2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation

Folding Repeated Instructions for Improving Token-based Code Clone Detection

Hiroaki Murakami, Keisuke Hotta, Yoshiki Higo, Hiroshi Igaki and Shinji Kusumoto
Graduate School of Information Science and Technology, Osaka University,
1-5, Yamadaoka, Suita, Osaka, 565-0871, Japan
Email: {h-murakm,k-hotta,higo,igaki,kusumoto}@ist.osaka-u.ac.jp

***Abstract*—A variety of code clone detection methods have been proposed before now. However, only a small part of them is widely used. Widely-used methods are line-based and token-based ones. They have high scalability because they neither require deep source code analysis nor constructing complex intermediate structures for the detection. High scalability is one of the big advantages in code clone detection tools. On the other hand, line/token-based detections yield many false positives. One of the factors is the presence of repeated instructions in the source code. For example, herein we assume that there are consecutive three printf statements in C source code. If we apply a token-based detection to them, the former two statements are detected as a code clone of the latter two statements. However, such overlapped code clones are redundant and so not useful for developers. In this paper, we propose a new detection method that is free from the influence of the presence of repeated instructions. The proposed method transforms every of repeated instructions into a special form, and then it detects code clones using a suffix array algorithm. The transformation prevents many false positives from being detected. Also, the detection speed remains. The proposed detection method has already been developed as a software tool, FRISC. We confirmed the usefulness of the proposed method by conducting a quantitative evaluation of FRISC with Bellon's oracle.**

***Index Terms*—Code clone detection, False positive reduction, Tool comparison**

I. Introduction

No software has no code clones. In order to detect code clones in the source code automatically, a variety of detection methods has been proposed in the past [1], [22]. At present, line-based and token-based detection methods are often used because of the following reasons:

- line/token-based detections have high scalability because they neither require deep source code analysis nor construct complex intermediate structures for the detection. Consequently, they are used in various contexts of software development. Also, they are used for detecting code clones from large-scale software [13], [17], a number of software [14], [18], [23], a number of consecutive revisions of software [9], [11], [16], [19];
- implementing line/token-based detection methods for multiple programming languages is easier than other detection methods like PDG-based ones. Popular line/token-based tools, CCFinder [13] and Simian [3], can handle widely-used languages such as C/C++, Java, COBOL.

On the other hand, automatic code clone detections by tools inherently produce false positives. Every detection method has its own unique definition of code clones, and it detects code clones based on the definition. However, developers do not need all code clones detected by tools.

Bellon, et al. compared *recall* and *precision* of seven detection tools by using oracle, which is a reference set of code clones [7]. As a result, they revealed the followings:

- high *recall* tools detect many code clones which implies that the detection results of those tools include many false positives [5], [13];
- high *precision* tools have low *recall* [6], [21]. Detecting a small number of false positives is their advantages but they miss many real code clones.

To summarize the above points, line/token-based detections have high scalability, and they can be applied to various contexts of software development. However, they yield many false positives. The authors think that the presence of repeated instructions in source code is a large factor of false positives detection based on our experiences of code clone related research. For example, if we detect code clones from the following example using a token-based approach with a suffix tree or suffix array algorithm, we will obtain a clone pair: one consists of the 1-2 lines code fragment; the other consists of the 2-3 lines code fragment. Both the code fragments in the clone pair are overlapped with each other. Detecting such a clone pair is meaningless.

```
1:  unsigned char division_mask;
2:  unsigned int division_offset;
3:  unsigned int division_size;
```

The above example is a repetition of consecutive variable declarations. If we tailor detection to ignore repeated instructions, the clone pair becomes undetected. Authors have revealed that there are various kinds of repeated instructions in the source code, and many code clones are detected in them with a token-based approach [10]. Consequently, ignoring repeated instructions prevents many false positives from being detected. This paper proposes a new code clone detection method focusing on not detecting false positives in repeated instructions. The contributions of this paper are as follows:

- this paper proposes a new code clone detection method producing less false positives, and it has been developed as a software tool, FRISC;
- we evaluated the proposed method on multiple open source software systems, and confirmed that ignoring

978-0-7695-4783-1/12 $26.00 © 2012 IEEE
DOI 10.1109/SCAM.2012.21

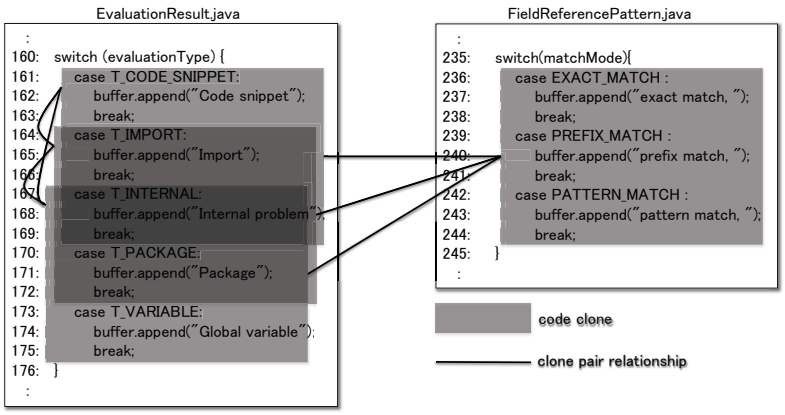


Fig. 1. Motivating example, which shows that many code clones are detected in repeated instructions

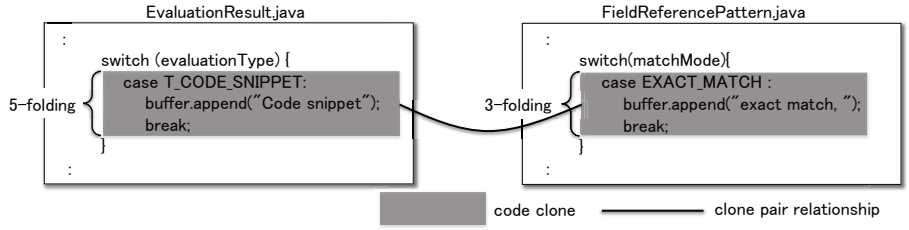


Fig. 2. Source code transformed from the one of Figure 1 by the folding operation

repeated instructions achieves higher *recall* and *precision* than not ignoring them;

- we compared FRISC with multiple existing tools, and we confirmed that FRISC has higher *recall* and *precision* than existing line/token-based detection tools.

