution dependencies*

I. Introducing execution-next link

We introduce a new kind of dependency into PDGs, which we call *execution-next link*. An execution-next link is an edge representing the order of executions of the program elements. There is an execution-next from node "v" to node "u" if the program element of "u" may be executed after the program element of "v" is executed. An execution-next link is exactly equivalent to an edge in control flow graphs. Execution-next links make it possible to detect consecutive program elements as clones even if they have neither data nor control dependency.

Figure 2 shows a PDG including execution-next links for the conventional PDG shown in Figure 1(b). By following execution-next links. slicings can reach node 2165 from node 2148 without going through nodes 2147 or 2166. Consequently, a code fragment from the line 2148 to the line 2165 can be detected as a clone.

II. Merging directly-connected equivalent nodes

We propose a technique to merge multiple nodes as a single node in PDGs. This technique is intended to reduce computational costs to detect similar subgraphs by decreasing the number of nodes in PDGs. Nodes that are directly connected via execution-next links are merged in this technique. For example, the nodes 2154 and 2155 in Figure 2 are merged by this technique. This proposed technique can reduce the following two types of the computational costs.

- If a merged node appears in the path of program slicing, the slicing cost is reduced. For example, in Figure 2, we need 3 hops from the node 2153 to the node 2156 mean while in Figure 3, only 2 hops are required for the slicing between the two nodes.

- If a merged node is used as a starting point of a slicing, the number of program slicings is decreased because some of nodes included in the same group are merged in this technique. For example, in Figure 2, the nodes 2154, 2155, 2163, and 2165 have the same hash values and they are grouped together. The number of program slicings starting from this group is ${}_4C_2 = 6$. However, most of the pairs are consecutive program statements in the source

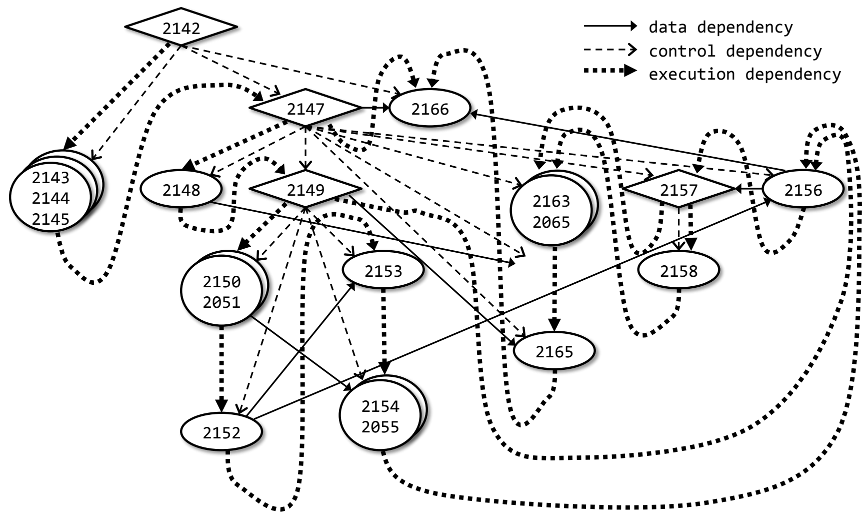


Figure 3: *PDG with merged nodes*

code, and pairwise slicings from such pairs does not detect human-wanted clones. On the other hand, if this proposed technique is applied to the PDG, each of the two pairs of nodes (2154, 2155) and (2163, 2165) is merged as a single node, respectively. Consequently, only a single pair of program slicings (${}_2C_2 = 1$) is performed from these nodes.

A set of nodes satisfying all the following conditions are merged as a single node in this proposed technique.

CONDITION1: there is a path from the node s to the node t,and each edge forming the path is execution-next link. Here, we assume such a path as R.

$$R \quad = \quad s, \cdots, t \tag{1}$$

CONDITION2: there is neither branch nor confluence on execition-next link in R.

CONDITION3: all the nodes in R have the same hash value.

CONDITION4: any path subsuming R does not satisfy both the CONDITION2 and CONDITION3.

Figure 3 is a PDG where this proposed technique has been applied to the PDG shown in Figure 2. In this example, the following groups of nodes in Figure 2 were merged.

- 2143, 2144, and 2145.
- 2150 and 2151.
- 2154 and 2155.
- 2163 and 2165.

The remainder of this subsection explains how we construct data, control, and execution-next links from/to merged nodes. We herein assume that node m was generated by merging all the nodes in R.

