Refactoring Clones: A New Perspective

Nikolaos Tsantalis, Giri Panamoottil Krishnan
Department of Computer Science and Software Engineering
Concordia University, Montreal, Quebec, Canada
nikolaos.tsantalis@concordia.ca, giri.krishnan@concordia.ca

***Abstract*—In this position paper we support that there is still great potential for advancements in the research area of software clone refactoring, and argue about some possible research objectives and directions through illustrative examples.**

I. Introduction

Software clone management comprises all activities of looking after and making decisions about consequences of copying and pasting [1]. According to Koschke [1], we can distinguish three main lines of clone management: a) *preventive*, which comprises activities to avoid new clones, b) *compensative*, which encompasses activities aimed at limiting the negative impact of existing clones that are to be left in the system, and c) *corrective*, which covers activities to remove clones from a system. Recent research work has focused more on the preventive and compensative aspects by providing techniques for clone tracking, incremental clone detection and clone consistency analysis [2], and much less on the corrective aspect of clone management.

A recent study by Tairas and Gray [3] on the clones detected in 9 open-source projects using the Deckard clone detection tool, revealed that only 10.6% of the detected clone groups could be refactored by the Eclipse IDE, while their technique (CeDAR) was able to refactor successfully 18.7% of them. Clearly, there is still great space to improve the percentage of clones that can be refactored. In this position paper, we will present cases that cannot be handled by existing clone refactoring techniques and suggest possible solutions as a means to initiate new research directions on the refactoring of software clones.

II. Related Work

In this section we will briefly discuss the current state-of-the-art in the refactoring of software clones. Higo et al. [4] and Choi et al. [5] propose a metric-based approach to examine whether a clone set can be refactored and suggest appropriate refactoring strategies. Tairas and Gray [3] extend the refactoring engine in Eclipse IDE, which supports only the parameterization of differences in local variable identifiers, with additional parameterized differences between field accesses, method calls and literals. This enables the refactoring of clones containing dissimilarities between different types of AST nodes. Hotta et al. [6] detect isomorphic subgraphs in the Program Dependence Graphs (PDGs) of two methods containing *Type-3* clones in order to extract their **common process** that can be pulled up to a base class. The remaining code fragments that do not belong to the detected isomorphic subgraphs constitute the **unique processes** that should remain in each derived class.

III. Research Directions

A. Determining Valid Clone Regions

Most existing clone refactoring techniques [3], [4] recognize that the presence of valid clone regions (i.e., the regions in which the clones expand) is an important condition to enable the refactoring of a clone group. A valid clone region is a region that does not contain incomplete statements. A statement is considered as incomplete if part of its expression(s) or body is not included in the clone region. Experience has shown that text-based and token-based clone detection tools may return invalid clone regions [3], [4]. To address this problem, Aries [4] extracts a subset of syntactically valid units from token-based clones, while CeDAR [3] employs AST-based clone detection tools in order to get appropriate input for their clone refactoring techniques. AST-based techniques return clones which are sub-trees within the abstract syntax tree of the code, and thus always form valid clone regions. However, a universal clone refactoring technique should be able to process results obtained from any clone detection tool.

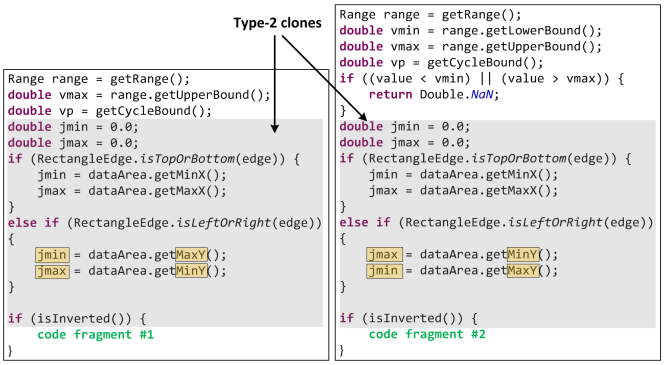

Fig. 1. Example of invalid clone regions (highlighted in gray).

The example shown in Figure 1 is a typical case of invalid clone regions (the last `if` statement is incomplete in both clones). Block-based regions have been used in method decomposition techniques [7] for determining alternative regions in which a slice may expand (i.e., block-based slicing), and therefore exploring alternative decomposition opportunities. We believe that block-based regions could be extremely useful for restricting software clones into valid regions that will enable their refactoring.

B. Intelligent Parameterization of Differences

Type-2 clones are syntactically identical code fragments that differ in variable identifiers, method call identifiers, literals, and types. *Type-2* clones can be refactored by mapping the differences among the clones of a clone group and introducing a parameter of appropriate type in the extracted method for each parameterized difference. After the extraction of the duplicated code, the methods that originally contained the clones call the extracted method by passing as arguments the values corresponding to the parameterized differences. The majority of clone refactoring tools support the parameterization of differences in local variable identifiers. Recently, CeDAR [3] introduced the parameterization of differences between different types of AST nodes (e.g., variables replaced with method calls). However, even this approach would be ineffective in the example of Figure 2, because an entire expression (i.e., `high - low`) is replaced with a method call. This example could be refactored only if parameterization took place at argument level (i.e., a higher level in the AST) and not at identifier level (i.e., AST leaves).

```
Rectangle2D rect = null;
if (orientation == HORIZONTAL) {
  low = Math.max(low,dataArea.getMinY());
  high = Math.min(high,dataArea.getMaxY());
  rect = new Rectangle2D.Double(
         dataArea.getMinX(), low,
         dataArea.getWidth(), high - low);
}
```

```
Rectangle2D rect = null;
if (orientation == HORIZONTAL) {
  low = Math.max(low,dataArea.getMinX());
  high = Math.min(high,dataArea.getMaxX());
  rect = new Rectangle2D.Double(
         low, dataArea.getMinY(),
         high - low, dataArea.getHeight());
}
```

Fig. 2. Example requiring a more advanced parameterization of differences.