II. Research Motivation

Figure 1 shows actual code clones detected from repeated instructions. In the left-side source file, there are five case entries and three case entries exist in the right-side one. If we detect code clones with using line/token-based detection tools, we will obtain six clone pairs. Every detected code clone is a hatching part in Figure 1. As shown in this example, many code clones are detected from repeated instructions.

Code clones in repeated instructions have the following characteristics:

1) both the code clones forming a clone pair are overlapped with each other. There is no reason to detect such a clone pair because both the code clones forming it point almost the same locations of the source code;
2) both the code clones forming a clone pair overlap with both the code clones forming another clone pair. We need not both the clone pairs because they point almost the same locations of the source code.

Detecting all code clones having the above characteristics enlarges detection results, so that we become unaware of code clones in other parts of the target system.

The proposed method can resolve the problem. Intuitively, the proposed method firstly folds repeated instructions, and then it detects code clones. By the folding operation, the source code in Figure 1 is transformed to the source code in Figure 2. Consequently, the proposed method detects only a single clone pair: one is a code clone from the 161th line to the 175th line of the left-side source file; the other is a code clone from the 236th line to the 243th line of the right-side source file. The proposed method identifies the two switch-statements as duplicated code without detecting code clones having the characteristics 1 and 2.

Also, Figure 3 shows an example of real code clones judged manually in the Bellon's experiment [7]. As shown in this figure, humans prefer code clones covering a whole of the repeated instructions rather than ones covering a part of them. In other words, human does not care the differences of the number of repetitions between the code fragments.

Herein, we define the following research question in order to confirm that the proposed method detects fewer false positives, and it detects more preferable code clones.

RQ1: Does folding repeated instructions improve *precision* and *recall* of detection results?

Currently, there is a variety of detection tools. In order to show the usefulness of the proposed method by comparing them, we define the following research question.

RQ2: Does code clone detection with folding repeated instructions have higher accuracy than existing tools?

III. Code Clone Detection with Folding Repeated Instructions

The proposed method consists of the following five steps.

STEP1: Lexical Analysis and Normalization

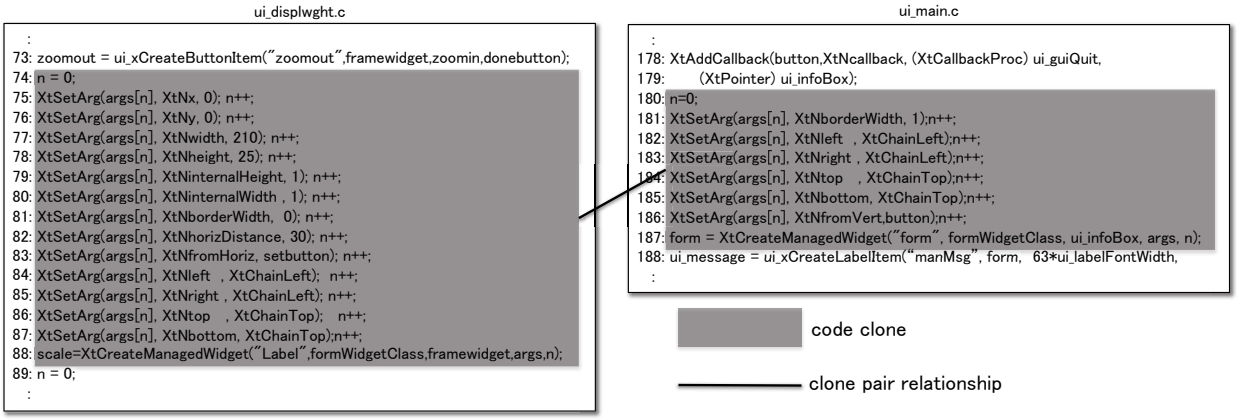


Fig. 3. Motivating example, which shows that human regards whole the repeated instructions as a code clone

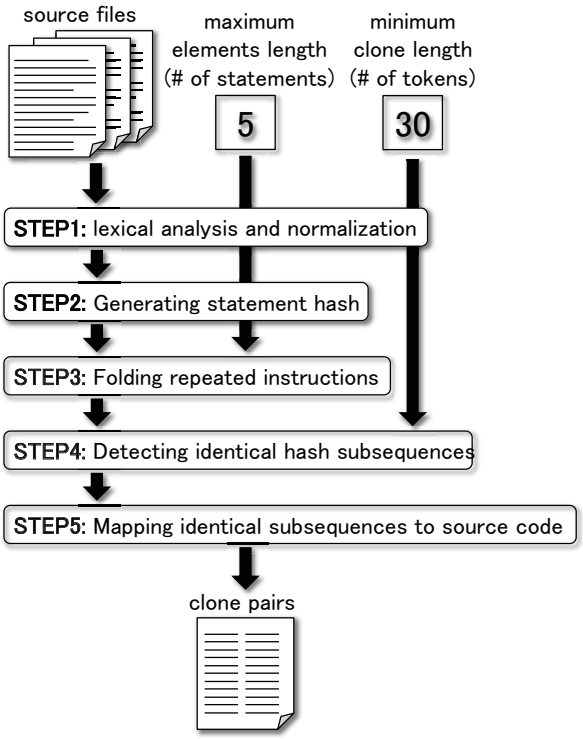


Fig. 4. Overview of the proposed method

STEP2: Generating Statement Hash
STEP3: Folding Repeated Instructions
STEP4: Detecting Identical Hash Sequences
STEP5: Mapping Identical Subsequences to Source Code

The proposed method takes the followings as its inputs:

- source code;
- maximum elements length (number of statements);
- minimum clone length (number of tokens).

The proposed method outputs a set of detected clone pairs. Figure 4 shows an overview of the proposed method. The remainder of this section explains every of the steps in detail.