Data dependency: assume that $data_{to}(p)$ is the set of nodes having data dependencies to node p, and P is the set of the nodes consisting of path R, the set of nodes having data dependencies to the node m is defined the following formula:

$$data_{to}(m) \quad = \quad \bigcup_{p \in P} data_{to}(p) \cap \overline{P} \tag{2}$$

$\overline{P}$ means a complement set of P, which means $P \cup \overline{P}$ are all the nodes in the PDG. In the same way, assume that $data_{from}(p)$ is the set of nodes having data dependencies from the node p, $data_{from}(m)$ is defined as the following formula:

$$data_{from}(m) \quad = \quad \bigcup_{p \in P} data_{from}(p) \cap \overline{P} \tag{3}$$

Control dependency: CONDITION2 ensures that all the nodes in P have control dependencies from the same node. In this proposed technique, we define that the merged node m also has a control dependency from the same node. Consequently, a set of nodes having control dependencies to m is defined as the following formula:

$$control_{to}(m) \quad = \quad control_{to}(s) \tag{4}$$

CONDITION2 also ensures that there is no conditional predicate in P. Consequently, the set of control dependencies from the merged node m always becomes empty:

$$control_{from}(m) \quad = \quad \emptyset \tag{5}$$

Execution-next link: CONDITION2 ensures that there is neither branch nor confluence of execution-next link in R, so that it is very easy to define the sets of execution-next links from/to the merged node m:

$$execute_{to}(m) \quad = \quad execute_{to}(s) \tag{6}$$
$$execute_{from}(m) \quad = \quad execute_{from}(t) \tag{7}$$

This proposed technique also deems to be effective for reducing false positives. The nodes merged in this technique are consecutive program elements in the source code, and they are repeated instructions. There is a research report that many false positives are detected from the presence of repeated instructions in automatic clone detection [33]. Murakami et al. proposed a technique to reduce the amount of false positives by folding repeated instructions in token-based clone detection [34]. Consequently, even in the PDG-based clone detection, false positives should be reduced by removing repeated instructions.

III. Matching by subgraph

We propose calculating a hash value from every subgraph in PDGs not from every node. Herein, a subgraph is an edge and two nodes of its both ends. The number of subgraph in a PDG equals to the number of edges in the PDG. If given two subgraphs satisfy all of the following conditions, their hash values become the same:

- dependency types of the two subgraphs are the same,
- starting nodes of the two subgraphs are identical (starting nodes have the same hash value), and

- end nodes of the two subgraphs are identical (end nodes have the same hash value).

The clone detection algorithm using subgraph matching is the same as the basic algorithm shown in Subsection II except the matching is changed from the node unit to the subgraph unit. We show the difference between detections by the two units.

STEP1 a hash value is calculated from every of the subgraphs in PDGs not from every node. Subgraphs having the same hash value are grouped together.

STEP2 pairwise slicings are performed. They start from pairs of subgraphs in the same group and they follow pairs of the same subgraphs.

Note that, the basic algorithm shown in Subsection II performs a node-based detection, which means the number of nodes in a pair of slices are the same. On the other hand, this proposed technique performs a subgraph-based detection, so that the number of subgraphs in a pair of slices are the same, but the number of nodes are not necessarily the same.

IV. Implementation

We have implemented a clone detection tool, **SCORPIO**, with the proposed techniques. The tool is open to the public[4]. Currently, SCORPIO deals with only Java language. However, it is not difficult to expand it to other programming languages.

V. Investigation into RQs

We have conducted an experiment in order to answer the three RQs presented in Section III. We show the RQs again here.

RQ1 is detection capability improved by adding a edge for each pair of two nodes whose elements are consecutive in the source code?

RQ2 is the number of clone references detected with gap information different from the number of ones without gap information?

RQ3 How many intertwined clones are included in detection results of the PDG-based technique?

[4]`https://github.com/YoshikiHigo/TinyPDG`

Table 3: *Target software*

Software	short name	LOC	# of clone references
netbeans-javadoc	netbeans	14,360	55
eclipse-ant	ant	34,744	30
eclipse-jdtcore	jdtcore	147,634	1,345
j2sdk1.4.0-javax-swing	swing	204,037	777

I. Clone references

We used freely available clone data from Bellon's web site[5] as a reference (a set of clones that should be detected). In the remainder of this section, we use the following terms:

clone candidates clones detected by SCORPIO, and

clone references clones included in the references

As described in Section III, the clone references do not include information about locations of gaps for non-contiguous clones. Consequently, authors investigated the source code of all the non-contiguous clones in the references one by one, and added the information of gap locations. The references with gap locations are open to the public in authors' web site[6]. In this experiment, we used the clone references with gap locations.

II. Evaluation Indicators

Investigations into the RQ1 and RQ2 are performed by measuring precision and recall of detections. In order to measure the values, we need to check whether each of the clone candidates matches either of the clone references or not. We use the *good* value and the *ok* value [21] for checking the matching. The definitions of *ok* and *good* values are presented in Appendix. We use 0.7 as the threshold, which is the same value used in the literature [21].

Assume that C is a detection result (a set of clone candidates), R is the set of the clone references, and $R_{good}(C)$ and $R_{ok}(C)$ are sets of clone references whose *good* or *ok* value with either of the clone candidates in C is equal to or greater than the threshold.

