2017 IEEE International Conference on Software Maintenance and Evolution

Flattening Code for Metrics Measurement and Analysis

Yoshiki Higo and Shinji Kusumoto
Graduate School of Information Science and Technology, Osaka University,
1-5, Yamadaoka, Suita, Osaka, 565-0871, Japan
Email:{higo, kusumoto}@ist.osaka-u.ac.jp

***Abstract*—When we measure code metrics or analyze source code, code normalization is occasionally performed as a preprocessing. Code normalization means removing untargeted program elements, formatting source code, or transforming source code with specific rules. Code normalization makes measurement and analysis results more significant. Existing code normalization mainly targets program elements not influencing program behavior (e.g., code comments and blank lines) or program tokens (e.g., variable names and literals). In this paper, we propose a new code normalization technique targeting program structure. Our proposed technique transforms a complex program statement to simple ones. We call this transformation flattening. By flattening code, we can obtain source code including only simple program statements. As applications of the code flattening, we report how it changes LOC metric and clone detection results.**

I. Introduction

Program source code is one of the main targets of research in the field of software engineering. There are various purposes of the research targeting source code.

Understanding: a purpose is helping to understand source code. Quantifying size, complexity, and characteristics of source code is a way to objectively grasp its status [1]. Dependencies among modules such as method invocation and class inheritance are worth being visualized for understanding [2], [3].

Reuse: on research for effective or efficient software reuse, source code is a main target. Identifying frequently-reused software components [4] and suggesting instructions such as method invocations for half-written code [5] are active research topics.

Suggestion: pointing out latent issues in source code or suggesting corrective strategies for them also have been studied for many years. Various methodologies have been proposed for identifying fault-prone modules [6], [7] and suggesting refactorings [8].

Metrics measurement and code clone (in short, *clone*) detection are fundamental techniques widely used in research on program source code. LOC, cyclomatic complexity, and C&K metrics suite [9] are well-known code metrics. There are many techniques to detect clones on different granularities such as file level, method level, and code fragment level [10].

In metrics measurement and clone detection, code normalization is occasionally performed to obtain more significant result. Herein, code normalization means transforming source code with specific rules. For example, in measuring LOC, removing code comments and blank lines and formatting source code are typical code normalization. In detecting clones, variables and literals are replaced with special tokens to absorb their differences. As just described, the targets of existing code normalization are peripheral program elements not influencing program behavior and program tokens.

In this paper, we propose a new code normalization technique, which is targeting program structure. The proposed technique transforms complex program statements to simple ones. In other words, source code including both complex and simple statements is transformed to the one including only simple statements. We call this transformation *flattening*. By flattening source code, the amount of functionality in every code line becomes homogenized. Thus, LOC values become more significant. The program structure is also uniformed by code flattening. Thus, more code fragments having identical/similar behavior are detected as clones.

In this paper, we also report how LOC value and clone detection results are changed by code flattening. We confirmed that there were many source files whose LOC values got increased greatly and much more clones were detected in flattened source files than original ones.

The main contributions of this paper are as follows:

- proposing a new code normalization technqiue,
- investigating how LOC and clone detection results are changed by the proposed technqiue, and
- investigating whether the proposed technique raises correlations between LOC and fault-prone modules or not.

II. Related Work

Code normalization is an often-used technqiue in clone detection [11], [12], [13], [14], [15], [16], [17]. In clone detection, variables are replaced with special tokens to ignore their differences. There are two typical ways for variable normalization; the first one is replacing all variables with the same special token; the other is replacing each variable with a different special token. The latter one is better from the viewpoint of detecting less false positives.

One way to support source code analysis is transforming source code to another form. Cordy proposed TXL, which is a framework to support source code analysis and source transformation tasks [18]. Maletic proposed srcML, which is a framework to present source code in the XML form [19]. By presenting source code in the XML form, existing various tools for XML documents can be applied.

Techniques to remove *goto* statements [20], [21] are a kind of structure normalization. Those techniques leverage control flow graphs. The structure of target source code is transformed so as not to include *goto* statements without changing the structure of the control flow graphs.