STEP1: Lexical Analysis and Normalization

In STEP1, all the target source files are transformed into token sequences. User-defined identifiers are replaced with special tokens to detect similar code fragments as code clones even if they include different variables.

STEP2: Generating Statement Hash

In STEP2, a hash value is generated from every statement in the token sequences. Herein, we define a statement as every subsequence between ";", "{", and "}". STEP2 transforms token sequences into hash sequences. Note that every hash has a weight, which means the number of tokens included in its statement.

STEP3: Folding Repeated Instructions

STEP 3 is the core of the proposed code clone detection method. Firstly, repeated subsequences are identified. Every of the identified repeated subsequences is divided into the first repeated elements and its subsequent repeated elements. Then, the subsequent repeated elements are removed from the hash sequences. The weights of deleted elements are added to the weights of their first repeated elements. Algorithm 1 shows the algorithm used for folding repeated subsequences. In the algorithm, *seq* is a hash sequence, and *max_elmt_length* is a maximum elements length. As a result of the algorithm application, all the repeated subsequences whose elements length is equal to or less than the threshold (the maximum elements length) are folded. Figure 5 shows how input source code is transformed into folded hash sequences. Why we use the threshold is that, if elements of repetitions are large, users might not want to treat them as repetitions. Using the threshold realizes more configurable code clone detections.

STEP4: Detecting Identical Hash Subsequences

Identical subsequences are detected from the folded hash sequences. If the sum of weights in an identical subsequence is smaller than the minimum token length, it is discarded.

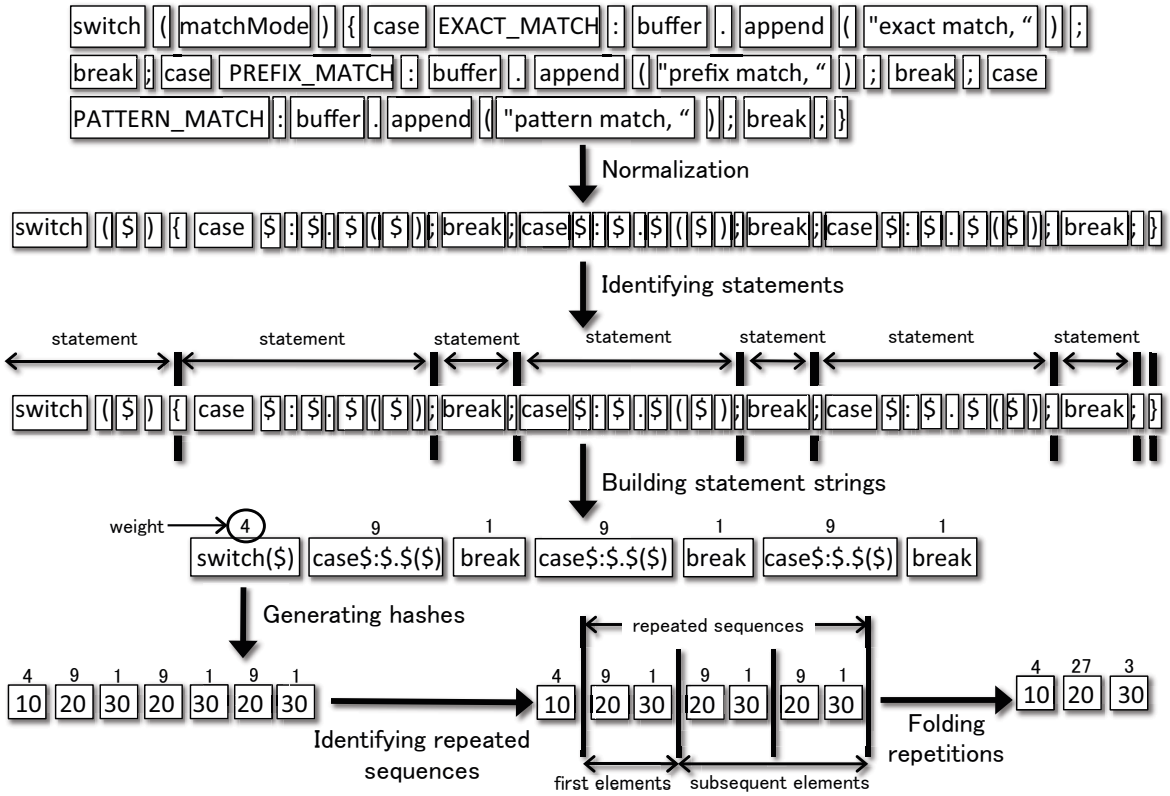


Fig. 5. Example how input source file is transformed into a folded hash sequence

STEP5: Mapping Identical Subsequences to Source Code

Identified subsequences detected in STEP4 are converted to location information in the source code (file path, start line, end line), which are clone pairs.

IV. Implementation

We have developed a software tool, **FRISC** (**F**olding **R**epeated **I**nstructions in **S**ource **C**ode), based on the proposed method. Currently, FRISC can handle Java and C. However, FRISC performs only lexical analysis as a language-dependent processing, so that it is not difficult to extend FRISC to other programming languages.

FRISC supports multi-thread processings. All the steps of the proposed method except STEP5 are processed in parallel. In STEP1, 2, and 3, every thread takes a source file and outputs its hash sequence one-by-one. This processing is performed for all the target source files. In STEP4, every thread detects identical hash subsequences from a different pair of hash sequences generated in STEP3. Of course, identical hash subsequences within a hash sequence are also detected. Current implementation does not perform STEP5 in parallel because it is relatively a lightweight processing. Hence, the detection speed of FRISC can be shortened with multi-thread processing drastically. FRISC accepts the number of threads as its command line option.

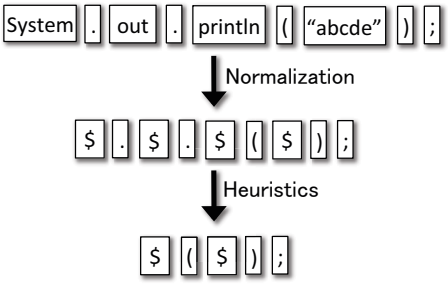


Fig. 6. Example of transformation with heuristics

FRISC uses some heuristics for identifying more significant code clones. Currently, they are as follows:

Shrinking user-defined identifiers connected with “.” By shrinking those identifiers, we can detect code clones even if the number of them are different. Figure 6 shows a transformation how such identifiers are shrinked.

