Master Thesis

Title

Study on Code Clone Detection and Modification Support

Supervisor
Prof. Shinji KUSUMOTO

by
Tasuku Nakagawa

February 2, 2021

Departmant of Computer Science
Graduate School of Information Science and Technology
Osaka University

Master Thesis

Study on Code Clone Detection and Modification Support

Tasuku Nakagawa

Abstract

A code clone (in short, clone) is a code fragment that is identical or similar to other code fragments in source code. The presence of clones has a negative impact on software maintenance. For example, if a code fragment contains a bug, it is necessary to consider whether its clones should be modified or not. Therefore, in software maintenance, clone detection techniques that automatically detect clones from the target software and clone modification support for the developers are important. We have conducted studies about clone detection and modification support and proposed three techniques and systems. In this thesis, we report these studies.

First, we report a study about clone detection. Clones with a large number of edits, such as inserting or deleting statements, are called large-variance clones. Large-variance clones have a negative impact on software maintenance as well as general clones. Hence, it is necessary to detect them as well as general clones. On the other hand, due to the recent large-scale software development, techniques that can detect clones from large inputs in a scalable manner are required. However, there is no technique or tool that achieves both large-variance clone detection and high scalability. Consequently, we proposed a scalable large-variance clone detection technique and introduced its implementation, called NIL. NIL is a token-based clone detector and uses an N-gram representation of token sequences, an inverted index, and LCS to detect clones. Our experimental results show that NIL has higher accuracy on large-variance clone detection than existing state-of-the-art tools, and it is also good at scalability.

Second, we report a study about clone modification support. Clone modification support for developers is important in software maintenance. An existing study proposed a system that notifies developers of clone change information to support efficient clone modification. However, the existing system is premised on one execution a day and is not designed to be triggered by external factors except for time. Therefore, the system is difficult to be executed triggered by development workflow. Consequently, we focused on a pull request (PR), a part of the development workflow, and proposed CLIONE, a clone modification system aimed to integrate into PR-based development. CLIONE detects code fragments that need modifications by tracking clones when creating PRs, and CLIONE also notifies the developers of the code fragments as PR comments. We evaluated CLIONE in five experiments and confirmed that CLIONE was useful for supporting both PR-based development and developers.

Finally, we report a study about clone modification support, especially refactoring support. An existing study proposed a technique to estimate the number of reducible lines of code (LoC) by merging clones to support clone refactoring. However, the existing technique has two issues; (1) it cannot correctly judge whether or not clones can be refactored, and (2) it cannot estimate the number of reducible LoC about overlapping clones. Consequently, we proposed a technique to

measure the number of reducible LoC by merging clones more correctly. The proposed technique automatically repeats (1) detecting clones, (2) merging clones, and (3) compiling and testing the source code so that the technique can measure reducible LoC on clones that can be actually merged with considering clone overlapping. Our experimental results showed the proposed technique was able to measure the number of reducible LoC more correctly than the existing one.

Keywords

Code Clone
Software Maintenance
Clone Detection
Large-Variance Clone
Scalability
Clone Modification Support
Pull Request Based Development
Refactoring
Reducible Lines of Code

Contents

List of Figures

List of Tables

Part I
Introduction

1 Background

Software development consists of the following five phases [1].

- Requirement phase
- Design phase
- Implementation phase
- Testing phase
- Maintenance phase

It has been said that the maintenance phase is the most expensive [2,3]. Maintenance of software systems is defined as modifying a software product after its delivery to correct faults, improve the performance or other attributes, or adapt the software to a modified environment [4,5]. According to literature [6], approximately 80% of the total cost is spent on software maintenance. To modify software products, modifying the source code is unavoidable. Therefore, in order to support source code modification in software maintenance, many studies, such as automated program repair [7,8], fault localization [9,10], fault prediction [11,12], refactoring [13,14], and program slicing [15,16], have been conducted. In addition, the presence of *code clones* has been pointed out as one of the factors that make software maintenance difficult [17].

A code clone (in short, clone) is a code fragment identical or similar to other code fragments in source code. Clones are generated for various reasons, such as copy-and-paste operations. If a code fragment contains a bug and needs a modification, we have to consider whether its clones need the same modification or not. Hence, software containing many clones is difficult to maintain [18]. Therefore, many studies about clones have been conducted to support software maintenance. Among them, we considered that the following two research areas are particularly important; clone detection and clone modification support.

Clone detection

It is not realistic to manually compare a large number of code fragments in software to judge whether they are clones or not. Therefore, many studies to automatically detect clones from the target software have been conducted [19–22]. Some clones are easy to detect, such as simple copies, while others require special techniques for detection, such as code fragments that are semantically similar but not syntactically similar. In addition, due to recent large-scale software development, scalable clone detection techniques are required [23]. In software maintenance, it is an important issue to propose techniques that can detect clones difficult to detect in a scalable manner.

Clone modification support

Developers modify clones as needed, such as simultaneous clone modification [24] or clone refactoring [25]. Simultaneous clone modification is modifying clones consistently at the same time. Refactoring is improving the internal structure of software without changing its external behavior [26]. Especially, clone refactoring is merging clones into a single module, such as a method or a class. In software maintenance, it is an important issue to automatically detect

clone changes and provide them for the developers to improve the developers' efficiency of these tasks. Besides, refactoring changes source code that is working fine, and there is a possibility that refactoring itself introduces a new bug into the source code [27]. Therefore, developers need a reasonable indicator to refactor source code. In software maintenance, it is also an important issue to provide the indicator for the developers.

For these issues, we have conducted three studies. In this thesis, we report the results.

First, in Part II, we report a study about scalable large-variance clone detection. A large-variance clone is a clone generated by inserting or deleting a large number of statements in scattered places in a copy-and-pasted code fragment. Most of the existing tools are aimed at detection strongly similar clones and suffer in large-variance clone detection. On the other hand, the scalability of existing techniques that are specialized to detect large-variance clones is limited. Consequently, we proposed a scalable large-variance clone detection technique. The proposed technique is token-based. Based on a feature of large-variance clones, the proposed technique detects large-variance clones by measuring the similarity from the LCS between token sequences of two code fragments. Moreover, based on another feature of large-variance clones, the proposed technique also achieves scalable clone detection by identifying clone candidates using N-gram representation of token sequences and an inverted index. We implemented the proposed technique as a software tool, NIL. We compared NIL with four state-of-the-art clone detectors, including existing large-variance clone detectors. The results show NIL has high accuracy on large-variance clone detection, high scalability, and equivalent accuracy on general Type-1, Type-2, and Type-3 clone detection.

Next, in Part III, we report a study about a clone modification support system aimed to integrate into pull request (in short, PR) based development [28]. An existing study proposed a system, Clone Notifier, which supports developers to conduct simultaneous clone modification and clone refactoring by regularly notifying them of clone change information of a target project [29]. However, Clone Notifier is premised on one execution a day and is not designed to be triggered by external factors except for time [30, 31]. Therefore, the system is difficult to be executed triggered by development workflow, such as modifying source code or merging branches. Consequently, we focused on a PR, a part of the development workflow, and proposed CLIONE, a clone modification support system aimed to integrate into PR-based development. When developers create a PR, CLIONE notifies code fragments that need modifications by tracking clones between the PR and detecting clone changes. We evaluated the usefulness of CLIONE in five experiments. The experimental results show that CLIONE is useful to support PR-based development and developers.

Finally, in Part IV, we report a study about an indicator for refactoring [32]. An existing study proposed to use the number of reducible lines of code (in short, LoC) by merging clones as an indicator to refactor clones [33]. In addition, the existing study proposed a technique to estimate the number of the reducible LoC using JDeodorant [34], which judges whether clones can be refactored or not, and a greedy algorithm that was proposed in an existing study [35]. However, since JDeodorant, which is used in the existing technique, does not actually merge clones, it cannot correctly judge whether clones can be refactored or not. Moreover, the greedy algorithm cannot measure the reducible LoC from overlapped clones. Therefore, we considered that the existing technique cannot estimate the number of the reducible LoC correctly. Consequently, we proposed a technique to measure the number of the reducible LoC more correctly. The proposed technique

automatically repeats (1) detecting clones, (2) merging clones, and (3) compiling and testing the source code. We experimented to apply the proposed technique to several Java projects. We also compared the number of reducible LoC the proposed techniques measure with the existing technique's ones. As a result, we confirmed that the proposed technique was able to measure reducible LoC more correctly than the existing technique.

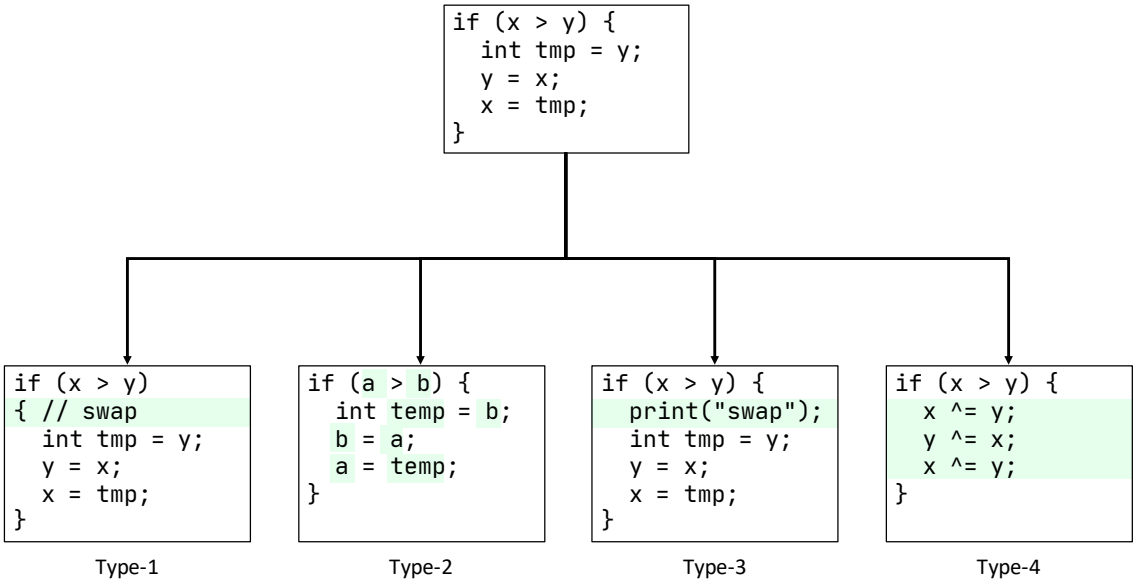


Figure 1: Examples of clone types

2 Preliminaries

2.1 Definition

A code fragment is a consecutive segment of source code. It can be represented by the tuple $(file\_name, start\_line, end\_line)$. A code block is a code fragment within braces. This study treats a function, which is a code block, as a clone detection unit, as done in previous studies [23,36,37]. Clones are code fragments identical or similar to other code fragments in source code. A pair of similar code fragments is called a clone pair. Clones are classified based on the degree of the similarity between them as follows.

Type-1 is an exact copy without modifications (except for white space and comments).

Type-2 is a syntactically identical copy; only variable, types, or function identifiers were changed.

Type-3 is a copy with further modifications; statements were changed, added, or removed.

Type-4 is a code fragment with not syntactical similarity but semantic similarity.

Figure 1 shows examples of clone types. Moreover, Type-3 clones are classified based on the degree of the similarity after pretty-printing and identifier/literal normalization as follows.

Very Strongly Type-3 is a Type3 clone whose similarity is in range [90, 100).

Strongly Type-3 is a Type3 clone whose similarity is in range [70, 90).

Moderately Type-3 is a Type3 clone whose similarity is in range [50, 70).

Weakly Type-3 is a Type3 clone whose similarity is in range [0, 50).

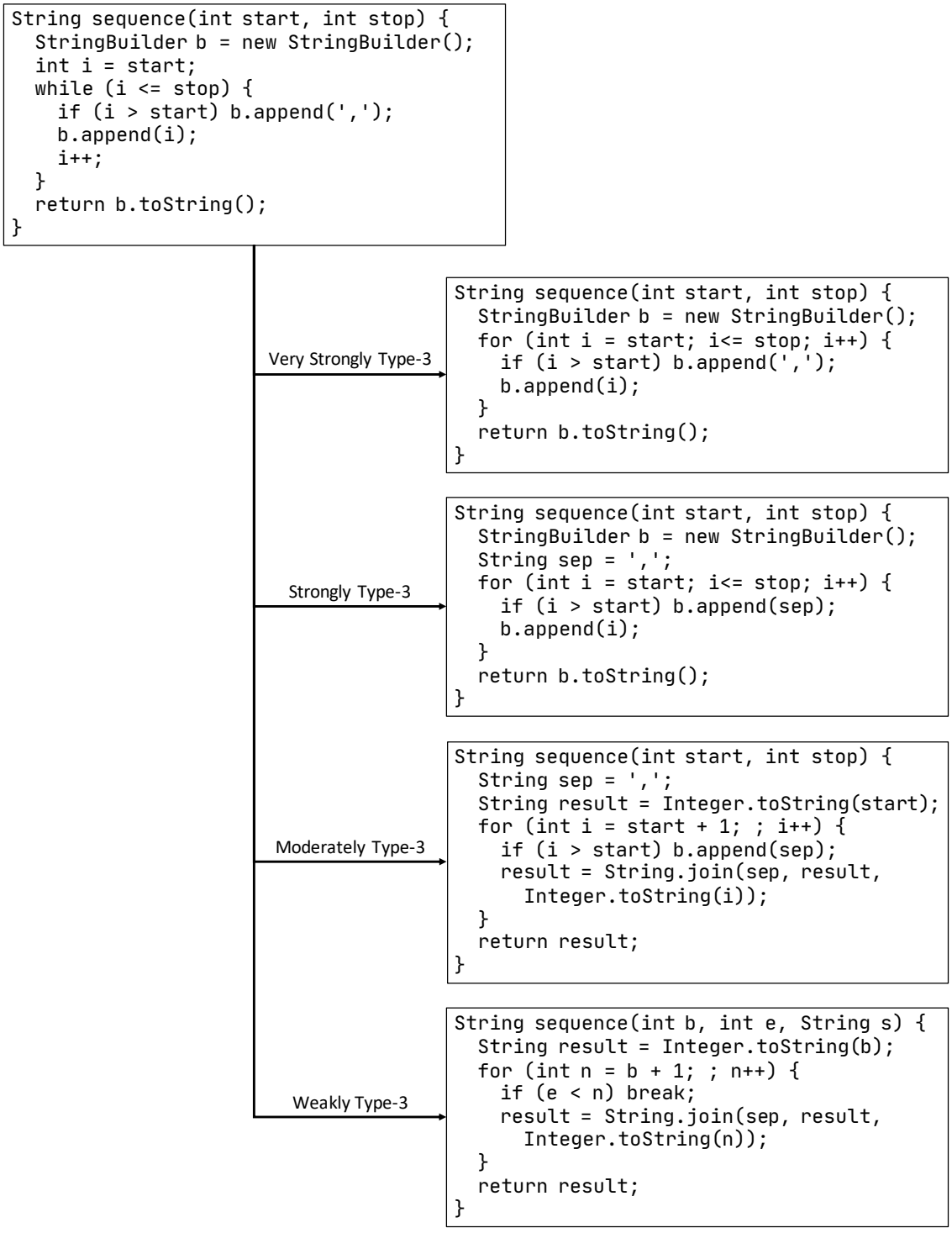


Figure 2: Examples of Type-3 clones

Figure 2 shows examples of these Type-3 clones.

The minimum length of clones is the minimum number of lines that a code fragment must be to be treated as a clone. It is often set to six lines or 50 tokens [19].

2.2 Causes of clone creation

Clones can be created or introduced in the following situations.

Copy-and-paste operations This is the most prevalent situation in which clones are created. Reusing code by copy-and-paste operations is common in software development because it is quite easy and enables us to develop software faster.

Stylized processing Processing used frequently (e.g., calculations of income tax, insertions in queues, and access to data structures) may cause code duplications.

Lack of suitable functions Developers may have to write similar processes with similar algorithms if they use programming languages that do not have abstract data types or local variables.

Performance improvement Developers can intentionally introduce code duplication to improve the performance of software systems if in-line expansion is not supported.

Automatically generated code Code generation tools automatically create code based on stylized code. As a result, if we use code generation tools to handle similar processes, the tools may generate similar code fragments.

Handling multiple platforms Software systems that can handle multiple operating systems or CPUs tend to include many clones in each platform's process handling.

Accident Different developers may write similar code accidentally. However, it is rare the amount of similar code generated accidentally becomes high.

2.3 Clone detection techniques

Clones have a negative impact on software maintenance. Many clone detection techniques and tools have been proposed and developed. Clone detection techniques can be categorized into the following categories.

Line-based techniques

Line-based techniques detect clones by comparing every line of code fragments as a string. They regard multiple consecutive lines that exceed a specified threshold as clones. Line-based techniques can detect clones quickly compared with other detection techniques because they do not require any preprocessing of the source code. However, they cannot detect clones that have different coding styles.

The techniques of Johnson [38] and Ducasse [39] are well-known line-based techniques. Their techniques compare every line of code after removing white space, tabs, and line breaks. Thus, they detect clones that have different coding styles and are language-independent.

Simian is one of the most famous and commonly used line-based clone detectors [40]. Simian

can handle about 20 programming languages (e.g., Java, C, and C++) and quickly detect clones.

Token-based techniques

First, token-based techniques transform source code into a token sequence. Then, they detect common sub-sequences of the tokens are clones. Compared to line-based techniques, token-based techniques are robust for code formatting. The detection speed of token-based techniques is inferior to that of line-based techniques. However, superior to the tree-based or graph-based techniques discussed below.

Kamiya et al. developed a token-based clone detector, named CCFinder [41], that is well-known and has been widely used by many developers. It replaces user-defined identifiers (e.g., method or variable names) with specific tokens. By preprocessing, CCFinder can detect Type-2 clones.

Li developed a token-based clone detector, named CP-Miner [42]. First, lexical and syntax analyses are performed on the source code. User-defined identifiers are replaced with specific tokens, as in CCFinder. The major difference between CP-Miner and CCFinder is the detection algorithms. In CP-Miner, hash values are calculated from every statement, and then a frequent pattern mining algorithm [43] is applied for detecting clones. In the frequent pattern mining algorithm, the hash values do not have to be consecutive. Thus, CP-Miner can detect Type-3 clones.

Tree-based techniques

In tree-based techniques, source code is transformed into a tree representation. Abstract syntax tree (AST) is one of the well-known tree representations. Tree-based techniques regard common sub-trees as clones, and thus these techniques also robust for code formatting. However, they have the disadvantages of requiring for detecting clones than do line-based and token-based techniques.

Baxter developed a tree-based clone detector, named CloneDR [44], which calculates various metrics based on ASTs and then detects clones by comparing metrics. Thus, CloneDR detects clones quickly in large software systems. CloneDR can also handle a lot of programming languages.

Koschke's technique [45] and Jiang's technique [46] are also tree-based approaches. In Koschke's technique, ASTs compared with a suffix tree algorithm to reduce the detection speed. Jiang's detector, named DECKARD, uses a locality-sensitive hashing algorithm [47] to detect clones. With the algorithm, DECKARD can detect Type-3 clones.

Graph-based techniques

In graph-based techniques, source code is transformed into a graph representation. Program dependence graph (PDG), one of the well-known graph representations, has data-dependence edges and control-dependence edges for each source code element. Graph-based techniques regard isomorphic sub-graph as clones. Because PDGs require a semantic analysis for their creation, these approaches require much more cost than other detection techniques. However, these approaches can detect clones with some differences that have no impact on the program behavior. Komondoor proposed the initial graph-based technique [48]. Komondoor's technique uses program slicing [15] to find isomorphic sub-graphs.

Krinke's technique [49] and Higo's technique [50] are also classified as graph-based techniques. Both of these techniques are designed to reduce detection cost. Krinke's technique sets the limit of the search range for finding isomorphic sub-graphs. Higo proposed a technique that aggregates nodes in PDGs under some conditions. Moreover, he introduced a new dependence edge named execution dependence edge for PDGs. By introducing the execution dependence edge, Higo's technique successfully detected clones that other graph-based techniques could not detect.

Text-based techniques

Text-based technique firstly normalizes the target source code and detects clones by comparing code fragments, such as methods or code blocks in the normalized source code.
NiCad [51] is a well-known text-based clone detection tool. First, it normalizes source code using TXL [52] and then detects clones by comparing code fragments based on Longest Common Subsequence algorithm. Moreover, text-based technique is used for detection clones generated by inserting or deleting a large number of statements in a copy-and-pasted code fragment. CCAligner [36] and LVMapper [37] also use TXL and detect such clones based on their own algorithms.

2.4 Large-gap clone

A large-gap clone is a clone generated by inserting or deleting a large number of statements in one place in a copy-and-pasted code fragment. Figure 3 shows an example of large-gap clones. In this example, a 10-line if-statement is inserted into Clone A (lines 4–13 of Clone B). Wang et al. [36] pointed out that existing clone detectors are incapable of large-gap clone detection because most target to the detection of near-miss clones. Wang et al. defined large-gap clone as follows. Consider two code blocks c_1 and c_2 with LOC values of L_1 and L_2, respectively, where $L_1 \leq L_2$. Let $\lambda = L_i/L_j$ (i.e., λ is the ratio of the code lengths of two code blocks). If c_1 and c_2 are Type-3 clones and the corresponding $\lambda \leq 0.7$, then these clones are large-gap clones. The clone pair shown in Figure 3 fits the definition of large-gap clones because the ratio of the code lengths of Clone A and Clone B is $12/22 \simeq 0.55 < 0.7$.

Wang et al. proposed CCAligner [36], a large-gap clone detector. CCAligner detects clones using a code window (a code fragment composed of k consecutive lines in a code block). First, CCAligner transforms code blocks into code windows. Then, it identifies clone candidates as pairs of code blocks that share at least one code window with considering e edit distance. Finally, it verifies clone candidates based on their similarity, which is calculated as follows:

$$sim(c_1, c_2) = \frac{|W_{c_1} \cap W_{c_2}|}{min(|W_{c_1}|, |W_{c_2}|)}$$

where c_1 and c_2 are two code blocks, and W_{c_1} and W_{c_2} are the corresponding sets of code windows, respectively.