978-1-5386-0992-7/17 $31.00 © 2017 IEEE
DOI 10.1109/ICSME.2017.65

```
…
256  public int read() throws IOException {
257    if (needAddSeparator) {
258      if (lastPos >= eolString.length()) {
259        lastPos = 0;
260        needAddSeparator = false;
261      } else {
262        return eolString.charAt(lastPos++);
263      }
264    }
265    while (getReader() != null) {
266      int ch = getReader().read();
267      if (ch == -1) {
268        nextReader();
269        if (isFixLastLine() && isMissingEndOfLine()) {
270          needAddSeparator = true;
271          lastPos = 1;
272          return eolString.charAt(0);
273        }
274      } else {
275          addLastChar((char) ch);
276          return ch;
277      }
278    }
279    return -1;
280  }
…
```

Fig. 1. Source code including complex statements

III. Reserch Motivation

The source code in Figure 1 includes both simple statements (e.g., the 259th line) and complex statements (e.g., the 269th line). The authors think, the fact that complex statements are mixed with simple ones have negative influences on metrics measurement and source code analysis.

As an example of metrics measurement, we consider measuring LOC. LOC is a metric to present a size of a given source file. In LOC measurement, blank lines and/or code comments in the files are ignored to obtain more significant values as a size metric.

In using the LOC metric for guessing the amount of functionality in the source file or for identifying fault-prone modules, the fact that complex statements are mixed with simple ones can be a factor that degrades its performance. For example, it is not easy to know whether a given 50-line code fragment consisting of complex statements includes more functionality than another given 100-line code fragment including only simple statements or not. Moreover, if we use the metric values (50 and 100), techniques of fault-prone module identification will regard that the l00-line code fragment has a greater probability of including faults than the 50-line one[1].

As an example of source code analysis, we consider detecting clones from source code. Detected clones are used to identify code fragments including the same faults or to be targets of code unification refactoring. Like this, clone detection techniques are used to detect code fragments having identical/similar behavior. Token-based and AST-based clone detection techniques are more often used than other techniques

[1]Using the LOC metric for guessing the amount of functionality may be an old-fasioned example. However, the LOC metric is still widely used. For example, in mining software repositories. the code churn metric is well used. Code churn is calculated from the LOC of changed code.

```
…
284        needAddSeparator = false;
285      } else {
286        int $24 = lastPos++;
287        char $25 = eolString.charAt($24);
288        return $25;
289      }
…
295      if (ch == -1) {
296        nextReader();
297        boolean $28 = isFixLastLine();
298        boolean $29 = isMissingEndOfLine();
299        if ($28 && $29) {
300          needAddSeparator = true;
301          lastPos = 1;
…
```

Fig. 2. Source code flattened by the proposed technique

such as semantic clone detection or graph-based clone detection because of their scalability. In token-based techniques, given source code is transformed to a sequence of tokens and identical subsequences are detected as clones. In AST-based techniques, given source code is parsed to generate ASTs. Sub-trees in the ASTs having identical/similar structures are detected as clones. Assume that there are two same-behavior code fragments: one includes some complex statements while the other consists of only simple statements. In such a case, existing clone detection techniques cannot detect the two code fragment as clones.

In this research, we are trying to transform complex structures in source code to simple ones without changing its external behavior. The authors call such a transformation *flattening* in this research. By flattening source code, complex statements are gone. Consequently, the amount of functionality in each line gets homogenized, so that the LOC becomes more significant values. Besides, same-behavior code fragments get more similar structures by flattening, which makes it easier to detect them as clones with clone detection techniques. In other words, code flattening enables a part of semantic clones to be detected by syntax-based clone detection techniques.

Figure 2 shows source code flattened by our proposed technique. Its original version is shown in Figure 1. For example, the 262nd line in the original code was transformed to the 286–288th lines. We can see that the postfix expression and the method invocation one are extracted from the complex return-statement.

IV. Proposed Technique: Code Flattening

The proposed technique depends on abstract syntax tree (in short, AST) implementation. Herein, we explain the proposed technique with Java Development Tools (in short, JDT). JDT is a widely-used Java source code analysis tool. It takes a set of source files and builds ASTs. The tool has a function to resolve name bindings.