Removing import and package statements We do not think that code clones in import and package statements are useful, and so they are removed in STEP1.

V. Overview of Investigation

We have conducted an investigation to answer the two research questions described in Section II. The investigation consists of two experiments.

Algorithm 1 Folding repetitions

Require: $seq, max\_elmt\_length(\geq 1)$
Ensure: folded seq

```
1: seq_length ← length(seq)
2: for i = 0 to max_elmt_length do
3:   left ← 0
4:   loop
5:     flg ← true
6:     index ← left
7:     tmpleft ← left
8:     count ← 0
9:     while count ≤ i and index < seq_length do
10:      if isSentenceEnd(seq[index]) then
11:        if flg then
12:          k ← index + 1; flg ← false
13:        end if
14:        count ← count + 1
15:      end if
16:      index ← index + 1
17:    end while
18:    if index > seq_length then
19:      break
20:    end if
21:    tmp ← seq[left..index − 1]
22:    count ← 0
23:    left ← index
24:    while count ≤ i and index < seq_length do
25:      if isSentenceEnd(seq[index]) then
26:        count ← count + 1
27:      end if
28:      index ← index + 1
29:    end while
30:    if index > seq_length then
31:      break
32:    end if
33:    tmp2 ← seq[left..index − 1]
34:    if tmp = tmp2 then
35:      seq ← seq[0..left − 1] + seq[index..seq_length]
36:      seq_length ← length(seq)
37:      left ← tmpleft
38:    else
39:      left ← k
40:    end if
41:  end loop
42: end for
```

- **Experiment A:** code clones are detected by FRISC with two settings: one is with the folding operation; the other is without it. Then, *recall* and *precision* of the two detections are calculated and compared.
- **Experiment B:** code clones are detected by FRISC and multiple other tools. Then, *recall* and *precision* of all the detection results are calculated and compared.

In order to calculate *recall* and *precision*, we need real code clones. Herein we use freely available code clone data in [2] as a reference set (a set of code clones to be detected). The reference set includes code clones information of eight software systems. Table I shows an overview of the target systems. In the remainder of this paper, we use the following terms:

Clone candidates: code clones detected by tools
Clone references: code clones included in the reference

We use the *good* value [7] to decide whether every of clone candidates matches any of clone references or not. In this investigation, We use 0.7 as the threshold, which is the same value used in the literature [7].

We calculate *recall* and *precision* for evaluating the detection capability. Assume that R is a detection result, S_{refs} is the set of clone references, and S_R is a set of clone candidates whose *good* values with an instance of clone references is equal to or greater than the threshold in R.

Recall of R ($Recall_R$) is defined as follows:

$$Recall_R = \frac{|S_R|}{|S_{refs}|} \quad (1)$$

Precision of R ($Precision_R$) is defined as follows:

$$Precision_R = \frac{|S_R|}{|R|} \quad (2)$$

This evaluation has a limitation on *recall* and *precision*. The clone references used in the experiments are not all the real code clones included in the target systems. Consequently, the absolute values of *recall* and *precision* are meaningless. *Recall* and *precision* can be used only for relatively comparing detection results. Moreover, we have to pay a special attention to *precision*. A low value of *precision* does not directly indicate that the detection result includes many false positives. A low value means that there are many clone candidates not matching any of the clone references; however, nobody knows whether they are truly false positives or not.

The remainder of this section summarizes the two experiments for investigating the RQs. The details of each experiment are described in Section VI and VII, respectively.

A. Summary of Experiment A

The *precision* with folding repeated instructions is averagely higher than the one without it by 29.8%. On the other hand,

TABLE I
TARGET SOFTWARE

Short name	Language	Lines of code	# of references
netbeans	Java	14,360	55
ant	Java	34,744	30
jdtcore	Java	147,634	1,345
swing	Java	204,037	777
weltab	C	11,460	275
cook	C	70,008	440
snns	C	93,867	1,036
postgresql	C	201,686	555

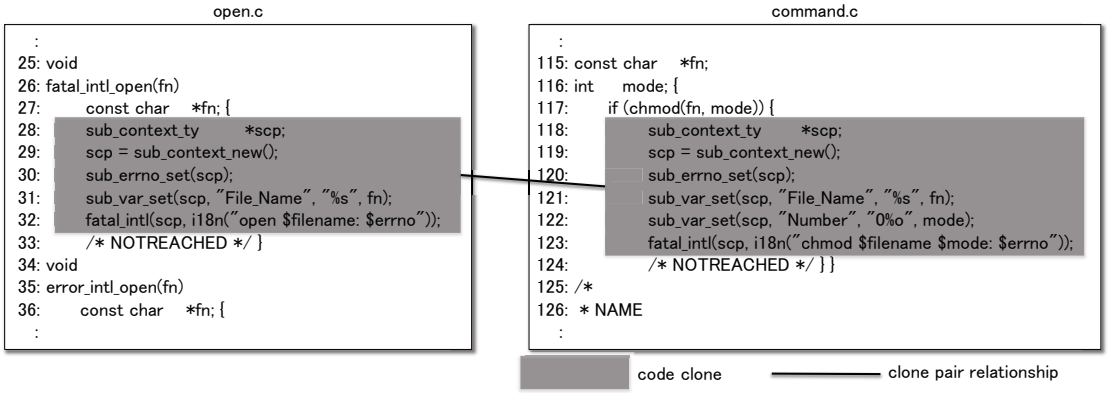
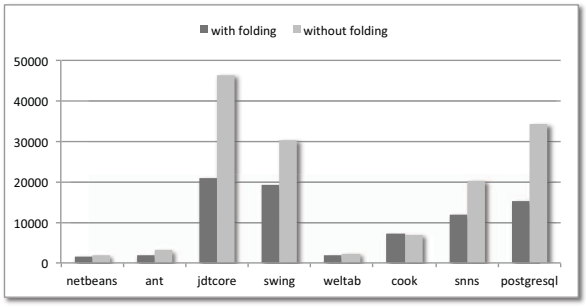
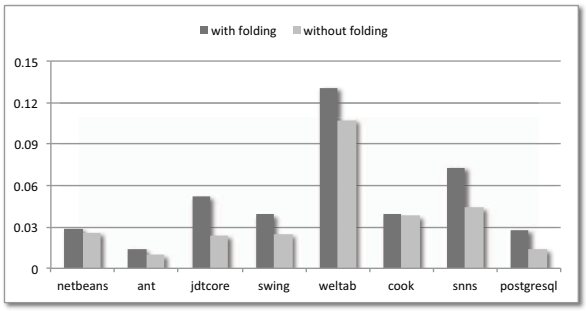


