Unification and Refactoring of Clones

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

***Abstract*—Code duplication has been recognized as a potentially serious problem having a negative impact on the maintainability, comprehensibility, and evolution of software systems. In the past, several techniques have been developed for the detection and management of software clones. Existing code duplication can be eliminated by extracting the common functionality into a single module. However, the unification and refactoring of software clones is a challenging problem, since clones usually go through several modifications after their initial introduction. In this paper we present an approach for the unification and refactoring of software clones that overcomes the limitations of previous approaches. More specifically, our approach is able to detect and parameterize non-trivial differences between the clones. Moreover, it can find an optimal mapping between the statements of the clones that minimizes the number of differences. We compared the proposed technique with a competitive clone refactoring tool and concluded that our approach is able to find a significantly larger number of refactorable clones.**

I. Introduction

Code duplication has been recognized as a potentially serious problem in software systems [1]. There is empirical evidence that duplicated code increases significantly the maintenance effort and cost [2], is associated with error-proneness due to the inconsistent changing of clones [3], and is more unstable than non-duplicated code [4]. Over the past years, the software clone research community has mainly focused on developing techniques for the detection of duplicated code [1] and creating tools for the management of detected clones, such as clone tracking, clone evolution and consistency analysis, and incremental clone detection [5]. However, the corrective aspect of clone management [6] (i.e., activities to remove clones from a system through refactoring) has not received as much attention from researchers.

In a previous work [7], we presented some limitations of existing clone refactoring tools. The first major limitation is that current tools can parameterize only a small subset of the differences that can be found in clones. The second limitation is that current approaches may return a non-optimal mapping between the statements of the clones. To facilitate the refactoring of duplicated code, an optimal mapping should not only contain the maximum number of possible mapped statements, but also the minimum number of differences between them. The **minimization of the differences** is of key importance for the refactoring of clones, since it directly affects the number of parameters that have to be introduced in the extracted method containing the common functionality, as well as the feasibility of the refactoring transformation. However, the current approaches focus only in maximizing the number of mapped statements and do not explore the entire search space of possible matches (i.e., they usually select the "first" or the "best" match based on a similarity function).

In this paper, we present a technique for the refactoring of software clones in Java programs that tackles the aforementioned limitations. Our approach takes as input two code fragments or even entire methods that have been detected as clones by clone detection tools and applies three steps to determine whether the clones or parts of them can be safely refactored. In the first step, it tries to find identical control dependence structures within the clones that will serve as candidate refactoring opportunities. In the second step, it applies a mapping approach that tries to maximize the number of mapped statements and at the same time minimize the number of differences between them. Finally, in the last step, the differences detected in the previous step are examined against a set of preconditions to determine whether they can be parameterized without changing the program behavior. Our technique supports the refactoring of *Type-1* clones (i.e., identical code fragments except for variations in whitespace, layout, and comments [1]), *Type-2* clones (i.e., structurally/syntactically identical fragments except for variations in identifiers, literals, types, layout and comments [1]), and *Type-3* clones (i.e., copied fragments with statements changed, added or removed in addition to variations in identifiers, literals, types, layout and comments [1]). Our statement matching approach is described in Section III. The proposed technique builds on top of a previous work [8], in which we presented our initial ideas on treating the unification of clones as an optimization problem. Apart from Section IV-B, all other sections contain new and unpublished material. Additionally, there have been made some improvements in the algorithms presented in Section IV-B.

To evaluate our approach, we compared it with CeDAR [9], a state-of-the-art tool in the refactoring of *Type-2* clones. We repeated the same experiment that they performed on the clones detected by Deckard [10]. The results have shown that our approach is able to find 83% more refactorable clones than CeDAR in the 7 Java open-source projects examined.

In summary, the contributions of the proposed technique are the following:
- It supports the detection and parameterization of non-trivial differences between duplicated code fragments.
- It can process clones detected from any clone detection tool even if they do not have an identical control dependence structure, or they do not expand over a valid block region.

978-1-4799-3752-3/14 © 2014 IEEE

CSMR-WCRE 2014, Antwerp, Belgium

- It treats the problem of finding a mapping between the statements of two clones as an optimization problem with two objectives, namely maximizing the number of mapped statements and at the same time minimizing the number of differences between the mapped statements.
- It defines preconditions that can be used to determine whether a clone group is safe to be refactored.