2.5 Large-variance clone

A large-variance clone is a clone generated by inserting or deleting a large number of statements in various places in a copy-and-pasted code fragment. Figure 12 shows an example of large-variance

```
 1 protected int run(Commandline cmd) {
 2   try {
 3     Execute exe = new Execute(new LogStreamHandler(this,
         Project.MSG_INFO, Project.MSG_WARN);
 4     exe.setAntRun(getProject());
 5     exe.setWorkingDirectory(getProject().getBaseDir());
 6     exe.setCommandline(cmd.getCommandline);
 7     exe.setVMLauncher(false);
 8     return exe.execute();
 9   } catch (java.io.IOException e) {
10     throw new BuildException(e, getLocation());
11   }
12 }
```

(a) Clone A

```
 1 protected int run(Commandline cmd) {
 2   try {
 3     Execute exe = new Execute(new LogStreamHandler(this,
         Project.MSG_INFO, Project.MSG_WARN);
 4     if (serverPath != null) {
 5       String[] env = exe.getEnvironment();
 6       if (env == null) {
 7         env = new String[0];
 8       }
 9       String[] newEnv = new String[env.length + 1];
10       System.arrayCopy(env, 0, newEnv, 0, env.length);
11       newEnv[env.length] = "SSDIR=" + serverPath;
12       exe.setEnvironment(newEnv);
13     }
14     exe.setAntRun(getProject());
15     exe.setWorkingDirectory(getProject().getBaseDir());
16     exe.setCommandline(cmd.getCommandline);
17     exe.setVMLauncher(false);
18     return exe.execute();
19   } catch (java.io.IOException e) {
20     throw new BuildException(e, getLocation());
21   }
22 }
```

(b) Clone B

Figure 3: Example of large-gap clones

```
protected String getPrompt(InputRequest request) {
  String prompt = request.getPrompt();
  if (request instanceof MultipleChoiceInputRequest) {
    StringBuffer sb = new StringBuffer(prompt);
    sb.append("(");
    Enumeration e = ((MultipleChoiceInputRequest) request)
       .getChoices().elements();
    boolean first = true;
    while (e.hasMoreElements()) {
      if (!first) {
        sb.append(",");
      }
      sb.append(e.nextElement());
      first = false;
    }
    sb.append(")");
    prompt = sb.toString();
  }
  return prompt;
}
```

(a) Clone A

```
protected String getPrompt(InputRequest request) {
  String prompt = request.getPrompt();
  String def = request.getDefaultValue();
  if (request instanceOf MultipleInputChoiceRequest) {
    StringBuilder sb = new StringBuilder(prompt).append("(");
    boolean first = true;
    for (String next : ((MultipleInputChoiceRequest) request)
      .getChoices()) {
      if (!first) {
        sb.append(",");
      }
      if (next.equals(def)) {
        sb.append('|');
      }
      sb.append(next);
      if (next.equals(def)) {
        ab.append('|');
      }
      first = false;
    }
    sb.append(")");
    return sb.toString();
  }
  else if (def != null) {
    return prompt + "[" + def + "]";
  }
  else {
    return prompt;
  }
}
```

(b) Clone B

Figure 4: Example of large-variance clones

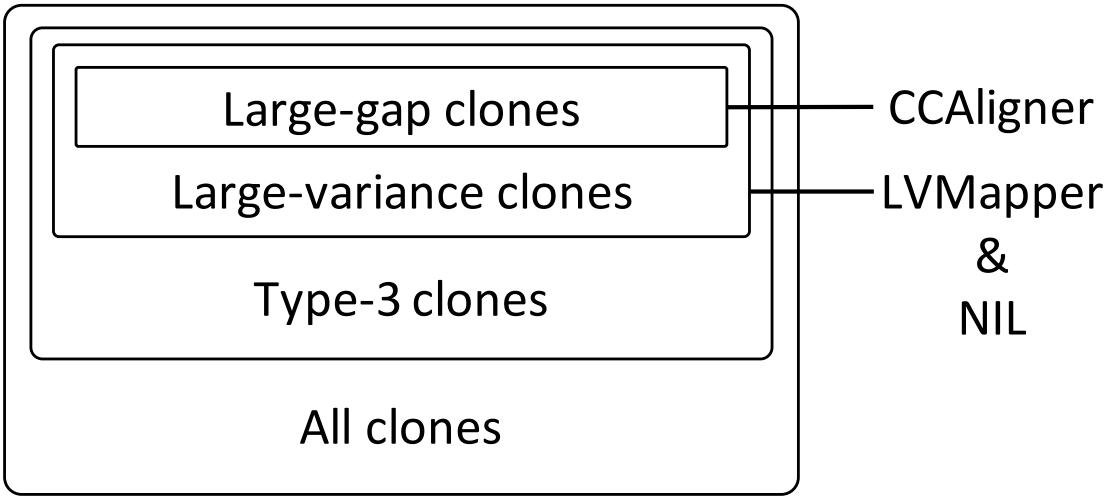


Figure 5: Relation among clone types and target clone types for several tools

clones. In this example, statements have been inserted into and deleted from various places in Clone A to create Clone B. Wu et al. [37] pointed out that CCAligner targets the detection of large-gap clones, making it incapable of large-variance clone detection. Wu et al. defined large-variance clones as clones whose code length ratio is less than 0.7. This means that large-gap clones are a special case of large-variance clones. They proposed LVMapper, a large-variance clone detector. Figure 5 shows that the relation among clone types and the clone types targeted by CCAligner, LVMapper, and NIL. CCAligner targets large-gap clones, whereas LVMapper and NIL target large-variance clones, which include large-gap clones.

LVMapper detects clones using code windows, just like CCAligner. Its clone detection has three phases, namely the locating, filtering and verifying phases. In the locating phase, LVMapper identifies pairs of code blocks that share at least one code window as clone candidates. Then, in the filtering phase, it calculates the proportions of common code windows for each clone candidate and removes clone candidates whose proportions are lower than filtering threshold θ. Finally, in the verifying phase, it verifies each clone candidate based on similarity measured using a common subsequence of lines between each clone candidate's code block pair.

2.6 Clone Notifier

Tokui et al. developed a tool, Clone Notifier, which supports developers to modify clones [29]. Clone Notifier takes two versions of a project as inputs, and based on the changes of the source code between the two versions, it classifies all clone sets detected in each version under following four categories.

Stable Clone set whose all clones have not been changed.

Changed Clone set whose some or all clones have been changed between the versions.

New Clone set whose all clones were added in the new version.

Deleted Clone set whose all clones were deleted in the old version.

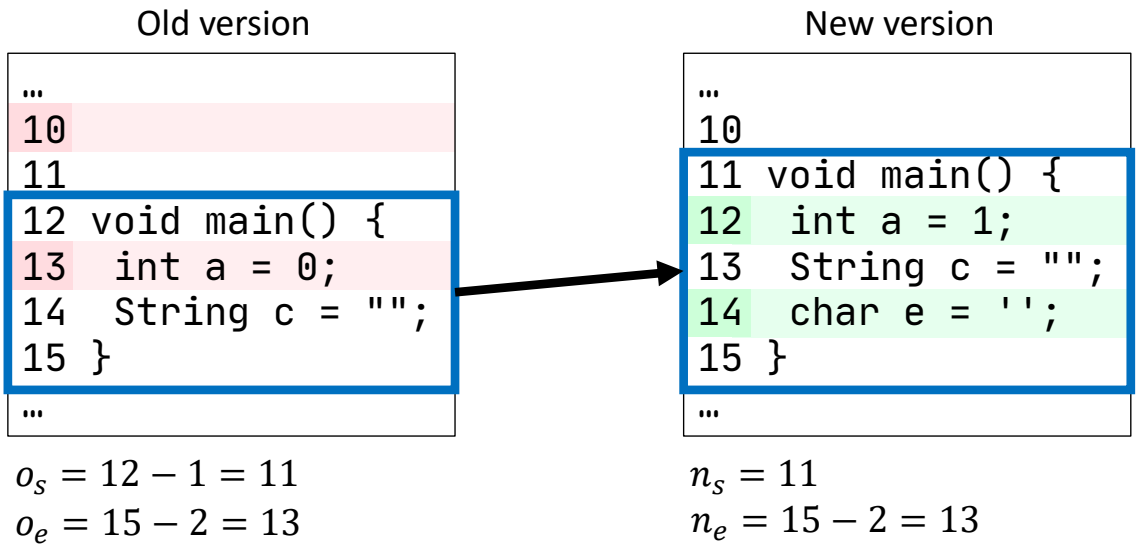


Figure 6: Example of clone tracking

First, Clone Notifier detects clones from each revision by using existing clone detectors [23,41,53]. Next, it tracks detected clones using *location overlaping function*, which an existing study [54] proposed.

Location overlapping function measures how much two code fragments $cf1$ and $cf2$ overlap each other ($0 \leq LO(cf1, cf2) \leq 1$). Clone Notifier uses the difference between the same file in each version, without the added and deleted lines. It computes the relative proportion of an overlapped region between $cf1$ and the calibrated $cf2$.

$$LO(cf1, cf2) = \frac{min(n_e, o_e) - max(n_s, o_s)}{n_e - n_s} \tag{1}$$

where $cf1$ in the old version spans from the line o_s to the line o_e, and the calibrated location of $cf2$ in the new version spans from the line n_s to the line n_e. If the location overlapping between the two clones is 0.3 or more, Clone Notifier tracks from the clone at the old version to the clone at the new version.

Figure 6 shows an example of clone tracking. In the figure, main method is tracked. In the old version, there are one and two deleted lines up to the start and end lines of main method, respectively. In the new version, there are zero and two added lines up to the start and end lines of main method, respectively. Therefore, based on $o_s = 12 - 1 = 11$, $o_e = 15 - 2 = 13$, $n_s = 11$, and $n_e = 15 - 2 = 13$, the result of location overlapping filtering is 1, and the method tracking succeeds.

Finally, Clone Notifier classifies clone sets under four categories (i.e., *Stable*, *Changed*, *Added*, and *Deleted*) from the results of clone tracking. Clone Notifier is designed to be premised on regular execution (e.g., once a day) [30].

2.7 Pull request based development

PR-based development is a development process using pull request (PR), which is one of the features of GitHub. PR is a feature that notifies developers of development information, such

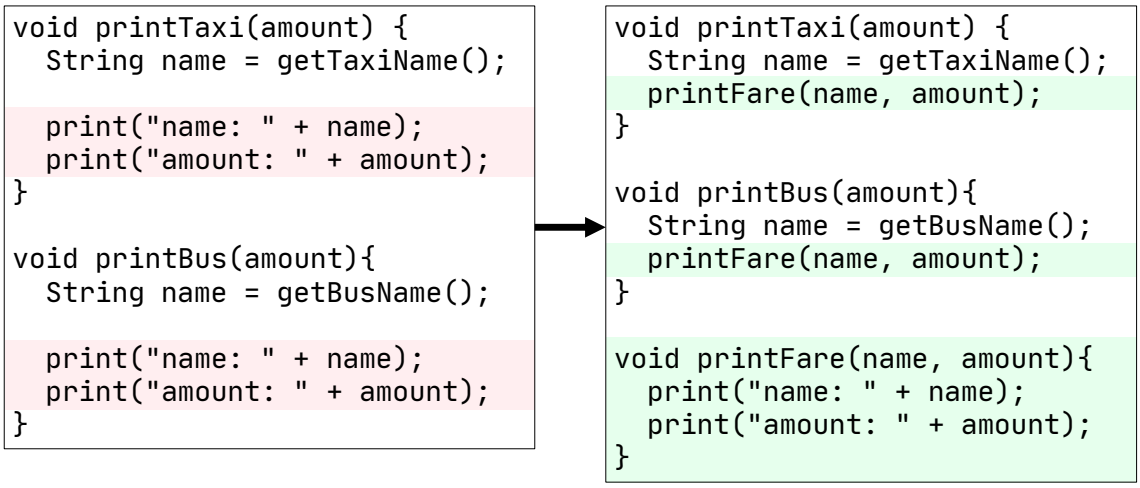


Figure 7: Example of extract method

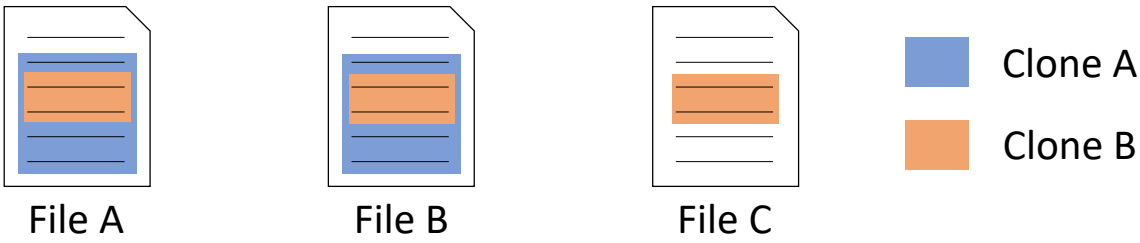
as source code changes before a branch is merged into another one. In PR-based development, the developers make source code edits, such as adding features, refactoring, bug fix on not the main branch but topic branches. Once the assigned work is done, the developer creates a PR and notify other developers of the change information, and if the changes have no problems, the topic branch is merged to the main branch. PR-based development is widely adopted by OSS development [55–57].

2.8 Clone refactoring

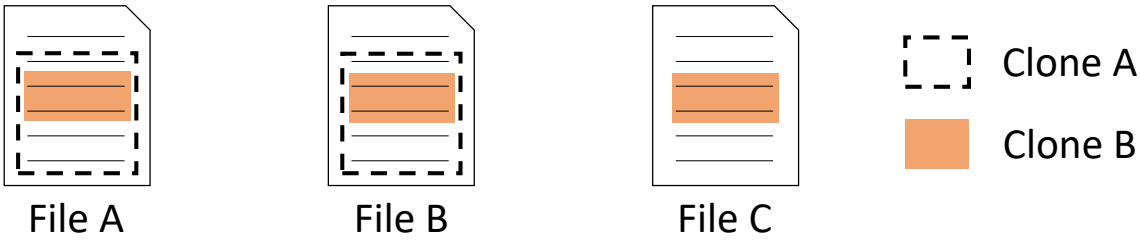
Refactoring is known as a promising technique to improve the internal structure of source code without changing its external behavior [26]. Duplicated code (clones) is one of the typical bad smells (code to be refactored). There are a variety of ways to refactor (merge) clones [25, 58, 59]. *Extract Method* refactoring is a simple yet well-known technique to remove clones. The original purpose of this refactoring pattern simplifies a long and/or complicated method by extracting a part of it as a new method. However, by applying *Extract Method* refactoring to clones, they can be removed. Figure 7 shows a simple example of *Extract Method* refactoring to two duplicated code fragments. Clones can be merged by extracting a clone set as a method. As a result, the number of potential bugs ans LoC can be reduced.

2.9 Clone overlapping

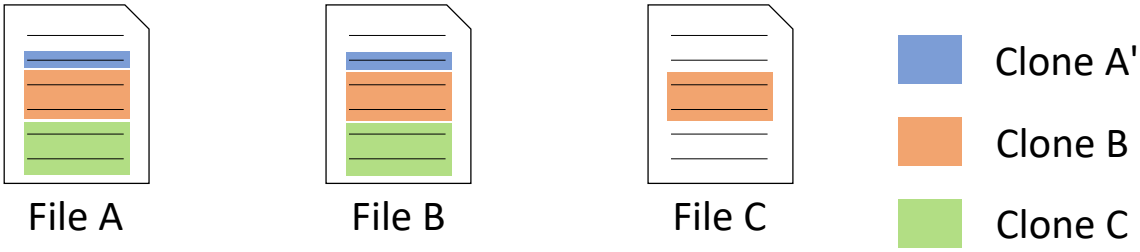
In this thesis, when two or more clones share one or more tokens, we say that "*the clones are overlapped with each other*". Figure 8(a) shows an example of overlapped clones. In this figure, clone A includes clone B. In clone refactoring, it may not be possible to refactor both of two clones overlapped with each other at the same time. Thus, when clones are overlapped like this figure, we need to do special care to estimate reducible LoC. For example, the existing technique in literature [35] estimates reducible LoC in the case of removing clone A and in case of removing clone B separately, and then remove only either A or B, whose reducible LoC is larger than the other. Literature [35] also proposes a more aggressive approach, which is shown in Figure 8(c).



(a) Overlapped clones



(b) Merging either one



(c) Dividing clones

Figure 8: Clone overlapping

In this approach, the overlapped parts and the non-overlapped parts are separately considered for refactoring.

2.10 Calculating reducible LoC

Herein, we introduce the definition of reducible LoC, which is proposed in the existing technique [35]. Herein, we assume that a clone set includes n code fragments, and each code fragment consists of C_{size} LoC. In a refactoring of merging the clone set, each clone is replaced with a method invocation, which is usually 1-line code. Consequently, reducible LoC C_{all} can be represented with the following formula.

$$C_{all} = n * C_{size} - n \tag{2}$$

In the case of Java, there is a 1-line code of method signature and open bracket "{" before a method body, and there is another 1-line code of close bracket "}". Thus, the LoC of a method becomes LoC of the method body plus two. An extracted code fragment (a clone) becomes a

body of the new method. Consequently, LoC of the extracted method can be represented with the following formula.

$$M = C_{size} + 2 \tag{3}$$

By using the two formula 2 and 3, the LoC difference between the original code and refactored code can be represented with the following formula.

$$S = C_{all} - M = (n-1) * C_{size} - n - 2 \tag{4}$$

In the existing technique [33], the above formula is used to estimate reducible LoC by removing clones.

Part II
Scalable Large-Variance Clone Detection

1 Background

A code clone (in short, clone) is a code fragment that is identical or similar to other code fragments in source code. Clones are generated by copying, pasting, and modifying code fragments for reuse [20, 60]. Clones are a major problem in software maintenance because they lead to bug propagation. Therefore, clone detection techniques, which automatically detect clones in the target codebase, are essential. Many clone detection techniques have been proposed [19, 21, 61], and applied in applications [62], such as refactoring [25, 58, 63, 64], debugging [42, 65, 66], and mining software repositories [67–69].

It is important for clone detection techniques to detect clones that have been heavily edited. A clone generated by inserting or deleting a large number of statements in one place in a copy-and-pasted code fragment is called a large-gap clone. Such clones are common in software development and should thus be detected along with general clones. Wang et al. pointed out that it is difficult for existing clone detectors to detect large-gap clones; they proposed a technique for detecting such clones and presented its implementation, called CCAligner [36]. Wu et al. pointed out that CCAligner targets only clones in which statement insertion or deletion is made in a single place and cannot detect clones in which modifications are scattered [37]. They called the latter type of clone large-variance clones and proposed LVMapper, a clone detector for large-variance clones.

It is also important for clone detection techniques to be scalable. Highly scalable clone detectors are required for analyzing large-scale projects or source files in an inter-project repository. Many scalable clone detectors have been proposed [23, 70, 71]. To achieve scalable clone detection, SourcererCC [23] and CloneWorks [70] use heuristics to reduce the number of code block comparisons needed to detect clones, and SAGA [71] uses a GPU to parallelize its clone detection process.

However, clone detectors that can detect clones with a large number of edits fail for large inputs [36] or require a long time to detect clones [37]. Scalable clone detectors target only identical or strongly similar clones (near-miss clones). They are incapable of detecting large-variance clones, in which many statements have been inserted or deleted. Therefore, the scalable detection of large-variance clones is challenging.

In this part, we propose a scalable technique for detecting large-variance clones and describe its implementation, called NIL[1], which uses an N-gram representation, an inverted index, and the longest common subsequence (LCS). NIL is a token-based clone detector. One of the features of large-variance clones is that the order of many tokens is preserved (i.e., the common subsequence between token sequences of large-variance clones is long). Hence, to detect large-variance clones, NIL measures the similarity between the token sequences of two code fragments based on the LCS. In addition, large-variance clones share many consecutive tokens. Hence, for scalable clone detection, NIL uses an N-gram representation of token sequences and an inverted index to reduce the number of code block comparisons needed to detect clones. First, NIL transforms code blocks extracted from source files into token sequences and creates an inverted index from the N-gram representation of the token sequences. Next, it identifies the clone candidates for each code block using the code block and the inverted index. Finally, it verifies the clone candidates by measuring the similarity between the code block and the clone candidates.

We evaluate NIL's (1) large-variance clone detection accuracy, (2) general Type-1, Type-2,

[1] A clone detector using N-gram, Inverted index, and LCS.

and Type-3 clone detection accuracy, and (3) scalability. We compared NIL with existing state-of-the-art tools, namely LVMapper [37], CCAligner [36], SourcererCC [23], and NiCad [51]. The experimental results show that NIL has a high precision of 87% in large-variance clone detection. It also has a high recall of 100%, as determined in our evaluation of large-variance clone detection using a mutation technique. In general clone detection, the accuracy of NIL is equivalent to that of the existing tools. In addition, we confirmed that NIL has high scalability; it can detect clones faster than the existing tools for large inputs (codebases with 250 MLOC).

The main contributions of this part are as follows.

1. We proposed a scalable technique for detecting large-variance clones. The proposed technique identifies clone candidates efficiently by using an N-gram representation of token sequences and an inverted index, and verifies clone candidates precisely by measuring the similarity between token sequences based on the LCS.

2. We implemented the proposed technique as a tool, called NIL. The executable file is available at `https://zenodo.org/record/4492665`.

3. We evaluated the usefulness of NIL through three experiments. The results show that NIL has high large-variance clone detection accuracy, high scalability, and equivalent general clone detection accuracy compared to that of existing state-of-the-art tools. We have published our experiment data to facilitate replication studies.

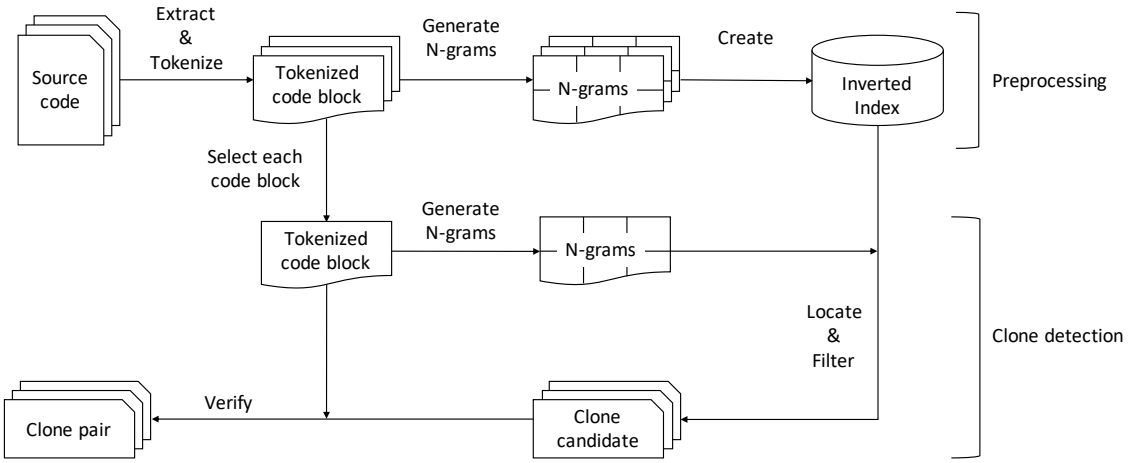


Figure 9: Overview of NIL

2 Approach

Figure 9 shows an overview of the proposed technique. The input is a set of source code files, and the output is the clone pairs in the source code. In the proposed technique, large-variance clones are detected based on the similarity between token sequences based on the LCS, taking advantage of the fact that the order of many tokens in a large-variance clone pair is preserved. In addition, large-variance clones share many consecutive tokens. Hence, to achieve scalable large-variance clone detection, the proposed technique reduces the number of code block comparisons by using an N-gram representation of token sequences and an inverted index. The proposed technique transforms code blocks in source code into token sequences in the *Preprocessing* phase and detects clones by comparing the token sequences in the *Clone detection* phase. In this part, we implemented the proposed technique as a tool, called NIL. NIL is written in the Kotlin language and currently targets only Java source code. The following subsections describe the *Preprocessing* and *Clone detection* phases.

2.1 Preprocessing

In the *Preprocessing* phase, NIL extracts code blocks from the target source code and transforms them into token sequences. NIL does not perform lexical analysis but simply divide each code block's text based on symbols (e.g., "+", "−", or braces), white spaces, or newlines, as done by SourcererCC. For example, when the code block shown in figure 4(a) is transformed into the token sequence, `protected`, `String`, `getPrompt`, `InputRequest`, `request`, `....` With this transformation, lexical analyzers for other languages do not need to be implemented to extend NIL. The token sequence transformation is fast because lexical analysis is not necessary. In addition, NIL has a relatively low rate of false positives because it does not normalize identifiers, such as variable and function names. However, it may not detect clones whose identifiers have been changed (i.e., Type-2 clones). We discuss the impact of the lack of identifier normalization in Section 3.