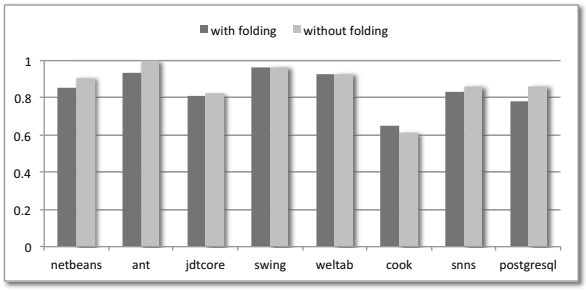
Fig. 8. A clone candidate newly detected by using the folding operation



(a) Number of clone candidates



(b) Precision



(c) Recall

Fig. 7. Number of clone candidates, precision, recall of Experiment A

the folding averagely decreased *recall* by 2.9%. The degree of *precision* increasing is about 10 times of the degree of *recall* decreasing. The execution time with the folding is almost the same as the execution time without it.

B. Summary of Experiment B

FRISC detected more clone references than any of the comparison tools in most cases. Especially, for five out of the eight systems, both the *precision* and *recall* of FRISC are greater than those of CCFinder, which is one of the most widely-used detection tools. Still, the *precision* of FRISC is lower than those of CloneDR [6] and CLAN [20] for all the target systems.

VI. Experiment A

The purpose of Experiment A is to reveal how the number of clone candidates, *precision*, and *recall* are changed by folding repeated instructions. In this experiment, we used the following thresholds.

Maximum elements length: 5 is for detection with the folding operation, and 0 is for detection without it.

Minimum clone length: 30

Figure 7(a) shows the number of clone candidates. The folding decreases the number of clone candidates for almost all the target software. Especially, for jdtcore, which is the software where most clone candidates were detected, the number of clone candidates dropped by about 54%. We browsed the source code of jdtcore, and found that it includes a large number of consecutive if-else-statements, consecutive case entries in switch-statements, and consecutive catch-statements. The folding prevented code clones from being detected from those repetitions in the source code.

Average decreasing rate of clone candidates was about 32%, and we found that, in the case of cook, the number of clone candidates was slightly increased. Figure 8 shows a clone candidate newly detected by using the folding operation. If we do not use the folding operation, the code fragment from the 28th line to the 31st line of the left-side source file is a code clone of the code fragment from the 118th line to the 121st line of the right-side source file. However, the length of the code fragment is 26 tokens, which is less than the minimum token length, 30. Hence, the clone pair was discarded. On the other

hand, when we used the folding operation, the code fragment from the 28th line to the 32nd line of the left-side source file was a code clone of the code fragment from the 118th line to the 123rd line of the right-side source file. The code fragments are greater than the minimum token length, so that the clone pair was output. This is a pair of gapped (type-3) code clones. In cook, there are many of those clone pairs, so that the number of clone candidates with the folding operation is greater than the number of clone candidates without it. We expect that there are such clone pairs in the other target systems too.

Figure 7(b) shows the *precision* of the detections with and without the folding. For all of the software, *precision* was improved. Concretely, we confirmed the followings:

- in the best case, *precision* increased by 53.8%,
- even in the worst case, *precision* increased by 2.6%,
- averagely, *precision* increased by 29.8%.

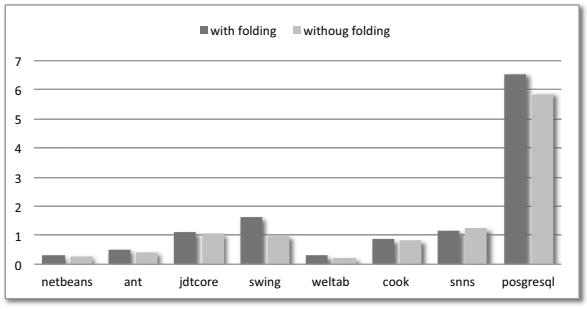
Figure 7(c) shows the *recall* of the detections with and without the folding. The changes of *recall* varied from the target software, unlike *precision*. For system cook, *recall* was improved by the folding. More clone references were detected with the folding operation. Also, for two systems, swing and weltab, *recall* remained unchanged. However, for the other five systems, *recall* was decreased. Averagely, *recall* dropped by 2.9%.

We investigated code clones that were detected without the folding but not detected with the folding to know why they became undetected. Table II shows the proportion of self-overlapping code clones that became undetected. It is easy to remove self-overlapping code clones even if the folding operation is not applied. For example, after detecting code clones, checking whether locations of code fragments of a clone pair is overlapped or not is a simple way. The proportions of self-overlapping code clones are very different from software systems, which means that checking the self-overlapping as a post processing of code clone detection is not enough to reduce false positives.

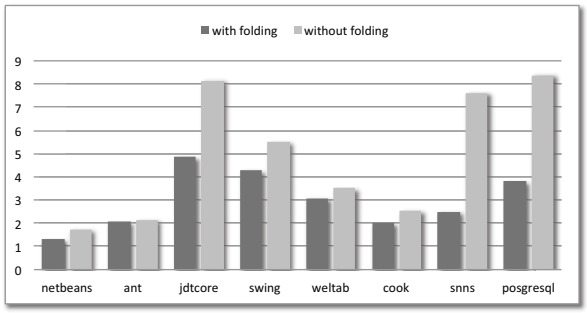
We also investigated the execution time of FRISC with and without the folding. Figure 9 shows the investigation result. Using the folding operation increased execution time of the sum of STEP1, STEP2, and STEP3; however, the difference was negligible. On the other hand, using the folding operation decreased execution time of STEP4. This is because hash sequences, which is the target of the STEP4 operation, were shortened by the folding operation.