II. Preliminaries

The core of our approach is built around the mapping of the *Program Dependence Graphs* (PDGs) corresponding to duplicated code fragments. The PDG [11] is a directed graph with multiple edge types, in which the nodes represent the statements of a function or method, and the edges represent control and data flow dependences between statements. A *control dependence* edge denotes that the execution of the statement at the end point of the edge depends on the control conditions of the control predicate statement (e.g., `if`, `for`) at the start point of the edge. A *data dependence* edge is always labeled with a variable *v* and denotes that the statement at the end point of the edge is using the value of *v*, which has been modified by the statement at the start point of the edge. If the data dependence is carried through a loop node *l*, then it is considered as a *loop-carried* dependence.

The PDG representation used in this paper is extended in two ways. First, we introduce composite variables [12] representing the state of the objects being referenced within the body of a method, and create additional data dependences for these variables by analyzing method calls that may modify or use the state of the referenced objects. Second, we add two more types of edges in the PDG, which are used in the examination of preconditions (Subsection IV-C). These edges are anti-dependences and output-dependences. An *anti-dependence* edge due to variable *v* denotes that the statement at the end point of the edge is modifying the value of *v*, which has been used by the statement at the start point of the edge (i.e, the opposite of a data dependence). An *output-dependence* edge due to variable *v* denotes that both statements at the start and end points of the edge modify the value of *v*.

Additionally, in a previous work [8], we introduced the *Control Dependence Tree* (CDT) in order to represent the control dependence structure of the duplicated code fragments. A CDT has the same structure as the *Control Dependence Graph* (CDG) [11] with the only difference being that it includes only the control predicate nodes of the PDG, while a CDG contains all nodes of the PDG.

III. PDG Node and Edge Compatibility

In this section, we describe our statement matching function that is based on the analysis of the Abstract Syntax Tree (AST) structure. This function is used in all our algorithms (see function *compatibleAST*) to examine if two statements are compatible and can be matched. In our approach we consider two PDG nodes as compatible, if they correspond to the same AST statement type and have a matching AST structure. However, we provide a high degree of freedom in the matching of expressions within the statements in order to make more flexible the unification of duplicated code with non-trivial differences. In the past, Tairas and Gray [9] extended the Eclipse IDE refactoring engine for duplicated code, which supports only the parameterization of differences in local variable identifiers, by allowing the matching of differences in field accesses, string literals, and method calls without arguments. The evaluation on the *Type-2* clones detected in 9 open-source projects using Deckard [10] has shown that the aforementioned changes in the matching of duplicated code increased the percentage of refactorable clones from 10.6% to 18.7% on average. In our AST matching implementation, we have significantly increased the number of expression types that could be parameterized, and additionally we allow the matching of different types of expressions. Table I contains the complete list of supported expression types. Any expression type in that list can be replaced with any other expression type as long as both expressions return the same class/primitive type or types being subclasses of a common superclass (excluding `Object`). In the case of control predicate nodes (e.g., `if`, `for` statements), the part of the AST structure being compared is only their conditional expression(s).

TABLE I
Supported Expression Types In AST Matching

Expression Type	Example
Method Invocation	`expr.method(arg0, arg1,...)`
Super Method Invocation	`super.method(arg0, arg1,...)`
String Literal	`"string"`
Character Literal	`'c'`
Boolean Literal	`true` or `false`
Number Literal	`5.6`
Null Literal	`null`
Type Literal	`Type.class`
Class Instance Creation	`new Type(arg0, arg1,...)`
Array Creation	`new Type[expr]`
Array Access	`array[index]`
Field Access	`this.identifier`
Super Field Access	`super.identifier`
Parenthesized Expression	`(expr)`
Simple Name	`identifier`
Qualified Name	`Type.identifier`
Cast Expression	`(Type)expr`
This Expression	`this`
Prefix Expression	`-expr`
Infix Expression	`expr1 + expr2`

* we also support the matching of an assignment expression, where the left-hand side is a field access, e.g., `field = value`, with the corresponding setter method invocation, e.g., `setField(value)`.

Our AST matching algorithm has been implemented by extending the `ASTMatcher` superclass provided in Eclipse JDT framework. Our implementation also returns a list of the differences detected between the matched PDG nodes that is used in the examination of preconditions (Subsection IV-C). Table II shows the difference types which are reported by our AST matching algorithm. The last two difference types, namely operator and variable type mismatches (with the exception of generic types) cannot be parameterized. In the cases where a difference refers to a property of a primary expression (e.g., method name mismatch, argument number mismatch,

missing caller expression), the entire primary expression (e.g., method invocation) should be parameterized.