The proposed technique works on ASTs built by JDT. For each of the target sub-trees in given ASTs, the proposed technique checks whether it is enough complex to be flattened or not. If the sub-tree is complex, its portion is removed from the original position and a new sub-tree including the portion is generated in a different position. The new sub-tree is a variable

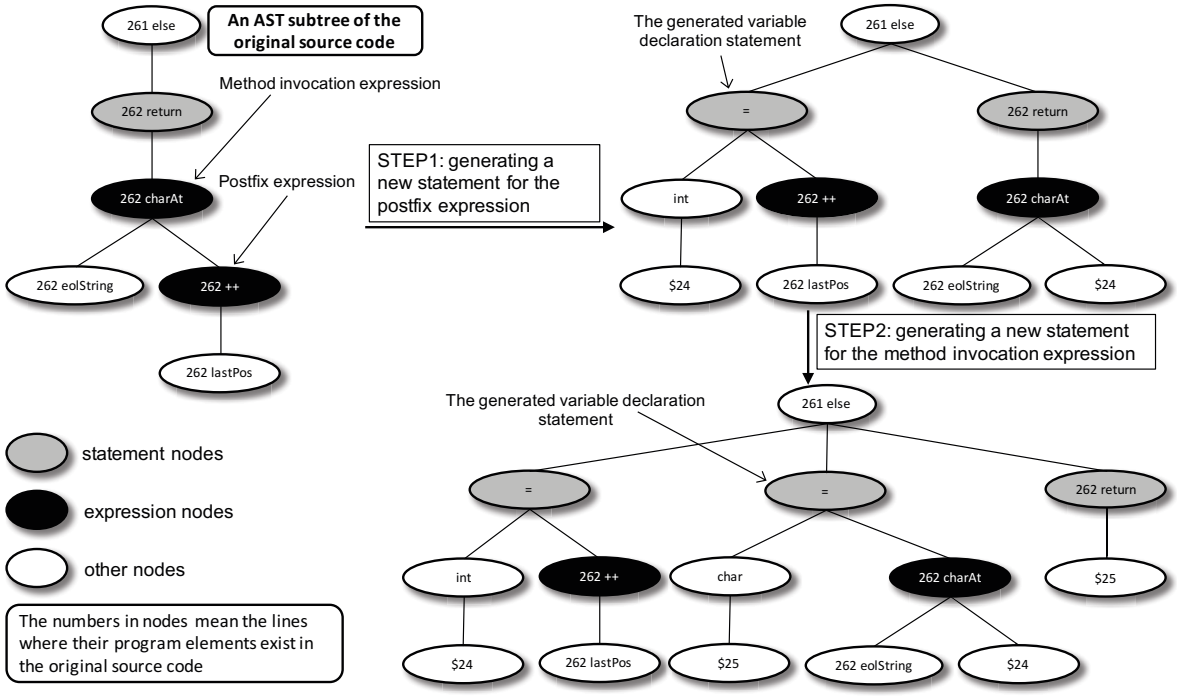

Fig. 3. AST transformation of the proposed technique

declaration statement (in short, VDS) and its right-hand code is the removed sub-tree. A reference to the new variable is added to the original position where the portion was removed. The newly introduced variable has a type that is the same as its right-hand code. This transformation strategy realizes that the source code can be compiled and executed even after it has been flattened.

The following AST nodes are targets to check whether they are complex or not:

- program statements,
- conditional predicates for branchs and loops, and
- initializers and updaters of for-statements.

If the above targets include multiple expressions, such expressions are extracted as new VDSs. Table I shows a list of target expressions. The following expressions are not included in the extraction targets because they are not complex:

- variable reference expression,
- literal expression, and
- *null* expression.

Figure 3 shows how code is flattened by the proposed technique. The upper-left tree is the AST sub-tree for the 261st and 262nd lines of Figure 1. Sub-trees are manupulated with depth-first search and postorder traversal. In this example, firstly the node of the postfix expression is regarded that it increases the complexity of the return statement. Thus, a new VDS for the postfix expression is added and the sub-tree of the postfix expression is replaced with a reference to the newly added variable $24. Then, the node of the method invocation expression is also regarded that it increases the complexity of the return-statement. This sub-tree is manipulated in the same way as the postfix expression.

Figure 4 is another example of flattened code. The left side is its original code. The conditional predicate of the if-statement is very complex, so that five VDSs are created to reduce the complexity of the conditional predicate. Like this, the proposed technique introduces more VDSs as target sub-trees are more complex.

In Figure 4, the conditional predicate of the for-statement is also a target of flattening. In a case of loop condition, a VDS is added to just before the loop and a substitution statement is added to the end inside the loop[2]. Both the VDS and the substitution statement are required to keep program behavior after code flattening.

