CloneDifferentiator: Analyzing Clones by Differentiation

Zhenchang Xing, Yinxing Xue, and Stan Jarzabek
School of Computing
National University of Singapore
{xingzc, yinxing, stan}@comp.nus.edu.sg

Abstract—Clone detection provides a scalable and efficient way to detect similar code fragments. But it offers limited explanation of differences of functions performed by clones and variations of control and data flows of clones. We refer to such differences as semantic differences of clones. Understanding these semantic differences is essential to correctly interpret cloning information and perform maintenance tasks on clones. Manual analysis of semantic differences of clones is complicated and error-prone. In the paper, we present our clone analysis tool, called CloneDifferentiator. Our tool automatically characterizes clones returned by a clone detector by differentiating Program Dependence Graphs (PDGs) of clones. CloneDifferentiator is able to provide a precise characterization of semantic differences of clones. It can provide an effective means of analyzing clones in a task oriented manner.

Keywords- Clone analysis, Graph differencing, Program dependence graph

I. INTRODUCTION

Code clones are inherent in software. They result from copy-paste-modify practice in software development, or are introduced by application of design patterns or similar requirements [4]. Researchers have presented many clone detection techniques [9][10] based on code metrics, token, Abstract Syntax Tree (AST), or Program Dependence Graph (PDG). The effectiveness of these clone detection techniques has been empirically and comparatively evaluated [3][11].

However, detecting clones is only part of the story. To perform a concrete maintenance task (such as refactoring [5]) that affects the clones, one must correctly interpret cloning information. One problem with clone detection techniques is that they return many clones with little or no additional information to aid the user in their interpretation. Consequently, the user has to manually inspect and classify the detected clones pair by pair, and determine the relevance of these clones to a given maintenance task. This manual analysis process is time consuming and error prone.

In this paper, we present our clone analysis tool, called CloneDifferentiator. Our tool eases analysis of clones by characterizing clones automatically based on their semantic differences. The underlying rationale of CloneDifferentiator is that similarities and differences are two sides of the same coin. We cannot interpret software clones (i.e. similarities) without understanding their differences precisely.

CloneDifferentiator raises the level of analysis of clones to PDG. PDG abstracts away textual and syntactic differences of clones; it captures semantic information of clones and the program context in which they occur, i.e. the methods that contain cloned code fragments, the program entities that these methods use, and the control/data flow of clones. This allows CloneDifferentiator to provide a precise characterization of semantic differences of clones.

We have conducted two case studies using CloneDifferentiator: refactoring JavaIO library and parameterizing Eclipse JDT-model unit tests [12]. Our studies show that automatically characterizing semantic differences of clones can provide an effective means of analyzing clones in a task oriented manner. For example, a maintainer bent on removing code duplication would be interested in clones that have no semantic differences or clones being “part of” others, because such clones can be removed by refactorings such as extract method [5]. Such candidate clones can be easily located based on the differentiation results of CloneDifferentiator.

The rest of this paper is structured as follows. Section II describes our CloneDifferentiator tool. Section III briefly discusses the results of our case studies. Section IV reviews related work. Section V concludes our work.

II. CLONEDIFFERENTIATOR

CloneDifferentiator begins with detecting clones in source code (Section II.A). Then, it generates PDGs for clones and applies graph matching technique to identify seven types of semantic differences of clones (Section II.B). CloneDifferentiator allows the user to filter and query which clones and what types of clone differences should be inspected in a task oriented manner (Section II.C). It presents the analysis results in a GUI that allows the user to visually inspect clones and their semantic differences (Section II.D). Note that the user will interleave querying and visual inspection of clones during their clone analysis session.

A. Detecting Clones

CloneDifferentiator does not make assumptions regarding clone detectors. The current implementation of CloneDifferentiator has been integrated with CloneMiner [2]. CloneDifferentiator allows the user to request clone detection for a whole Java project or for some selected packages and/or source files.
CloneMiner is then invoked to detect clones in the selected project/packages/files. CloneMiner first finds simple clones (i.e., similar fragments of contiguous codes), using a token-based clone detection technique similar to CCFinder [7]. Then, it mines simple clones with frequent itemset mining [6] to detect clones across large program units, such as cloned methods that contain one or more simple clones.

CloneMiner reports clones as clone classes \( \{c_1, \ldots, c_n\} \) where \( c_i (1 \leq i \leq n) \) is referred to as a clone instance. A pair of such clone instances is referred to as a clone pair. CloneDifferentiator stores the clones reported by CloneMiner in a relational database and presents the detection results in a Clone TableView (not shown in this paper) for further analysis.

