Blanker: A Refactor-Oriented Cloned Source Code Normalizer

Davide Pizzolotto
Osaka University, Japan
davidepi@ist.osaka-u.ac.jp

Katsuro Inoue
Osaka University, Japan
inoue@ist.osaka-u.ac.jp

***Abstract*—Refactoring is widely practiced by developers and has become a key factor in order to increase the maintainability of software. However, code clones pose a threat in any refactor process due to the fact that a developer should edit identical portions of code more than once. Despite the numerous researches in this topic, most of the results are focused on discovering type-3 and type-4 clones, that require an higher effort to be refactored and removed.**

In this paper we present our tool, Blanker, that searches and unifies equivalent statements available in the language before feeding the source to an existing code clone detector limited to type-2 clones. This step acts as a normalization step and produces refactorable results without the error introduced by potentially unrelated added statements (like in type-3 clones), that would be unsuitable for refactoring purposes, and with added flexibility compared to checking for identical code portions (like in type-2 clones).

We used NiCad to detect clones before and after our normalization step and found up to 10% more type-2 clones after our normalization, all of them being refactor candidates.

***Index Terms*—Semantic Similarity Detection, Code Normalization, Source Transformation, Semantically Normalized Clones**

I. Introduction

Refactoring is the process of changing the code structure without changing its behavior. One of the most widely performed refactoring activity is the Extract Method, that simplifies methods by moving existing portions of code into a new method that can be reused [1]. This allows a developer to resolve a detected code clone by extracting it into a method and reusing it different times. For this reason, code clone detectors invest an important role in this scenario, given their ability to quickly identify a code portion that may be a refactoring candidate.

However, not always the output of a code clone detector can be used. In fact, a code clone detector usually categorize clones into four categories: *type-1* which are portions of code completely identical, *type-2* which are portions identical except for the variable naming, *type-3* almost identical except for a few statements and *type-4* which are different portions of code but with the same behavior [2]. Considering these categories, only the *type-1* and *type-2* can be easily and, almost automatically, refactored. Instead, *type-3* clones need to be manually checked and require an effort by the programmer that needs to check if the extra statements can be safely removed or not. This is true also for *type-4* clones that require even more work by the programmer, in addition to being extremely difficult to detect [2]. Our tool aims at normalizing some features provided by the programming language without changing the semantic, so a clone that would normally be categorized as a *type-3* or *type-4* can be categorized as *type-2* and thus easily refactored. As an example, Figure 1 shows a snippet taken from the class `XYBlockRenderer.java`[1] of JFreeChart: we can note the missing `else` keyword on the transformed version, that has been removed for consistency while retaining the same semantic.

This study follows the work of Ragkhitwetsagul and Krinke that showed how compiling and decompiling a source file can greatly increase the amount of detected clones, given that the compilation acts as a sort of "normalization step" [3]. However, the compilation/decompilation routine is not always applicable and can introduce errors in languages like C or could lead to false positives also in the Java language, as the aforementioned authors already discovered (i.e. by replicating some methods of the parent class of an inner class). To solve the problem we propose Blanker, that aims at replacing this compilation/decompilation routine with a plain code transformation. We replicated the setup and analyzed the various discoveries of Ragkhitwetsagul et al., then we manually implemented the transformations performed by the compilation step. Despite working with Java, in order to replicate the original study, we also ensured that our tool can perform the transformation on C source files.

Throughout the paper we will refer to easy-to-refactor clones. With this name we intend type-2 clones that can be solved by a simple Extract Method process, instead of requiring an in-depth analysis of the extra statements like the type-3 clones.

To evaluate Blanker we replicated their exact same scenario by using the same three open source projects, namely *JUnit*, *JFreeChart* and *Apache Tomcat*, and *NiCad* [4] as code clone detector. We demonstrated that despite our tool missing a small amount of clones, compared to the compilation/decompilation approach, it has virtually no false positives and provides the added flexibility of being applicable to languages where the decompilation step may introduce errors.