Next, NIL generates N-grams from each token sequence. An N-gram is a chunk of consecutive N tokens [72]. Figure 10 shows an example of generating 3-grams from the code block shown

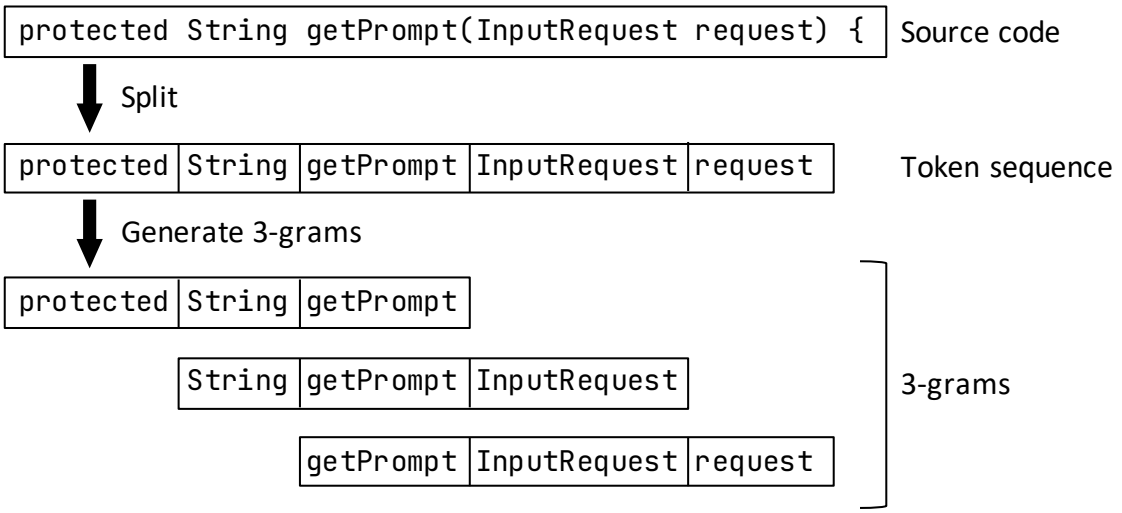


Figure 10: Example of generating 3-grams

in figure 4(a). In this example, three 3-grams are generated from the five tokens of on the first line in the code block. Even though large-variance clones include many modifications (statement insertions and deletions), many tokens other than the statements match consecutively (i.e., many N-grams match). Therefore, using N-grams is effective for scalable large-variance clone detection.

Then, NIL creates an inverted index from the generated N-grams. An inverted index is an information retrieval technology that allows the fast retrieval of documents that contains a word given as a query [73]. It is often used in clone detection techniques [74]. NIL uses a dictionary whose keys are the hash values of N-grams, and values are the code blocks containing the corresponding N-gram as an inverted index. All code blocks containing an N-gram can be quickly obtained by looking up the hash value of the N-gram in the inverted index. Therefore, a pair of code blocks that share an N-gram (i.e., the pair is possibly a large-variance clone pair) can be obtained quickly using the inverted index.

However, an inverted index consumes a lot of memory. Hence, creating an inverted index from all code blocks may lead to large memory consumption. To avoid this, we apply partial inverted indexes [70]. Code blocks are divided into several groups and an inverted index is created for each group (i.e., a partial inverted index), instead of creating a single inverted index for all code blocks. Figure 11 shows the concept of partial inverted indexes. First, code blocks extracted from source code are divided into n groups (Step 1), where n is set to a value such that the memory consumption of a partial inverted index is manageable. Next, an inverted index is created from one group of code blocks (Step 2). Based on the partial inverted index created in Step 2 and all code blocks, clone pairs between the code blocks in the group and all code blocks are detected in Step 3 (the clone detection process is described in the following Section 2.2). Steps 2 and 3 are performed for each group of code blocks.

2.2 Clone detection

After the *Preprocessing* phase, NIL performs *Clone detection* using the inverted index created in the *Preprocessing* phase and all code blocks. *Clone detection* is divided into three phases, namely

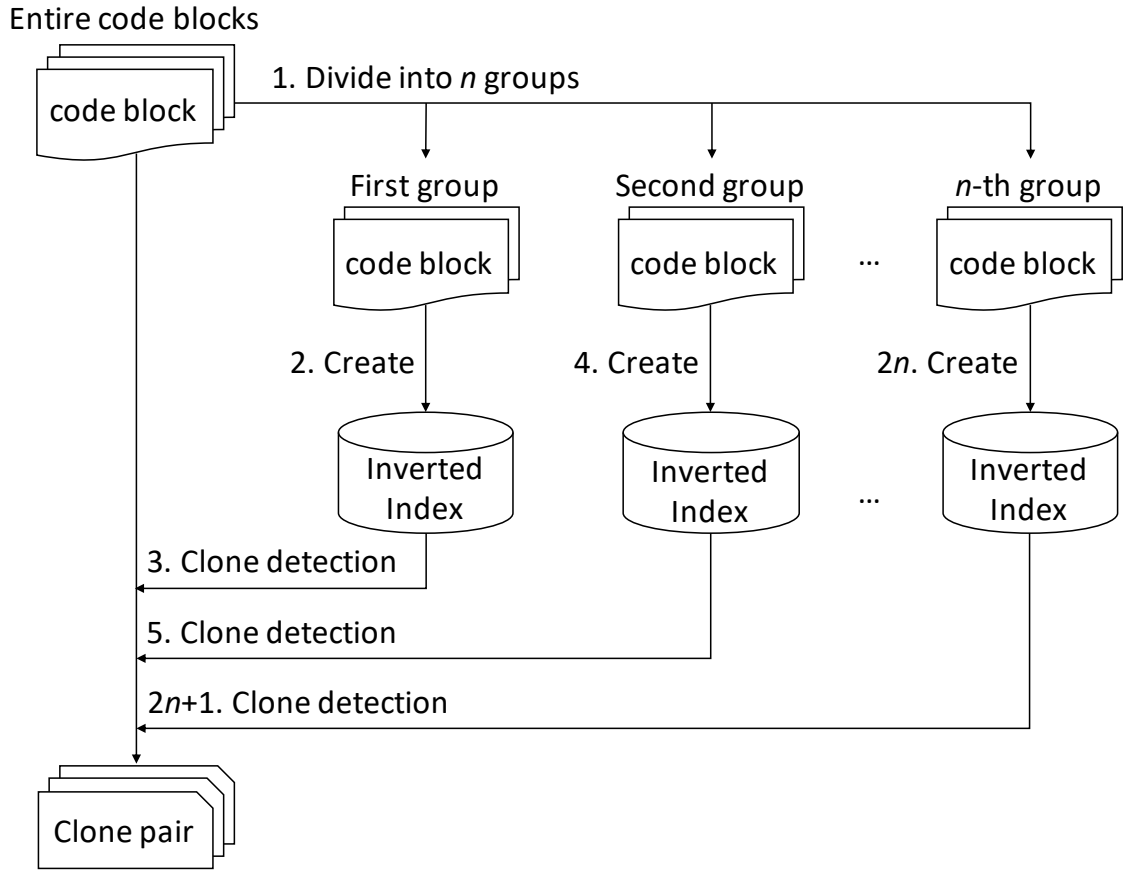


Figure 11: Concept of partial inverted indexes

location, filtration, and verification, as done by LVMapper [37]. First, NIL selects a code block from all code blocks prepared in the *Preprocessing* phase as the target code block. Then, in the location and filtration phases, NIL identifies the clone candidates of the target code block using an N-gram and the inverted index. Next, in the verification phase, NIL verifies that the target code block and the clone candidates are clone pairs by calculating the LCS. These phases are performed for each code block to detect all clone pairs in the target source code. Algorithm 1 shows the *Clone detection* algorithm. The three phases of *Clone detection* are described in detail below.

2.2.1 Location phase

In the location phase, NIL collects the clone candidates of the target code block using the inverted index. Lines 3–11 in Algorithm 1 are the location phase.

First, NIL generates N-grams from the token sequence of the target code block. $M-N+1$ N-grams are generated from a token sequence with length M. Next, a hash value is calculated for each N-gram. This hash value is used as a query when looking up values in the inverted index. Finally, NIL applies the hash values to the inverted index and collects code blocks that contain the N-gram whose hash value is the same as the given hash value. The obtained code blocks are referred to as the clone candidates of the target code block.

2.2.2 Filtration phase

In the filtration phase, NIL removes code blocks that unlikely to be clones from the clone candidates collected in the location phase. Lines 13–22 in Algorithm 1 are the filtration phase. It is necessary to reduce the number of clone candidates for scalable and fast clone detection because NIL performs the LCS calculation, which is a time-consuming process, in the verification phase. NIL filters clone candidates based on a feature of large-variance clones.

As described in Section 2.1, the two code blocks of a large-variance clone pair share a certain number of N-grams. Hence, if two code blocks share few N-grams, the pair is unlikely to be a large-variance clone pair. Based on this feature, NIL calculates $filtration\_sim$, defined below, between the target code block and each clone candidate.

$$filtration\_sim(c_1, c_2) = \frac{common\_ngrams(c_1, c_2)}{min(ngrams(c_1), ngrams(c_2))}$$

$$common\_ngrams(c_1, c_2) = |ngrams(c_1) \cap ngrams(c_2)|$$

where c_1 and c_2 are two code blocks with lengths $|c_1|$ and $|c_2|$, respectively. $ngrams(c_1)$ and $ngrams(c_2)$ are the numbers of N-grams generated from code blocks c_1 and c_2, respectively. Because of the large number of statement insertions and deletions in large-variance clones, the two code blocks may have significantly different token sequence lengths. We use min in the denominator so that $filtration\_sim$ can be properly calculated even such cases. NIL removes clone candidates whose $filtration\_sim$ is less than filtration threshold θ.

2.2.3 Verification phase

In the verification phase, NIL checks whether the target code block and each clone candidate are a true large-variance clone pair. Lines 24–32 in Algorithm 1 are the verification phase. As mentioned in Section 2.2.2, one of the features of large-variance clones is that the common subsequence between token sequences of large-variance clones is long even if there are a large number of insertions and deletions. Therefore, NIL calculates the LCS between the target code block and each clone candidate and measures the similarity of the pair based on the length of the LCS. The similarity function $verification\_sim(c_1, c_2)$ is expressed as following

$$verification\_sim(c_1, c_2) = \frac{lcs(c_1, c_2)}{min(|c_1|, |c_2|)}$$

where c_1 and c_2 are token sequences with lengths $|c_1|$ and $|c_2|$, respectively, and $lcs(c_1, c_2)$ is the length of the LCS between c_1 and c_2.

We use min as the denominator of the similarity function to detect large-variance clones, as done in the studies on CCAligner [36] and LVMapper [37] even if the lengths of the token sequences differ greatly.

Other clone detectors [37, 51] also use the LCS to measure similarity. However, they calculate line-based LCS whereas NIL calculates token-based LCS. In general, the token sequence of a code block is longer than its line sequence. The time complexity of a method for LCS calculation based

on dynamic programming is $O(|A| \times |B|)$, indicating a very long computation time for a large input size [75]. To reduce time complexity, NIL uses the Hunt-Szymanski algorithm [76]. With this algorithm, NIL can calculate the LCS in $O(r \log |A| + |B| \log |B|)$, where A and B are token sequences ($|A| \leq |B|$) and r is the number of pairs of common tokens between A and B.

Algorithm 1: Clone Detection

Input: C is a list of tokenized code blocks $\{c_1, c_2, \ldots, c_n\}$, Inverted Index I of C, N for size of N-gram, θ for filtering threshold, δ for verifying threshold

Output: All clone pairs CP

$CP \leftarrow \phi$;
for all each c_i in C **do**
  // Location phase
  // CC represents clone candidates
  $CC \leftarrow \phi$
  for $j = 1, 2, \ldots, (c_i.len - N + 1)$ **do**
    // $c_i[j]$ is j-th token in c_i's token sequence
    $n\_gram = concat(c_i[j], c_i[j+1], \ldots, c_i[j+N-1])$;
    $key = hash(n\_gram)$;
    // *get* is a function that returns values to which a given key is mapped in a given dictionary
    $CC = CC \cup get(I, key)$;
  end for

  // Filtration phase
  for all each cc_j in CC **do**
    /* *common_ngrams* is a function that computes the number of common N-grams between two given code blocks */
    $cn = common\_ngrams(c_i, cc_j)$;
    $m = min(c_i.len, cc_j.len)$;
    $filtration\_sim = cn/(m - N + 1)$;
    if $filtration\_sim < \theta$ **then**
      $CC = CC \setminus \{cc_j\}$;
    end if
  end for

  // Verification phase
  for all each cc_j in CC **do**
    /* *lcs* is a function that computes the length of the LCS between token sequences of two given code blocks */
    $lcs\_len = lcs(c_i, cc_j)$;
    $verification\_sim = lcs\_len/min(c_i.len, cc_j.len)$;
    if $verification\_sim \geq \delta$ **then**
      $CP = CP \cup (c_i, cc_j)$;
    end if
  end for
end for
return CP;

3 Evaluation

We evaluated NIL in terms of

- large-variance clone detection accuracy,
- general clone detection accuracy, and
- scalability.

In the following subsections, we first optimize the N-gram size based on a balance between recall and execution time. Next, we evaluate large-variance clone detection accuracy in terms of precision and recall. Then, we evaluate general clone detection using two commonly used benchmarks. Finally, we evaluate scalability by measuring execution time for various input sizes. Additionally, we compare the above results to those for four state-of-the-art tools [23, 36, 37, 51]. Table 1 shows these clone detectors and their settings. These settings were taken from the prior studies [23, 36, 37]. Note that the threshold δ for verification of LVMapper is variable and that δ takes the following values depending on the number of the lines of clone l.

$$\delta = \begin{cases} 0.7 & \text{if } 6 < l \leq 10, \\ 1 - 0.03 \times l & \text{if } 10 < l \leq 20, \\ 0.4 & \text{if } 20 < l \end{cases}$$

3.1 Summary

First, we summarize the results of this evaluation. We found that NIL has a high precision of 87% and a high recall of 100% in large-variance clone detection. These values are the highest among the tested large-variance clone detectors [36, 37]. In general Type-1, Type-2, and Type-3 clone detection, NIL's accuracy is equivalent to that of the existing clone detectors, including large-variance clone detectors, and its precision is higher than that of large-variance clone detectors. Moreover, we confirmed that NIL is the fastest at detecting clones in large codebases (1–250 MLOC) among the tested clone detectors.

Table 1: Settings for various clone detectors

Tool	Settings
LVMapper	Min length 6 lines, window size $k = 3$, filtering threshold $\theta = 0.1$, verification threshold δ is dynamic.
CCAligner	Min length 6 lines, window size $q = 6$, edit distance $e = 1$, min 60% similarity.
SourcererCC	Min length 6 lines, min 70% similarity.
NiCad	Min length 6 lines, max length 20,000 lines, blind renaming, identifier abstraction, min 70% similarity.

3.2 Parameter setting

NIL requires three parameters, namely N for N-grams, filtration threshold θ, and verification threshold δ. We set δ to 0.7, which is often used in clone detectors [23, 51]. We set θ to 0.1, as done for LVMapper. N must be carefully selected because it has a large impact on performance (e.g., execution time and clone detection accuracy). If N is set to a too small value, the recall of clone detection will increase because more code blocks will share the same N-grams. However, because more code blocks are identified as clone candidates in the location phase, the number of comparison targets in the filtration and verification phases increases, resulting in a longer execution time. Therefore, to optimize the N, we executed NIL with $N = 1$–9 and measured the execution time and clone detection recall for each N value. We used BigCloneEval [77] to measure recall. BigCloneEval automatically measures the recall of clone detectors using BigCloneBench [78].

Table 2 shows the results for each N value. For $N < 5$, an increase in N significantly reduces the execution time without lowering recall by more than one point compared to when $N = 1$. For $N = 5$ and 6, execution time decreases but recall also decreases. For $N > 6$, execution time does not significantly decrease. Therefore, considering the balance between execution time and recall, $N = 5$ is the optimal value.

Table 2: Recall and execution time results for each N value

N	1	2	3	4	5	6	7	8	9
Type-2	97.5	97.5	97.5	97.5	96.6	96.3	96.1	95.8	93.3
Very Strongly Type-3	93.5	93.5	93.5	93.5	93.5	93.1	92.9	92.7	92.5
Strongly Type-3	68.4	68.4	68.4	68.3	67.1	65.3	64.6	63.7	62.0
Moderately Type-3	11.2	11.2	11.2	11.1	10.6	9.9	9.3	8.7	7.8
Execution time	$>$ 24h	2h 13m 52s	21m 42s	5m 56s	2m 58s	1m 13s	1m 4s	1m 2s	57s

3.3 Large-variance clone detection

We evaluated the large-variance clone detection accuracy of NIL in terms of precision and recall and compared the results to those for existing large-variance and large-gap clone detectors, namely CCAligner and LVMapper.

3.3.1 Precision

Precision is the ratio of correct clones detected to all clones detected. A clone detector with higher precision provides more accurate results. In general, precision is measured via a manual validation of the clones detected by the target tool. In this part, we used Ant and Maven, which were used in the prior studies on CCAligner [36] and LVMapper [37] to measure precision. JDK1.2.2 and OpenNLP, also used in the above studies, were not used here because we were not able to find the source code for JDK1.2.2, which has been end-of-lifed and we considered OpenNLP to be unsuitable for manual validation because it is a machine learning library whose source code is difficult to read (e.g., it includes repetitive array manipulation code). We used the following procedure to measure the precision of each tool:

1. we input the target source code into each tool,
2. we randomly selected 100 large-variance clone pairs with more than 10 lines for each tool and target system, and
3. we manually confirmed whether the clone pairs were correct.

To remove bias in the manual validation, the detected large-variance clone pairs of a given tool were validated without knowledge of the tool used for detection. Table 3 shows the number of files and total LOC for Ant and Maven.

Table 4 shows the number of large-variance clone pairs detected by each tool and the precision for each tool[2]. The number of large-variance clones detected by NIL detected was almost same as that of LVMapper and more than that of CCAligner. The manual validation results indicate that NIL had a high average precision of 87% whereas LVMapper and CCAligner had low average precision values of about 60% and 40%, respectively. We considered this difference in precision to be due to LVMapper and CCAligner normalizing the identifiers in code blocks. After checking the large-variance clones detected by LVMapper and CCAligner, we found that code blocks with consecutive assignment statements, such as constructors, and consecutive `if`-statements were incorrectly detected as large-variance clones. In contrast, NIL detected large-variance clones more precisely because it does not perform identifier normalization.

Table 3: Target systems

Name	# Files	LoC
Ant 1.10.1	895	109,073
Maven 3.5.0	698	60,471

[2]Manual validation descriptions are available at `https://zenodo.org/record/4490845`

In addition, NIL had a higher precision than that of LVMapper and detected a similar number of large-variance clone pairs, indicating that it can detect many large-variance clones that LVMapper cannot. After checking large-variance clones that NIL detected but LVMapper did not, we found that the large-variance clone pairs share a small number of consecutive lines. LVMapper regards three consecutive lines in code blocks as code windows (see Section 2.4) and identifies clone candidates based on these code widows. Therefore, if a pair of code blocks shares a little or no code window, LVMapper cannot detect the pair as a large-variance clone pair. In contrast, NIL can detect such large-variance clones because it uses an N-gram representation of token sequences.

3.3.2 Recall

Recall is the ratio of clones detected by a tool to the total number of true clones in the target codebase. A clone detector with a higher recall can detect true clones more exhaustively. To evaluate recall, all true clone pairs in the target codebase are required. However, it is not realistic to manually check all code block pairs in a system to determine the total number of true clone pairs.

Therefore, we generated large-variance clone pairs automatically using mutation techniques to evaluate recall. Mutation techniques are frequently used for clone detector recall evaluation [79]. The studies on CCAligner [36] and LVMapper [80] used them to evaluate large-variance clone detection recall. In this part, we reproduced large-variance clones by randomly inserting various numbers of statements into original code blocks (i.e., the large-variance clones are mutants of the original code blocks). We targeted JDK6 and Apache Commons[3] because they were used in the study on CCAligner [36]. We randomly selected 100 functions with 15–50 lines in these systems as the original code blocks. A minimum length 15 lines is often used for recall evaluation using mutation techniques [23, 79]. We used a maximum length of 50 lines because if the number of lines of the original code blocks is too large, even if a large number of statements are inserted into the code blocks, the generated clones will not be large-variance clones. For example, if 20 lines of statements are inserted into an original code block, if the number of lines of a code block is 100, the ratio of lines of the clone pair is $100/120 > 0.7$, and thus the clone pair does not satisfy the large-variance clone definition. To reproduce large-variance clone pairs, 1–20 one-line statements

Table 4: Large-variance clone detection results

Tool	System	# Large-variance clones	Precision (%)
NIL	Ant	354	86.0
	Maven	398	88.0
LVMapper	Ant	355	64.0
	Maven	389	60.0
CCAligner	Ant	184	43.0
	Maven	284	40.0

[3]A project that provides open-source reusable Java components

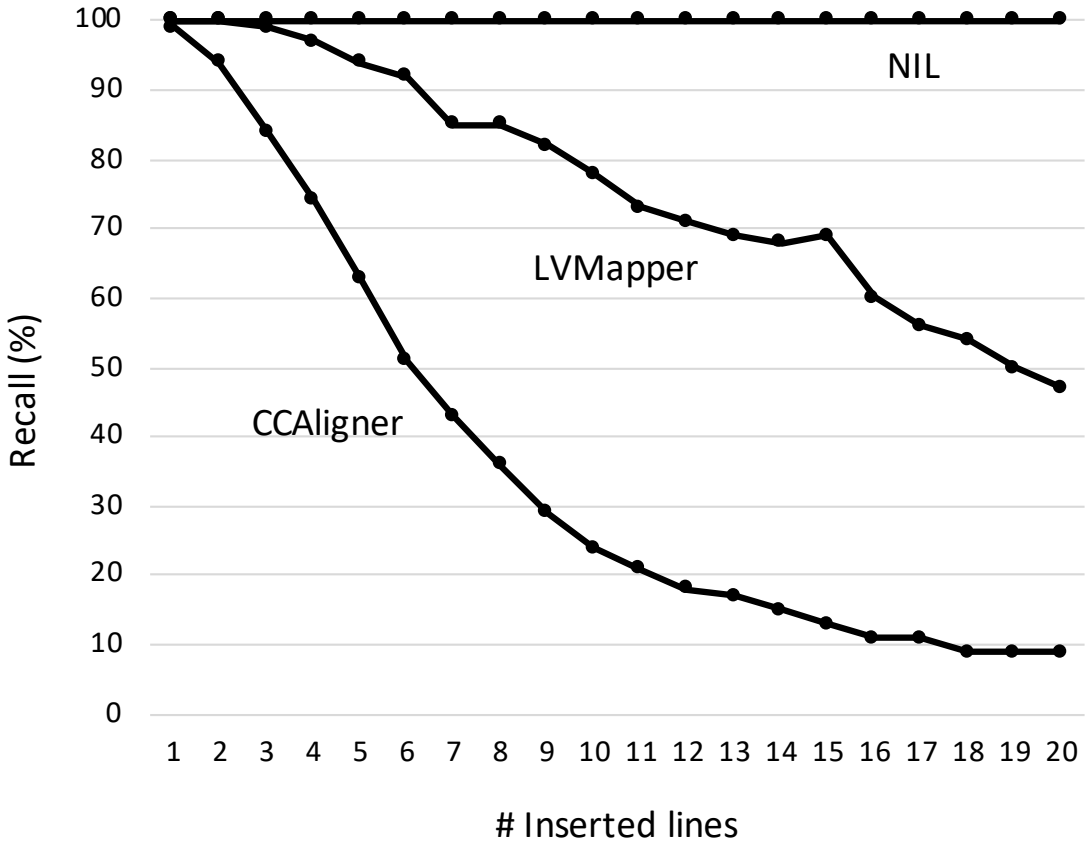


Figure 12: Recall results for various numbers of inserted lines

were inserted into each code block at random locations. 20 clone pairs were generated per original code block, for a total of 2,000 clone pairs, including large-variance clone pairs[4].

Figure 12 shows the recall of NIL, LVMapper, and CCAligner for clone pairs generated by inserting various numbers of statements. NIL detected all generated clone pairs. This is because even though many statements are inserted into a code block, the order of many tokens between the large-variance clone pair is preserved, and thus the clone pair shares a certain number of N-grams and has a long common subsequence. In summary, using N-gram-based clone candidate identification and token-LCS-based clone validation is effective in large-variance clone detection.