B. Differentiating Clones

After clone detection, the user of CloneDifferentiator can select all or some clone classes in Clone TableView and request the differentiation of selected clones. We briefly discuss here how CloneDifferentiator differentiates clones. Readers are referred to our technical report [12] for detailed discussion on the differentiation process.

Upon the user’s request, CloneDifferentiator first use Wala [15], a static analysis library for Java, to build intra-method PDGs for the clone instances of selected clone classes. The nodes of a Wala-generated PDG represent operation (e.g., method invocation), branch, parameter, and constant statements. The edges of a PDG represent the control and data dependences between statements. These PDGs capture semantic information of clone instances and the program context in which they occur.

Then, given a clone class \( \{c_1, \ldots, c_n\} \), let \( PDG_1 \) and \( PDG_2 \) be the PDGs of clone instances \( c_i \) and \( c_j \) (\( i \neq j, 1 \leq i, j \leq n \)). CloneDifferentiator applies GenericDiff [13] to compute the differences between \( c_i \) and \( c_j \). Next, it merges the two PDGs into a unified PDG (see an example in Figure 3). The unified PDG is constructed by first creating the matched parts of two PDGs, and then appending the unmatched statements and dependences on the basis of the matched parts of two PDGs.

Finally, CloneDifferentiator analyzes the unified PDG to detect instances of seven types of semantic differences of clones, which reveal differences of functions performed by clones and variations of control and data flows of clones:

- **Differential Property, Unmatched Operation Pair,** and **Unmatched Block Pair:** Structurally similar but functionally different clones;
- **Partially Matched Branch** and **Additional Branch:** Inconsistencies or subtle differences of control flows of clones;
- **Additional Operation** and **Additional Block:** Subtle or “big” differences in functions performed by clones

CloneDifferentiator stores the differentiation results of clones in a relational database, including the PDGs of clone instances that have been differentiated, the PDG differencing results shown in Figure 2 as an example. It reports some lexical differences between the code block (line 33-35) of listFiles(FileFilter) and the code block (line 34-35) of listFiles(FilenameFilter). In contrast, such differences are abstracted away in PDGs and thus are not reported by CloneDifferentiator. On the other hand, because lexical differencing compares clones as lines and chars, it cannot report the subtle differences in the condition checking \( filter==null \) and the actual methods being invoked \( filter.accept() \) (line 34 of the two methods), as reported by CloneDifferentiator.

Furthermore, the differencing results of CloneDifferentiator can be used to automatically filter and query clones in a task oriented manner, as described in Section II.C. In contrast, because lexical differences ignore semantic information to which they correspond, they have to be manually examined and interpreted. For example, lexical differencing identifies the differences between FileFilter (line 28 of listFiles(FileFilter)) and FilenameFilter (line 29 of listFiles(FilenameFilter)). However, lexical differencing cannot automatically determine that the detected lexical difference is due to the type difference of two parameters.

C. Filtering and Querying Clones

CloneDifferentiator is equipped with a set of simple filters for selecting clones of interest to the user based on the types and number of their semantic differences. Furthermore, semantic differences of clones can serve as basic building blocks for creating specific queries for identifying candidate clones that are potentially relevant to a specific maintenance task. In this section, let us review two scenarios from our case studies of cloning using CloneDifferentiator. More scenarios can be found in our technical report [12].

In JavaIO study, we would like to identify clones that can be easily removed by **extract method** refactoring [5]. To that end, we define query for searching for clones that have **no**
semantic differences or clones being “part of” others. The “part of” relations between clones identify situations in which the PDG of one clone instance is the subgraph of the PDG of the other instance, i.e. only one of the two compared PDGs has unmatched statements and dependencies. PipedInputStream.checkStateForReceive() and PipedReader.receive(int) is a pair of “part of” clones returned by our query. Both PipedInputStream and PipedReader need to check the pipe state in several places before starting receiving data. The developer of PipedInputStream recognized the repetition of this state checking and extracted the method checkStateForReceive(), while the developer of PipedReader did not. This clone suggests the opportunity to extract state checking logic from PipedReader.receive(int) into a method.

In the case study of JDT-model unit tests, we would like to identify cloned test methods that can be parameterized using seed-values testing pattern [14]. Seed-values pattern provides concrete input values for parameterized unit tests. To that end, we define query for searching for cloned test methods that have only differential properties and/or unmatched constant pairs. This query is motivated by the fact that with traditional unit testing, several tests with variant input values are usually developed to achieve a good coverage of the unit under test. The differences among these test variants are manifested as differential-properties and unmatched constants. In JDT-model unit tests, test methods for features such as searching and formatting often contain such differences, which can be parameterized as seed values.

