EasyMerge - A New Tool for Code Clones Refactoring

Shengying Pan
School of Computer Science
University of Waterloo
s5pan@uwaterloo.ca

Haocheng Qin
School of Computer Science
University of Waterloo
h7qin@uwaterloo.ca

Yahui Chen
School of Computer Science
University of Waterloo
y556chen@uwaterloo.ca

ABSTRACT

Code clones are common in medium to large scale software projects. Oftentimes, unnecessary clones cause troubles to code base maintenance and code reusability. Over past decades, many techniques and approaches have been proposed to detect code clones. However, how to refactor clones is still a very challenging topic to software engineers. Even text-wise identical code clones can be semantically different when they are referring variables and calling functions outside. And the problem is more complex when scopes and dependencies are involved. Furthermore, not all clones shall be refactored as they may be part of independent libraries. And refactoring is not only about removing duplicate code but also fixing all reference errors caused by such deletion thereafter. As a result, we need adaptive clone refactoring tools that can locate unnecessary clones and alert possible implications in the procedure to help software engineers be more efficient and make less mistakes in the refactoring process.

In this paper, we introduce EasyMerge, a new tool to refactor code clones. EasyMerge is built on top of AST-based anti-unification clone detection algorithm and refactors software projects written in Python. It adds intelligence to code clone refactoring by categorizing clones and generating refactoring recommendations based on evaluation of external variable references and function calls. It copes with features of Python language and can deal with clone fragments across different scopes. It ensures functionality-wise consistency before and after refactoring and provide warnings when potential refactoring could lead to unwanted complexity or cause readability issues.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Maintenance—*Restructuring, reverse engineering, and reengineering*

General Terms

Algorithms

Keywords

Software engineering, clone detection, code refactoring, recommendation system, Python

1. INTRODUCTION

In software development, it's very common seeing developers reuse code fragments by copying and pasting with or without minor adaptation. Moreover, for large scale projects, developers are often too lazy to browse existing source files so that they may rewrite similar or even identical functions which were already in the code base. As a result, software systems often contain sections of code that are very similar, called code clones.

Previous research shows that a significant fraction (between 7% and 23%) of the code in a typical software system has been cloned [1] [11]. Many code clones in code bases are unnecessary duplications. Code duplication can be a significant drawback, leading to bad design, and increased probability of bug occurrence and propagation. As a result, it can significantly increase maintenance cost, and form a barrier for software evolution. By detecting, categorizing, and removing code clones, we can produce easier to understand, cleaner, and more reusable code.

Clone detection has been an avid research topic in the field of software engineering for decades. Fortunately, several automated techniques for detecting code clones have already been proposed. However, how to deal with detected clones, e.g. how to distinguish necessary clones from unnecessary ones and how to refactor code to remove unnecessary clones still remain a big problem in not only commercial but also academic domain. In this paper, we try to classify code clones and build a recommendation system called EasyMerge to help developers merge unnecessary clones on top of current state-of-the-art clone detection approach.

More specifically, we pick CloneDigger [4], an anti-unification duplicate code detection tool as our underlying clone detector. CloneDigger is one of the best available clone detection tools for its overall performance, coverage of multiple clone types, and availability. EasyMerge integrates CloneDigger as the pre-processing tool, analyze its output clone pairs, and recommend possible merges which can remove unnecessary clones without changing functionality of code base, creating reference conflicts, nor causing troubles to future

code understanding and reusing.

The rest of the paper is structured as follows: we first go through the basics, background, and current state of clone detection and code clone refactoring in general. Then we introduce and discuss the fundamentals of CloneDigger and the anti-unification algorithm it is using to detect clones. Afterwards, we explain EasyMerge's work flow and underlying techniques. And at the end, we set up testing environment and discuss the experimental results of running EasyMerge against several open source projects of different scales.