On the other hand, as shown in Figure 12, the recall of LVMapper and CCAligner decreased with increasing number of inserted lines. LVMapper can also detect clones in which a large number of statements are inserted because it verifies clones based on line-based LCS. However, our evaluation results show that the recall of LVMapper decreased with increasing number of inserted lines. This is because LVMapper failed to identify many large-variance clones in its locating phase. LVMapper identifies a pair of code blocks that share some code windows (see Section 2.4) as a clone pair. Therefore, with increasing number of inserted lines, the number of shared code windows decreases, and thus LVMapper failed to identify a pair of code blocks as a large-variance clone pair. CCAligner

[4]The generated large-variance clone pairs are available at `https://zenodo.org/record/4491016`

uses e-mismatch code windows to identify clone candidates, and this affects its recall. In addition, CCAligner uses code windows in verification and thus fails to detect large-variance clones.

3.4 General clone detection

We evaluated the general Type-1, Type-2, and Type-3 clone detection accuracy of NIL using two benchmarks, namely Mutation Framework [79] and BigCloneEval [77]. In addition, we compared the results of NIL to those of existing state-of-the-art tools, namely CCAligner, LVMapper, SourcererCC, and NiCad.

3.4.1 Mutation Framework

Mutation Framework automatically generates clone pairs based on mutation techniques. We executed Mutation Framework with all the default settings and input the generated clones[5] into each clone detector. Table 5 shows the results of recall for each tool measured by Mutation Framework. NIL detected all clone pairs generated by Mutation Framework.

3.4.2 BigCloneEval

BigCloneEval [77] automatically measures the recall of clone detectors using BigCloneBench [78]. We also measured precision, as done in the prior studies [23, 36, 37]. For each tool, we randomly selected 400 of the detected clone pairs from BigCloneBench and validated them manually. The clones were shuffled, and the validation was conducted without knowledge of the clone source.

Table 6 shows the results of recall[6] and precision for each tool on BigCloneBench[7]. As shown, NIL has a high recall of 96% for Type-2 clone detection even though it does not normalize identifiers in the *Preprocessing* phase. NIL had the second highest recall of Moderately Type-3 clones, which contain large-variance clones, behind only LVMapper. We considered that this is because normalizing identifiers is necessary to detect most Moderately Type-3 clones.

The precision (see bottom of Table 6) of both LVMapper and CCAligner, which are large-variance and large-gap clone detectors, was low. In contrast, that of NIL was very high (94%). Even though SourcererCC and NiCad also had high precision, they had poor large-variance clone detection performance. Therefore, compared to the existing tools, NIL has equivalent general clone detection accuracy and higher precision than that of the existing large-variance clone detectors.

Table 5: Recall results for Mutation Framework

Tool	NIL	LVMapper	CCAligner	SourcererCC	NiCad
Type-1	100.0	100.0	100.0	100.0	100.0
Type-2	100.0	100.0	100.0	100.0	100.0
Type-3	100.0	99.9	99.9	100.0	100.0

[5]The generated clone pairs are available at `https://zenodo.org/record/4491052`

[6]Note that in our experiments, BigCloneEval reported different recall of the existing clone detectors from the prior studies. Type-1 recall of some clone detectors was 99.9% because BigCloneBench contains faulty clone pairs [81].

[7]Manual validation descriptions are available at `https://zenodo.org/record/4493069`

3.5 Scalability

We evaluated the scalability of NIL using codebases with various sizes and compared the execution time of NIL to those of the existing tools. We used IJaDataset [82], a large inter-project Java dataset, as done in the prior studies [23, 36, 37]. We created datasets with 1, 10, 100, and 250 MLOC[8]. We used CLOC [83] to measure the LOC of the datasets. A computer with a quad-core CPU and 12 GB of memory was used for the evaluation, as done in the prior studies [23, 36].

Table 7 shows the execution times for each tool for various input sizes. As shown, the execution time of NIL is the shortest for all input sizes. Even though both LVMapper and SourcererCC detected clones from the 250-MLOC codebases, their execution times are longer than three days, indicating poor scalability. In addition, CCAligner and NiCad was not able to complete detecting clones from the 100- and 10-MLOC codebases, respectively, indicating their limited scalability. Therefore, NIL has the highest scalability.

Moreover, we examined how effective the location and filtration phases are for NIL's scalability. Figure 13 shows growth in the number of clone candidates with the increased number of code blocks. Note that this figure is a logarithmic graph. "Naive" is comparing all pairs of code blocks for clone detection[9]. As shown in the figure, both the location and filtration phases were able to drastically reduce the number of clone candidates. For example, when there were 10,000 code blocks, the number of clone candidates was able to be reduced from 49,995,000 in Naive to 1,449,634 in the location phase and 54,000 in the filtration phase. Therefore, the two phases are very effective for scalable clone detection.

Table 6: Recall and precision results for BigCloneBench

Tool	NIL	LVMapper	CCAligner	SourcererCC	NiCad
Type-1 Recall	99.9	99.9	99.8	93.8	99.9
Type-2 Recall	96.6	99.2	98.9	96.6	99.2
Very Strongly Type-3 Recall	93.5	98.1	97.4	68.2	98.4
Strongly Type-3 Recall	67.1	81.8	69.0	59.0	69.7
Moderately Type-3 Recall	10.6	19.1	10.0	4.8	0.5
Precision	94.0	58.5	33.7	99.2	80.2

Table 7: Scalability results

Tool	NIL	LVMapper	CCAligner	SourcererCC	NiCad
1 M	10s	29s	52s	3m 1s	1m 48s
10 M	1m 38s	13m 38s	26m 3s	27m 37s	—
100 M	1h 38m 29s	17h 23m 39s	—	19h 38m 5s	—
250 M	7h 40m 7s	3d 13h 47m 39s	—	5d 6h 55m 1s	—

[8]These datasets and an executable file of NIL are available at `https://zenodo.org/record/4491208`.

[9]The curve can be represented using $y = x(x-1)/2$ quadratic function where x is the number of code blocks in a project and y is the number of candidate comparisons carried out to detect all clone pairs.

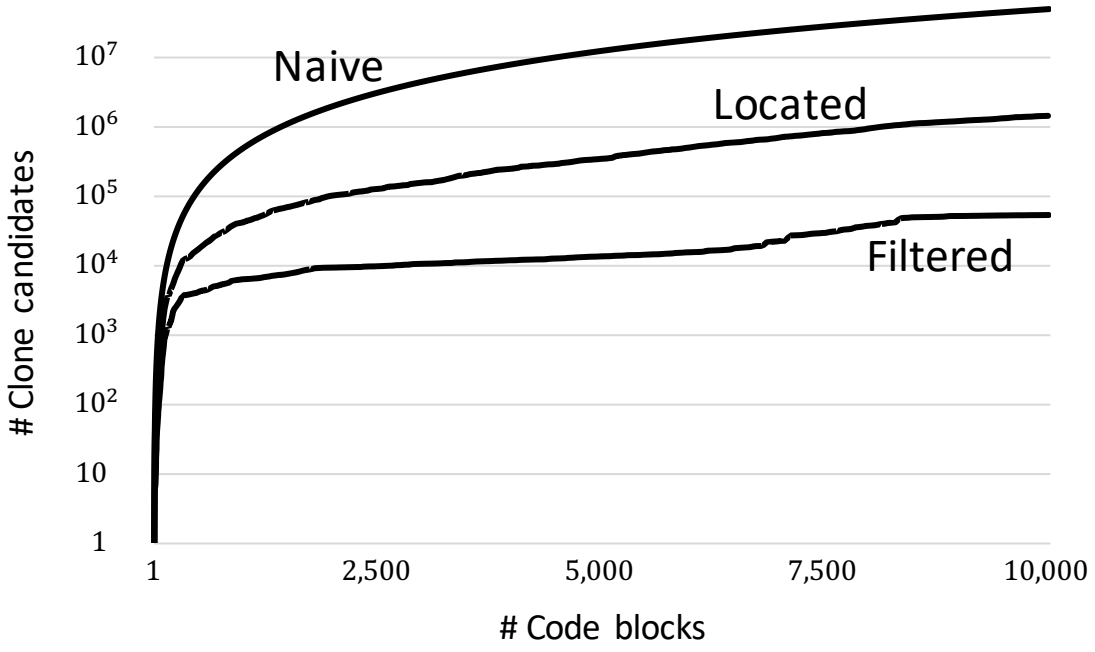


Figure 13: Growth in the number of clone candidates with the increased number of code blocks

4 Threats to Validity

To measure precision for each tool, we manually validated clones each tool detected, as done in the prior studies [23, 36, 37]. Because the clone detector names were not disclosed during the manual validation, there was no bias in the evaluation. However, because the criteria for whether a pair of code fragments is a clone pair can vary, manual validation by other researchers may yield different values. To ensure the validity of this study, the clones used in the manual validation are made public so that a third party can conduct replication studies.

In this study, we used the widely used benchmarks, BigCloneEval [77] and Mutation Framework [79] to evaluate the recall of clone detectors. Different results may be obtained using other benchmarks [19, 84, 85].

In this study, we targeted only the Java language. Different results may be obtained for other languages.

It is known that the accuracy and execution time of a clone detector is greatly influenced by its settings [86]. In this study, we optimized the N value for N-grams. However, the filtration threshold θ and the verification threshold δ were set based on values used by other clone detectors. The results can be improved by optimizing these values for NIL.

5 Related Works

Many clone detection techniques have been proposed to achieve various goals.

5.1 Complicated Type-3 clone detection

In addition to large-variance clones, since complicated Type-3 clones are difficult to detect, some techniques specialized for detecting them have been proposed.

Program dependence graph [87] (in short, PDG) is frequently used for complicated Type-3 clone detection. Krinke was the first to use PDGs for clone detection [49]. His technique detected isomorphic parts of PDGs as clones. He reported that PDG-based clone detection was good at recall and precision. Higo et al. pointed out that Kirinke's techniques suffered in the execution time and detecting contiguous clones. To enhance PDG-based clone detection, they introduced two PDG specializations and three heuristics into the PDG-based clone detection technique [50]. Zou et al. pointed out that the PDG-based clone detection techniques have still been quite time-consuming and missed many clones due to their exact graph matching using subgraph isomorphism. They proposed CCGraph [88], using an approximate graph matching algorithm based on the reforming Weisfeiler-Lehman graph kernel [89].

An intermediate representation (in short, IR) is also used for complicated Type-3 clone detection. Some syntactical differences (e.g., `for`-loop and `while`-loop) in source code are transformed into the same or similar instructions in IRs of the source code. Selim et al. proposed a clone detection technique using Java IRs generated by Soot [90, 91]. Their technique transforms Java source code into their IRs and detects clones on the IRs using existing tools [40, 41], which were not able to detect Type-3 clones. They reported that they detected Type-3 clones by executing the tools on IRs of source code. Caldeira et al. also proposed a clone detection technique using IRs [92]. They devised a C-like IR based on LLVM and ran NiCad [51] on it. Their experimental results suggested that IRs were beneficial for improving clone detection and IRs had a large impact on complicated Type-3 clones.

Machine learning is also useful for complicated Type-3 clone detection. Saini et al. proposed a clone detector, Oreo, for Weakly Type-3 clones [93]. It combines machine learning, information retrieval techniques, and software metrics to detect clones.

However, those three techniques based on PDG, IR, or machine learning do not always detect large-variance clones. For example, a prior study showed that Oreo has a higher precision but lower recall in large-variance clone detection than those of LVMapper [37]. Moreover, these techniques require a long execution time and are limited in scalability. PDG-based clone detection requires a long time to construct PDGs and perform subgraph isomorphism. IR-based clone detection requires a long time to transform source code into IRs. Machine learning-based clone detection is necessary to complete training before clone detection.

5.2 Scalable clone detection

Kamiya et al. proposed CCFinder [41] and its successor, CCFinderX [94]. CCFinder transforms the target source code into a token sequence, normalizes identifiers, such as variable names and literals, and then uses a suffix tree algorithm to detect matching token sequences as clones. As

shown in prior studies [23, 71], CCFinderX has high scalability and can detect clones even for a 100-MLOC codebase.

Ishihara et al. proposed a scalable method-level clone detection technique [95]. The technique hashes each normalized method and detects methods whose hash values are the same as clones. They reported that they detected clones in a large codebase (3.5 MLOC) in 3.5 hours.

Hummel et al. proposed ConQat [74, 96], an index-based clone detector. ConQat creates clone indexes by hashing consecutive six lines of source code and detects the clone indexes whose hash values are the same as clones. ConQat is capable of distributed processing in clone detection, so it can be applied for ultra-large codebase (2.9 GLOC) using cluster computing.

However, those scalable clone detectors cannot detect gapped clones due to their algorithms. Though there are several tools for scalable near-miss clone detection [23,70,71,97], they still cannot detect complicated Type-3 clones, including large-variance clones. In this part, we proposed NIL, which achieves both large-variance clone detection and scalability.

6 Conclusion

In this part, we proposed a clone detection technique for the scalable detection of large-variance clones from a large codebase and described its implementation, called NIL. NIL uses N-grams, an inverted index, and the LCS to detect large-variance clones. Our experimental results show that NIL has higher precision and recall in large-variance clone detection than those of existing large-variance and large-gap clone detectors. In addition, the general Type-1, Type-2, and Type-3 clone detection accuracy of NIL is equivalent to that of existing state-of-the-art tools. Moreover, NIL can detect clones from large codebases more quickly than do existing clone detectors.

As future works, we consider doing research on software engineering applications such as code recommendation and completion, refactoring, and bug propagation for large-variance clones using NIL.

Part III

Clone Modification Support for Pull Request Based Development

1 Background

There are many studies on simultaneous modification and refactoring support for clones [24, 25] because clones have been pointed out as one of the major problems in the maintenance process of software development. An existing study proposed a clone change notification tool, Clone Notifier, to improve the efficiency of these tasks [29]. Clone Notifier takes two versions of a target project as inputs and notifies developers of information about clones changed between the versions. Clone Notifier is designed for industry use [30] and is premised on one execution a day. On the other hand, Clone Notifier is difficult to be executed triggered by development workflow, such as modifying source code or merging branches, because it is not designed to be triggered by external factors except for time. For this reason, we think there are the following issues when considering the use of Clone Notifier in development, such as OSS development, in which modifying source code or merging branches are frequently conducted [98].

- The first issue is that even if code fragments need modifications due to clone changes, the code fragments will not be notified promptly. Because Clone Notifier is executed once a day, once Clone Notifier has been executed, there is a day interval until the next execution. Suppose Clone Notifier users set its execution interval shorter (e.g., five minutes) to solve this issue. In that case, the information of clones is notified to the users frequently even if the source code is not modified. Hence, the notification probably annoys the users. On the other hand, if code fragments to be modified are notified to the developers in line with the development workflow, such as source code modifications, the developers will be able to respond to the code fragments promptly without feeling such annoying.

- The second issue is that a large number of notifications are possibly sent to developers at once. If a large number of source code modifications are conducted, and as a result, many clones may be changed improperly (i.e., many code fragments need modifications), the information is notified all at once. In this case, the users will be forced to check a large amount of clone change information, and they may overlook serious clone changes. If clone change notification is triggered by development workflow, code fragments to be modified are notified in line with each source code modification so that the developers can avoid overlooking serious clone changes.

- The third issue is that code fragments to be modified are merged into the main branch when developing with version control systems (in short, VCS). In such development, the main branch should be bug-free and always ready for release. Therefore, code fragments to be modified that may contain bugs should be checked before the topic branch is merged. However, Clone Notifier, which is premised on regular execution, is difficult to detect such code fragments before merging topic branches. If clone change notification is triggered by development workflow, the possibility of bugs merged into the main branch can be reduced because such code fragments are detected before merging topic branches.

Consequently, in this part, we propose a clone modification support system named CLIONE[10], which aims to integrate into pull request based development to solve these issues. CLIONE detects

[10] `https://github.com/T45K/CLIONE`

code fragments that need modifications by tracking clones when creating pull requests (in short, PR), and CLIONE also notifies the code fragments to developers as PR comments. Moreover, we made three improvements for more accurate clone change tracking than Clone Notifier.

In order to evaluate CLIONE, we conducted five experiments. We confirmed the followings from the experimental results.

1. There are 21.5% of PRs in which clones have been modified non-simultaneously (i.e., some of clones were modified but the others were not), indicating that CLIONE is useful for PR-based development.

2. The execution time of CLIONE was sufficiently short compared to CI tools widely used in PR-based development, indicating that the barrier to introduce CLIONE into PR-based development is low.

3. Due to the clone tracking improvements, CLIONE was able to track clones correctly compared to Clone Notifier in 34.6% of PRs in which clones have been modified non-simultaneously.

4. We conducted proposals the way of clone modifications using CLIONE and questionnaire survey to developers. From the results, CLIONE is also useful for developers.

```
- private IRubyObject any_pBlockless(ThreadContext context) {
-   for (int i = 0; i < realLength; i++) {
-     if (eltOk(i).isTrue()) return context.runtime.getTrue();
-   }
-
-   return context.runtime.getFalse();
- }

+ private IRubyObject any_pBlockless(ThreadContext context, IRubyObject[] args) {
+   IRubyObject pattern = args.length > 0 ? args[0] : null;
+   if (pattern == null) {
+     for (int i = 0; i < realLength; i++) {
+       if (eltOk(i).isTrue()) return context.runtime.getTrue();
+     }
+   } else {
+     for (int i = 0; i < realLength; i++) {
+      if (pattern.callMethod(context, "===", eltOk(i)).isTrue())
+         return context.runtime.getTrue();
+     }
+   }
+
+   return context.runtime.getFalse();
+ }
```

(a) any_pBlockless method

```
  private IRubyObject all_pBlockless(ThreadContext context) {
    for (int i = 0; i < realLength; i++) {
      if (!eltOk(i).isTrue()) return context.runtime.getFalse();
    }

    return context.runtime.getTrue();
  }
```

(b) all_pBlockless method

Figure 14: Two methods in JRuby

2 Research Motivation

Figure 14 shows two methods in JRuby. In PR#5096[11], while any_pBlockless method was modified, all_pBlockless method was not modified. This non-simultaneous modification introduced a bug that caused an error when calling a Ruby API, which internally calls all_pBlockless method in JRuby into the main branch. As a result, JRuby, which included this bug, was released at version 9.2.0.0. This bug was fixed in PR#5298[12].

If the developers used CLIONE, they could avoid introducing the bug caused by non-simultaneous modification of clones into the main branch. In development using VCS, the main branch should be bug-free and always ready for release. Therefore, a clone modification system aimed to integrate into development workflow is needed to keep main branch bug-free.

[11] https://github.com/jruby/jruby/pull/5096
[12] https://github.com/jruby/jruby/pull/5298

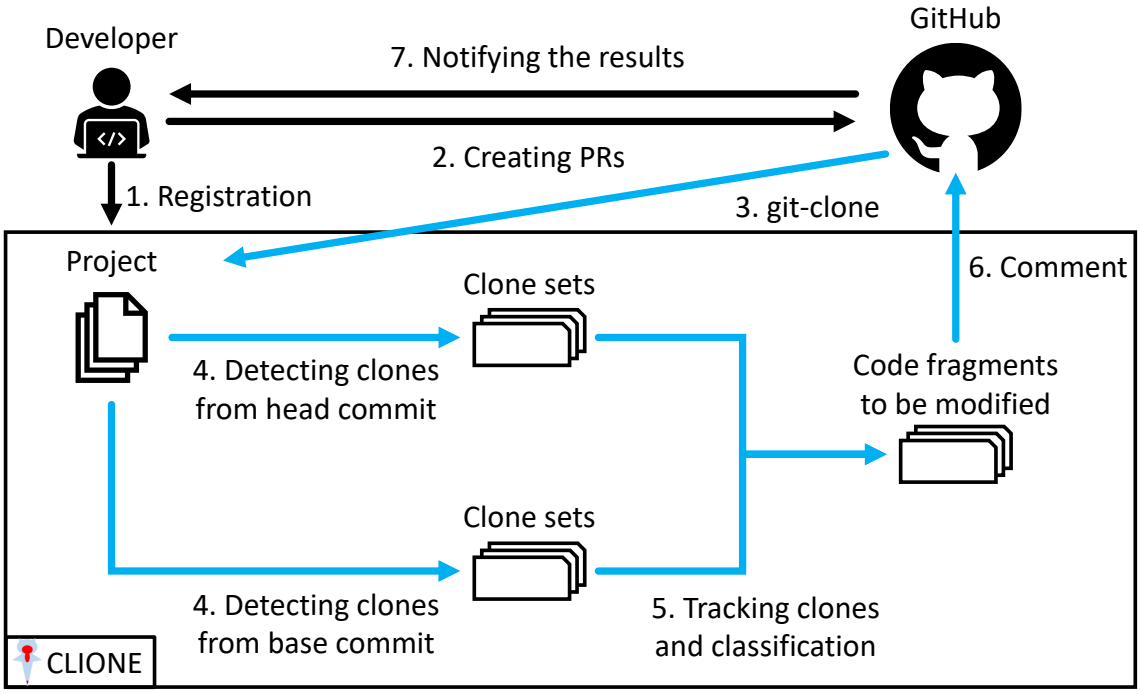


Figure 15: Overview of CLIONE. Blue lines means processes CLIONE performs.

3 Proposed System: CLIONE

In this part, we propose CLIONE, a clone modification support system aimed to integrate into development workflow. CLIONE detects code fragments that need modifications by tracking clones automatically at the time of creating PRs. CLIONE is a server-side application which uses GitHub Apps [99] and receives HTTP requests which GitHub sends when developers create PRs. Thanks to GitHub Apps, it is easy to install CLIONE on GitHub projects.

3.1 Overview

Figure 15 shows an overview of CLIONE. First, developers who want to use CLIONE register their GitHub accounts and their repositories with CLIONE. After this registration, whenever the developer creates a PR, GitHub sends an HTTP request to CLIONE, and CLIONE is executed. CLIONE tracks clones between the head commit of the PR and the commit at the point where the PR branch was created (in short, base commit).

When the developer creates a PR, CLIONE downloads (git-clone) the project onto its local environment. Next, CLIONE detects clones from the head and the base commits of the PR, respectively, by using NiCad [51] or SourcererCC [23]. Why we select these clone detectors is each of them can detect Type-3 clones precisely.

After clone detection on each commit, CLIONE tracks clones to detect clone changes between the two commits. Clone tracking is used to determine if the clone has been modified between the two commits. As well as Clone Notifier, in order to track clones between the two commits, CLIONE calculates the overlapping location of clones, based on the location overlapping function of Kim

et al. [54]. Moreover, in order to suppress the execution time of CLIONE, CLIONE does not track clones in source files that have not been modified between the two commits. The reason is that such clones can be determined to be unchanged without tracking them.

After clone tracking, based on changes of clones, CLIONE detects code fragments that need modifications. Specifically, CLIONE treats clone sets where some clones were modified but the others were not modified and clone sets where some or all of clones were newly added as code fragments to be modified. Moreover, in CLIONE, we made three improvements for more accurate clone tracking than Clone Notifier. We explain the improvements in Section 3.2.

Finally, CLIONE makes a PR comment for each the code fragments to be modified to notify the results to the developer. Figure 16 shows an example of PR comment about a non-simultaneously modified clone set. In the top of the figure (a), the changed piece of the changed code fragment is shown in the diff format, the whole of changed code fragment is shown in the middle (b), and the non-changed code fragment is shown in the bottom (c). By getting notifications of the results as PR comments, developers can receive feedback promptly.

3.2 Improvements of clone tracking

In CLIONE, we made three improvements to the clone change tracking technique used in Clone Notifier,

1. code fragment tracking,
2. file rename detection, and
3. clone change judgment by comparing token sequences.