In the reimained of the paper Section II shows our approach at implementing Blanker along with the engineering aspects

[1] `org/jfree/chart/renderer/xy/XYBlockRenderer.java`

```java
if (r == null) {
    return null;
}
else {
    return new Range(r.getLowerBound() + this.yOffset,
            r.getUpperBound() + this.blockHeight + this.yOffset);
}
```

(a) Original Code

```java
if (r == null) {
    return null;
}

    return new Range(r.getLowerBound() + this.yOffset,
            r.getUpperBound() + this.blockHeight + this.yOffset);
```

(b) Transformed Code

Fig. 1. Example of transformation taken from JFreeChart

and Section III describes the evaluation of the tool. Section IV describes related works and Section V closes the paper.

II. Approach

Blanker follows a linear workflow depicted in Figure 2 and composed by the following phases:

Parse: The original source file is parsed and the position of every semantic structure is recorded.

Categorize: The parsing result is analyzed in order to find possible refactoring candidates.

Rewrite The possible normalizations highlighted in the previous step are applied to a new file.

Detect: The code clone detector is applied to the rewritten file.

Remap: Possible discrepancies between the transformed file and the original file are addressed in this step.

The following subsections explains every component in detail. However, in order to better understand the following phases, it is worth precising that we built Blanker upon the results of Ragkhitwetsagul et al. [3]. In their previous results they showed that most of the normalizations performed by the compilation/decompilation process are applied to if-else statements, so, naturally, throughout our entire analysis, we focused greatly on those structures.

A. Parse

In order to transform a source code file, the first step requires identifying the semantic structures composing it. We built our tool aiming at negligible speed, so a single-pass token parser using *flex*[2] as lexer and *bison*[3] as parser was built. The great challenge imposed by this phase was the creation of a grammar expressive enough to recognize the required structures, namely `if` and `else` blocks, without having to implement the entire java grammar. In order to solve this problem we chose each semicolon as a statement representation and recorded the position of `if`, `else` and `return` keywords only. Additionally we used curly braces to represent list of statements and let the lexer consume the condition following every `if` keyword. This grammar assumes the input file being a valid java file, a reasonable assumption for our tool, and works fine without modifications also for the C language, while avoiding having to deal with things such as parentheses, assignments and operators.

[2]https://github.com/westes/flex

[3]https://www.gnu.org/software/bison/

B. Categorize

This phase is used to analyze the parsed data and discover actual structures that could be refactored. After analyzing the results of Ragkhitwetsagul et al. and replicating their experiments we determined that the most prominent normalizations performed by the compiler/decompiler combo were:

1) removal of `else` keyword after an `if` block terminating with a `return`. An example of this can be seen in Figure 3 where the else keyword can be omitted and the same logic is kept. It is worth noting that this particular check is also part of LLVM's coding standard recommendations under the name of *readability-else-after-return*
2) Returning an equality or inequality between two variables is transformed into an if block with the equality as the condition and return true or return false as body
3) Returning a conjunctive boolean formula is splitted into an "explicit short circuit evaluation". In order words, every variable is checked by itself in an if block and if the variable does not hold, false is returned. An example of the transformed code for `return a && b;` is shown in Figure 4.
4) `final` keywords lacking consistency. Sometimes a variable could be final but this keyword is not present, but it is present in a cloned snippet somewhere else.

Another common normalization that, however, we did not address was declaration and assignment of a variable in a single line or in different lines, possibly interleaved by other statements.

C. Rewrite

This phase is the actual transformation of the structure described in Section II-B. A key requirement of this phase was keeping as much as possible the file similar to the original one, the reason being explained in Section II-D. In order to accomplish this, we exploited several facts: firstly the whitespaces and newlines being meaningless in both C and Java. Moreover, these are ignored also by the code clone detector of our choice. Given this we could easily transform categories 1) and 4) described in Section II-B just by patching the redundant parts with whitespaces. Additionally we exploited the fact that also newlines are meaningless in both C and Java, allowing us to write multiple statements in one line. Categories 2) and 3) in Section II-B requires replacing a single statement with multiple ones, and this language feature allows us to maintain