D. Presenting Clones and Their Semantic Differences

CloneDifferentiator has incorporated three ways to present its analysis results to the user: a CloneDiff TreeView that summarizes clone classes that have been analyzed, a PDG Viewer for graphically inspecting clone differences in unified PDG, and a CloneDiff Compare Editor for inspecting clones and their differences in source code editors.

CloneDifferentiator lists all the clone classes that have been differentiated or those of interest to the user returned by some specific queries in a CloneDiff TreeView (not shown in this paper). Clone classes can be easily navigated to through the tree. For each clone class, TreeView summarizes the types and number of semantic differences it has. Clone classes can be sorted by the name of cloned methods or the types and number of their semantic differences.

Double-clicking a clone pair in CloneDiff TreeView opens a PDG Viewer that visualizes the unified PDG of the selected cloned pair. This view allows the user to graphically inspect the PDGs of clones and their differences, especially differences of data and control dependences. For example, Figure 3 shows partially the unified PDG of the cloned methods listFiles(FilenameFilter) and listFiles(FileFilter). The black nodes and edges represent matched statements and dependences in the PDGs of the two methods, while the green and red nodes/edges represent unmatched statements and dependences of the two methods, respectively.

In this example, the program control flows from the matched branch (i<ss.length) directly to the unmatched branch (filter==null) in listFiles(FilenameFilter), while in listFiles(FileFilter) the control flows first to the instantiation of a File object and then to the unmatched branch (filter==null). Thus, CloneDifferentiator reports the unmatched branch filter==null in listFiles(FilenameFilter) as partially matched branch, while the unmatched branch filter==null in listFiles(FileFilter) as additional branch.

Double-clicking a clone pair in CloneDiff TreeView also opens a CloneDiff Compare Editor (see Figure 1). This editor displays the source code of the cloned methods side by side. CloneDifferentiator extends Eclipse compare editor to highlight code segments based on the semantic differences of the selected clone pairs, using different fonts, backgrounds or underlines. The CloneDiff Compare Editor can be configured to highlight only specific type(s) of differences of interest to the user. Hovering mouse over a highlighted code segment pops up a short description that summarizes the semantic differences in the hovered code segment.

III. EVALUATION

We have evaluated our CloneDifferentiator tool on two Java software systems: JavaIO library and Eclipse JDT-model unit tests. In JavaIO study, we used CloneDifferentiator to identify clones that can be refactored using extract method refactoring or Java generics. We also used CloneDifferentiator to investigate the inconsistent programming styles in JavaIO library. In JDT-model-unit-tests study, we used CloneDifferentiator to identify clones that can be parameterized using seed values, state machine, or assume/assert invariants testing patterns [14]. Our evaluation shows that CloneDifferentiator can provide a precise characterization of the important semantic differences of clones. This enables developers to query and inspect clones in a task oriented manner, by quickly removing “irrelevant” clones in the detection results, rather than browsing the detected clones pair by pair [12].

IV. RELATED WORK

Researchers have presented many techniques to detect clones. Roy and Cordy [10] and Koschke [9] provide comprehensive surveys of existing work. Roy and Cordy [11] and Bellon et al. [3] report the detailed quantitative evaluation of clone detection techniques. Clone detection tools often provide only limited explanation of clones through lexical differencing or code metrics. For example, Balazinska et al. [1] define a classification schema based on the differences between token sequences forming the clones. CCFinder [7] generates metrics for clones to aid understanding of clones. To better understand the characteristics of cloning, Kapser and Godfrey [8] propose a clone classification through syntactic analysis of locality of clones. Our CloneDifferentiator raises the analysis of clones to PDG and exploits robust graph differencing technique to differentiate clones and detect seven types of semantic differences in clones. CloneDifferentiator helps to analyze clones from the perspective of their differences.

V. CONCLUSION AND FUTURE WORK

We cannot judge clones without understanding their differences precisely. This paper presented our clone analysis tool, CloneDifferentiator, which can automatically provide a
precise characterization of clones by differentiating the PDGs of clones. CloneDifferentiator can simplify the interpretation and analysis of clones in a task oriented manner. Furthermore, it allows us to investigate in more detail the characteristics of code cloning from a new perspective (i.e., how clones are different). It opens new opportunities to better apply clone detection techniques in various software maintenance settings.

REFERENCES