3.2.1 Code fragment tracking

First, CLIONE tracks not only clone instances but also code fragments. Clone Notifier treats only clone instances (code fragments that are detected as clones) as targets of tracking. Thus, if code fragments that are detected as clones in the old version are not detected in the new version, the code fragments are classified deleted clone instances by Clone Notifier. On the other hand, in some cases, a code fragment is not detected as a clone because it has been modified. For example, figure 17(a) shows two methods detected as clones in the base commit, but not detected as clones in the head commit because one of them has been modified. In this case, Clone Notifier classifies the clone set (i.e., the two methods) as *Deleted* because the methods are not detected as clones in the head commit. However, this clone set should be classified as non-simultaneous modification because only main1 method was modified. As a result, developers may overlook this non-simultaneous modification.

On the other hand, CLIONE tracks not only clone instances but also code fragments, such as methods or code blocks. Even if clones are not detected in one of the commits, the changes of clones can be detected by tracking code fragments themselves. Figure 17(b) shows an example of code fragment tracking. In this example, CLIONE tracks the methods themselves. As a result, CLIONE can detect a non-simultaneously modified clone set (figure 17(c)) and classify it appropriately.

3.2.2 File rename detection

Second, CLIONE detects file name changes. Clone Notifier tracks clones in files with the same name in the input two versions. On the other hand, in software development and maintenance, file names are often changed. When a file is renamed between versions, Clone Notifier determines that the file was deleted in the old version and added in the new version. Therefore, when a file is renamed, Clone Notifier incorrectly classifies clones in the file as *Deleted* in the old version and as *Added* in the new version, even if the contents of the file have not changed.

On the other hand, CLIONE uses Git's rename detection function to track files based on the similarity of their contents, even when the file name has been changed. As a result, the code fragments in the file can be tracked, reducing the misclassification of clones.

3.2.3 Clone change judgment by comparing token sequences

Third, CLIONE judges whether clones are changed or not by comparing their token sequences. In Clone Notifier, a clone is considered modified if there are lines in the clone that have been added or deleted between two versions. On the other hand, even if changes made in clones do not affect the program behavior, such as formatting or comments, Clone Notifier judges that the clones have been changed. In particular, when the source code format is changed, a large number of clones may be judged as having changed because the changes will cover a wide area of the entire source code. Therefore, judging whether clones have been changed based on the presence or absence of added or deleted lines may be burdensome for developers to check because a large amount of clone change information will be notified even if the changes do not affect the behavior of the program.

On the other hand, CLIONE converts the tracked code fragments into token sequences and compares them to judge if the code fragments have been modified. Therefore, CLIONE can judge whether clones have been modified while ignoring changes that do not affect its behavior, such as formatting or comment changes. This allows us to avoid notifying developers of changes in clones that they do not need to check.

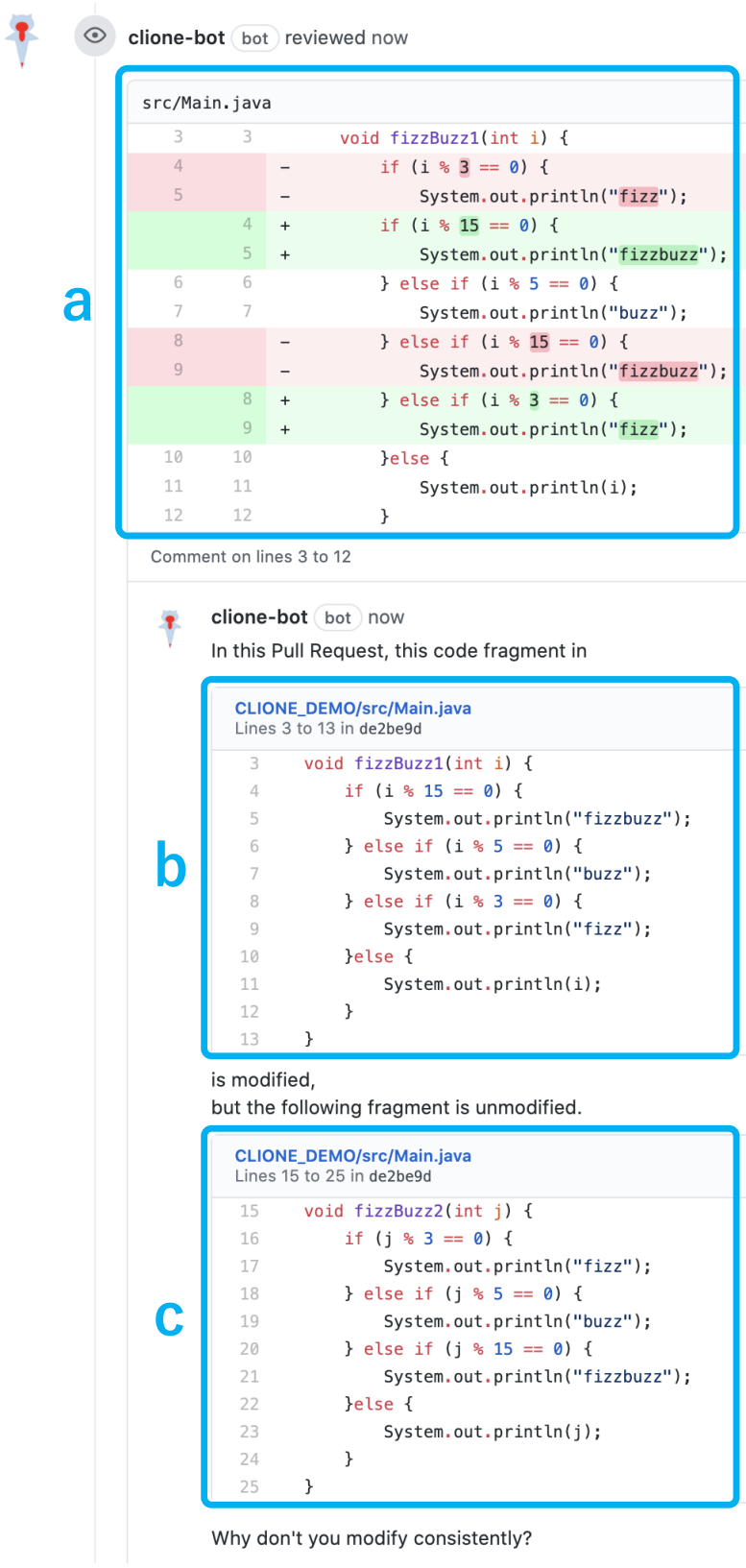
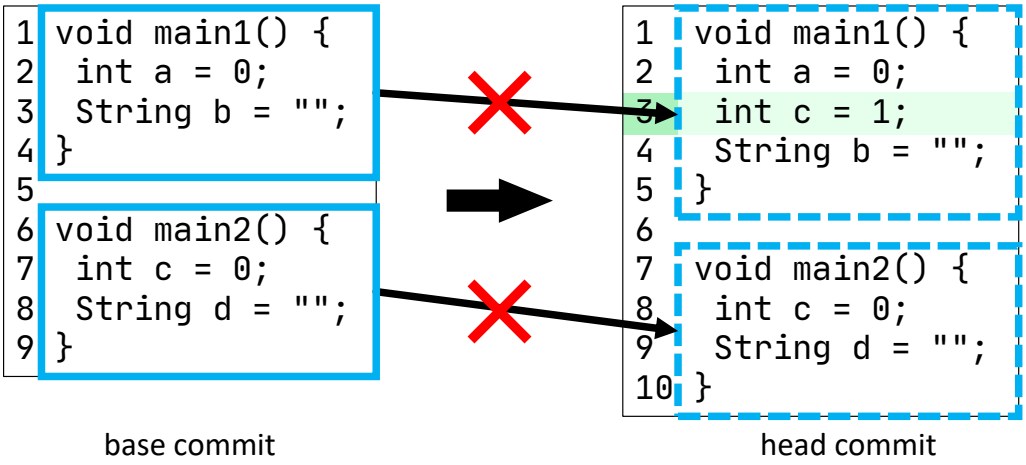
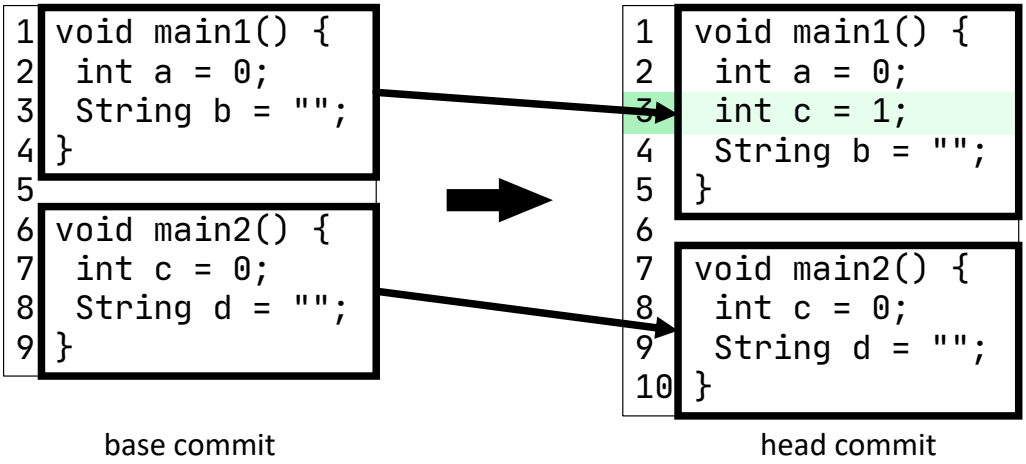


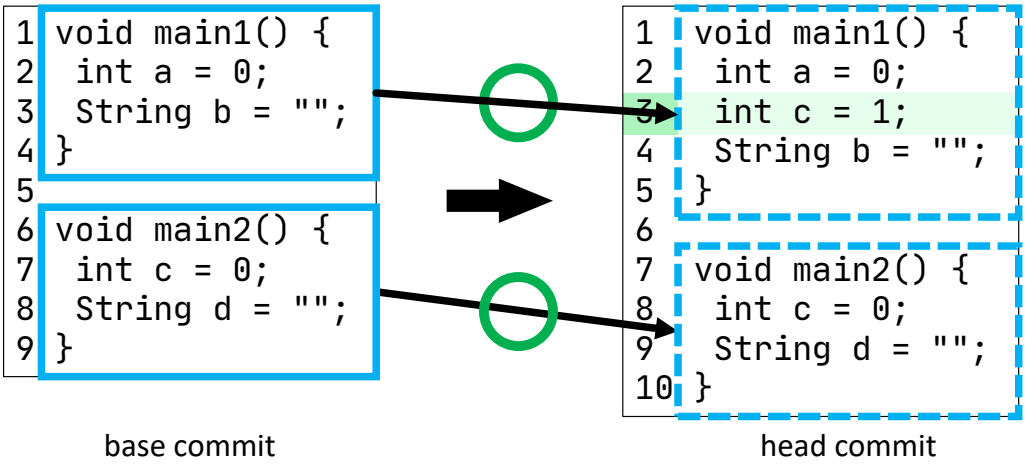
Figure 16: Example of comment about a non-simultaneously modified clone sets



(a) Clone instance tracking



(b) Code fragment tracking



(c) Accurate clone change tracking

Figure 17: Improvement of clone tracking

4 Evaluation

In this part, we conducted five experiments to evaluate CLIONE.

4.1 Experiment 1

In experiment 1, in order to evaluate the usefulness of introducing CLIONE into PR-based development, we investigated the proportions of PRs in which clones have been modified non-simultaneously, which CLIONE targets to notify. In this experiment, we manually executed CLIONE on merged PRs of target projects and investigated whether clones had been modified simultaneously or not for each PR.

In this experiment, we selected three OSS as our experimental targets. Table 14 shows the names of the projects, the number of PRs merged by 20/7/2020, and the number of PRs in which at least a Java file was changed (in short, target PRs). We selected those OSS because they are often targeted in clone research [100–102] and developed in PR-based development on GitHub.

Table 9 shows the results of experiment 1. In this table, "improper PRs" means PRs in which clones were modified non-simultaneously. All the projects have PRs in which clones were modified non-simultaneously, and their proportions are 11.9%–30.4%. Some of these non-simultaneous modifications can cause bugs, as introduced in Section 2. Hence, developers need to check non-simultaneously modified clones and determine whether they need modifications before merging branches. If the developers use CLIONE, they are notified of non-simultaneously modified clones for each PR created, so that they can easily deal with them. Therefore, CLIONE is useful to support clone modifications in PR-based development.

Table 8: Target OSS

Name	# PRs	# Target PRs
JRuby[13]	2,248	292
JUnit4	848	236
Gson	317	69

Table 9: Results of experiment 1

Name	# Target PR	# improper PR	Proportion
JRuby	292	89	30.4%
JUnit4	236	28	11.9%
Gson	60	10	16.7%
Total	588	127	21.5%

[13] JRuby is a multi-module project, and we targeted only core module of JRuby.

4.2 Experiment 2

In experiment 2, in order to evaluate whether CLIONE can be introduced into PR-based development, we measured response time from PR creation to CLIONE notification[14]. Additionally, we compared the response time of CLIONE with the execution time of CI tools[15], which are widely used in PR-based development [55]. If the response time of CLIONE is shorter than the execution time of CI tools, the barrier to introducing CLIONE to PR-based development is low.

In this experiment, we manually reproduced the target PRs of JRuby, one of the OSS targeted in Experiment 1, and measure the response times of CLIONE. We targeted JRuby because JRuby is the only one that uses CI tools for the automated build. By all rights, we should reproduce all of the target PRs. However, we considered that manually reproducing them is hard task. Thus, we targeted only the five latest PRs merged by 6/10/2020 and at least a Java file was changed.

Table 10 shows the results of experiment 3. As shown in the table, in all PRs, each response time of CLIONE is much shorter, less than half of the corresponding CI tool execution time. The CI execution time varies so much depending on the PR because JRuby builds in two different CI tools depending on the PR changes. We also checked the breakdown of the CLIONE response time and found that the PR#6424 took 50 seconds (67.4%) for git-clone, 25 seconds (29.0%) for clone detection, and 1 second (1.6%) for others. The same trend was observed for the response time in other PRs. This indicates that CLIONE specific processes such as clone tracking and classification account for a very small percentage of CLIONE's response time. Also, since git-clone accounts for the majority of CLIONE's response time, we can improve the implementation of CLIONE. For example, only the differences are downloaded to the CLIONE environment (git-pull) instead of downloading the full target project every time a PR is created.

In addition, we conducted an experiment to estimate the upper limit of the response time of CLIONE. As mentioned in Section 3.1, CLIONE does not track clones in files that have not been modified. Therefore, the more files are modified in the PR, the more clones are tracked, and the response time of CLIONE increases. Hence, CLIONE response time is expected to be the largest when all the Java files in the project have been modified. In order to estimate the upper limit of CLIONE response time, we created a PR with the change of adding blank lines at the end of all Java files in JRuby[16], and measured the response time of CLIONE. A total of 1,459 Java files were

Table 10: Response times of each PR

PR number	CLIONE response time	CI tool execution time
6424	1m 26s	4m 35s
6412	1m 28s	14m 35s
6410	1m 23s	12m53s
6407	1m 18s	4m 20s
6401	1m 17s	13m 26s

[14]Our environment where CLIONE is executed is as follows; CPU: Intel Xeon E5-2620, Memory: 32GB, and OS: Ubuntu 18.04.

[15]Continuous Integration tools. They automatically build and test projects at the time of creating PRs or pushing commits.

[16]`https://github.com/T45K/jruby_clione/pull/6`

modified in this PR. The response time of CLIONE for this PR was 3 minutes and 37 seconds, which is an increase from the response time in table 10 but shorter than the execution time of CI tools.

In summary, CLIONE can notify code fragments in a short time, which is faster than the execution time of CI tools, and CLIONE also can be easily introduced into PR-based development.

4.3 Experiment 3

In experiment 3, in order to evaluate our improvements of clone tracking, we compared the results of CLIONE's clone tracking and Clone Notifier's one. In this experiment, we manually executed CLIONE and Clone Notifier on the target PRs of three projects shown in table 14, checked the clone tracking results of each tool, and counted PRs in which the clone tracking results are different.

As a result, we confirmed that there were 44 PRs (34.6% of improper PRs) in total in which CLIONE tracked clones more accurately than Clone Notifier. On the other hand, there was no clone where Clone Notifier succeeded in tracking but CLIONE failed. Table 11 shows a breakdown of the improvements that contributed to the different clone track results. As shown in table 11, the number of PRs with different clone tracking results by code fragment tracking is the highest. On the other hand, the number of PRs with improved tracking results for file rename detection and clone change judgment by comparing token sequences is small. However, in the PRs with improved tracking results, CLIONE was able to correctly classify a large number of clone changes while Clone Notifier misclassified a large number of clone changes due to file name changes and format changes. In other words, although the number of improved PRs is small, these improvements can significantly reduce the burden on developers because the developers are not notified of the tracking results of a large number of clones that are not considered necessary to be checked in that PR.

In summary, thanks to these three improvements, CLIONE can track clones more accurately than Clone Notifier and provide more useful clone change information.

4.4 Experiment 4

In experiment 4, in order to evaluate the usefulness of CLIONE for developers, we detected non-simultaneously modified clones from past PRs of OSS using CLIONE, modified them, and proposed them to the developers of OSS as PRs. When the proposed PR was merged, if the OSS developers had used CLIONE, they would have dealt with the non-simultaneously modified clones when the PR was created. Therefore, CLIONE is considered to be useful for the developers.

Table 11: Breakdowns of improvements

Improvement	# PRs with different results
Code fragment tracking	35
File rename detection	3
Clone change judgment by comparing token sequences	6
Total	44

In this experiment, we detected non-simultaneous modifications from OSS satisfying the following conditions to confirm the usefulness of CLIONE for developers of diverse OSS.

- The OSS is developed in PR-based development on GitHub.
- The OSS is written in Java.
- The OSS has more than 5,000 stars on GitHub[17].

We selected OSS written in Java because we were familiar with Java so that we can easily modify detected non-simultaneous modifications. Eventually, we created 15 PRs from 14 OSS.

Table 12 shows the list of 15 created PRs and their status as of 3/11/2020. Ten of them were merged, one was closed, and four have been still opened. As for ten merged PRs, if the OSS developers had used CLIONE, they could have dealt with the clones when creating the PRs. On the other hand, one of them was closed. As for the closed PRs, the developers considered that the clones were not needed modifications while we considered that they should be modified.

In summary, though the developers need to decide whether or not to modify the code fragments, CLIONE is useful for the developers because it can notify the developers of the code fragments to be modified when the PR is created.

[17]Star is a feature for GitHub users to save their favorite repositories. It can be said that the more stars a repository has, the more popular the repository is.

Table 12: List of created PRs

Name	URL	Modification details	Status
JRuby	https://github.com/jruby/jruby/pull/6360	Change API	Merged
JUnit4	https://github.com/junit-team/junit4/pull/1671	Boxing variables	Merged
fastjson	https://github.com/alibaba/fastjson/pull/3444	Move code	Merged
RxJava	https://github.com/reactivex/Rxjava/pull/7080	Move method	Merged
Jenkins	https://github.com/jenkinsci/jenkins/pull/4945	Extract method	Merged
Guava	https://github.com/google/guava/pull/4036	Remove dead code	Merged
Glide	https://github.com/bumptech/glide/pull/4353	Move code	Merged
libGDX	https://github.com/libgdx/libgdx/pull/6201	Modify error message	Merged
checkstyle	https://github.com/checkstyle/checkstyle/pull/8872	Extract method	Merged
druid	https://github.com/alibaba/druid/pull/3956	Change API	Merged
JRuby	https://github.com/jruby/jruby/pull/6361	Remove dead code	Closed
Druid	https://github.com/apache/druid/pull/10414	Insert precondition	Open
Zuul	https://github.com/Netflix/zuul/pull/900	Remove dead code	Open
Jacoco	https://github.com/jacoco/jacoco/pull/1099	Move code	Open
Jackson	https://github.com/fasterXML/jackson-core/pull/642	Insert if statement	Open

4.5 Experiment 5

In experiment 5, in order to evaluate the usefulness of CLIONE for developers, we conducted a questionnaire survey about notification contents of CLIONE to developers. Concretely, we executed CLIONE on past merged PRs of kGenProg [103], which is our OSS, selected ten of them as targets, and conducted a questionnaire survey on the content of notification to four developers who created the PRs. Table 13 shows the list of the target PR numbers and the content of each PR. The contents of the questionnaire are as follows.

1. Were you aware of the clones changed in this PR?
2. If you were aware of the clones, why did you not modify them (i.e., modify them simultaneously or merge them)?
3. If you have been notified of the clones from CLIONE when creating the PR, did you modify them?

Hereafter, we introduce the responses to the questionnaire and discuss them.

In question(1), we got two responses (PR#633 and #663) that the developer was not aware of the clone changes. Also, both responded to question (3) that they would modify the clones if they were notified of them at the time of creating the PR. We checked the current implementation of kGenProg, and both of the clones had been modified. Therefore, it can be said that if CLIONE had been used, the developers would have been able to modify the code fragments to be modified when creating the PRs.

On the other hand, we got eight responses that the developer was aware of the clone changes. Hereafter, we classify these eight responses and discuss them in more detail. First, for the three cases (PR#154, #442, and #482), the answer to question (2) was "there is no need to merge the clones." We checked the clones, and all of them were about six lines. Hence, we considered that developers do not treat clones with a small number of lines as refactoring targets. Therefore,

Table 13: Questionnaire target PRs

PR number	How clones were changed
40	Newly added
76	Newly added
154	Newly added
442	Newly added
449	Newly added
482	Newly added
491	Newly added
496	Newly added
633	Non-simultaneously modified
663	Newly added

improving CLIONE to filter newly added clones by the number of their lines or tokens would prevent sending such notifications that will not be useful to developers.

Second, for the two cases (PR#491 and #496), the answer to question (2) was "I thought merging clones would make readability low." The newly added clones in these PRs were clones that performed similar processing but had different parent classes, making it difficult to pull up the methods. Moreover, forcing to extract them into a single method could worsen the readability. On the other hand, this may be due to the poor design of inheritance relations in the added code fragments, which makes it difficult to merge clones. Actually, a refactoring that improves the design of the code fragments in these PRs has been conducted[18]. Therefore, it can be considered that CLIONE can not only detect code fragments to be modified but also point out the poor design of the project.

Third, for the one case (PR#40), the answer to question (2) was "I did not dare merge the clones for the sake of implementation speed." This answer indicates that it depends on the developer to decide whether to merge clones or not. On the other hand, such ad hoc implementation is called *technical debt* [104], which should be modified as soon as possible [105]. Creating *Issue* is one of the ways of dealing with technical debt. Issue is a feature of GitHub, and we can manage tasks or projects and ask the other developers for help using Issue. By managing tasks to modify technical debts as Issues, we can easily modify them later. Therefore, we consider that implementing Issue creation function in CLIONE can support the management of such code fragments to be modified.

Fourth, for the one case (PR#76), the answer to question (2) was "It is difficult to merge the clones because the types of the variables between the clones are different." The cause is that general clone detectors normalize not only identifier names but also type names. Therefore, if CLIONE uses clone detectors that do not normalize type names, CLIONE can avoid detecting changes of such clones that are not suitable for refactoring targets.

Finally, for the one case (PR#449), the answer to question (2) was "I wanted to merge the clones, but didn't know how." The clones had a common parent class, and only part of the process of them was different. We checked them, and they have been merged in *Template Method Pattern* [106]. In this regard, by implementing Issue creation function in CLIONE as described earlier, developers can ask for help from other developers by creating Issue even if they find it difficult to merge clones at the time of PR creation. In addition, he answered question (3) as "CLIONE should suggest me how to merge the clones."

These results show that CLIONE can provide useful notifications to developers. Even if the developers judge the information to be unhelpful at this point, with some improvements to CLIONE, CLIONE can avoid providing unhelpful notifications or assist developers in other ways. Therefore, CLIONE is useful for developers' clone modification support.

[18] `https://github.com/kusumotolab/kGenProg/pull/501`

5 Threats to Validity

In this study, the experiments were conducted for a specific OSS. Therefore, different results may be obtained if different projects are targeted. In addition, in Experiment 3, the clone tracking results of CLIONE and Clone Notifier were checked and compared manually. Therefore, the results of Experiment 3 may depend on our subjectivity.