2. BACKGROUND

2.1 Clone Detection

Roy, Cordy, and Koschke have done a great work [12] writing an overview paper explaining the basics of clone detection, and providing a complete comparison of essential strengths and weaknesses of both individual tools and techniques and alternative approaches in general. It gives us all the needed preliminaries to focus on clone refactoring rather than spending time working on the detection part. We begin with a basic introduction to clone detection terminology in Roy's paper.

Definition 1. (Code Fragment). A code fragment (CF) is any sequence of code lines (with or without comments). It can be of any granularity, e.g., function definition, begin-end block, or sequence of statements. A CF is identified by its file name and begin-end line numbers in the original code base and is denoted as a triple ($CF.FileName$, $CF.BeginLine$, $CF.EndLine$).

Definition 2. (Code Clone). A code fragment $CF2$ is a clone of another code fragment $CF1$ if they are similar by some given definition of similarity, that is, $f(CF1) = f(CF2)$ where f is the similarity function (see clone types below). Two fragments that are similar to each other form a clone pair $(CF1, CF2)$, and when many fragments are similar, they form a clone class or clone group.

Definition 3. (Clone Types). There are two main kinds of similarity between code fragments. Fragments can be similar based on the similarity of their program text, or they can be similar based on their functionality (independent of their text). The first kind of clone is often the result of copying a code fragment and pasting into another location. In the following we provide the types of clones based on both the textual (Type 1 to 3)[3] and functional (Type 4)[6, 9] similarities:

- **Type-1:** Identical code fragments except for variations in whitespace, layout and comments.
- **Type-2:** Syntactically identical fragments except for variations in identifiers, literals, types, whitespace, layout and comments.
- **Type-3:** Copied fragments with further modifications such as changed, added or removed statements, in addition to variations in identifiers, literals, types, whitespace, layout and comments.
- **Type-4:** Two or more code fragments that perform the same computation but are implemented by different syntactic variants.

There are many different techniques to detect code clones. In Roy's paper [12], they covered tools of textual approaches, lexical approaches, syntactic approaches, semantic approaches and hybrids. Different tools have advantages and disadvantages for different facets (usage, interaction, language, clone information, technical aspect, adjustment, processing, and evaluation) and behave quite differently for some scenarios. As clone detection is still an on-going research, it's hard to say one tool is the best at the moment. However, for the purpose of building EasyMerge, we want to pick a tool that is cross-platform, freely available, efficient, covering multiple clone types, and can handle different editing scenarios decently. Thus we chose CloneDigger [4] after a comprehensive comparison.

2.2 Clone Refactoring

On the other hand, for clone refactoring, most proposed solutions use pre-defined metrics as the refactoring guidance to help users determine whether the code clones are suitable for refactoring. For example, Balazinska [2] proposed to use 21 metrics to measure the suitability for refactoring. Higo [7] developed ARIES tools, using the code detection tool CCFinder and environment analysis tool Gemini to find the clones can be refactored. And Schulze [13] proposed a new metric DIST (distance of the code clones).

For the actual merging process, currently existing research mainly focus on using refactoring patterns [5], especially "Extract Method" and "Pull Up Method". "Extract Method" means that a fragment of source code is extracted and redefined as a new method. "Pull Up Method" means that the same methods defined in child classes are pulled up to its parent class. In general, they are both merging similar code blocks into one newly created function, and "Pull Up Method" only differs from "Extract Method" as it places the newly created function into the super class when code blocks are inside functions sharing the same parent class.

Unfortunately, code clones may be coupled with surrounding code, its inheritance tree, or other independent classes/code blocks. For instance, code fragments inside clone pairs may contain function calls or variable references which are not defined inside the fragments. Furthermore, functions and variables of the same names may be in fact different at different locations of the code base. For example, if we have a code clone fragment of following:

```
for x in range(0, 3):
    a = x + 1
    b = f(x)
    c = d + x
```