TABLE I
EXTRACTION TARGET JDT NODES

ArrayAccess	ArrayCreation
ArrayInitializer	CastExpression
ClassInstanceCreation	ConstructorInvocation
ExpressionMethodReference	InfixExpresssion
InstanceofExpression	MethodInvocation
PostfixExpression	PrefixExpression
SuperConstructorInvocation	SuperMethodInvocation

V. EXPERIMENT

We conducted an experiment with our tool[3]. In this experiment, we investigate how LOC and clone detection results

[2]In the strict sense, there is no definition of substitution statement in Java specification. Herein, a substitution statement means an expression statement including a substitution operator.

[3]https://github.com/YoshikiHigo/JCodeFlattener/

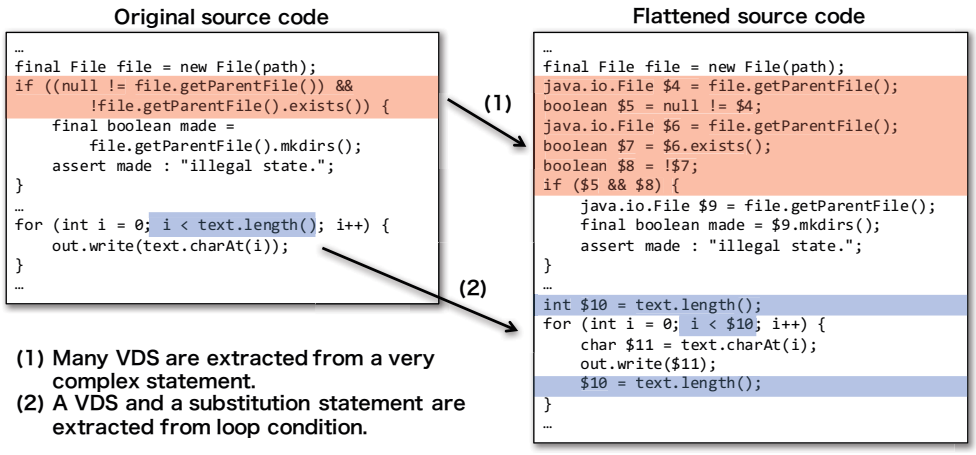

Fig. 4. An example of code frattening with the implemented tool

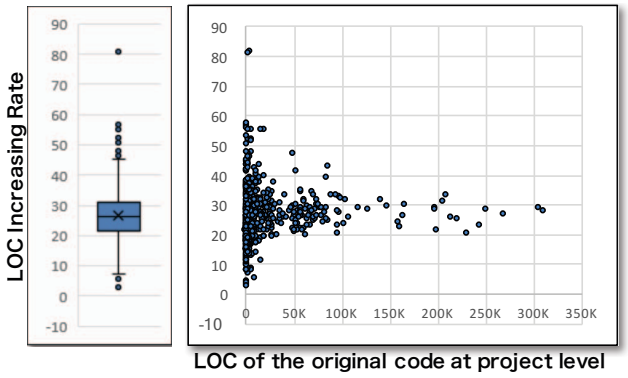

Fig. 5. Results of LOC Increasing Rate calculation on project level

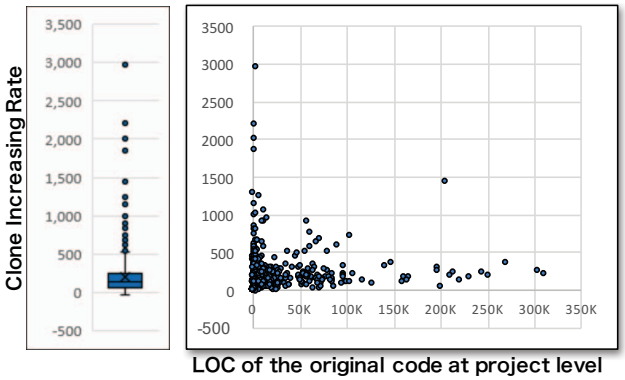

Fig. 6. Results of Clone Increasing Rate calculation on project level

are affected by code flattening. We leveraged the dataset that we created and published in our previous research [22]. The dataset is open to the public in our web site[4]. The dataset includes two packages. Each package includes source code of 84 or 500 projects. Table II includes more detailed numerical information. The dataset includes neither test cases nor automatically generated code. They were removed manually.