TABLE II
DETECTED DIFFERENCES BETWEEN MATCHED NODES

Difference Type	Example	
Variable Identifier	`int x = y;`	`int x = z;`
Literal Value	`String s = "s1";`	`String s = "s2";`
Method Name	`expr.foo(arg);`	`expr.bar(arg);`
Argument Number	`foo(arg0, arg1);`	`foo(arg0);`
Caller Expression	`expr.foo(arg);`	`foo(arg);`
Array Dimension	`int x = a[i];`	`int x = a[i][j];`
AST Compatible	`int x = foo();`	`int x = 5;`
Operator	`int x = y + z;`	`int x = y - z;`
Variable Type	`int x = 5;`	`double x = 5;`

Regarding PDG edge compatibility, two PDG edges are considered compatible if they connect nodes which are compatible (i.e., the nodes in the starting and ending points of the edges, respectively, should be compatible with each other) and they have the same dependence type (i.e., they are both control or data flow dependences). In the case of control dependences, both should have the same control attribute (i.e., True or False). In the case of data dependences, the data attributes should correspond to variables having the same name, or to variables detected as renamed during the AST compatibility check of the attached nodes. Finally, if both data dependences are *loop-carried*, then the loop nodes through which they are carried should be compatible too.

IV. TECHNIQUE

The proposed technique comprises three major steps, which will be presented in detail in the following sections.

A. Input

The proposed technique is able to process two different forms of input:

1) Two code fragments within the body of the same method, or different methods, reported as clones by a clone detection tool.
2) Two method declarations considered to be duplicated or containing duplicate code fragments.

Our goal is to find *isomorphic* CDTs within the duplicated code fragments. In the first case, we are constructing the control dependence subtrees corresponding to each code fragment, while in the second case we are constructing the CDTs corresponding to each method declaration. However, there is no guarantee that the constructed CDTs will be isomorphic in either case. As a result, we developed an algorithm that takes as input two CDTs and finds all non-overlapping *largest common subtrees* [13] within the CDTs. Each resulting subtree match will be further investigated as a separate clone refactoring opportunity. Initially, we collect from the two CDTs all leaf nodes, which either do not have siblings, or all of their siblings are also leaf nodes. Next, we extract all matching (i.e., AST compatible) pairs between the collected leaf nodes from the two CDTs. Each leaf node pair is given as input to Algorithm 1, which returns a subtree match as a solution.

In a nutshell, the algorithm first compares the sibling nodes of the node pair given as input to find matching sibling pairs. For each matching sibling pair it performs a top-down tree match and examines if the resulting subtree match is "exactly paired" (Algorithm 1, line **13**). Two subtrees are considered as exactly paired if there is a *one-to-one correspondence* between their nodes (i.e., a *bijection*). In set theory, there is a bijection from set X to set Y when every element of X is paired with exactly one element of Y, and every element of Y is paired with exactly one element of X. If all matching sibling pairs lead to exactly paired top-down subtree matches, then the parent nodes of the node pair given as input are visited. Finally, if the parent nodes match, then Algorithm 1 is recursively executed with the new parent node pair as input. The proposed algorithm essentially applies a combination of bottom-up and top-down tree matching techniques [13] and guarantees that the returned subtree match will be exactly paired. We designed the algorithm to return only exactly paired subtree matches in order to avoid inconsistencies or gaps in the control dependence structure of the matched subtrees.

B. PDG Mapping

In the previous step of our approach, we described an algorithm that extracts isomorphic subtrees from the CDTs of the clones given as input. In the current step, we present an approach for the optimal mapping of the PDGs corresponding to the extracted CDTs.