CLIONE uses NiCad and SourcererCC to detect clones. If CLIONE uses other clone detectors, the experimental results may differ from this study.

Clone tracking and change classification by CLIONE is independent of the programming language. Therefore, CLIONE can be applied to any project written in a programming language from which the clone detector used by CLIONE can detect clones. However, in this study, only OSS written in Java is used as the experimental target. Therefore, different results may be obtained when applying CLIONE to other languages.

6 Related Works

6.1 PR-based development support system

Some studies that integrate existing source code analysis techniques into PR-based development to improve software development efficiency are conducted.

Alizadeh et al. proposed RefBot, a system that integrates automated refactoring into PR-based development [107]. RefBot automatically refactors the target project every time a PR is created. It also presents the results of the refactoring to the developer. The developer can interactively select the following options for the refactoring done by RefBot. (1) Refactor the source code as-is. (2) Reflect only the position, and the the developer does the refactoring itself. (3) Do not reflect the refactoring.

Carvalho et al. proposed C-3PR, a system that integrates static analysis into PR-based development [108]. C-3PR executes existing static analysis tools on the target project for each PR creation and automatically corrects the output warnings.

Alizadeh and Carvalho have experimented with their system in real-world development and concluded that integrating automated refactoring and static analysis into PR-based development is useful for developers, respectively. CLIONE is a system that integrates clone modification support into PR-type development, and we also concluded that CLIONE is useful for developers.

6.2 Clone modification support

Since clones are considered one of the factors that affect software maintenance, many studies have been conducted to support clone modifications.

Nguyen et al. proposed JSync, a tool to support simultaneous modification of clones [109]. JSync is implemented as an Eclipse plug-in that targets Subversion repositories and automatically detects non-simultaneous modifications of clones at commit time. It also calculates the differences in the abstract syntax tree between the modified and unmodified clones and suggests to the developer how to modify the clones.

Wit et al. proposed CLONEBOARD, a tool for managing clones by recording copy-and-paste operations of code fragments [110]. CLONEBOARD is also implemented as an Eclipse plug-in. When a copied-and-pasted code fragment is modified, the original code fragment is suggested to the developer as the target of modification.

The major difference between these tools and CLIONE is that CLIONE supports PR-based development. Since PR-based development is widely adopted in today's OSS development, we consider supporting PR-based development highly important.

We also consider that these methods and CLIONE are not necessarily exclusive. For example, we consider that more useful developer support can be provided by integrating JSync's suggestion of clone modification methods into CLIONE.

7 Conclusion

In this part, we proposed a clone modification support system, CLIONE, for integration into PR-based development. CLIONE detects code fragments that need to be modified due to clone changes between the base commit and the head commit of a PR when the PR is created. We also made three improvements to the clone tracking method used by Clone Notifier in order to track clones more accurately. Five experiments were conducted to evaluate CLIONE from three different perspectives. The experimental results showed that CLIONE is useful for PR-based development and developer, and CLIONE can also track clones more accurately than Clone Notifier.

As a future work, we plan to improve the implementation of CLIONE. Specifically, as described in Section 4.5, we plan to implement a function to filter out clones that are not suitable for refactoring, create Issue, and recommend how to modify code fragments detected by CLIONE.

Part IV

Measurement Reducible Lines of Code Based on Automated Merging Clones

1 Background

Changes are occasionally (or even continuously) added to the source code after software systems have been released. A bunch of changes to the source code deteriorates its quality (e.g., collapsing its design, decreasing the readability), so that the maintenance cost gets more expensive [111–113]. The maintenance cost of the source code is often estimated from its size or complexity [113]. In the case of large-scale software systems, an enormous amount of money is required. If users of a software system can estimate its number, they should be able to decide whether they continue to use it or replace it with a new one [114].

A factor of deteriorating the quality of source code is the presence of clones. Clones get involved in both software development and maintenance [115,116]. The presence of clones makes the source code redundant so that inconsistencies in source code tend to happen unintentionally [54]. Thus, from the perspective of maintainability of the source code, merging clones is important.

Merging clones is a well-known refactoring. Refactoring is defined as a set of operations to improve the internal structure of the source code without altering its external behavior [26]. *Extract Method* refactoring, which is one of the most often performed refactorings, is a set of operations to extract a code fragment in an existing method as a new method. If duplicated code fragments are extracted as a new method, the duplication is removed from the source code. Removing clones by refactoring makes it easier to keep consistencies in the source code because we do not have to put the same changes on duplicated code in multiple places. However, there is a possibility that refactoring itself introduces a new bug in the source code [27]. Consequently, removing all clones without any special reason is not realistic: a reasonable indicator is required whether or not given clones should be refactored. The number of lines of code (LoC) reduced by refactoring clones are used for the indicator.

An existing study proposed a technique to estimate how many LoC can be reduced by refactoring clones [33] The existing technique judges whether clones can be refactored or not using JDedorant [34], which Tsantalis et al. developed, and calculates reducible LoC based on the lines of refactorable clones. The technique utilizes the position information of clones (the path of the file including a given clone, start line and end line of the given clone). If multiple clones are overlapped with each other, the technique selects only one of them using the greedy algorithm.

However, we think there are two issues in the existing technique.

- The first issue is that the existing technique does not correctly judge whether or not a clone can be refactored. In JDeodorant, several conditions are set and only the clones that match all the conditions are judged to be refactorable. However, even if a clone matches all the conditions set by JDeodorant, there is a possibility that it cannot be refactored. It is also considered that refactoring is possible even for clones that do not meet the criteria [33]. On the other hand, it is not realistic to manually check the refactorability of all the large number of clones detected, as this would require a lot of effort.

- The second issue is that in the case that clones are overlapping, the existing technique only considers merging one of the overlapped clones. However, even if two clones are overlapping, both of them may be able to be merged. Thus, an estimated reducible LoC may become a completely different number from the actual reducible LoC.

Therefore, in this part, we propose a new technique to calculate reducible LoC. The proposed technique performs a loop of (1) detecting clones, (2) removing a set of clones, and (3) compiling and testing the edited source code as long as the LoC of the source code gets decreased by the refactorings. Due to its nature, the proposed technique is completely free from the above two issues. We implemented a software tool based on the proposed technique and applied it to several open source software systems. The purpose of the application is comparing the proposed technique to the existing one. As a result, we confirmed that the proposed technique was able to measure reducible LoC more correctly than the existing technique.

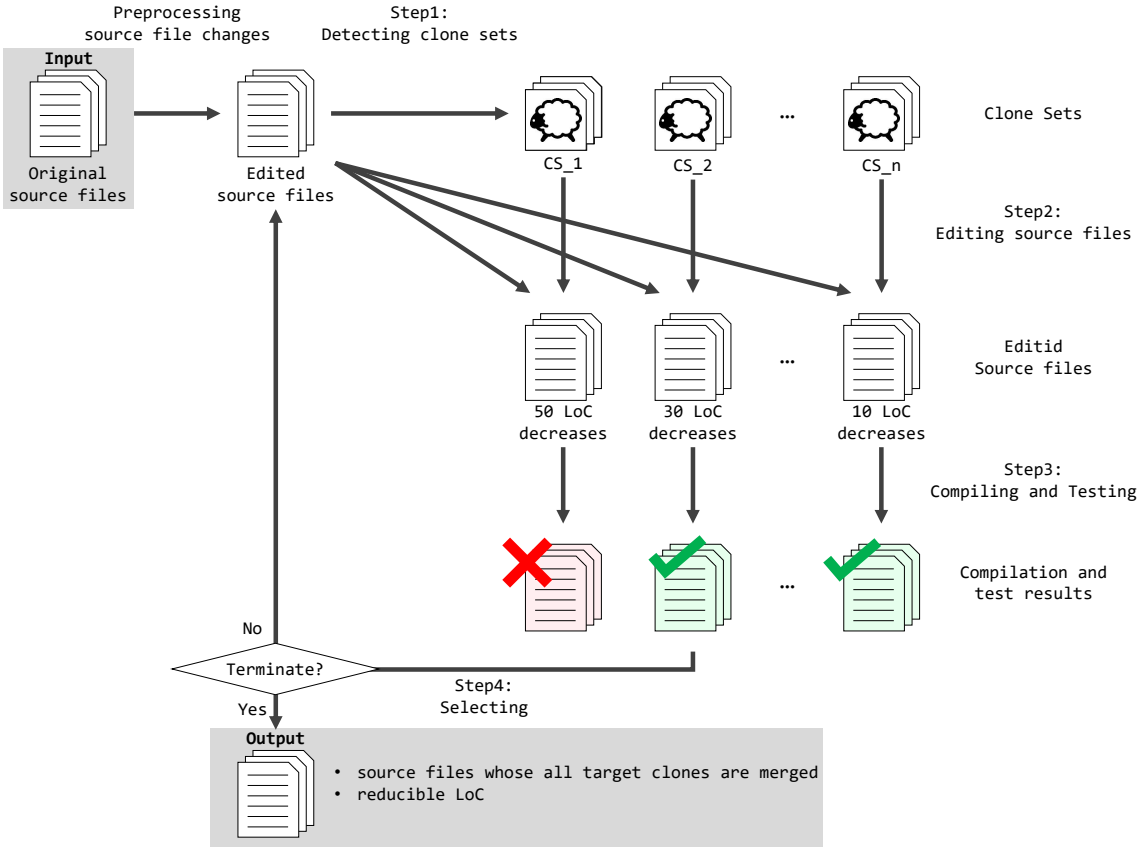


Figure 18: Overview of proposed technique

2 Proposed Technique

We propose a new technique to calculate LoC that can be reduced by merging clones. The technique repeats (1) detecting clones, (2) editing source files, (3) compiling, and (4) testing. This technique enables users to obtain actual LoC that can be reduced by merging clones, not just estimating it. Moreover, this technique has an important feature: even if multiple clones are overlapped, the technique tries to merge each of them while the existing study [33] only considers one of the overlapped clones. The reasone why the proposed technique can deal with overlapped clones is the technique performs clone detection and editing source files repeatedly. For example, in the case of figure 8(a), in which clone A contains clone B, the technique first merges clone A and then detect clones again. Therefore, the technique can try to merge clone B, which have been contained in clone A. Figure 18 shows an overview of the proposed technique. The input of the technique is a set of source files. The output is a set of source files in which clones have been merged as much as possible and their reducible LoC. Please note that this technique does not merge clones if merging them does not reduce LoC of the source files.

The technique is composed of two processes shown below. Both the processes are performed fully automatically.

- Preprocessing source file changes

- Clone merging

2.1 Preprocessing source file changes

First, a new class is generated. The new class is used as a utility class for placing merged clones. More concretely, each merged clone is declared as a static method in the class. Besides, target source files are edited to avoid compile error due to uninitialized local variables. Additionally, the source files are reformatted to avoid inconsistencies of reducible LoC due to coding style. In total, four changes are made in Preprocessing source file changes.

2.2 Clone merging

Clone merging is composed of four steps shown below. All of the steps are performed fully automatically.

Step1: detecting clone sets

Step2: editing source files

Step3: compiling and testing edited source files

Step4: selecting edited source files

In Step1, clone sets in the target source files are detected. In Step2, source files are edited to merge one of the clone sets that were detected in Step1. A method extracted to merge clones is placed in the Java class described in Subsection 2.1. In Step3, the source files edited in Step2 are compiled and tested to verify the external behavior. Hereafter, we call edited source files which succeed in compiling and testing and whose LoC gets reduced *selectable source files*. Step2 and Step3 are performed on each of the clone sets detected in Step1. In Step4, the selectable source files whose reduced LoC is the largest is selected. Then, Step1–Step4 are repeatedly performed on the selected source files. When either of the following termination conditions is satisfied, *clone merging* terminates and outputs the reducible LoC.

- No clone sets are detected.
- No source files edited in Step2 are selectable.

3 Implementation

We implement our technique as a tool. The tool is written in Java and targets Java source files.

3.1 Preprocessing source file changes

3.1.1 Generating a new Java class

A class in which extracted methods are placed is newly generated. When the extracted method in the new class is called from the original place, it is invoked with its fully qualified name.

3.1.2 Initializing local variables

On the Java language specification, it is prohibited to reference uninitialized local variables. Thus, when an uninitialized local variable is passed to the extracted method as an argument, a compile error occurs. Developers may initialize local variables and pass them to the method when developers merge clones. In this part, all local variables which do not have `final` modifier and are uninitialized at the time of their declaration are initialized. In the case that the local variable is primitive type except for boolean type, it is initialized as `0`. In the case of boolean type, it is initialized as `false`. In the case of reference type, it is initialized with `null`.

3.1.3 Reformatting

In general, coding conventions are specified in each project. For example, some conventions specify using many line breaks and other conventions specify a small number of line breaks. If code fragments of convention violation exist, it is difficult to calculate a reducible LoC properly. Figure 19 shows examples in which different reducible LoC are calculated due to the coding style.

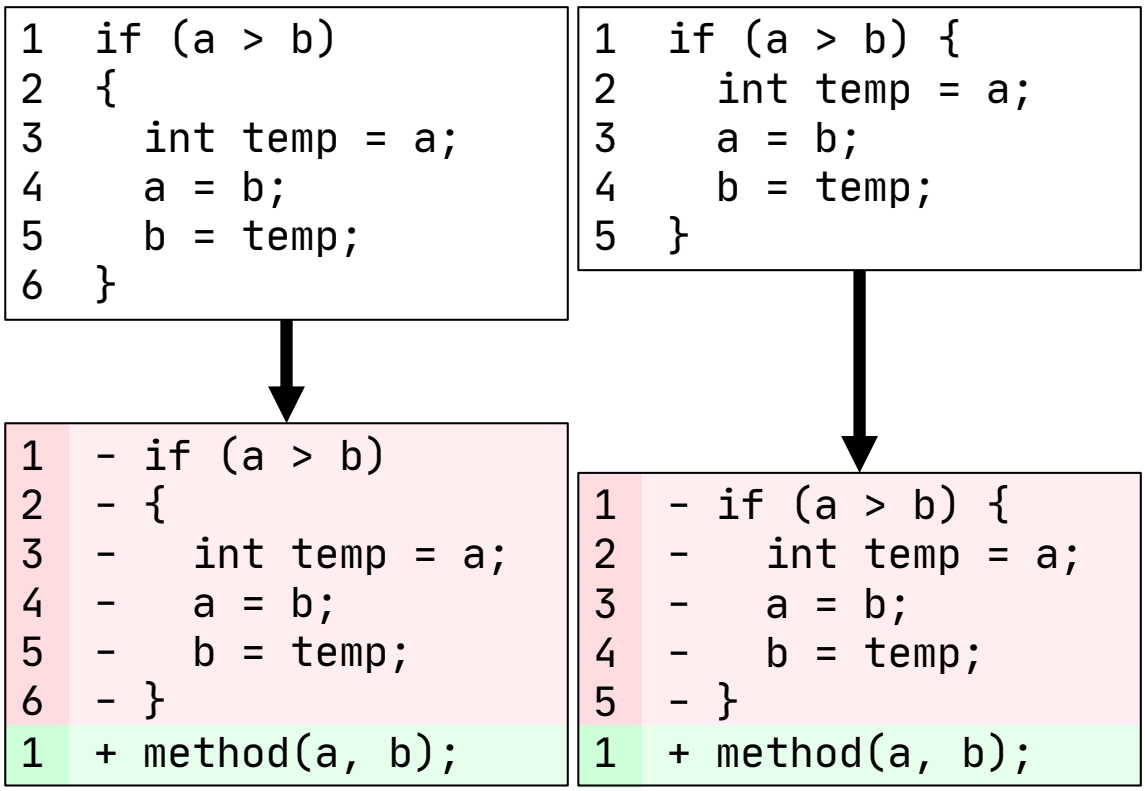
Figure 19(a), *Extract Method* refactoring reduces 5 LoC, but Figure 19(b), *Extract Method* refactoring reduces 4 LoC. Thus, it is necessary to unify coding style by using formatter. In an ideal world, we should prepare the formatters which reproduce coding conventions of each of the target projects, but of course, it is impossible. Thus, we use the default Eclipse formatter [117].

3.2 Clone merging

3.2.1 Step 1: Detecting clone sets

Figure 20 shows an example of detecting clones. Clones are detected in block level. We regard the following statements as blocks. Each following statement is defined as a derived class of class `Statement` in Eclipse JDT [118].

- Block
- DoStatement
- EnhancedForStatement
- ForStatement
- IfStatement



(a) 5 LoC reduce

(b) 4 LoC reduce

Figure 19: Different reducible LoC

- SwitchStatement
- SynchronizedStatement
- TryStatement
- WhileStatement

Block-level clones are more coarse-grained than clones that are detected by token-based techniques. The number of block-level clones is less than the number of token-based clones [119]. However, block-level clones have a remarkable feature, which is that they are better candidates of *Extract Method* refactoring because the code fragments are syntactic chunks [5]. The proposed technique uses Eclipse JDT to parse source files and identify blocks. If a block includes a `return` statement, it is not detected as a clone because it is difficult to extract it as a new method [120].

The proposed technique normalizes identified blocks according to the rules shown below because the larger number of clones can be detected by applying the rules.

- Identifiers are normalized as "$" + "number".
- The same identifier is normalized as the same normalized name.
- All literals are normalized as "$".

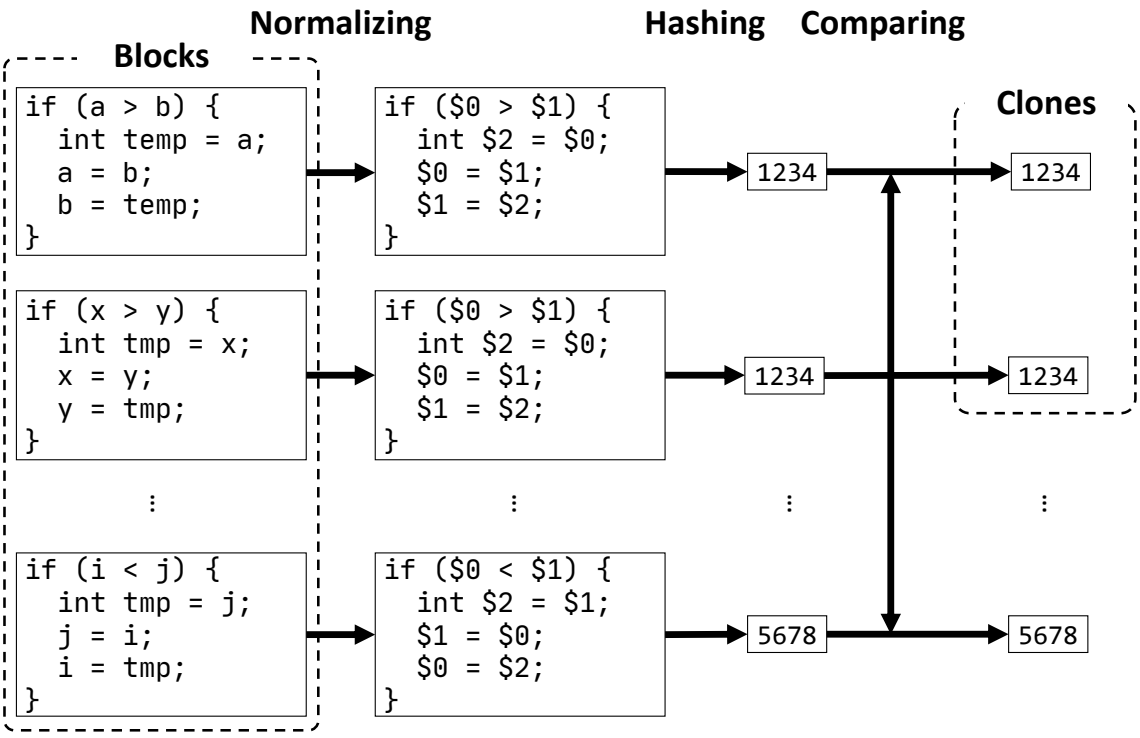


Figure 20: Clone detection

- Qualified names are normalized as identifiers.
- Class name is not normalized.
- Method name is not normalized.

The same identifier is normalized as the same normalized name to avoid false detection as much as possible. The reason why class and method names are not normalized is to avoid detecting clones whose differences are class or method names because it is difficult to merge clones with different type names or calling methods.

After the normalization, a hash value is calculated from each block. The proposed technique uses SHA256 hashing algorithm [121]. Since SHA256 outputs 256-bit hash value, hash collisions hardly occur.

Finally, the proposed technique compares the hash values of blocks to detect blocks of the same hash values as clones.

3.2.2 Step 2: Editing source files

Using *Extract Method* refactoring on one of the clone sets detected in Step1, source files are automatically edited. Figure 21 shows an example of automatically editing source files.

It is possible to reduce LoC because each of the clones is replaced to a method invocation by extracting clones as a method. This extracted method is declared as a `static` method in the class made in *Preprocessing source files changes* (see Section 3.1). The method is invoked with its fully qualified name.

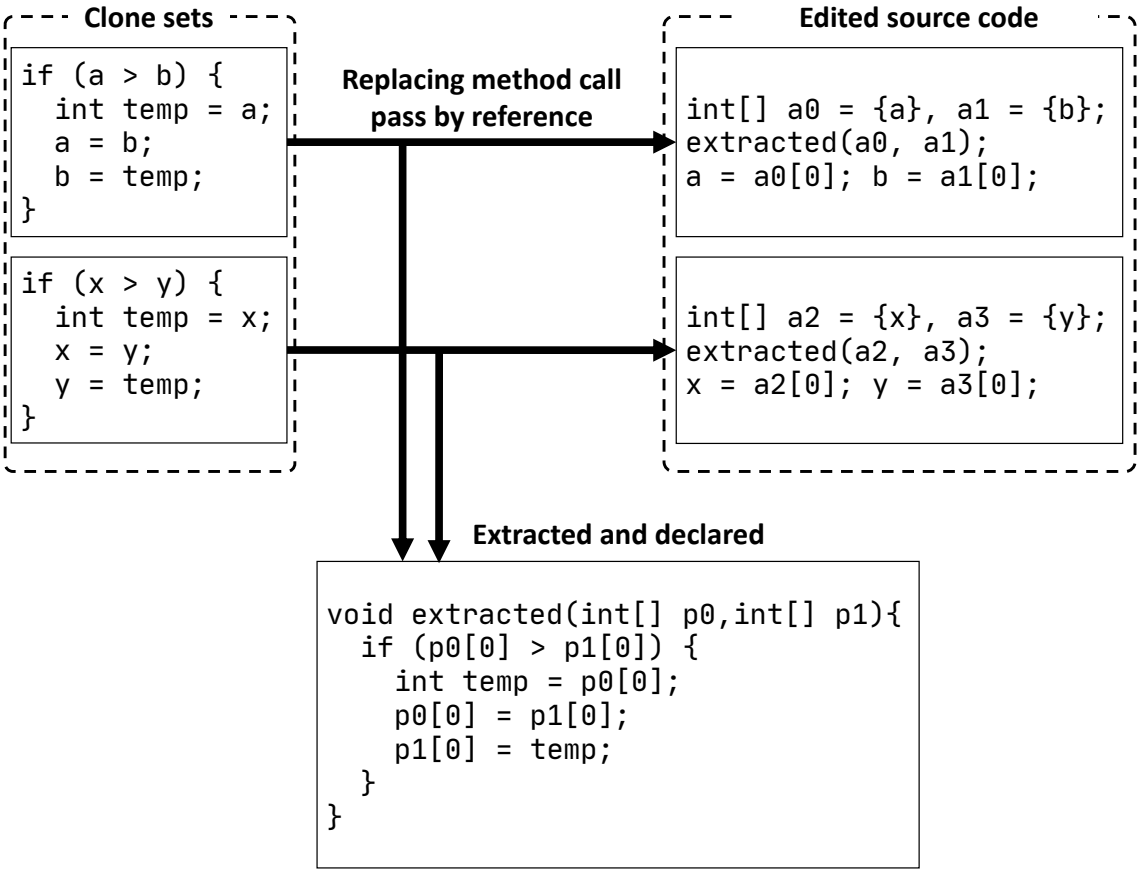


Figure 21: Editing source files

In practice, we need to take care of changes of variables on *Extract Method* refactoring. For example, when only one of the variables is changed in the code fragment, *Extract Method* refactoring can be performed by returning the variable and assigning it in the caller place. However, in the Java language specification, it is impossible to return two or more parameters simultaneously. *Extract Method* refactoring on such clones is not realistic. Unfortunately, examining whether or not each variable is changed in the target code fragment requires deep source code analysis. Thus, In this part, we implemented our tool to pass arguments by reference to the extracted method to keep the external behavior. Our tool uses an array type to pass arguments by reference. Before invoking the extracted method, arrays whose types are the same of variables in the target code fragment are newly defined and initialized with each of the values. These arrays are passed to the method. In the method, the array element at index zero is referenced. After invoking the method, each array element at index zero is assigned back to each variable. These processes are devices of implementation for automated refactoring, not for manual refactoring.