Figure 5 shows how LOC at project level was changed by code flattening. The LOC Increasing Rate is calculated with the following formula. LOC_{ori} and LOC_{fla} represent LOC of the original code and flattened code, respectively.

$$IncreasingRate_{LOC} = 100 * \frac{LOC_{fla} - LOC_{ori}}{LOC_{ori}}$$

[4]http://sdl.ist.osaka-u.ac.jp/~higo/fse2014/

TABLE II
TARGET PROJECTS IN EXPERIMENT A

Package	APACHE	UCI
# projects	84	500
# files	66,724	60,548
Total LOC	11,545,556	10,073,635

As shown in this figure, the dispersion of the LOC Increasing Rate becomes larger for smaller projects. If original code size is 25K LOC or less, the code size was increased more than one-and-a-half times for several projects. On the other hand, for 100K LOC or larger projects, the LOC Increasing Rate falls roughly within the range of 20% and 35%. The results imply that we should not use LOC as an indicator of our decision makings, especially for small projects.

Figure 6 shows how the number of detected clones is affected by code flattening. The Clone Increasing Rate is calculated with the following formula. $CLONE_{ori}$ and $CLONE_{fla}$ represent the number of clones detected from the original code and flattened code, respectively.

$$IncreasingRate_{CLONE} = 100 * \frac{CLONE_{fla} - CLONE_{ori}}{CLONE_{ori}}$$

As shown in this figure, the Clone Increasing Rate varies more widely in smaller projects. This is the same tendency as LOC metric. For 25K or smaller projects, 10 times or more clones were detected in some cases. On the other hand, for most of 100K or larger projects, the Clone Increasing Rate is less than five. The results show that code flattening allows clone detection tools to detect much more clones.

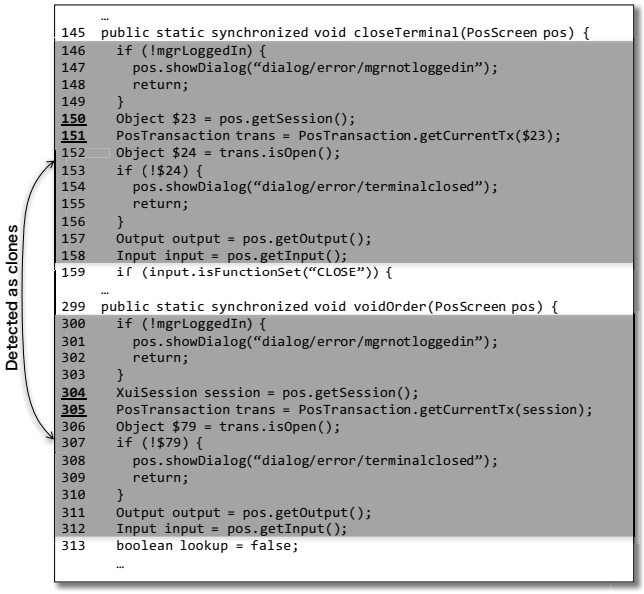

Fig. 7. Newly detected clones by flattening code

Figure 7 shows a pair of clones detected from the flattened code but not detected from the original code. The 150th and 151st lines were a single statement in the original code. On the other hand, the 304th and 305th lines are in this form in the original code. The differences prevent the clones from being detected by the clone detection tool. By flattening the code, the differences were removed and the code fragments were detected as clones, especially for small projects.

Code flattening has a side effect in clone detection. More trivial clones are detected from flattened code because code flattening increases the number of tokens, the number of statements, and the number of AST nodes. In this experiment, we do not check ratio of trivial clones in the detection results.

VI. Conclusion

In this paper, we proposed a code flattening technique, which dissolves a complex program statement to multiple simple ones. By flattening code, the amount of functionality in every line gets homogenized. Thus, values of code metrics such as LOC become more significant. Besides, more clones are detected from flattened source code.

The proposed technique has been implemented and it is open to the public in our web site[5]. We applied the tool to 584 open source software in total and found that there were many source files whose LOC became much greater by code flattening. We found that the code flattening allowed much more clones to be detected. Furthermore, we found that the LOC of flattened code was more useful than the one of original code in the context of fault-prone module identification.

The next step of our research is developing methodologies of refactorings for flattened code because small-size clones are ocassionaly generated by code flattening. By merging such small-size clones, source code will get more suited for metrics measurement and clone detection.

We are also going to compare clone detection results on flattened code with semantic clone detection results and program-dependence-graph-based clone detection results. We consider code flattening enables syntax-based detecton techniques to detect a part of semantic clones and graph-based clones.