C. Refactoring of Type-3 Clones

The refactoring of *Type-3* clones is challenging due to the presence of unmatched statements (i.e., replaced, added, or removed statements). The PDG is the most appropriate representation for the problem of *Type-3* clone refactoring [8]. First, it can reduce the ambiguity of statement matching, since statement similarity can be assessed not only based on textual or AST-structure similarity, but also based on the matching of incoming/outgoing control and data dependencies. Second, the PDG can be used to determine whether the unmatched statements between the clones (i.e., statements in gaps) can be moved before or after the execution of the duplicated code by examining whether this move alters the original dependencies of the graph [7]. Therefore, the problem of finding the common statements between *Type-3* clones can be transformed into finding the **Maximum Common Subgraph** (MCS) in their PDGs. Previous techniques [8], [9] have applied backward and forward slicing on PDGs to detect isomorphic subgraphs. However, these approaches return a single mapping solution, which might not be optimal. On the other hand, MCS techniques create a search tree by exploring the entire solution space. An optimal solution can be determined by finding the subgraph in the search tree that has the maximum number of mapped nodes/edges and the minimum number of structural differences between the mapped nodes.

The clone on the left side of Figure 3 contains two additional statements compared to the clone on the right side. These statements define two variables, namely `lineVisible` and `shapeVisible`, which are used as arguments in the `LegendItem` constructor call that follows. PDG analysis can reveal that these two statements can be moved before the beginning of the left-side clone, thus enabling the extraction of the entire duplicated code fragment in a single method.

```
- duplicated code fragment #1 -
boolean lineVisible =
    getItemLineVisible(series, 0);
boolean shapeVisible =
    getItemShapeVisible(series, 0);
LegendItem result = new LegendItem(
  label, description, toolTipText,
  urlText, shapeVisible, shape,
  getItemShapeFilled(series, 0),
  fillPaint, shapeOutlineVisible,
  outlinePaint, outlineStroke, lineVisible,
  new Line2D.Double(-7.0, 0.0, 7.0, 0.0),
  getItemStroke(series, 0),
  getItemPaint(series, 0));
- duplicated code fragment #2 -
```

```
- duplicated code fragment #1 -
GAP
LegendItem result = new LegendItem(
    label, description, toolTipText,
    urlText, true, shape,
    getItemShapeFilled(series, 0),
    fillPaint, shapeOutlineVisible,
    outlinePaint, outlineStroke, false,
    new Line2D.Double(-7.0, 0.0, 7.0, 0.0),
    getItemStroke(series, 0),
    getItemPaint(series, 0));
- duplicated code fragment #2 -
```

Fig. 3. Example of *Type-3* clones with a gap.

D. Refactoring Sub-clones

All existing approaches are unable to refactor clone fragments that compute more than one variable, since the extracted method in which the duplicated code will be moved may return at most one variable (in Java programming language). In the example of Figure 1, we can observe that both clones contain the computation of two variables, namely `jmin` and `jmax`. These clones can be refactored only by extracting separately the computation of each variable. This can be achieved by decomposing the original clones into **sub-clones** having a distinct functionality [7]. In this particular example, the `if` and `else if` conditional structures will have to be duplicated in the two extracted methods, since they are required for the computation of both `jmin` and `jmax` variables. However, the number of duplicated statements will be significantly reduced from 8 statements before refactoring to just 2 statements after refactoring.

REFERENCES

[1] R. Koschke, "Frontiers of software clone management," in *Frontiers of Software Maintenance*, 2008, pp. 119–128.

[2] H. A. Nguyen, T. T. Nguyen, N. H. Pham, J. Al-Kofahi, and T. N. Nguyen, "Clone management for evolving software," *IEEE Transactions on Software Engineering*, vol. 38, no. 5, pp. 1008–1026, 2012.

[3] R. Tairas and J. Gray, "Increasing clone maintenance support by unifying clone detection and refactoring activities," *Inf. Softw. Technol.*, vol. 54, no. 12, pp. 1297–1307, Dec. 2012.

[4] 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*, vol. 20, no. 6, pp. 435–461, 2008.

[5] E. Choi, N. Yoshida, T. Ishio, K. Inoue, and T. Sano, "Extracting code clones for refactoring using combinations of clone metrics," in *Proceedings of the 5th International Workshop on Software Clones*, 2011, pp. 7–13.

[6] K. Hotta, Y. Higo, and S. Kusumoto, "Identifying, tailoring, and suggesting form template method refactoring opportunities with program dependence graph," in *Proceedings of the 16th European Conference on Software Maintenance and Reengineering*, 2012, pp. 53–62.

[7] N. Tsantalis and A. Chatzigeorgiou, "Identification of extract method refactoring opportunities for the decomposition of methods," *J. Syst. Softw.*, vol. 84, no. 10, pp. 1757–1782, Oct. 2011.

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

[9] Y. Higo and S. Kusumoto, "Code clone detection on specialized PDGs with heuristics," in *Proceedings of the 15th European Conference on Software Maintenance and Reengineering*, 2011, pp. 75–84.