We are calling function **f()** where the definition of **f()** can be different from fragment to fragment. Moreover, the value of variable **d** is potentially different, too. The current approach to solve this problem is to pass such external functions and

variables as parameters to newly generated methods from the "Extract Method" refactoring pattern. For instance:

```
def helper(p1, p2):
    for x in range(0, 3):
        a = x + 1
        b = p1(x)
        c = p2 + x
helper(f, d)
```

However, this creates two new problems. Firstly, in this example, the scope of b was changed. It is now inside the helper function and no longer available to code after the clone fragment which could potentially cause reference errors. Secondly, when the length of the clone fragment is too long, we could possibly end up with a helper function of tons of parameters which in turn make the code after refactoring harder to understand and is against our goal to improve readability. For most clone refactoring approaches, this gives bad scores to the evaluation metrics and prevent merging. And a possible alternative approach is to return b and such variables from the "Extracted Method" which introduces even more complexity.

For recently proposed refactoring approach [10] and the tools mentioned at the beginning of this subsection, they count the number of such external references in a code clone fragment as the key factor of their refactoring guideline metrics. And clone fragments of too many such occurrences will not be recommended for refactoring. Furthermore, in Li's work [10], after standardization, it can also be used to refactor some semantical similar but syntax-wise different clones by moving different function references to extracted method's parameters.

Additionally, results from clone detectors are usually clone pairs but it's very common more than two fragments are clones of each other. Moreover, overlapping clone fragments often exist in large scale code bases, too. For example, it's not difficult to find a code block of ABCDEF where BC is a clone fragment of one other block and CDE is a clone fragment of another. For both cases, merging one set of clones will in turn break clones in another set. And we need to take care of all reference errors caused by code deletion. Thus, our clone refactoring tool must handle all these potential issues without introducing too much computational overhead.

Unlike most of current clone refactoring approaches that use token-based CCFinder [8] for clone detection, EasyMerge uses a newer detection tool called CloneDigger. Compared to CCFinder, CloneDigger is cross-platform and freely available as an open source project. Moreover, based on evaluation from Roy [12], CloneDigger performs no worse than CCFinder in all detection scenarios and is actually better in certain scenarios e.g. a programmer copies a function that calculates the sum and product of a loop variable and calls another function, foo() with these values as parameters three times, making changes in whitespace in the first fragment (S1(a)), changes in commenting in the second (S1(b)), and changes in formatting in the third (S1(c)). In the following section, we will go over the basic ideas behind CloneDigger and its anti-unification algorithm.

3. CLONEDIGGER

CloneDigger [4] was an anti-unification code clone detector introduced by Bulychev and Minea in 2008. It's one of the latest open source clone detection tool.

3.1 Overview

Techniques for detecting duplicate code can be classified according to several criteria. Code can be viewed as similar based on syntactic criteria or at a semantic level. CloneDigger considers only syntactic similarity. The algorithm is approached based on abstract syntax trees. The algorithm of finding duplicates consists of several phases. In the beginning all statements are partitioned into clusters using anti-unification distance and the code is abstractly viewed as sequence of cluster identifiers. All pairs of identical sequences of cluster IDs, which have similar statements in corresponding positions, are globally checked for similarity using anti-unification distance.

3.2 Preliminaries

Anti-unification is a general expression of two given terms. Let E_1 and E_2 be two terms. Anti-unification E is a generalization of E_1 and E_2 if there exist two substitutions σ_1 and σ_2 such that $\sigma_1(E) = E_1$ and $\sigma_2(E) = E_2$. The most specific generalization of E_1 and E_2. The most specific generalization of E_1 and E_2 is called anti-unifier. The anti-unifier tree of two trees T_1 and T_2 is obtained by replacing some subtrees in T_1 and T_2 by special nodes.m containing term placeholders which are marked with integers, such as $?_n$. An anti-unifier stores only the common top-level tree structure and therefore details may be ignored.

The anti-unification distance is defined as follows: let U be the anti-unifier of two trees T_1 and T_2 with substitutions σ_1 and σ_2. σ_1 and σ_2 are mappings from the set $\{?_1, ?_2, ..., ?_n\}$ to substituting trees. Anti-unification distance between T_1 and T_2 as a sum of sizes of substituting trees in σ_1 and σ_2. The distance doesn't allow the permutation of siblings or changing the number of child nodes.