Recalls of C by using *good* and *ok* values are defined as follows:

$$Recall_{good}(C) = \frac{|R_{good}(C)|}{|R|} \tag{8}$$

$$Recall_{ok}(C) = \frac{|R_{ok}(C)|}{|R|} \tag{9}$$

Precisions of C by using *good* and *ok* values are defined as follows:

$$Precision_{good}(C) = \frac{|R_{good}(C)|}{|C|} \tag{10}$$

$$Precision_{ok}(C) = \frac{|R_{ok}(C)|}{|C|} \tag{11}$$

This experiment has a limitation related to recall and precision. The clone references used in this experiment are not all clones included in the target systems. Consequently, the absolute values of recall and prevision are meaningless. They can be used only for relatively comparing multiple detection results.

III. Investigation for RQ1

In order to investigate RQ1, we detected clone candidates with the following three types of PDGs:

conventional PDG PDGs including data and control dependencies,

[5] `http://www.bauhaus-stuttgart.de/clones/`

[6] `http://sdl.ist.osaka-u.ac.jp/~h-murakm/2013_clone_references_with_gaps/`

execution PDG PDGs including execution-next links in addition to data and control dependencies, and

merged PDG PDGs having merged nodes.

In Bellon's study, each tool detected clones including at least 6 lines. Thus, we specified 6 nodes as the minimum threshold of SCOPIO's clone detection. Table 4 shows the number of clone candidates detected from each of the systems. For all of the systems, the execution and merged PDGs detected much more clone candidates than the conventional PDG. The number of clone candidates with the execution PDG is roughly the same as the number of clones with the merged PDG.

Table 5 shows clone references detected with each of the PDGs. The number of clone references detected with the execution or merged PDG is much more than the one with the conventional PDG.

Figure 4 shows recall and precision with the three PDGs. The recall values of the execution and merged PDGs are at least 2-fold higher than the ones of the conventional PDGs for all the systems. On the other hand, the precision values of the execution and merged PDGs are higher than the ones of the conventional PDGs in some cases meanwhile the conventional PDGs higher values than the other two PDGs in the other cases.

We calculated f-measure for comparing the detection accuracy of the three kinds of PDGs. Figure 5 shows the calculation result. All of the f-measure of the execution and merged PDGs are higher than the conventional PDG except the execution PDG of jdtcore with *ok* value. Especially, in the cases of netbeans and swing with *ok* value, f-measure of the execution and merged PDGs is 2-fold or more than the conventional PDG.

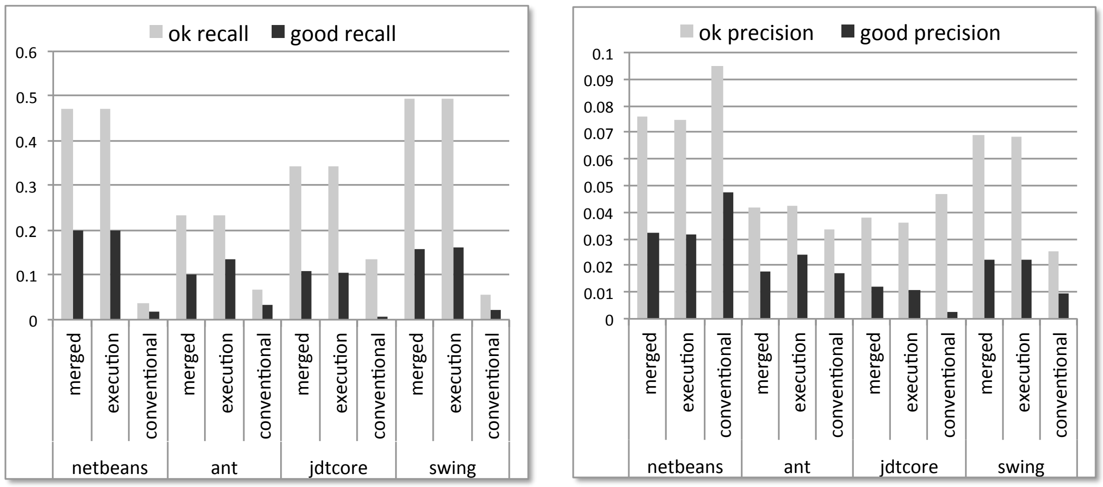
Consequently, our answer to **RQ1** is **YES**. By introducing execution-next links to PDGs, the number of detected clone candidates and clone references, so that recall of detection was improved. On the other hand, there were cases that precision of detection was decreased because of many clone candidates that are not matched with clone references. In the f-measure comparison, there was only one case that the value of the execution or merged PDG was lower than the conventional PDG. For the other 15 cases, f-measure was improved. Especially, for the 10 cases in them, the value of f-measure became 2-fold or more.