In order to check the effective of the multi-thread implementation, we investigated the execution time with 1, 2, 4, 8 threads. Figure 10 shows the investigation result. We can see that, multi-thread processing could reduce the execution time drastically if the execution time with a single thread exceeded 10 seconds.

Consequently we answer RQ1 as follows: *using the folding operation decreased the number of clone candidates by about 32 % averagely. The decreasing caused the improvement of precision, averagely 29.8%. However, it also caused missing some clone references. the average of decreasing recall is 2.9%.* We can conclude that the folding is a useful approach



(a) sum of STEP1, STEP2, and STEP3 (sec.)



(b) STEP4 (sec.)

Fig. 9. Execution time of normalization (STEP1, STEP2, STEP3) and detection (STEP4)

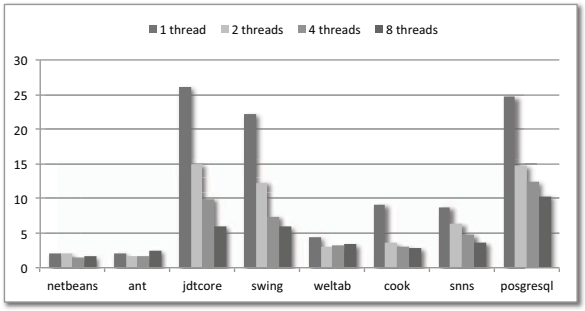


Fig. 10. Execution time with 1, 2, 4, and 8 threads (sec.)

to prevent false positives from being detected meanwhile it misses some clone references.

VII. Experiment B

The purpose of Experiment B is to reveal whether FRISC detects code clones more precisely than existing tools or not.

TABLE II
RATE OF SELF-OVERLAPPING CODE CLONES THAT BECAME UNDETECTED BY USING THE FOLDING OPERATIONS

Software	Self-overlapping
netbeans	75.2%
ant	15.1%
jdtcore	43.8%
swing	37.8%
weltab	52.7%
cook	89.1%
snns	75.5%
postgresql	79.5%

TABLE IV
THE NUMBER OF CLONE CANDIDATES. EVERY "-" MEANS THAT THE DETECTION TOOL COULD NOT FINISH CODE CLONE DETECTION BECAUSE SCALABILITY ISSUE.

Software	FRISC	Dup	CloneDR	CCFinder	CLAN	Duploc	Duplix	Nicad
netbeans	1,636	344	33	5,552	80	223	-	24
ant	1,992	245	42	865	88	162	-	19
jdtcore	21,007	22,589	3,593	26,049	10,111	710	-	1,142
swing	19,115	7,220	3,766	21,421	2,809	-	-	1,804
weltab	1,968	2,742	186	3,898	101	1,754	1,201	160
cook	7,200	8,593	1,438	2,964	449	8,706	2,135	159
snns	11,925	8,978	1,434	18,961	318	5,212	12,181	352
postgresql	15,356	12,965	1,452	21,383	930	-	-	352

In this experiment, we chose the detection tools shown in Table III as targets for the comparison. All the tools except Nicad were used in the experiment conducted by Bellon et al. [7], and we used their experimental result in this experiment.

The remaining tool, Nicad, is a detection tool developed by Roy and Cordy [21]. Nicad detects code clones on block-level or function-level. It identifies duplicated token sequences from every pair of blocks/functions by using LCS (Longest Common Subsequence) algorithm. If the rate of detected duplications of a pair is greater than a threshold, the pair is regarded as a code clone pair.

Table IV shows the number of clone candidates detected by the tools. The number of clone candidates considerably varies from tool to tool. Also, we can see that line/token-based tools found many more clone candidates than the other tools.

Figure 11 shows *precision* and *recall* of all the tools on all the target systems. The *recall* of FRISC is the best in all the tools for five out of the eight systems. FRISC could detect most clone references for the systems. For two of the remaining systems, ant and snns, FRISC placed the second position. In the worst case, cook, FRISC placed the third position.

In order to reveal what kinds of clone references detected by the comparison tools were not detected by FRISC, we extracted all of those clone references from all the target systems. Then, we randomly selected 100 instances from them, and we browsed their source code. As a result, they were categorized as follows (the numbers in parentheses mean number of clone pairs falling into the category):

- **A(71):** clone references including some gaps;
- **B(17):** clone references being less than 30 tokens;
- **C(11):** clone references locating in repeated instructions;
- **D(1):** clone references including unmatched modifiers.

In token-based detection, identical subsequences are detected as code clones. Gapped (type-3) code clones are not detected by naive token-based detection. Consequently, it is quite natural that 71 clone references falling into category A were not detected by FRISC. However, if we adopt some techniques like Roy et al. [21] or Juergens et al. [12] to detect such clone references, FRISC may detect some of those clone references.

TABLE III
TOOLS USED FOR COMPARISON

Developer	Tool	Detection method
Baker	Dup [4]	token-based
Baxter	CloneDR [6]	AST-based
Kamiya	CCFinder [13]	token-based
Merlo	CLAN [20]	metrics-based
Rieger	Duploc [8]	line-based
Krinke	duplix [15]	PDG-based
Roy	Nicad [21]	block-based