3.2.3 Step 3: Compiling and testing edited source files

In Step3, source files edited in Step2 are compiled and tested. At first, compilation is performed. After compilation gets success, test runs. If both compilation and test get success, edited source

files are recorded as selectable source files. If either compilation or test fails, edited source files are not recorded.

3.2.4 Step 4: Selecting edited source files

After Step2 and Step3 are performed on each of the clone sets detected in Step1, Step4 is performed. In Step4, the selectable source files whose reduced LoC is the largest is selected. Then, Step1–Step4 are repeatedly performed on the selected source files.

4 Experiment

4.1 Overview

In order to evaluate the proposed technique, we applied the proposed technique to open source projects and compared the results obtained with the results of the existing technique. We also investigated the differences between the clone sets merged by the proposed technique and the existing technique. In this part, we selected the same five projects as the existing study [33], except for Columba and Xerces, which could not be compiled on our environment[19]. Table 14 shows the names, versions, and total number of LoC of the target projects. In this experiment, we omitted the test code and tutorial code of the projects. The number of LoC was measured after applying the formatter described in Section 3.1.3.

4.2 Experimental results

4.2.1 Comparison with the existing technique

Table 15 shows the number of merged clone sets, the number of reduced LoC, the execution time as the results of executing the proposed technique, and the table also shows the number of refactorable clone sets and the number of estimated reducible LoC as the results of executing the existing technique. As shown from the table, in all projects, the number of clone sets actually merged by the proposed technique is smaller than the number of clone sets estimated as refactorable by the existing technique. In addition, for all projects except jEdit, the number of reduced LoC measured by the proposed technique is smaller than the number of reducible LoC estimated by the existing technique. We considered that this is because JDeodorant judged many of the clone sets that did not pass compilation or testing when merged to be refactorable, and as a result, the existing technique estimated the number of LoC that can be reducible for those clone sets. Table 16 shows the number of clone sets that pass compilation and testing when merged (CS 1 in the table), for which the proposed technique measures the number of LoC to be reduced, and the table also shows the number of clone sets that JDeodorant judges as refactorable (CS 2 in the table), for which the existing technique estimates the number of lines to be reduced, and their CS 2 in the table), the number of clone sets that JDeodorant judged to be refactorable (CS 2 in the table). As shown from the table, for each project, the number of clone sets that pass compilation and testing when merged is less than the number of clone sets judged to be refactorable by JDeodorant. Hence,

Table 14: Experimental targets

Name	Version	# Total LoC
Ant	1.10.7	231,634
jEdit	5.5.0	161,329
JFreeChart	1.5.0	210,823
JMeter.Core	5.2	82,360
JRuby.Core	9.2.9.0	360,724

[19]Ubuntu 18.04, OpenJDK 1.8.0_222, Ant 1.10.5, CPU: 8 cores 3.6GHz, Memory: 64GB

the number of reduced lines measured by the proposed technique may be lower than the number of reducible lines estimated by the existing technique. However, since the proposed technique merges only the clone sets that pass the compilation and testing when merged, the proposed technique can measure the number of reduced LoC more accurately than the existing technique.

Table 15: Results of comparison

Name	# Detected CS	Proposed technique			Existing technique	
		# Merged CS	# Reduced LoC	Execution time	# Refactorable CS	# Reducible LoC
Ant	583	15	456	1h 15m	155	1246
jEdit	421	58	548	34m	91	494
JFreeChart	810	176	1491	2h 54m	250	1963
JMeter.Core	103	15	47	10m	34	115
JRuby.Core	747	54	570	1h 40m	210	1142

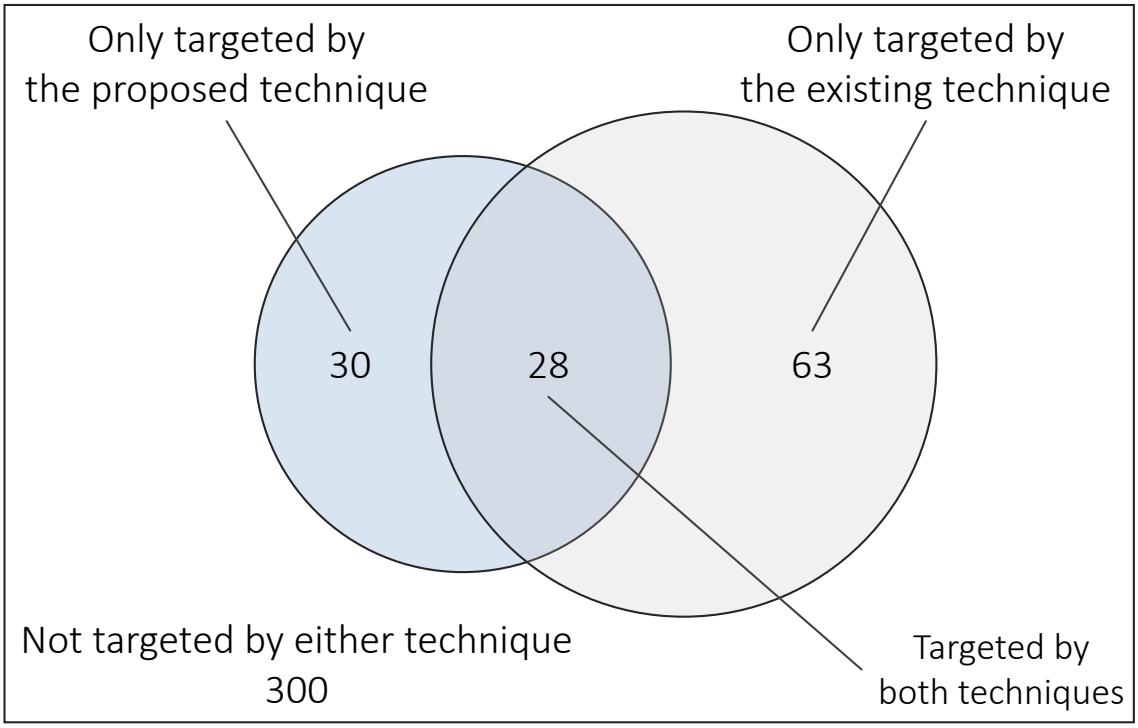


Figure 22: Venn diagram representing clone sets that are targeted to merge by the proposed technique and/or the existing one

4.2.2 Investigation of differences in clone sets targeted to merge by each technique

The reason for the difference in the number of LoC reduced in the Section 4.2.1 is that the clone sets targeted to merge are different when calculating the number of reduced LoC. Therefore, we investigated the difference in the clone sets targeted to merge by each technique. We selected the clone sets detected from jEdit as the target. Figure 22 shows the difference in the number of clone sets merged by the proposed technique and the number of clone sets estimated as refactorable by the existing technique. As shown from the figure, the clone sets merged by the proposed technique and the clone sets estimated as refactorable by the existing technique have some clone sets in common but also have some different clone sets. As an example, we introduce total of four clone sets that were targeted to merge by the proposed technique but not by the existing technique and that were targeted to merge by the existing technique but not by the proposed technique, vice versa.

Figure 23 is a code fragment in `org/gjt/sp/jedit/Buffer.java` that is a clone of a code fragment in `org/gjt/sp/jedit/View.java`. The variables `waitSocket` used in this source code are both of type `Socket`. In the proposed technique, this set of clones was successfully compiled and tested after merging (i.e., it is actually refactorable). On the other hand, the existing technique judged that this clone set was not refactorable. JDeodorant, which is used in the existing technique, has several conditions to determine which clones can be refactored. One of the conditions is that "field variables cannot be parameterized." The variable `websocket` used in Figure 23 is a field variable, that condition was violated, and the existing technique was considered the clone set to

```
1600 if (waitSocket != null) {
1601  try {
1602   waitSocket.getOutputStream().write('¥0');
1603   waitSocket.getOutputStream().flush();
1604   waitSocket.getInputStream().close();
1605   waitSocket.getOutputStream().close();
1606   waitSocket.close();
1607  } catch {IOException e) {
1608   // Log.log(Log.ERROR,this,io);
1609  }
1610 }
```

Figure 23: Example of a clone that was actually refactorable, but the existing technique did ot target to merge because the variables in the clones were field variables

be not refactorable. Thus, there were clone sets that the existing technique judged to be not refactorable because it violated JDeodorant's condition. However, they were actually refactorable, and the proposed technique was able to merge them to measure reduced LoC.

Figure 24 shows a code fragment in `org/gjt/sp/gui/ExtendedGridLayout.java`, which is a clone (hereafter, clone A) of a code fragment in the same class. This clone set consisted of two code fragments. On the other hand, the code fragments in lines 775–779, 781–785, and 787–792 of Figure 24 are also clones (hereafter, clone B). This clone set consisted of six code fragments. In this example, clone A contains clone B. Using the formula (4) described in section 2.10, we can expect a reduction of 16 lines when merging clone A and 17 lines when merging clone B. Therefore, the existing technique that uses the greedy algorithm only targeted clone B and was not able to target clone A. On the other hand, the proposed technique was able to both clone A and B, because it first merged clone B and then detects clone A after replacing the code fragment of clone B with a method call. Thus, there were clone sets that were not targeted to merge by the existing technique due to the greedy method but were merged by the proposed technique.

Figure 25 is a code fragment in `org/gjt/sp/jedit/GUIUtilities.java`, which is a clone of the code fragment in `org.gjt.sp.jeit.gui.ActionBar.java`. Both classes had `dispose` methods, but there was no inheritance relationship between the two classes. Therefore, the clone set was

Table 16: The number of CS 1 and CS 2

Name	# CS 1	# CS 2	# Intersection
Ant	225	292	105
jEdit	163	211	118
JFreeChart	377	539	275
JMeter.Core	48	68	40
JRuby.Core	60	396	40

```
773 for (int col = fromCol; col < toCol; col++) {
774  int minimumColWidth = minimumColWidths[col];
775  if ((Integer.MAX_VALUE - minimumColWidth) < ...) {
776   currentMinimumColWidth = Integer.MAX_VALUE;
777  } else {
778   currentMinimumColWidth += minimumColWidth;
779  }
780  int preferredColWidth = preferredColWidths[col];
781  if ((Integer.MAX_VALUE - preferredColWidth) < ...) {
782   currentPreferredColWidth = Integer.MAX_VALUE;
783  } else {
784   currentPreferredColWidth += preferredColWidth;
785  }
786  int maximumColWidth = maximumColWidths[col];
787  if ((Integer.MAX_VALUE - maximumColWidth) <  ...) {
788   currentMaximumColWidth = Integer.MAX_VALUE;
789  } else {
790   currentMaximumColWidth += maximumColWidth;
791  }
792 }
```

Figure 24: Example of a clone that the existing technique did not target to merge because of overlapping

not refactorable. However, the existing techniques judged that the clone sets were refactorable and targeted them to merge. On the other hand, since the proposed technique failed to compile source code after merging, the clone set was excluded from the targets to merge.

Figure 26 is a code fragment in `org/gjt/sp/jedit/TextUtilities.java`, which is a clone of a code fragment in the same class. This clone set was judged to be not refactorable by the proposed technique because it failed to compile source code after merging. However, the clone set was judged to be refactorable by the existing technique. The variable `buf` is used in lines 965, 966, 969, and 972 of this code fragment. In one of the two code fragments in the clone set, the type of `buf` variable is `StringBuilder`, while in the other, the type of `buf` variable is `StringBuffer`. Both of these two types inherit from the `AbstractStringBuilder` class. Therefore, the clone set can be refactored by declaring the parameter of an extracted method as `AbstractStringBuffer` type. On the other hand, since the proposed technique is implemented to generate a method from one of the code fragments in the clone set as described in 3.2.2, it is difficult to declare the types of the parameters in an abstract way. This is one of the future works in the implementation.

In summary, though the proposed technique needs some implementation improvements, compared to the existing technique, the proposed technique calculates the number of reduced LoC by using only the clone sets that can be merged. In other words, the proposed technique can calculate the number of reduced LoC more accurately than the existing technique.

```
1324 if (splash != null) {
1325  splash.dispose();
1326  splash = null;
1327 }
```

Figure 25: Example of a clone that was actually not refactorable because of the type of the variable between the clones has no inheritance relationship, but hte existing technique targeted to merge

```
961 String word = st.nextToken();
962 if (lineLength == leadingWhitespaceWidth) {
963  // do nothing
964 } else if (lineLength + word.length() + 1 > ...) {
965  buf.append('¥n');
966  buf.append(leadingWhitespace);
967  lineLength = leadingWhitespaceWidth;
968 } else {
969  buf.append(' ');
970  lineLength++;
971 }
972 buf.append(word);
973 lineLength += word.length();
```

Figure 26: Example of a clone that was actually refactorable, but the proposed technique did not target to merge because our implementation are insufficient

5 Limitations

5.1 Limitations in executing the proposed technique

In this part, we showed that the proposed technique was able to calculate the number of reduced LoC more accurately than the existing technique. However, when the conditions for calculating the number of reducible LoC are limited, it is necessary to use the existing technique instead of the proposed technique. For such situations, we discuss the execution environment and execution time of both techniques.

5.1.1 Limitations in the execution environment

In order to execute the proposed technique, the target software must be compiled and tested. Therefore, it is difficult to execute the proposed technique if the environment for compiling and testing the target software is limited and the number of computers that can provide the environment is limited. In such a case, the existing technique should be used.

On the other hand, in order to execute the existing technique, the external tools CCFinderX [41] and JDeodorant [34] are required to run the existing technique. Therefore, if the uses do not have the equipment to use these external tools, they should use the proposed technique.

5.1.2 Limitations in the execution time

In this part, we showed that the proposed technique was able to measure the number of reduced LoC more accurately than the existing technique for software with hundreds of thousands of LoC in a practical time. Therefore, we consider that the proposed technique should be used instead of the existing technique for software with hundreds of thousands of LoC.

On the other hand, the proposed technique should not be used to measure the number of reducible LoC for large software that requires a long time to compile and test. This is because the proposed technique repeatedly compiles and tests the software and thus may not be able to measure the number of reduced LoC in a practical time. The proposed technique can be used when the target software uses incremental builds, which makes the second and subsequent compile and test faster. However, in other cases, the use of the existing technique should be considered.

5.2 Limitations in the implementation

In this part, various limitations are given to the implementation of the proposed technique. The limitations given are as follows.

5.2.1 All extracted methods are declared in one class

In our implementation, all methods extracted by refactoring are declared in a single class. This reason this limitation was given that when merging clones that span multiple classes, we thought it would be difficult to automatically determine the class to declare the extracted methods. When the developer performs refactoring, the extracted method will be declared in the appropriate class based on the developer's judgment.

5.2.2 Clones are detected block by block

In our implementation, the clones are detected in block-level. The reason this limitation was given is that we considered that clones detected in block-level are easier to extract as a method. When developers conduct the refactoring, not only blocks but also continuous statement lines are targeted to refactor.

5.2.3 Code fragments that contain `return` statements are not detected as clones

In our implementation, code fragments that contain `return` statements are not detected as clones. The reason this limitation was given is that we thought it would be difficult to automatically perform extract method on code fragments that contain `return` statements. This limitation does not mean that developers will not refactor the clones that contain `return` statement.

5.2.4 Arguments are passed by reference to the extracted method

In the implementation of this part, arguments are passed by reference to the extracted methods. This is because we thought it would be difficult to automatically determine whether the variables given as arguments are changed in the method. When developers conduct refactoring, they should check whether the variables given as arguments are changed in the method and return the variable that is changed in the method.

6 Conclusion

In this part, we proposed a technique to measure the number of reduced LoC more accurately based on the results of clone refactoring. The proposed technique repeatedly and automatically performs clone detection, merging them, source code compilation, and testing on the target source code. The experimental results show that the proposed technique can merge only clones that can be actually refactored and to calculate the number of reduced LoC more accurately than the existing technique.

The following subsections are our future works.

Utilizing other refactoring patterns

At this moment, the proposed technique utilizes *Extract Method* pattern to remove clones. However, there are various ways to remove clones: for example, *Pull Up Method* or *Form Template Method*. Considering other refactoring patterns to remove clones, calculated reducible LoC will get closer to the real value when developers appropriately remove clones.

Improving the implementation

As described in Section 4.2.2, we encountered compilation errors in the experiment. However, most of such compilation errors are avoidable if we devise an implementation. If we do so, we will be able to get better reducible LoC.

Measuring the execution time on large software

In this part, we showed that the proposed technique was able to measure the number of reduced LoC in a practical time for software with hundreds of thousands of LoC. On the other hand, we did not measure the execution time for large-scale software that requires a long time for compilation and testing. It is important to confirm whether or not the proposed technique can be applied to such software in the future.

Acknowledgements

This thesis has been completed with the help of many people.

First of all, I would like to appreciate my supervisor, Professor Shinji Kusumoto. I was very fortunate to be able to do research under him. His support was very diverse, such as daily research activities, attending conferences, and submitting papers. Moreover, he has been a sympathetic supporter of mine and always cheered me up.

I would like to express the most gratitude to Associate Professor Yoshiki Higo. A large part of this thesis has been made possible with his three years of dedicated and meticulous guidance. His advice was sometimes too stern to be painful for me, but it has steadily made me grow. Thanks to his support, I have eventually been able to decide on a research topic and conducted the experiments on my own.

I would like to express my deep appreciation for Assistant Professor Shinsuke Matsumoto. I was greatly inspired by his philosophy, aesthetics, and presentation technique. Thanks to him, my writing, diagram, and presentation skills have been greatly improved. I think these skills will be useful after graduation.

I would like to express my thanks to Tomoko Kamiya, a clerical assistant. She took care of the clerical works so that I was able to focus on my research.

I would like to appreciate Professor Katsuro Inoue because the many studies he has done were the driving force behind my own research. Moreover, he gave me much helpful advice in various situations, such as Computer Science Seminar.

I would like to express my thanks to Ryo Arima, Akito Tanikado, Keigo Naito, Hiroyuki Matsuo, and Miwa Sasaki, who graduated in 2019. They mentored me as a newcomer to Kusumoto laboratory. Without their help, my research would not have been as successful as it was.

My deep appreciation goes to Hiroto Tanaka, Masayuki Doi, Nozomi Nakajima, and Junnosuke Matsumoto, who graduated in 2020. They have been a great help to me over the past two years. Especially, Masayuki Doi supported my research activities, such as writing papers and Junnosuke Matsumoto gave much technical advice to me. My foundation of research activities has consisted of the knowledge they gave. I would like to greater gratitude to them.

I would like to much appreciation for first-year master's students, Naoto Ichikawa, Ryoko Izuta, Sho Ogino, Akira Fujimoto, and Aoi Maejima and fourth-year undergraduate students, Masashi Iriyama, Kanta Kotou, Masayuki Taniguchi, Tomoaki Tsuru, and Hiroto Watanabe, for their grateful support. Thanks to their efforts to improve the laboratory environment, I was able to do my research comfortably. I was also able to gain valuable experience in teaching the younger students.

I would like to express my special thanks to second-year master's students, Hideaki Azuma, Tetsushi Kuma, Yuya Tomida, and Kaisei Hanayama. I spent the longest time with them in Kusumoto laboratory. Their presence has been a great encouragement to me in my long laboratory life. Especially, Yuya Tomida was generous with his technical support for me. I am very fortunate to have good friends.

Finally, I sincerely appreciate my parents for their 24-years support.

References

[1] Richard N Burns and Alan R. Dennis. Selecting the appropriate application development methodology. *ACM SIGMIS Database: the DATABASE for Advances in Information Systems*, 17(1):19–23, 1985.

[2] Rajiv D Banker, Srikant M Datar, Chris F Kemerer, and Dani Zweig. Software complexity and maintenance costs. *Communications of the ACM*, 36(11):81–95, 1993.

[3] Alexander J Rostkowycz, Václav Rajlich, and Andrian Marcus. A case study on the long-term effects of software redocumentation. In *Proceedings of 20th IEEE International Conference on Software Maintenance*, pages 92–101, 2004.

[4] Manishankar Mondal, Md Saidur Rahman, Ripon K Saha, Chanchal K Roy, Jens Krinke, and Kevin A Schneider. An empirical study of the impacts of clones in software maintenance. In *Proceedings of 2011 IEEE 19th International Conference on Program Comprehension*, pages 242–245, 2011.

[5] Yoshiki Higo, Yasushi Ueda, Toshihro Kamiya, Shinji Kusumoto, and Katsuro Inoue. On Software Maintenance Process Improvement Based on Code Clone Analysis. In *Proceedings of International Conference on Product Focused Software Process Improvement*, pages 185–197, 2002.

[6] Ghazi Alkhatib. The maintenance problem of application software: An empirical analysis. *Journal of Software Maintenance: Research and Practice*, 4(2):83–104, 1992.

[7] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. *Ieee transactions on software engineering*, 38(1):54–72, 2011.

[8] Matias Martinez and Martin Monperrus. Astor: Exploring the design space of generate-and-validate program repair beyond genprog. *Journal of Systems and Software*, 151:65–80, 2019.

[9] W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42(8):707–740, 2016.

[10] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. On the accuracy of spectrum-based fault localization. In *Proceedings of Testing: Academic and industrial conference practice and research techniques-MUTATION*, pages 89–98, 2007.

[11] Marco D'Ambros, Michele Lanza, and Romain Robbes. An extensive comparison of bug prediction approaches. In *Proceedings of 2010 7th IEEE Working Conference on Mining Software Repositories*, pages 31–41, 2010.

[12] Cagatay Catal and Banu Diri. A systematic review of software fault prediction studies. *Expert systems with applications*, 36(4):7346–7354, 2009.

[13] Tom Mens and Tom Tourwé. A survey of software refactoring. *IEEE Transactions on software engineering*, 30(2):126–139, 2004.

[14] Frank Simon, Frank Steinbruckner, and Claus Lewerentz. Metrics based refactoring. In *Proceedings 5th european conference on software maintenance and reengineering*, pages 30–38. IEEE, 2001.

[15] Mark Weiser. Program slicing. *IEEE Transactions on software engineering*, (4):352–357, 1984.

[16] Keith Brian Gallagher and James R Lyle. Using program slicing in software maintenance. *IEEE transactions on software engineering*, 17(8):751–761, 1991.

[17] Elmar Juergens, Florian Deissenboeck, Benjamin Hummel, and Stefan Wagner. Do code clones matter? In *Proceedings of 2009 IEEE 31st International Conference on Software Engineering*, pages 485–495, 2009.

[18] Bruno Lague, Daniel Proulx, Jean Mayrand, Ettore M Merlo, and John Hudepohl. Assessing the benefits of incorporating function clone detection in a development process. In *Proceedings International Conference on Software Maintenance*, pages 314–321, 1997.

[19] Stefan Bellon, Rainer Koschke, Giulio Antoniol, Jens Krinke, and Ettore Merlo. Comparison and evaluation of clone detection tools. *IEEE Transactions on software engineering*, 33(9):577–591, 2007.