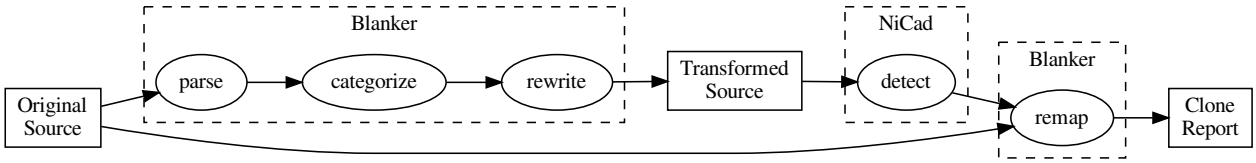

Fig. 2. Overall structure of Blanker

```
if (r == null) {
    return null;
} else {
    return new Range(...);
}
```

Fig. 3. Categorization 1). In this case the else keyword is redundant

```
if (!a) {
    return false;
if (!b) {
    return false;
}
return true;
```

Fig. 4. Categorization 3). This could be written as `return a && b;`

a one to one mapping between the original cloned lines and the transformed ones.

D. Detect and Remap

At this point, the Detect phase applies the code clone detector to our transformed files. In our implementation, however, the normalized files are not overwritten, so the report generated by the code clone detector will present wrong paths. This is a major problem nonetheless, given that the entire normalization process should be invisible to the user, and thus the remap phase is used to map every reference of the normalized files in the code clone detector report back to the original files. This also gives an indication why in Section II-C was important to keep the file as similar as possible to the original one: replacing a statement with one spanning more lines or less lines means having to remap also every line reported by the code clone detector.

III. Evaluation

Being this study based on the results of the paper Ragkhitwetsagul et al., we replicated their evaluation setup. The same three open source project were considered for this evaluation, namely *JUnit v4.13*, *JFreeChart v1.5.0* and *Apache Tomcat v9.0*. These projects are listed ordered by size, spanning from 9.7k LoC of JUnit to 241.9k LoC of Tomcat.

The code clone detector used for this evaluation was NiCad v5.2. Our tool can be used with any code clone detector, but we decided to use NiCad in order to compare with the previous study. Studying the effect on other code clone detectors, especially the ones not using a token-based approach, will be considered as future work. Unlike the previous study, however, we did not analyze type-3 clones: our goal is providing an easy-to-refactor clone, thus limiting the evaluation to type-1 and type-2. The type-3 clones with the noise generated by the additional statements are considered as out-of-scope for our purpose. The configurations used are thus the default of NiCad named *type1* and *type2c*, the latter being type 2 clones with consistent naming.

The Research Questions we wanted to answer are the following, similar to the ones of the original study:

- **RQ1**$_{agreement}$: How many clone pairs are reported by both approaches? How many are exclusive to the plain file and to the normalized file?
- **RQ2**$_{accuracy}$: How is the detection performance of our tool compared to the compilation/decompilation approach presented in the study of Ragkhitwetsagul et al.?

Despite not conducting a real performance study, we also want to highlight that the processing time impact our tool is negligible: in the three project we measured this time to be, on average, half a millisecond per file, with every file being independent of the others thus enabling multithread processing. Also for big projects this translates in a normalization step order of magnitude faster than the clone detection process which usually requires several seconds.

A. Agreement

In order to answer RQ1 we ran NiCad with both *type1* and *type2c* configurations on the testing repositories, firstly without any normalization and then with normalizations applied. Table I depicts the results for *type2c*, despite, by nature, these vary greatly depending on the considered project and the coding style. *type1* clones are not reported given that the results are absolutely identical. This is expected, given that is really unlikely that an user writes some code semantically different and keeps the same variable naming. By analyzing the *type2c* results instead, we can notice that the normalized version reports more clones. We manually analyzed the normalized clones reported and confirmed that they are a superset

TABLE I
COMPARISON OF THE AMOUNT OF DETECTED CLONES BY NICAD WITHOUT AND WITH OUR NORMALIZATION APPLIED

Project	Original clones	Normalized clones	Variation
JUnit	6	6	+0.0%
JFreeChart	373	397	+6.43%
Apache Tomcat	242	275	+13.64%

of the non-normalized version. Every clone pair reported is actually refactorable, however, despite this, we have to precise that the category 4) explained in Section II-B could in practice produce unrefactorable results, given that we simply removed every `final` keyword instead of performing a full constness propagation analysis to ensure semantic preservation. However, this problem was not highlighted in the three projects and requires further analyses. We can thus answer RQ1 as follows:

Processing the files with Blanker helps the code clone detector to find up to 10% more clones. In the case studies no false positives and no false negatives were detected compared to the original source detection.