Table 4: *Number of clone candidates*

	netbeans	ant	jdtcore	swing
conventional	21	59	3,835	1,723
execution	349	165	12,675	5,639
merged	343	168	12,030	5,568

Table 5: *Number of detected clone references (the number in parentheses is the total number of clone references)*

	netbeans (55)		ant (30)		jdtcore (1,345)		swing (777)	
	good	ok	good	ok	good	ok	good	ok
conventional	1	2	1	2	10	181	16	44
execution	11	26	4	7	139	459	125	384
merged	11	26	3	7	148	459	124	384



(a) Recall (b) Precision

Figure 4: *Recall and Precision for merged, execution, and conventional PDGs*

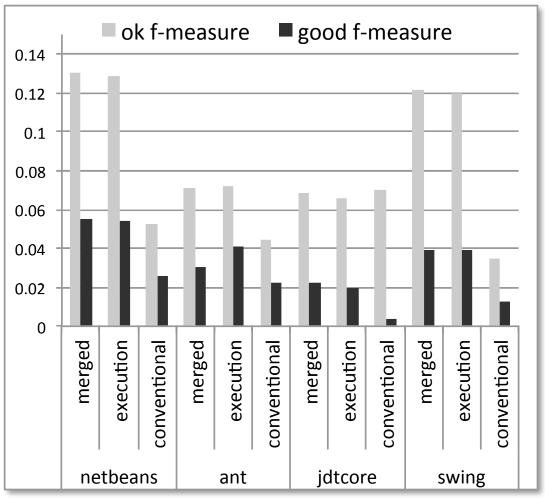


Figure 5: *F-measure for merged, execution, and conventional PDGs*

IV. Investigation for RQ2

In order to answer RQ2, we also investigated how many clone references were detected without locations of gaps. In the investigation for RQ2, we used the information of the start line and the end line for clone candidates and clone references, and we assume that all the lines between the start and end lines are included in clones. Table 6 shows the number of detected clone references. The values in Table 6 are the same or greater than the values in Table 5. That means matching without locations of gaps sometimes yields incorrect results.

In order to compare the difference between clone references detected with and without locations of gaps quantitatively, we calculated $Diff_{good}$ and $Diff_{ok}$ with the following formulae.

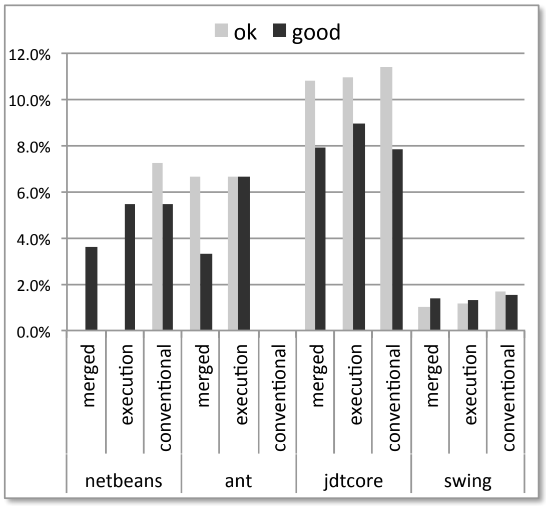


Figure 6: *Difference between with-gap and without-gap*

Herein, we assume that $R^{nogap}_{good}(C)$ and $R^{nogap}_{ok}(C)$ are a set of clone references detected without locations of gaps.

$$Diff_{good} = \frac{||R_{good}(C)| - |R^{nogap}_{good}(C)||}{|R|} \times 100 \quad (12)$$

$$Diff_{ok} = \frac{||R_{ok}(C)| - |R^{nogap}_{ok}(C)||}{|R|} \times 100 \quad (13)$$

Figure 6 shows the calculation result. There are only three cases that there is no difference between the clone references with and without locations of gaps. For the other 21 cases there are differences, and in the maximum case, the difference is over 13%.

Consequently, our answer to **RQ2** is **YES**. There are difference between clone references detected with and without locations with gaps in 21 out of 24 cases. In some of such cases, the difference is over 10%. Hence, we can say that it is important to consider locations of gaps for more accurate evaluation. The Bellon's references with locations of are open to the public in authors' web site[7].

[7] `http://sdl.ist.osaka-u.ac.jp/~h-murakm/2013_clone_references_with_gaps/`

Table 6: *Number of detected clone references (without gaps)*

Software	netbeans (55)		ant (30)		jdtcore (1,345)		swing (777)	
	good	ok	good	ok	good	ok	good	ok
conventional	4	6	1	2	115	334	28	57
execution	14	26	2	9	259	606	135	393
merged	13	26	2	9	254	604	135	392

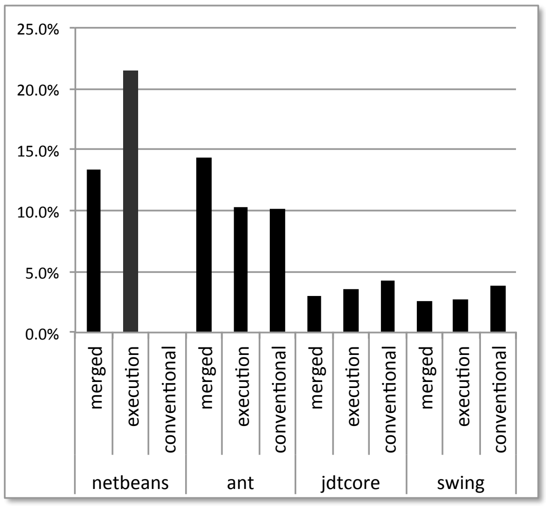


Figure 7: *Rate of intertwined candidates for all Difference between with-gap and without-gap*

V. Investigation for RQ3

In order to answer RQ3, we investigated how many intertwined clones were included for each of the detection results. Herein, we define that intertwined clones do not satisfy the following formula.

$$EL(c_1) < SL(c_2) \quad \vee \quad SL(c_1) > EL(c_2) \tag{14}$$

We counted the number of clones that do not satisfying the formula. Table 7 shows the result. There are intertwined clones in all the detection results except the conditional PDG of ant.