[20] Chanchal Kumar Roy and James R Cordy. A survey on software clone detection research. *Queen's School of Computing TR*, 541(115):64–68, 2007.

[21] Andrew Walker, Tomas Cerny, and Eungee Song. Open-source tools and benchmarks for code-clone detection: past, present, and future trends. *ACM SIGAPP Applied Computing Review*, 19(4):28–39, 2020.

[22] Abdullah Sheneamer and Jugal Kumar Kalita. A Survey of Software Clone Detection Techniques. *International Journal of Computer Applications*, 2016.

[23] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K Roy, and Cristina V Lopes. SourcererCC: Scaling code clone detection to big-code. In *Proceedings of the 38th International Conference on Software Engineering*, pages 1157–1168, 2016.

[24] Yoshiki Higo, Yasushi Ueda, Shinji Kusumoto, and Katsuro Inoue. Simultaneous Modification Support based on Code Clone Analysis. In *Proceedings of 14th Asia-Pacific Software Engineering Conference*, pages 262–269, 2007.

[25] Yoshiki Higo, Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Refactoring support based on code clone analysis. In *Proceedings of International Conference on Product Focused Software Process Improvement*, pages 220–233, 2004.

[26] Martin Fowler. *Refactoring: improving the design of existing code.* Addison-Wesley Professional, 2018.

[27] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. A field study of refactoring challenges and benefits. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, pages 1–11, 2012.

[28] Tasuku Nakagawa, Yoshiki Higo, and Shinji Kusumoto. CLIONE: Clone Modification Support for Pull Request Based Development. In *Proceedings of the 27th Asia-Pacific Software Engineering Conference*, pages 455–459, 2020.

[29] Shogo Tokui, Norihiro Yoshida, Eunjong Choi, and Katsuro Inoue. Clone Notifier: Developing and Improving the System to Notify Changes of Code Clones. In *Proceedings of 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering*, pages 642–646, 2020.

[30] Yuki Yamanaka, Eunjong Choi, Norihiro Yoshida, Katsuro Inoue, and Tateki Sano. Applying clone change notification system into an industrial development process. In *Proceedings of 2013 21st International Conference on Program Comprehension*, pages 199–206, 2013.

[31] Yuki Yamanaka, Eunjong Choi, Norihiro Yoshida, Katsuro Inoue, and Tateki Sano. Industrial Application of Clone Change Management System. In *Proceedings of the 6th International Workshop on Software Clones*, pages 67–71, 2012.

[32] Tasuku Nakagawa, Yoshiki Higo, Matsumoto Junnosuke, and Shinji Kusumoto. Measurement Reducible Lines of Code Based on Automated Merging Code Clones. *IPSJ Journal*, 2021 (to appear).

[33] Takuya Ishizu, Norihiro Yoshida, Eunjong Choi, and Katsuro Inoue. Analyzing the Amount of Reducible Source Code Based on the Refactorability of Software Clones. *IPSJ Journal*, 60(4):1051–1062, 2019.

[34] Nikolaos Tsantalis, Davood Mazinanian, and Giri Panamoottil Krishnan. Assessing the Refactorability of Software Clones. *IEEE Transactions on Software Engineering*, 41(11):1055–1090, 2015.

[35] Norihiro Yoshida, Takuya Ishizu, Bufurod Edwards III, and Katsuro Inoue. How slim will my system be? estimating refactored code size by merging clones. In *Proceedings of the 26th Conference on Program Comprehension*, pages 352–360, 2018.

[36] Pengcheng Wang, Jeffrey Svajlenko, Yanzhao Wu, Yun Xu, and Chanchal K Roy. CCAligner: a token based large-gap clone detector. In *Proceedings of the 40th International Conference on Software Engineering*, pages 1066–1077, 2018.

[37] Ming Wu, Pengcheng Wang, Kangqi Yin, Haoyu Cheng, Yun Xu, and Chanchal K Roy. LVMapper: A Large-variance Clone Detector Using Sequencing Alignment Approach. *IEEE Access*, 8:27986–27997, 2020.

[38] J Howard Johnson. Substring Matching for Clone Detection and Change Tracking. In *Proceedings of 1994 International Conference on Software Maintenance*, pages 120–126, 1994.

[39] Stéphane Ducasse, Matthias Rieger, and Serge Demeyer. A language independent approach for detecting duplicated code. In *Proceedings of IEEE International Conference on Software Maintenance*, pages 109–118, 1999.

[40] Simian. `https://www.harukizaemon.com/simian/`.

[41] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.

[42] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. CP-Miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on software Engineering*, 32(3):176–192, 2006.

[43] Rakesh Agrawal and Ramakrishnan Srikant. Mining sequential patterns. In *Proceedings of the 11th International Conference on Data Engineering*, pages 3–14, 1995.

[44] Ira D Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *Proceedings of International Conference on Software Maintenance*, pages 368–377, 1998.

[45] Rainer Koschke, Raimar Falke, and Pierre Frenzel. Clone detection using abstract syntax suffix trees. In *Proceedings of 2006 13th Working Conference on Reverse Engineering*, pages 253–262, 2006.

[46] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones. In *Proceedings of the 29th International Conference on Software Engineering*, pages 96–105, 2007.

[47] Loïc Paulevé, Hervé Jégou, and Laurent Amsaleg. Locality sensitive hashing: A comparison of hash function types and querying mechanisms. *Pattern recognition letters*, 31(11):1348–1358, 2010.

[48] Raghavan Komondoor and Susan Horwitz. Using Slicing to Identify Duplication in Source Code. In *Proceedings of Static Analysis*, pages 40–56, 2001.

[49] Jens Krinke. Identifying similar code with program dependence graphs. In *Proceedings Eighth Working Conference on Reverse Engineering*, pages 301–309, 2001.

[50] Yoshiki Higo and Shinji Kusumoto. Code Clone Detection on Specialized PDGs with Heuristics. In *Proceedings of 2011 15th European Conference on Software Maintenance and Reengineering*, pages 75–84, 2011.

[51] James R Cordy and Chanchal K Roy. The NiCad Clone Detector. In *Proceedings of 2011 IEEE 19th International Conference on Program Comprehension*, pages 219–220, 2011.

[52] James R Cordy. The TXL source transformation language. *Science of Computer Programming*, 61(3):190–210, 2006.

[53] Kazuki Yokoi, Eunjong Choi, Norihiro Yoshida, and Katsuro Inoue. Investigating Vector-based Detection of Code Clones Using BigCloneBench. In *Proceedings of 2018 25th Asia-Pacific Software Engineering Conference*, pages 699–700, 2018.

[54] Miryung Kim, Vibha Sazawal, David Notkin, and Gail Murphy. An empirical study of code clone genealogies. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 187–196, 2005.

[55] Yue Yu, Huaimin Wang, Vladimir Filkov, Premkumar Devanbu, and Bogdan Vasilescu. Wait for it: Determinants of pull request evaluation latency on github. In *Proceedings of 2015 IEEE/ACM 12th working conference on mining software repositories*, pages 367–371, 2015.

[56] Oleksii Kononenko, Tresa Rose, Olga Baysal, Michael Godfrey, Dennis Theisen, and Bart De Water. Studying pull request merges: a case study of shopify's active merchant. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, pages 124–133, 2018.

[57] Georgios Gousios, Martin Pinzger, and Arie van Deursen. An exploratory study of the pull-based software development model. In *Proceedings of the 36th International Conference on Software Engineering*, pages 345–355, 2014.

[58] Nikolaos Tsantalis, Davood Mazinanian, and Shahriar Rostami. Clone refactoring with lambda expressions. In *Proceedings of 2017 IEEE/ACM 39th International Conference on Software Engineering*, pages 60–70, 2017.

[59] Keisuke Hotta, Yoshiki Higo, and Shinji Kusumoto. Identifying, tailoring, and suggesting form template method refactoring opportunities with program dependence graph. In *Proceedings of 2012 16th European Conference on Software Maintenance and Reengineering*, pages 53–62, 2012.

[60] Rainer Koschke. Survey of research on software clones. In *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2007.

[61] Hou Min and Zhang Li Ping. Survey on software clone detection research. In *Proceedings of International Conference on Management Engineering, Software Engineering and Service Sciences*, page 9–16, 2019.

[62] Aakanshi Gupta and Bharti Suri. A Survey on Code Clone, Its Behavior and Applications. In *Proceedings of Networking Communication and Data Knowledge Engineering*, pages 27–39, 2018.

[63] Manishankar Mondal, Chanchal K Roy, and Kevin A Schneider. A survey on clone refactoring and tracking. *Journal of Systems and Software*, 159:110429, 2020.

[64] Salah Bouktif, Giuliano Antoniol, Ettore Merlo, and Markus Neteler. A Novel Approach to Optimize Clone Refactoring Activity. In 1885-1892, editor, *Proceedings of the 8th annual conference of Genetic and Evolutionary computation*, 2006.

[65] Lingxiao Jiang, Zhendong Su, and Edwin Chiu. Context-based detection of clone-related bugs. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 55–64, 2007.

[66] Debarshi Chatterji, Jeffrey C Carver, Beverly Massengil, Jason Oslin, and Nicholas A Kraft. Measuring the Efficacy of Code Clone Information in a Bug Localization Task: An Empirical

Study. In *Proceedings of 2011 International Symposium on Empirical Software Engineering and Measurement*, pages 20–29, 2011.

[67] Miryung Kim, Vibha Sazawal, David Notkin, and Gail Murphy. An Empirical Study of Code Clone Genealogies. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 187–196, 2005.

[68] Takashi Ishio, Yusuke Sakaguchi, Kaoru Ito, and Katsuro Inoue. Source File Set Search for Clone-and-Own Reuse Analysis. In *Proceedings of 2017 IEEE/ACM 14th International Conference on Mining Software Repositories*, pages 257–268, 2017.

[69] Mathieu Nayrolles and Abdelwahab Hamou-Lhadj. CLEVER: Combining Code Metrics with Clone Detection for Just-in-Time Fault Prevention and Resolution in Large Industrial Projects. In *Proceedings of the 15th International Conference on Mining Software Repositories*, pages 153–164, 2018.

[70] Jeffrey Svajlenko and Chanchal K Roy. CloneWorks: A Fast and Flexible Large-Scale Near-Miss Clone Detection Tool. In *Proceedings of International Conference on Software Engineering Companion*, pages 177–179, 2017.

[71] Guanhua Li, Yijian Wu, Chanchal K Roy, Jun Sun, Xin Peng, Nanjie Zhan, Bin Hu, and Jingyi Ma. SAGA: Efficient and Large-Scale Detection of Near-Miss Clones with GPU Acceleration. In *Proceedings of 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering*, pages 272–283, 2020.

[72] Andrei Z. Broder, Steven C. Glassman, Mark S. Manasse, and Geoffrey Zweig. Syntactic clustering of the web. *Computer Networks and ISDN Systems*, 29(8):1157–1166, 1997.

[73] Donald Ervin Knuth. *The art of computer programming*. Pearson Education, 1997.

[74] Benjamin Hummel, Elmar Juergens, Lars Heinemann, and Michael Conradt. Index-based code clone detection: incremental, distributed, scalable. In *Proceedings of 2010 IEEE International Conference on Software Maintenance*, pages 1–9, 2010.

[75] Lasse Bergroth, Hakonen Harri, and Raita Timo. A survey of longest common subsequence algorithms. In *Proceedings of 7th International Symposium on String Processing and Information Retrieval*, pages 39–48, 2000.

[76] James W. Hunt and Thomas G. Szymanski. A Fast Algorithm for Computing Longest Common Subsequences. *Communications of the ACM*, 20(5):350—353, 1977.

[77] Jeffrey Svajlenko and Chanchal K Roy. BigCloneEval: A Clone Detection Tool Evaluation Framework with BigCloneBench. In *Proceedings of 2016 IEEE International Conference on Software Maintenance and Evolution*, pages 596–600, 2016.

[78] Jeffrey Svajlenko and Chanchal K Roy. Evaluating clone detection tools with BigCloneBench. In *Proceedings of 2015 IEEE International Conference on Software Maintenance and Evolution*, pages 131–140, 2015.

[79] Jeffrey Svajlenko and Chanchal K Roy. The Mutation and Injection Framework: Evaluating Clone Detection Tools with Mutation Analysis. *IEEE Transactions on Software Engineering*, pages 1–1, 2019.

[80] Ming Wu, Pengcheng Wang, Kangqi Yin, Haoyu Cheng, Yun Xu, and Chanchal K. Roy. LVMapper: A Large-variance Clone Detector Using Sequencing Alignment Approach, 2019.

[81] Katsuro Inoue, Yuya Miyamoto, Daniel M German, and Takashi Ishio. Code Clone Matching: A Practical and Effective Approach to Find Code Snippets. *arXiv preprint arXiv:2003.05615*, 2020.

[82] Ambient Software Evolution Group: IJaDataset 2.0. `http://secold.org/projects/seclone`.

[83] CLOC. `http://cloc.sourceforge.net/`.

[84] Farouq Al-Omari, Chanchal K Roy, and Tonghao Chen. SemanticCloneBench: A Semantic Code Clone Benchmark using Crowd-Source Knowledge. In *Proceedings of International Workshop on Software Clones*, pages 57–63, 2020.

[85] Yusuke Yuki, Yoshiki Higo, Keisuke Hotta, and Shinji Kusumoto. Generating clone references with less human subjectivity. In *Proceedings of International Conference on Program Comprehension*, pages 1–4, 2016.

[86] Tiantian Wang, Mark Harman, Yue Jia, and Jens Krinke. Searching for Better Configurations: A Rigorous Approach to Clone Evaluation. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, page 455–465, 2013.

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

[88] Yue Zou, Bihuan Ban, Yinxing Xue, and Yun Xu. Ccgraph: a pdg-based code clone detector with approximate graph matching. In *Proceedings of 2020 35th IEEE/ACM International Conference on Automated Software Engineering*, pages 931–942, 2020.

[89] Nino Shervashidze, Pascal Schweitzer, Erik Jan Van Leeuwen, Kurt Mehlhorn, and Karsten M Borgwardt. Weisfeiler-lehman graph kernels. *Journal of Machine Learning Research*, 12(9), 2011.

[90] Gehan MK Selim, King Chun Foo, and Ying Zou. Enhancing Source-Based Clone Detection Using Intermediate Representation. In *Proceedings of 2010 17th Working Conference on Reverse Engineering*, pages 227–236, 2010.

[91] Soot. `http://soot-oss.github.io/soot/`.

[92] Pedro M Caldeira, Kazunori Sakamoto, Hironori Washizaki, Yoshiaki Fukazawa, and Takahisa Shimada. Improving Syntactical Clone Detection Methods through the Use of an Intermediate Representation. In *Proceedings of 2020 IEEE 14th International Workshop on Software Clones*, pages 8–14, 2020.

[93] Vaibhav Saini, Farima Farmahinifarahani, Yadong Lu, Pierre Baldi, and Cristina V. Lopes. Oreo: Detection of Clones in the Twilight Zone. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 354—-365, 2018.

[94] CCFinderX. `http://www.ccfinder.net/`.

[95] Tomoya Ishihara, Keisuke Hotta, Yoshiki Higo, Hiroshi Igaki, and Shinji Kusumoto. Inter-Project Functional Clone Detection Toward Building Libraries - An Empirical Study on 13,000 Projects. In *Proceedings of 2012 19th Working Conference on Reverse Engineering*, pages 387–391, 2012.

[96] Elmar Juergens, Florian Deissenboeck, and Benjamin Hummel. Clonedetective—a workbench for clone detection research. In *Proceedings of 2009 IEEE 31st International Conference on Software Engineering*, pages 603–606, 2009.

[97] Manziba Akanda Nishi and Kostadin Damevski. Scalable code clone detection and search based on adaptive prefix filtering. *Journal of Systems and Software*, 137:130–142, 2018.

[98] Joseph Feller and Brian Fitzgerald. *Understanding Open Source Software Development*. Addison-Wesley Longman Publishing Co., Inc., USA, 2002.

[99] GitHub Apps. `https://developer.github.com/apps/`.

[100] Tasuku Nakagawa, Yoshiki Higo, Junnosuke Matsumoto, and Shinji Kusumoto. How Compact Will My System Be? A Fully-Automated Way to Calculate LoC Reduced by Clone Refactoring. In *Proceedings of 2019 26th Asia-Pacific Software Engineering Conference*, pages 284–291, 2019.

[101] Chaiyong Ragkhitwetsagul and Jens Krinke. Using compilation/decompilation to enhance clone detection. In *Proceedings of 2017 IEEE 11th International Workshop on Software Clones*, pages 1–7, 2017.

[102] Kyohei Uemura, Akira Mori, Eunjong Choi, and Hajimu Iida. Tracking Method-Level Clones and a Case Study. In *Proceedings of 2019 IEEE 13th International Workshop on Software Clones*, pages 27–33. IEEE, 2019.

[103] Yoshiki Higo, Shinsuke Matsumoto, Ryo Arima, Akito Tanikado, Keigo Naitou, Junnosuke Matsumoto, Yuya Tomida, and Shinji Kusumoto. kGenProg: A High-performance, High-extensibility and High-portability APR System. In *Proceedings of 2018 25th Asia-Pacific Software Engineering Conference*, pages 697–698, 2018.

[104] Philippe Kruchten, Robert L Nord, and Ipek Ozkaya. Technical Debt: From Metaphor to Theory and Practice. *IEEE Software*, 29(6):18–21, 2012.

[105] Francesca Arcelli Fontana, Vincenzo Ferme, and Stefano Spinelli. Investigating the impact of code smells debt on quality code evaluation. In *Proceedings of 2012 3rd International Workshop on Managing Technical Debt*, pages 15–22, 2012.

[106] Erich Gamma. *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.

[107] Vahid Alizadeh, Mohamed Amine Ouali, Marouane Kessentini, and Meriem Chater. RefBot: Intelligent Software Refactoring Bot. In *Proceedings of 2019 34th IEEE/ACM International Conference on Automated Software Engineering*, pages 823–834, 2019.

[108] Antônio Carvalho, Welder Luz, Diego Marcílio, Rodrigo Bonifácio, Gustavo Pinto, and Edna Dias Canedo. C-3PR: A Bot for Fixing Static Analysis Violations via Pull Requests. In *Proceedings of 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering*, pages 161–171, 2020.

[109] Hoan Anh Nguyen, Tung Thanh Nguyen, Nam H Pham, Jafar Al-Kofahi, and Tien N Nguyen. Clone Management for Evolving Software. *IEEE Transactions on Software Engineering*, 38(5):1008–1026, 2011.

[110] Michiel De Wit, Andy Zaidman, and Arie Van Deursen. Managing code clones using dynamic change tracking and resolution. In *Proceedings of 2009 IEEE International Conference on Software Maintenance*, pages 169–178, 2009.

[111] Bennett P. Lientz and E. Burton Swanson. *Software Maintenance Management*. Addison-Wesley Longman Publishing Co., Inc., 1980.

[112] Stephen G Eick, Todd L Graves, Alan F Karr, J Steve Marron, and Audris Mockus. Does code decay? assessing the evidence from change management data. *IEEE Transactions on Software Engineering*, 27(1):1–12, 2001.

[113] Thomas M. Pigoski. *Practical Software Maintenance: Best Practices for Managing Your Software Investment*. Wiley Publishing, 1st edition, 1996.

[114] Harry M. Sneed. Planning the reengineering of legacy systems. *IEEE software*, 12(1):24–34, 1995.

[115] Anfernee Goon, Yuhao Wu, Makoto Matsushita, and Katsuro Inoue. Evolution of code clone ratios throughout development history of open-source C and C++ programs. In *Proceedings of 2017 IEEE 11th International Workshop on Software Clones*, pages 1–7, 2017.

[116] Rainer Koschke and Saman Bazrafshan. Software-clone rates in open-source programs written in C or C++. In *Proceedings of 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering*, volume 3, pages 1–7, 2016.

[117] Eclipse. `https://www.eclipse.org/`.

[118] Eclipse Java development tools. `https://www.eclipse.org/jdt/`.

[119] Keisuke Hotta, Jiachen Yang, Yoshiki Higo, and Shinji Kusumoto. How Accurate Is Coarse-grained Clone Detection?: Comparison with Fine-grained Detectors. In *Inproceedings of 8th International Workshop on Software Clones*, pages 1–18, 2 2014.

[120] Raghavan Komondoor and Susan Horwitz. Effective, automatic procedure extraction. In *Proceedings of 11th IEEE International Workshop on Program Comprehension*, pages 33–42, 2003.

[121] National Institute of Standards and Technology. Secure Hash Standard. 2015.

Appendix

A Available files

An executable file of NIL https://zenodo.org/record/4492665

Source code of NIL https://github.com/kusumotolab/NIL

Manual validation descriptions of large-variance clones https://zenodo.org/record/4490845.

Large-variance clones used in Section 3.3.2 https://zenodo.org/record/4491016.

Clones used in Section 3.4.1 https://zenodo.org/record/4491052.

Manual validation descriptions of general clones https://zenodo.org/record/4493069.

Codebases with various sizes https://zenodo.org/record/4491208.

Source code of CLIONE https://github.com/T45K/CLIONE

B Hunt-Szymanski Algorithm

Algorithm 2: Hunt-Szymanski Algorithm

Input: element array $A[1:n]$, $B[1:n]$;

Output: Printing the LCS between A and B

1: // Identifiers
2: integer array $THREASH[0:n]$;
3: list array $MATCHLIST[1:n]$;
4: pointer array $LINK[1:n]$;
5: pointer PTR;
6: // Step 1: build linked list
7: **for** $i := 1$ step 1 until n **do**
8: set list array $MATCHLIST[i] := \langle j_1, j_2, \ldots, j_p \rangle$ such that $j_1 > j_2 > \ldots > j_p$ and $A[i] = B[j_q]$ for $1 \leq q \leq p$;
9: **end for**
10: // Step 2: initialize $THREASH$ array
11: $THREASH[0] := 0$;
12: **for** $i := 1$ step 1 until n **do**
13: $THREASH[i] := n + 1$;
14: **end for**
15: $LINK[0] := null$;
16: // Step 3: compute successive $THREASH$ values
17: **for** $i := 1$ step 1 until n **do**
18: **for all** j on $MATCHLIST[i]$ **do**
19: find k such that $THREASH[k-1] < j \leq THREASH[k]$;
20: **if** $j < THREASH[k]$ **then**
21: $THREAS[k] := j$;
22: $LINK[k] := newnode(i, k, LINK[k-1])$;
23: **end if**
24: **end for**
25: **end for**
26: // Step 4: recover the LCS in reverse order
27: $k := largestksuchthatTHREASH[k] \neq n + 1$;
28: $PTR := LINK[k]$;
29: **while** $PTR \neq null$ **do**
30: print (i, j) pair pointed to by PTR;
31: advance PTR;
32: **end while**

C List of Publications

1. Tasuku Nakagawa, Yoshiki Higo, Junnosuke Matsumoto, and Shinji Kusumoto. Measurement Reducible Lines of Code Based on Automated Merging Code Clones. IPSJ Journal, 2021 (to appear).

2. TasukuNakagawa, Yoshiki Higo, and Shinji Kusumoto. CLIONE: Code Clone Modification Support System aimed to integrate into Pull Request Based Developement. IEICE Journal, 2021 (conditional accepted).

3. Tasuku Nakagawa, Yoshiki Higo, Junnosuke Matsumoto, and Shinji Kusumoto. How Compact Will My System Be? A Fully-Automated Way to Calculate Loc Reduced by Clone Refactoring. In *Proceedings of the 26th Asia-Pacific Software Engineering Conference*, pages 284–291, 2019.

4. Tasuku Nakagawa, Yoshiki Higo, and Shinji Kusumoto. CLIONE: Clone Modification Support for Pull Request Based Developement. In *Proceedings of the 27th Asia-Pacific Software Engineering Conference*, pages 455–459, 2020.