[5]https://github.com/YoshikiHigo/JCodeFlattener/

Acknowledgment

This work was supported by MEXT/JSPS KAKENHI 25220003 and 17H01725.

References

[1] N. E. Fenton and S. L. Pfleeger, *Software Metrics: A Rigorous and Practical Approach, Third Edition*, 3rd ed. Boston, MA, USA: PWS Publishing Co., 2014.

[2] R. Koschke, "Software Visualization in Software Maintenance, Reverse Engineering, and Re-engineering: A Research Survey," *Journal of Software Maintenance*, vol. 15, no. 2, pp. 87–109, 2003.

[3] M.-A. D. Storey, D. Čubranić, and D. M. German, "On the Use of Visualization to Support Awareness of Human Activities in Software Development: A Survey and a Framework," in *Proc. of SOFTVIS*, 2005, pp. 193–202.

[4] M. Acharya, T. Xie, J. Pei, and J. Xu, "Mining API Patterns As Partial Orders from Source Code: From Usage Scenarios to Specifications," in *Proc. of ESEC/FSE*, 2007, pp. 25–34.

[5] V. Raychev, M. Vechev, and E. Yahav, "Code Completion with Statistical Language Models," in *Proc. of PLDI*, 2014, pp. 419–428.

[6] C. Catal and B. Diri, "A Systematic Review of Software Fault Prediction Studies," *Expert Systems with Applications*, vol. 36, no. 4, pp. 7346–7354, 2009.

[7] V. B. Livshits and M. S. Lam, "Finding Security Vulnerabilities in Java Applications with Static Analysis," in *Proc. of USENIX*, 2005, pp. 18–18.

[8] T. Mens and T. Tourwé, "A Survey of Software Refactoring," *IEEE Transactions on Software Engineering*, vol. 30, no. 2, 2004.

[9] S. R. Chidamber and C. F. Kemerer, "A Metrics Suite for Object Oriented Design," *IEEE TSE*, vol. 20, no. 6, pp. 476–493, 1994.

[10] D. Rattan, R. Bhatia, and M. Singh, "Software Clone Detection: A Systematic Review," *Information and Software Technology*, vol. 55, no. 7, pp. 1165–1199, 2013.

[11] H. A. Basit, S. J. Puglisi, W. F. Smyth, A. Turpin, and S. Jarzabek, "Efficient Token Based Clone Detection with Flexible Tokenization," in *Proc. of ESEC/FSE*, 2007, pp. 513–516.

[12] S. Ducasse, O. Nierstrasz, and M. Rieger, "On the effectiveness of clone detection by string matching: Research articles," *Journal of Software Maintenance and Evolution*, vol. 18, no. 1, pp. 37–58, 2006.

[13] B. Hummel, E. Juergens, L. Heinemann, and M. Conradt, "ndex-based Code Clone Detection: Incremental, Distributed, Scalable," in *Proc. of ICSM*, 2010, pp. 1–9.

[14] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner, "Do code clones matter?" in *Proc. of ICSE*, 2009, pp. 485–495.

[15] E. Juergens, F. Deissenboeck, and B. Hummel, "CloneDetective – A Workbench for Clone Detection Research," in *Proc. of ICSE*, 2009, pp. 603–606.

[16] H. Murakami, K. Hotta, Y. Higo, H. Igaki, and S. Kusumoto, "Folding Repeated Instructions for Improving Token-Based Code Clone Detection," in *Proc. of SCAM*, 2012, pp. 64–73.

[17] 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 ICPC*, 2008, pp. 172–181.

[18] J. R. Cordy, "The TXL Source Transformation Language," *Science of Computer Programming*, vol. 61, no. 3, pp. 190–210, 2006.

[19] J. I. Maletic, M. L. Collard, and A. Marcus, "Source Code Files As Structured Documents," in *Proc. of IWPC*, 2002, pp. 289–292.

[20] Z. Ammarguellat, "A Control-Flow Normalization Algorithm and Its Complexity," *IEEE TSE*, vol. 18, no. 3, pp. 237–251, 1992.

[21] F. Zhang and E. H. D'Hollander, "Using Hammock Graphs to Structure Programs," *IEEE TSE*, vol. 30, no. 4, pp. 231–245, 2004.

[22] Y. Higo and S. Kusumoto, "How Should We Measure Functional Sameness from Program Source Code? An Exploratory Study on Java Methods," in *Proc. of FSE*, 2014, pp. 294–305.