But first, we will use an example to motivate the need for optimizing the mapping of PDGs, so that the number of differences between the mapped PDG nodes is minimum. Figure 1 illustrates two code fragments taken from methods `drawDomainMarker` and `drawRangeMarker`, respectively, found in class `AbstractXYItemRenderer` of the *JFreeChart* open-source project (version 1.0.14). These two methods contain over 90 duplicated statements extending through their entire body. However, for the sake of simplicity, we have included only a small portion of the duplicated code. Figure 1 depicts a possible mapping of the statements as obtained from the PDG-based clone detection approaches discussed in section VI-A. These techniques always select one match in the case of multiple possible node matches (e.g., statement 67 on the left side can be mapped to statements 68, 71, 80, and 83 on the right side), which, in the mapping of Figure 1, coincides with the 'first' match according to the actual order of the statements. As it can be observed from Figure 1, the mapping is maximum, since all 25 statements have been successfully mapped; however, it contains a large number of differences between the mapped statements. The minimization of the differences is of key importance for the refactoring of clones, since it directly affects the number of parameters that have to be introduced in the extracted method containing the common functionality, as well as the feasibility of the refactoring transformation. Figure 2 depicts the optimal mapping, which is again maximum in terms of the number

```
1  Function bottomUpMatch(nodePair, solution)
       Data: nodePair represents a pair of matching CDT
             nodes (node_i, node_j)
       Result: solution contains a set of CDT node pairs
               representing a complete subtree match
2      append nodePair to solution
3      siblings_i ← nodePair.node_i.siblings
4      siblings_j ← nodePair.node_j.siblings
5      matchedSiblings ← ∅
6      tempSolution ← ∅
7      foreach sibling_i ∈ siblings_i do
8          foreach sibling_j ∈ siblings_j do
9              if compatibleAST(sibling_i, sibling_j) and
10             not alreadyMatched(sibling_j) then
11                 pair ← (sibling_i, sibling_j)
12                 pairs ← topDownMatch(pair)
13                 if exactlyPairedSubtrees(pairs) then
14                     add pair → matchedSiblings
15                     append pairs to tempSolution
16                     break // first-match
17                 end if
18             end if
19         end foreach
20     end foreach
21     if |matchedSiblings| = |siblings_i| = |siblings_j|
       then
22         append tempSolution to solution
23         parent_i ← nodePair.node_i.parent
24         parent_j ← nodePair.node_j.parent
25         if compatibleAST(parent_i, parent_j) then
26             pair ← (parent_i, parent_j)
27             bottomUpMatch(pair, solution)
28         end if
29     end if
30 end
```

Algorithm 1: Recursive function returning the maximum exactly paired subtree match starting from a given node pair.

of mapped statements, but it has also the minimum number of differences between the mapped statements. Clearly, the bodies of the `if`/`else if` statements in the left and right side of Figure 2 are 'symmetrical' to each other. Consequently, parameterizing the differences in the conditional expressions of the 'symmetrical' `if`/`else if` statements makes easier the refactoring of the clones and introduces less parameters to the extracted method.

The core of our PDG mapping technique is a divide-and-conquer algorithm that breaks the initial mapping problem into smaller sub-problems based on the control dependence structure of the isomorphic CDTs extracted in the previous step. In a nutshell, Algorithm 2 performs a bottom-up processing of every level in the CDTs. At each level it uses all possible pairwise combinations of the matching control predicate nodes as starting points for a *Maximum Common Subgraph* (MCS) algorithm that is restricted in mapping the PDG subgraphs

```
60 if (im.getOutlinePaint() != null &&
       im.getOutlineStroke() != null) {
61   if (orientation == VERTICAL) {
62     Line2D line = new Line2D.Double();
63     double y0 = dataArea.getMinY();
64     double y1 = dataArea.getMaxY();
65     g2.setPaint(im.getOutlinePaint());
66     g2.setStroke(im.getOutlineStroke());
67     if (range.contains(start)) {
68       line.setLine(start2d, y0, start2d, y1);
69       g2.draw(line);
       }
70     if (range.contains(end)) {
71       line.setLine(end2d, y0, end2d, y1);
72       g2.draw(line);
       }
     }
73   else if (orientation == HORIZONTAL) {
74     Line2D line = new Line2D.Double();
75     double x0 = dataArea.getMinX();
76     double x1 = dataArea.getMaxX();
77     g2.setPaint(im.getOutlinePaint());
78     g2.setStroke(im.getOutlineStroke());
79     if (range.contains(start)) {
80       line.setLine(x0, start2d, x1, start2d);
81       g2.draw(line);
       }
82     if (range.contains(end)) {
83       line.setLine(x0, end2d, x1, end2d);
84       g2.draw(line);
       }
     }
   }
```

```
61 if (im.getOutlinePaint() != null &&
       im.getOutlineStroke() != null) {
62   if (orientation == VERTICAL) {
63     Line2D line = new Line2D.Double();
64     double x0 = dataArea.getMinX();
65     double x1 = dataArea.getMaxX();
66     g2.setPaint(im.getOutlinePaint());
67     g2.setStroke(im.getOutlineStroke());
68     if (range.contains(start)) {
69       line.setLine(x0, start2d, x1, start2d);
70       g2.draw(line);
       }
71     if (range.contains(end)) {
72       line.setLine(x0, end2d, x1, end2d);
73       g2.draw(line);
       }
     }
74   else if(orientation == HORIZONTAL) {
75     Line2D line = new Line2D.Double();
76     double y0 = dataArea.getMinY();
77     double y1 = dataArea.getMaxY();
78     g2.setPaint(im.getOutlinePaint());
79     g2.setStroke(im.getOutlineStroke());
80     if (range.contains(start)) {
81       line.setLine(start2d, y0, start2d, y1);
82       g2.draw(line);
       }
83     if (range.contains(end)) {
84       line.setLine(end2d, y0, end2d, y1);
85       g2.draw(line);
       }
     }
   }
```