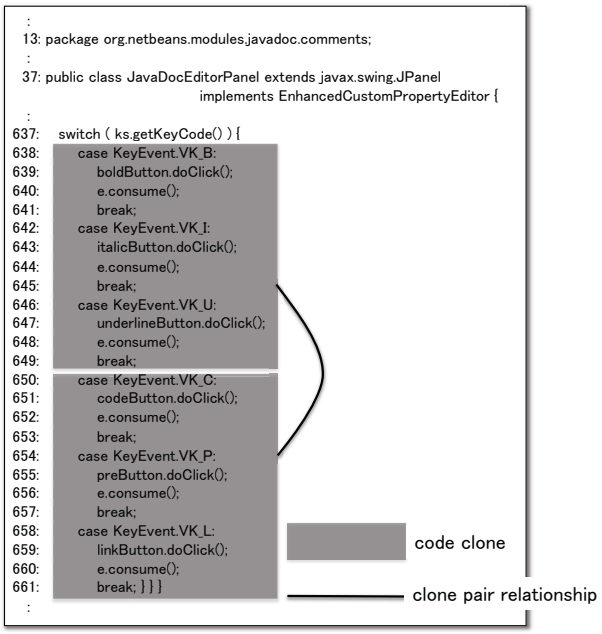
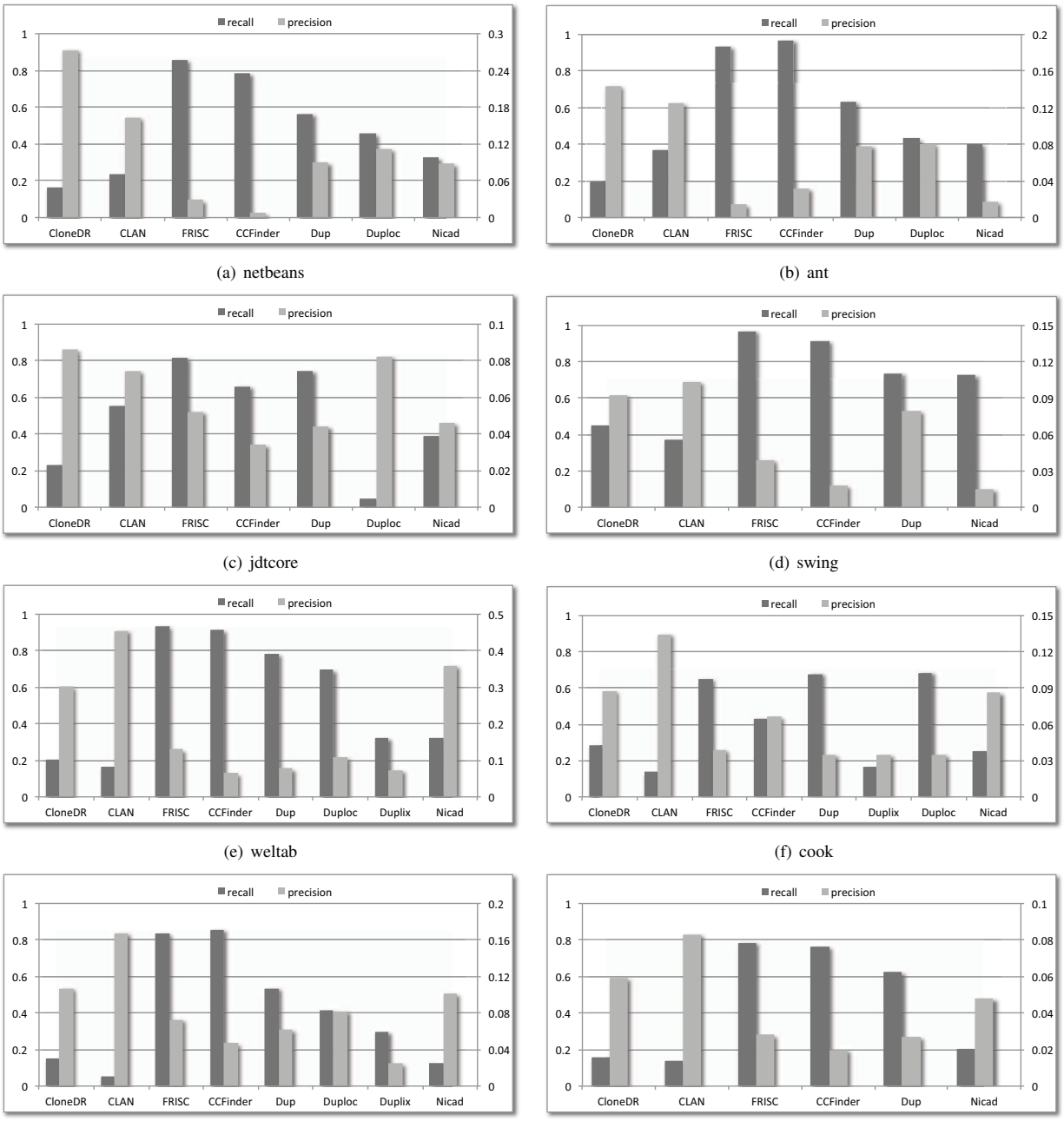


Fig. 12. A clone pair located in a repeated instructions

Clone references falling into category B are smaller than 30 tokens. In the Bellon's experiment, the minimum threshold of clone references is six lines, which is not a token-based threshold but a line-based one. However, FRISC takes a token-based threshold. In this experiment, FRISC took 30 tokens as the minimum clone length. Consequently, some clone references were not detected by FRISC.

Figure 12 shows a clone reference falling into C category. There are six case entries in this switch-statement. The former three entries form a code clone of the latter three entries. The proposed method folds the six case entries into a single entry, so that no clone pair is detected. However, it is possible to detect such a clone pair with the proposed method. If the sum of weights of a folded sequence is more than twice of the



(a) netbeans (b) ant

(c) jdtcore (d) swing

(e) weltab (f) cook

(g) snns (h) postgresql

Fig. 11. Precision and recall of all the detection tools for all the target software systems

minimum token length, there is a clone pair in it.

A clone reference was not detected because there is an unmatched modifier in it (category D): a code clone has "`final`" modifier in a method declaration; the correspondent does not have. Currently, FRISC does not normalize modifiers. However, it is not difficult to remove modifiers so as to detect such a clone reference.

The result shows that *precision* of the proposed method is not so high as the other token-based tools. However, we must notice that clone references used in this experiment is not all the real code clones in the systems. A low *precision* using the clone references does not directly mean that its detection result includes many false positives.