Figure 7 shows the rate of intertwined clones in the detection results. The rates are different from system to system, and more than 20% of clone candidates are intertwined clones in the maximum case.

Figure 8 shows a pair of intertwined clones detected from jdtcore. The lines started with "++" form a clone, and the lines started with –" form its corresponding clone. Each of the clones implements a procedure for Java field or method, respectively. These clones can be potential targets of refactoring such as extracting them as a new method [23] or reordering their statements for improving readability [35].

Consequently, our answer to **RQ3** is **Depending on target software**. There are intertwined cloned in all the detection results except the conventional PDG of ant. The rates of intertwined clones in the detection results varied from system to system, the rate was between 3 and 22%.

Table 7: *Number of intertwined clone candidates*

	netbeans	ant	jdtcore	swing
conventional	0	6	163	66
execution	75	17	459	156
merged	46	24	365	143

```
   this.interfacesCount = u2At(readOffset);
++ readOffset += 2;
   if (this.interfacesCount != 0) {
     this.interfaceNames = new char[this.interfacesCount][];
     for (int i = 0; i < this.interfacesCount; i++) {
       this.interfaceNames[i] = getConstantClassNameAt(u2At(readOffset));
       readOffset += 2; } }
   // Read the this.fields, use exception handlers to catch bad format
++ this.fieldsCount = u2At(readOffset);
-- readOffset += 2;
++ if (this.fieldsCount != 0) {
     FieldInfo field;
     this.fields = new FieldInfo[this.fieldsCount];
++   for (int i = 0; i < this.fieldsCount; i++) {
       field = new FieldInfo(reference, this.constantPoolOffsets, readOffset);
++     this.fields[i] = field;
++     readOffset += field.sizeInBytes(); } }
    // Read the this.methods
--  this.methodsCount = u2At(readOffset);
    readOffset += 2;
--  if (this.methodsCount != 0) {
      this.methods = new MethodInfo[this.methodsCount];
      MethodInfo method;
--    for (int i = 0; i < this.methodsCount; i++) {
        method = new MethodInfo(reference, this.constantPoolOffsets, readOffset);
--      this.methods[i] = method;
--      readOffset += method.sizeInBytes(); } }
    // Read the attributes
```

Figure 8: *Intertwined clones detected from* jdtcore

VI. Threats To Validity

I. Locations of gaps

In the experiment of this paper, we used a set of clones that should be detected. The set that we used is the Bellon's references, but locations of gaps were added by authors themselves. If someone else adds locations of gaps, the result (locations of gaps) might be different from our result. Our result is open to the public. Anyone can access to them.

II. No comparison with other detection techniques

In this experiment, we did not compare the proposed technique with other detection techniques intentionally. There are three reasons for that. The first reason is that the proposed techniques are for enhancing the PDG-based detection and the authors would like to concentrate on comparing proposed PDGs and conventional PDGs.

The second reason is that in Bellon's references, there are some clones each of which consists of multiple methods. Token-based or line-based detection techniques can detect such clones if they ignore boundaries of methods. On the other hand, the PDG-based detection can never find such clones. However we do not think incapability of detecting such clones is a drawback of the PDG-based detection. If we would like to detect such clones, we can use token-based or line-based techniques. That is, the authors would not like to have compared techniques of different categories in this paper.

The third reason is that Bellon's references do not include locations of gaps, which means it is impossible to fairly compare our PDG-based detection with the other techniques in the context of using locations of gaps.

Instead of comparing the proposed technique with other detection techniques such as line-based or token-based ones, we focuses on intertwined clones in this paper. Intertwined clones are included in detection result of the PDG-based technique meanwhile other detection techniques do not detect them. We also showed that intertwined clones can be potential targets of refactoring such as extracting them as a new method or reordering program statements in them.

VII. Related Work

Komondoor et al. first applied program slicing to clone detection [26]. In their method, program statements and control predicates are nodes of PDGs. Backward slicing is mainly used in the identification, and forward slicing is performed only from matching predicates. Their method was applied to several open-source software systems written in the C language. The application result demonstrated the capability of program slicing to detect non-contiguous clones.