Fig. 1. Non-optimal mapping with 25 mapped nodes and 24 differences.

```
60 if (im.getOutlinePaint() != null &&
       im.getOutlineStroke() != null) {
61   if (orientation == VERTICAL) {
62     Line2D line = new Line2D.Double();
63     double y0 = dataArea.getMinY();
64     double y1 = dataArea.getMaxY();
65     g2.setPaint(im.getOutlinePaint());
66     g2.setStroke(im.getOutlineStroke());
67     if (range.contains(start)) {
68       line.setLine(start2d, y0, start2d, y1);
69       g2.draw(line);
       }
70     if (range.contains(end)) {
71       line.setLine(end2d, y0, end2d, y1);
72       g2.draw(line);
       }
     }
73   else if (orientation == HORIZONTAL) {
74     Line2D line = new Line2D.Double();
75     double x0 = dataArea.getMinX();
76     double x1 = dataArea.getMaxX();
77     g2.setPaint(im.getOutlinePaint());
78     g2.setStroke(im.getOutlineStroke());
79     if (range.contains(start)) {
80       line.setLine(x0, start2d, x1, start2d);
81       g2.draw(line);
       }
82     if (range.contains(end)) {
83       line.setLine(x0, end2d, x1, end2d);
84       g2.draw(line);
       }
     }
   }
```

```
61 if (im.getOutlinePaint() != null &&
       im.getOutlineStroke() != null) {
74   if(orientation == HORIZONTAL) {
75     Line2D line = new Line2D.Double();
76     double y0 = dataArea.getMinY();
77     double y1 = dataArea.getMaxY();
78     g2.setPaint(im.getOutlinePaint());
79     g2.setStroke(im.getOutlineStroke());
80     if (range.contains(start)) {
81       line.setLine(start2d, y0, start2d, y1);
82       g2.draw(line);
       }
83     if (range.contains(end)) {
84       line.setLine(end2d, y0, end2d, y1);
85       g2.draw(line);
       }
     }
62   else if (orientation == VERTICAL) {
63     Line2D line = new Line2D.Double();
64     double x0 = dataArea.getMinX();
65     double x1 = dataArea.getMaxX();
66     g2.setPaint(im.getOutlinePaint());
67     g2.setStroke(im.getOutlineStroke());
68     if (range.contains(start)) {
69       line.setLine(x0, start2d, x1, start2d);
70       g2.draw(line);
       }
71     if (range.contains(end)) {
72       line.setLine(x0, end2d, x1, end2d);
73       g2.draw(line);
       }
     }
   }
```

Fig. 2. Optimal mapping with 25 mapped nodes and 2 differences.

containing the nodes nested under the currently processed pair of control predicate nodes. After the examination of all possible matching combinations the best sub-solution (i.e., the solution with the maximum number of mapped nodes and the minimum number of differences between them) is appended to the final solution.

The detection of the *Maximum Common Subgraph* is a well known NP-complete problem for which several optimal and suboptimal algorithms have been proposed in the literature. Conte et al. [14] compared the performance of the three most representative optimal algorithms, which are based on depth-first tree search. All three algorithms have a factorial worst case time complexity with respect to the number of nodes in the graphs, in the order of $\frac{(N_2+1)!}{(N_2-N_1+1)!}$, where N_1 and N_2 are the numbers of nodes in graphs G_1 and G_2, respectively [14]. The differences among the three algorithms actually lie only in the information used to represent each state of the search space, and in the kind of the heuristic adopted for pruning search paths [14].