B. Accuracy

In order to answer RQ2 we also ran the compilation/decompilation method and compared the original and normalized clones against that. Our normalized method provides no false positives, but, although not providing false negatives compared to the original source code, the compilation/decompilation provided some clone reports that both the original and our normalized version failed to detect. In the case of JFreeChart these comprise assignment and declaration of variables interleaved by a different amounts of statements, swapped if-else branches and a loop converted from while to for (done by the decompiler normalization step). Although each of these appearing only once, they are a clear sign that in order to discover even more clones a flow analysis may be required. On the other hand, the compilation/decompilation suffers from false positives, in particular related to the duplication of inner class methods that happens during compile time and not in the code written by the user. We can thus answer RQ2 as follows:

Our tool provides virtually no false positives at the cost of missing some clones. The compilation/decompilation approach suffers both false positives and negatives but provides a better spectrum of results if compared in conjunction with the original source code

IV. RELATED WORKS

Clone detection is an active area in Software Engineering and several approaches have been proposed, ranging from token-based techniques such as NiCad [4], SourcererCC [5] and CCFinder [6] to structure analysis tools such as Deckard [7].

Multiple studies have been conducted on the compiled version of a software: Selim et al. worked with an Intermediate Representation of Java (Jimple) by adapting a token based clone detector [8], Kononenko et al. used another adaptation to work at bytecode level [9], Davis and Godfrey analized the compiled assembly of a program using string matching [10].

Finally, Ragkhitwetsagul and Krinke used the compilation to achieve normalization and decompilation to avoid adapting existing clone detectors to work at a lower level [3].

V. CONCLUSION

In this paper we presented Blanker, a code normalization tool useful to change some structures in order to detect more easy-to-refactor clones. Blanker proved to be effective at presenting a correct superset of type-2 clones, including clones that resemble type-2 clones but are classified as type-3 due to some extra keywords. This can thus be used as a better report for refactorable type-2 clones.

Our tool in fact, provides a type-2 report with no drawbacks, false positives or false negatives compared to running a code clone detector on the original files. If a low false negative ratio is required, the compilation/decompilation approach could provide more results due to the flow analysis performed by the compiler, even though a much higher effort is required to actually intersect and analyze the results.

As future work we plan to conduct an extensive analysis on the various categorization performed, and implement extra ones in addition to flow analysis.

ACKNOWLEDGMENTS

This work was supported by JSPS KAKENHI Grant Number 18H04094.

REFERENCES

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

[2] 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*, vol. 74, no. 7, pp. 470–495, 2009.

[3] C. Ragkhitwetsagul and J. Krinke, "Using compilation/decompilation to enhance clone detection," in *2017 IEEE 11th International Workshop on Software Clones (IWSC)*. IEEE, 2017, pp. 1–7.

[4] C. K. Roy and J. R. Cordy, "Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization," in *2008 16th iEEE international conference on program comprehension*. IEEE, 2008, pp. 172–181.

[5] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "Sourcerercc: Scaling code clone detection to big-code," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 2016, pp. 1157–1168.

[6] 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*, vol. 28, no. 7, pp. 654–670, 2002.

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

[8] G. M. Selim, K. C. Foo, and Y. Zou, "Enhancing source-based clone detection using intermediate representation," in *2010 17th Working Conference on Reverse Engineering*. IEEE, 2010, pp. 227–236.

[9] O. Kononenko, C. Zhang, and M. W. Godfrey, "Compiling clones: What happens?" in *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 2014, pp. 481–485.

[10] I. J. Davis and M. W. Godfrey, "From whence it came: Detecting source code clones by analyzing assembler," in *2010 17th Working Conference on Reverse Engineering*. IEEE, 2010, pp. 242–246.