Krinke proposed a fine-grained PDG, which he applied to clone detection [27]. In the fine-grained PDG, nodes are mapped one-to-one onto nodes of an abstract syntax tree, which means that the number of nodes in fine-grained PDGs is much greater than the number of nodes in traditional PDGs. Consequently, it takes longer to detect clones using fine-grained PDGs. In order to release this problem, he proposed a *k*-limited search in which similar subgraphs are searched within *k* hops.

Jia et al. proposed a hybrid method to detect non-contiguous clones [36]. In this method, contiguous clones are firstly detected using a suffix-comparison algorithm such as the line-based or token-based detection technique. Secondly, surrounding statements having control or data dependencies with the detected clones are merged if the corresponding clones also have the same surrounding statements. Their case study revealed that their hybrid method was able to detect non-contiguous clones more rapidly than Duplix, which is an implementation of Krinke's method [27]. The difference between the proposed method and their hybrid method is that their method requires core parts (contiguous clones being longer than a certain length) to detect non-contiguous clones, whereas the proposed method does not.

Gabel et al. proposed a scalable detection method for semantic code clones [37]. Their method is an enhanced version of an AST-based detection method, Deckard [38]. They defined the *semantic thread*, which was used for mapping and detecting similar subgraphs in PDGs, and for detecting similar subtrees in ASTs.

VIII. Conclusion

In this paper, we revisited characteristics of the PDG-based clone detection, and proposed a new technique for improving its detection capabilities. The key idea is introducing new links between every pair of two nodes whose program elements are consecutive in the source code. We implemented a PDG-based detection tool, SCORPIO, based on the proposed technique, and it is open to the public. In this paper, we described the result comparing detection results between the proposed technique and the conventional PDG-based technique. We used Bellon's references for the comparison. Before the comparison, we added locations of gaps into Bellon's references because they had not included gap locations. Hence, we was able to evaluate detection accuracy of the PDG-based detection with considering gaps in clones.

We confirmed the followings with the experiment: (1) the proposed technique improved detection capability of the PDG-based technique, (2) using clone references with locations of gaps yields different evaluation results from one without them in most case, and (3) intertwined clones are included in detection results of the PDG-based technique.

In the future, we are going to detect clones whose program elements are included in multiple methods. In order to archive such detection, we need to detect clones from system dependency graphs, which is made by connecting PDGs of methods based on method invocations in the source code. Similar subgraph are detected from the PDG of entire of the system. However, a system PDG is much larger than a method PDG, which mean we need techniques for reducing computational costs. We actually built a system PDG from 10000-line software and applied the conventional PDG-based detection technique to it. But, detecting clones lasted more than several hours. Detecting clones from system dependency graphs is challenging research topic.

We also conduct more deep investigations on intertwined clones. Although many research efforts conducted experiments on clone evolution, none of them used the PDG-based detection technique. Thus, in their experiments, intertwined clones were not investigated. Intertwined clone is a special kinds of clones ant we might find interesting findings from investigations on it.

Acknowledgements

This study has been supported by Grants-in-Aid for Scientific Research (S) (25220003), Grant-in-Aid for Exploratory Research (24650011) from Japan Society for the Promotion of Science, and Grand-in-Aid for Young Scientists (A) (24680002) from the Ministry of Education, Culture, Sports, Science and Technology.

References

[1] D. Rattan, R. Bhatia, M. Singh, Software clone detection: A systematic review, Information and Software Technology 55 (7) (2013) 1165 – 1199.

[2] C. K. Roy, J. R. Cordy, R. Koschke, Comparison and evaluation of code clone detection techniques and tools: A qualitative approach, Science of Computer Programming 74 (7) (2009) 470–495.

[3] C. Kapser, M. W. Godfrey, "Cloning considered harmful" considered harmful: patterns of cloning in software, Empirical Software Engineering 13 (6) (2008) 645–692.

[4] M. Kim, V. Sazawal, D. Notkin, G. Murphy, An empirical study of code clone genealogies, in: Proceedings of the 10th European Software Engineering Conference held jointly with 13th International Symposium on Foundations of Software Engineering, 2005, pp. 187–196.

[5] N. Göde, R. Koschke, Frequency and risks of changes to clones, in: Proceedings of the 33rd International Conference on Software Engineering, 2011, pp. 311–320.

[6] K. Hotta, Y. Sano, Y. Higo, S. Kusumoto, Is duplicate code more frequently modified than non-duplicate code in software evolution?: an empirical study on open source software, in: Proceedings of the Joint ERCIM Workshop on Software Evolution and International Workshop on Principles of Software Evolution, 2010, pp. 73–82.

[7] J. Krinke, Is cloned code older than non-cloned code?, in: Proceedings of the 5th International Workshop on Software Clones, 2011, pp. 28–33.

[8] M. Mondal, C. K. Roy, M. S. Rahman, R. K. Saha, J. Krinke, K. A. Schneider, Comparative stability of cloned and non-cloned code: an empirical study, in: Proceedings of the 27th Annual ACM Symposium on Applied Computing, 2012, pp. 1227–1234.