3.3 Duplicate Code Detection Algorithm

The method of finding duplicate code consists of three phases:

step 1 Partitioning similar statements into clusters

Identify similar statements using anti-unification and partition them into clusters. A two-pass clustering algorithm is used. The first pass of the algorithm compare each new statement with the anti-unifiers of all existing clusters. The function *add_cost* is used to compute the cost of adding a tree T to the cluster consisting of n trees with anti-unifier au. Let au' be the result of anti-unification of T and au with substitutions σ_1 and σ_2: $\sigma_1(au') = au$, $\sigma_2(au') = T$. *add_cost* is defined as $n \times |\sigma_1| + |\sigma_2|$. During the second pass all the statements re traversed again and for each statement we search for the most similar pattern from the set produced in the previous pass using the anti-unification distance. After the first phase each statement is marked with its cluster ID, two statements with the same cluster ID are considered similar in this preliminary view.

step 2 Finding pairs of identical cluster sequences All pairs of sequences of statements, which are large enough, are identically labeled.

step 3 Examining code sequences for overall similarity All candidates are checked as a whole using anti-unification distance. The occurrences of the same variable refers to one leaf in the abstract syntax tree increase the quality of the algorithm.

Now we can move on to introduce EasyMerge and how it works behind the scene.

4. EASYMERGE

The entire EasyMerge program is written with Python 2.7.5 and the user interface is powered by PyQt 4.10.4. It takes a folder of Python written source tree as input, look for all refactorable clones and recommend potential merges.

4.1 User Interface

EasyMerge has a simple and straightforward user interface, as shown in figure.

4.2 Preprocessing

The first step of EasyMerge is to run CloneDigger at target source tree, take its output and format and standardize it for our own usage. The core of this step is to classify code clones and get the ones that we not only can but also want to merge.

5. EXPERIMENTAL RESULTS

6. REFERENCES

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

[2] M. Balazinska, E. Merlo, and M. Dagenais. Advanced clone-analysis to support object-oriented system refactoring. *in: Proceedings of the Seventh Working Conference on Reverse Engineering, WCRE 2000*, pages 98–107, 2000.

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

[4] P. Bulychev and M. Minea. Duplicate code detection using anti-unification. *Spring Young Researchers Colloquium on Software Engineering, SYRCoSE 2008*, pages 4–7, 2008.

[5] M. Fowler. Refactoring: improving the design of existing code. *Addison Wesley*, 1999.

[6] M. Gabel, L. Jiang, and Z. Su. Scalable detection of semantic clones. *in: Proceedings of the 30th International Conference on Software Engineering, ICSE 2008*, pages 321–330, 2008.

[7] Y. Higo, S. Kusumoto, and K. Inoue. A metric-based approach to identifying refactoring opportunities for merging code clones in a java software system. *Journal of Software Maintenance and Evolution: Research and Practice*, 20(6):435–461, 2008.

[8] 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, 2002.

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

[10] X. Li, X. Su, P. Ma, and T. Wang. Refactoring structure semantics similar clones combining standardization with metrics. *in: Proceedings of International Conference on Soft Computing Techniques and Engineering Application Advances in Intelligence Systems and Computing*, 250:361–367, 2014.

[11] C. K. Roy and J. R. Cordy. An empirical study of function clones in open source software systems. *in: Proceedings of the 15th Working Conference on Reverse Engineering, WCRE 2008*, pages 81–90, 2008.

[12] 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, Special Issue on Program Comprehension, ICPC 2008*, 74(7):470–495, 2009.

[13] S. Schulze, M. Kuhlemann, and M. Rosenmüller. Towards a refactoring guideline using code clone classification. *in: Proceedings of the 2nd Workshop on Refactoring Tools, WRT 2008*, page 6, 2008.