We answer RQ2 as follows: *the proposed method could detect more clone references than any of the other detection tools used for comparison in most cases. However, it detects many clone candidates as well as other token-based detection tools. Therefore, it may detects many false positives.*

VIII. Related Work

Koschke proposed using decision tree for filtering out false positives [14]. In his experiment, the learned decision tree had the following two metrics as its conditions.

- Parameter Similarity: rate how much both the code fragments in a clone pair use the same variables and literals.
- NR: rate of tokens not included in any repetition in a code fragment.

In order to build a decision tree, we must prepare training data. However, in his experiment, its accuracy was very high. The categorization error is below 0.1%.

Higo et al. proposed a metric *RNR* for filtering out false positives from a detection result [10]. Intuitively, *RNR* is an average of Koschke's *NR* value of every code clone in a clone class. In their experiment, 0.5 was an appropriate value as the threshold of *RNR*. By using the value, 2/3 of false positives were filtered out.

The biggest difference between the proposed method and their methods should be that: the proposed method transforms source code to a special form not to detect false positives; their methods filter out false positives after they are detected. In other words, the proposed method optimizes source code for code clone detection, so that there are code clones newly detected by the optimization. In our experiment, recall with the folding operation was actually higher than recall without it. On the other hand, their methods do not improve *recall* of detection results.

IX. Conclusion

In this paper, we proposed a new token-based code clone detection method. The proposed method folds repeated instructions for preventing false positives from being detected. The proposed method was developed as an actual tool, FRISC. We applied FRISC to eight open source software systems, and we confirmed the followings:

- the folding operation reduces false positives;
- there are some real code clones newly detected by the folding operation;
- detection time with the folding operation is hardly different from detection time without it;
- FRISC detects more real code clones than any other comparison tools used in Bellon's benchmark in most cases; and still,
- FRISC detects more false positives the other tools.

In the future, we are going to investigate where tools detect false positives. We expect that most false positives are detected from limited kinds of code patterns. If we can ignore code clones detected from those code patterns, false positives are drastically decreased.

Acknowledgment

This study has been supported in part by Grants-in-Aid for Scientific Research (A) (21240002), Grant-in-Aid for Exploratory Research (23650014), and Grand-in-Aid for Young Scientists (A) (24680002) from the Japan Society for the Promotion of Science.

References

[1] Clone Detection Literature. http://www.cis.uab.edu/tairasr/clones/literature/.
[2] Detection of Software Clones. http://bauhaus-stuttgart.de/clones/.
[3] Simian. http://www.harukizaemon.com/simian/.
[4] B. S. Baker. Parameterized Duplication in Strings: Algorithms and an Application to Software Maintenance. *SIAM Journal on Computing*, 26(5):1343–1362, Oct. 1997.
[5] B. S. Baker. Finding Clones with Dup: Analysis of an Experiment. *IEEE Transactions on Software Engineering*, 33(9):608–621, Sep. 2007.
[6] I. Baxter, A. Yahin, L. Moura, M. Anna, and L. Bier. Clone Detection Using Abstract Syntax Trees. In *Proc. of International Conference on Software Maintenance 98*, pages 368–377, Mar. 1998.
[7] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and Evaluation of Clone Detection Tools. *IEEE Transactions on Software Engineering*, 31(10):804–818, Oct. 2007.
[8] S. Ducasse, M. Rieger, and S. Demeyer. A Language Independent Approach for Detecting Duplicated Code. In *Proc. of the International Conference on Software Maintenance 99*, pages 109–118, Aug. 1999.
[9] N. Göde and R. Koschke. Frequency and Risks of Changes to Clones. In *Proc. of the 33th International Conference on Software Engineering*, May 2011.
[10] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue. Method and Implementation for Investigating Code Clones in a Software System. *Information and Software Technology*, 49(9-10):985–998, Sep. 2007.
[11] K. Hotta, Y. Sano, Y. Higo, and S. Kusumoto. Is Duplicate Code More Frequently Modified than Non-duplicate Code in Software Evolution?: An Emprical Study on Open Source Software. In *Proc. of the 4th International Joint ERCIM/IWPSE Symposium on Software Evolution*, pages 73–82, Sep. 2010.
[12] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner. Do code clones matter? In *Proc. of the 30th International Conference on Software Engineering*, May 2009.
[13] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code. *IEEE Transactions on Software Engineering*, 28(7):654–670, July 2002.
[14] R. Koschke. Large-Scale Inter-System Clone Detection Using Suffix Trees. In *Proc. of the 16th European Conference on Software Maintenance and Reengineering*, pages 309–318, Mar. 2012.
[15] J. Krinke. Identifying Similar Code with Program Dependence Graphs. In *Proc. of the 8th Working Conference on Reverse Engineering*, pages 301–309, Oct. 2001.
[16] J. Krinke. Is Cloned Code more stable than Non-Cloned Code? In *Proc. of the 8th IEEE International Working Conference on Source Code Analysis and Manupulation*, pages 57–66, Oct. 2008.
[17] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code. *IEEE Transactions on Software Engineering*, 32(3):176–192, Mar. 2006.
[18] S. Livieri, Y. Higo, M. Matsushita, and K. Inoue. Very-Large Scale Code Clone Analysis and Visualization of Open Source Program Using Distributed CCFinder: D-CCFinder. In *Proc. of the 29th International Conference on Software Engineering*, pages 106–115, May 2007.
[19] A. Lozano, M. Wermelinger, and B. Nuseibeh. Evaluating the relation between changeability decay and the characteristics of clones and methods. In *Proc. of the 23rd International Conference on Automated Software Engineering*, pages 100–109, Sep. 2008.
[20] J. Mayrand, C. Leblanc, and E. Merlo. Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics. In *Proc. of the International Conference on Software Maintenance 96*, pages 244–253, Nov. 1996.
[21] C. K. Roy and J. R. Cordy. Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *Proc. of the 16th IEEE International Conference on Program Comprehension*, June 2008.
[22] C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 74(7):470–495, May 2009.
[23] W. Shang, B. Adams, and A. E. Hassan. An Experience Report on Scaling Tools for Mining Software Repositories Using MapReduce. In *Proc. of the 25th International Conference on Automated Software Engineering*, pages 275–284, Sep. 2010.