[9] Y. Higo, S. Kusumoto, How often do unintended inconsistencies happened? –deriving modification patterns and detecting overlooked code fragments–, in: Proceedings of the 28th International Conference on Software Maintenance, 2012, pp. 222–231.

[10] Z. Li, S. Lu, S. Myagmar, Y. Zhou, Cp-miner: Finding copy-paste and related bugs in large-scale software code, IEEE Transactions on Software Engineering 32 (3) (2006) 176–192.

[11] H. A. Basit, U. Ali, S. Haque, S. Jarzabek, Things structural clones tell that simple clones don't, in: Proceedings of the 28th International Conference on Software Maintenance, 2012, pp. 275–284.

[12] H. A. Nguyen, T. T. Nguyen, N. H. Pham, J. Al-Kofahi, T. N. Nguyen, Clone management for evolving software, IEEE Transactions on Software Engineering 38 (5) (2012) 1008–1026.

[13] T. Yamamoto, M. Matsushita, T. Kamiya, K. Inoue, Measuring similarity of large software systems based on source code correspondence, in: Proceedings of the 6th International Conference on Product Focused Software Process Improvement, 2005, pp. 530–544.

[14] T. Ishihara, K. Hotta, Y. Higo, H. Igaki, S. Kusumoto, Inter-project functional clone detection toward building libraries - an empirical study on 13,000 projects, in: Proceedings of the 19th Working Conference on Reverse Engineering, 2012, pp. 387–391.

[15] N. Göde, R. Koschke, Incremental clone detection, in: Proceedings of the 13th European Conference on Software Maintenance and Reengineering, 2009, pp. 219–228.

[16] B. Hummel, E. Juergens, L. Heinemann, M. Conradt, Index-based code clone detection: incremental, distributed, scalable, in: Proceedings of the 26th International Conference on Software Maintenance, 2010, pp. 1–9.

[17] R. Koschke, Large-scale inter-system clone detection using suffix trees and hashing, Journal of Software: Evolution and Process (2013) n/a–n/a.

[18] J. Ossher, H. Sajnani, C. Lopes, File cloning in open source java projects: The good, the bad, and the ugly, in: Proceedings of the 27th International Conference on Software Maintenance, 2011, pp. 283–292.

[19] Y. Sasaki, T. Yamamoto, Y. Hayase, K. Inoue, Finding File Clones in FreeBSD Ports Collection, in: Proceedings of the 7th Working Conference on Minging Software Repositories, 2010, pp. 102–105.

[20] J. Svajlenko, I. Keivanloo, C. K. Roy, Scaling Classical Clone Detection Tools for Ultra-Large Datasets: An Exploratory Study, in: Proceedings of the 7th International Workshop on Software Clones, 2013, pp. 16–22.

[21] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, E. Merlo, Comparison and evaluation of clone detection tools, IEEE Transactions on Software Engineering 33 (9) (2007) 577–591.

[22] K. Hotta, Y. Higo, S. Kusumoto, Identifying, tailoring, and suggesting form template method refactoring opportunities with program dependence graph, in: Proceedings of the 16th European Conference on Software Maintenance and Reengineering, 2012, pp. 53–62.

[23] M. Fowler, Refactoring: improving the design of existing code, Addison-Wesley Longman Publishing Co., Inc., 1999.

[24] M. Weiser, Program slicing, in: Proceedings of the 5th international conference on Software engineering, 1981, pp. 439–449.

[25] J. Ferrante, K. J. Ottenstein, J. D. Warren, The program dependence graph and its use in optimization, ACM Transactions on Programming Languages and Systems 9 (3) (1987) 319–349.

[26] R. Komondoor, S. Horwitz, Using slicing to identify duplication in source code, in: Proceedings of the 8th International Symposium on Static Analysis, 2001, pp. 40–56.

[27] J. Krinke, Identifying similar code with program dependence graphs, in: Proceedings of the 8th Working Conference on Reverse Engineering, 2001, pp. 301–309.

[28] B. S. Baker, On finding duplication and near-duplication in large software systems, in: Proceedings of the 2nd Working Conference on Reverse Engineering, 1995, pp. 86–95.

[29] I. D. Baxter, A. Yahin, L. Moura, M. a. Sant'Anna, L. Bier, Clone detection using abstract syntax trees, in: Proceedings of the 14th International Conference on Software Maintenanc e, 1998, pp. 368–377.

[30] T. Kamiya, S. Kusumoto, K. Inoue, CCFinder: A multilinguistic token-based code clone detection system f or large scale source code, IEEE Transactions on Software Engineering 28 (7) (2002) 654–670.

[31] J. Mayrand, C. Leblanc, E. Merlo, Experiment on the automatic detection of function clones in a software system using metrics, in: Proceedings of the 12th International Conference on Software Maintenance, 1996, pp. 244–253.

[32] S. Ducasse, M. Rieger, S. Demeyer, A language independent approach for detecting duplicated code, in: Proceedings of the 15th International Conference on Software Maintenance, 1999, pp. 109–118.

[33] Y. Higo, T. Kamiya, S. Kusumoto, K. a. Inoue, Method and implementation for investigating code clones in a software system, Information and Software Technology 49 (9-10).

[34] H. Murakami, K. Hotta, Y. Higo, H. Igaki, S. Kusumoto, Folding Repeated Instructions for Improving Token-based Code Clone Detection, in: Proceedings of the 12th Working Conference on Source Code Analysis and Manipulation, 2012, pp. 64–73.

[35] Y. Sasaki, Y. Higo, S. Kusumoto, Reordering program statements for improving readability, in: Proceedings of the 17th European Conference on Software Maintenance and Reengineering, 2013, pp. 361–364.

[36] Y. Jia, D. Binkley, M. Marman, J. Krinke, M. Matsushita, Kclone: A proposed approach to fast precise code clone detection, in: Proceedings of the 3rd International Workshop on Software Clones, 2009.

[37] M. Gabel, L. Jiang, Z. Su, Scalable detection of semantic clones, in: Proceedings of the 30th international conference on Software engineering, 2008, pp. 321–330.

[38] L. Jiang, G. Misherghi, Z. Su, S. Glondu, Deckard: Scalable and accurate tree-based detection of code clones, in: Proceedings of the 29th international conference on Software Engineering, 2007, pp. 96–105.

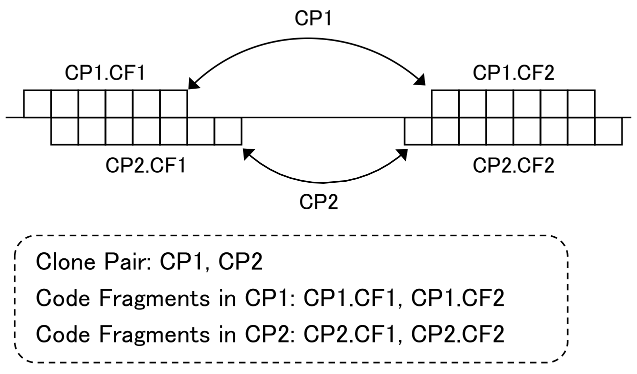

CP1
CP1.CF1
CP1.CF2
CP2.CF1
CP2.CF2
CP2
Clone Pair: CP1, CP2
Code Fragments in CP1: CP1.CF1, CP1.CF2
Code Fragments in CP2: CP2.CF1, CP2.CF2

Figure 9: *Sample calculation of good and ok values*

A. Definitions

Definition1 The rate of overlapping between two code fragments f_1 and f_2 is defined as the following formula. Assume that $lines(f)$ is a set of lines included in code fragment f.

$$overlap(f_1, f_2) = \frac{|lines(f_1) \cap lines(f_2)|}{|lines(f_1) \cup lines(f_2)|} \tag{15}$$

Definition2 The rate how code fragment f_1 is contained by another code fragment f_2.

$$contain(f_1, f_2) = \frac{|lines(f_1) \cap lines(f_2)|}{|lines(f_1)|} \tag{16}$$

Definition3 *good* value between two clone pairs (p_1 and p_2) is calculated by the following formula.

$$\begin{aligned} good(p_1, p_2) &= min(overlap(p_1.f_1, p_2.f_1), \\ &\quad overlap(p_1.f_2, p_2.f_2)) \end{aligned} \tag{17}$$

Figure 9 shows a situation that two clone pairs (p_1 and p_2) exists in the source code. In this situation, *good* value becomes $\frac{5}{8}$

$$good(p_1, p_2) = min(\frac{5}{8}, \frac{6}{8}) = \frac{5}{8}$$

If the threshold is 0.7, the two clone pairs are regarded as unmatched because of $\frac{5}{8} \leq 0.7$.

Definition4 *ok* value between two clone pairs (p_1 and p_2) is calculated by the following formula.

$$\begin{aligned} ok(CP_1, CP_2) \quad = \quad & min(max(contained(CP_1.CF_1, CP_2.CF_1), \\ & contained(CP_2.CF_1, CP_1.CF_1)), \\ & max(contained(CP_1.CF_2, CP_2.CF_2), \\ & contained(CP_2.CF_2, CP_1.CF_2))) \end{aligned} \qquad (18)$$

In the case of Figure 9, *ok* value becomes $\frac{5}{6}$

$$ok(p_1, p_2) = min(max(\frac{5}{6}, \frac{5}{7}), max(\frac{6}{6}, \frac{6}{8})) = \frac{5}{6}$$

If the threshold is 0.7, the two clone pairs are regarded as matched because of $0.7 \leq \frac{5}{6}$.

B. Effectiveness of merging PDG nodes

Table 8 shows the number of comparisons of pairwise slicings. By introducing execution-next link, the number of comparisons increased for all of the systems, however the merging technique release the increase in cost.

Table 8: *Number of comparisons*

	netbeans	ant	jdtcore	swing
conventional	16,992	140,595	14,166,700	7,055,758
execution	96,127	356,357	39,539,472	12,552,543
merged	81,194	340,413	35,192,633	11,928,501