The reason we apply this divide-and-conquer approach is that the direct application of a MCS algorithm on the original problem (i.e., the complete graphs) is likely to cause

```
1  Function PDGMapping(CDT_i, CDT_j)
       Data: Two isomorphic CDTs
       Result: The final mapping solution as finalSolution
2      level_i ← CDT_i.maxLevel
3      level_j ← CDT_j.maxLevel
       /* an initially empty solution      */
4      finalSolution ← ∅
5      while level_i ≥ 0 and level_j ≥ 0 do
6          cpNodes_i ← nodes at level_i of CDT_i
7          cpNodes_j ← nodes at level_j of CDT_j
8          foreach cp_i ∈ cpNodes_i do
9              mcsStates ← ∅
10             foreach cp_j ∈ cpNodes_j do
11                 if compatibleAST(cp_i.parent, cp_j.parent)
12                 and compatibleAST(cp_i, cp_j) then
13                     mapping ← (cp_i, cp_j)
14                     root ← createState(mapping)
15                     search(root, mapping)
16                     get the maximum common subgraph
                       from root & add it to mcsStates
17                 end if
18             end foreach
19             select the best state from mcsStates &
               append it to finalSolution
20         end foreach
21         decrement level_i
22         decrement level_j
23     end while
24 end
```

Algorithm 2: A divide-and-conquer PDG mapping process based on control dependence structure.

a combinatorial explosion. As the number of possible matches for the nodes increases, the width of the search tree constructed by the MCS algorithm grows rapidly as a result of the numerous combinatorial considerations to be explored. In order to reduce the risk of combinatorial explosion, we decided to take advantage of the control dependence structure of the duplicated code fragments.

Figure 3 shows the CDTs for the duplicated code fragments of Figure 1. In level 2 of the CDTs, node 67 on the left side can be mapped to nodes 68, 71, 80, and 83 on the right side. Consequently, there are four possible matching nodes for node 67 and four node pairs to be used as starting points. All sub-solutions resulting from the aforementioned starting points have the same number of mapped nodes, but only the sub-solution generated from starting point (67, 80) has the minimum number of differences (equal to zero).

For the implementation of our MCS search technique (Algorithm 3), we have adopted the McGregor algorithm [15], because it is simpler to implement and has a lower space complexity, in the order of $O(N_1)$, since only the states associated to the nodes of the currently explored path need to be stored in memory. The other two algorithms require the construction of the association graph between the two given

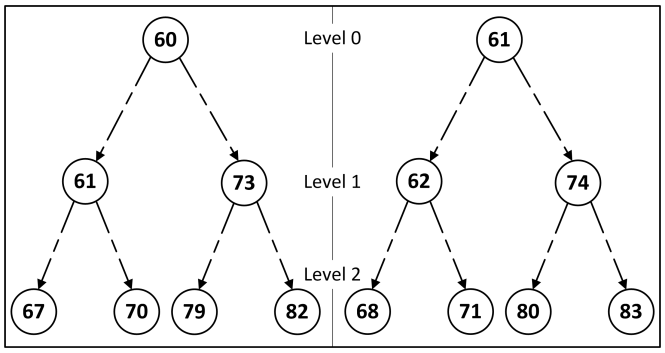


Fig. 3. The Control Dependence Trees for the code fragments of Figure 1.

```
1  Function search(pState, nodeMapping)
       Data: pState represents a parent state in the tree
       nodeMapping represents a pair of PDG nodes
       (node_i, node_j) that have been already mapped
       Result: Builds recursively a search tree.
       The leaf nodes in the deepest level are states
       corresponding to maximum common subgraphs
       /* get incoming & outgoing edges    */
2      Edges_i ← node_i.inEdges ∪ node_i.outEdges
3      Edges_j ← node_j.inEdges ∪ node_j.outEdges
4      foreach edge_i ∈ Edges_i do
5          if edge_i ∉ pState.visitedEdges then
6              add edge_i → pState.visitedEdges
7              foreach edge_j ∈ Edges_j do
8                  if compatibleEdges(edge_i, edge_j) then
9                      vN_i ← edge_i.otherEndPoint
10                     vN_j ← edge_j.otherEndPoint
11                     if compatibleAST(vN_i, vN_j) and
12                     not alreadyMapped(vN_i) and
13                     not alreadyMapped(vN_j) then
14                         mapping ← (vN_i, vN_j)
15                         state ← createState(mapping)
16                         if not pruneBranch(state) then
17                             add state → pState.children
18                             search(state, mapping)
19                         end if
20                     end if
21                 end if
22             end foreach
23         end if
24     end foreach
25 end
```

Algorithm 3: Recursive function building a search tree.

graphs, which in the worst case can be a complete graph with a space complexity in the order of $O(N_1 \cdot N_2)$. Given two PDGs, namely PDG_i and PDG_j, Algorithm 3 enforces the following constraints:

1) An edge of PDG_i is traversed only once in each path of the search tree (line **5**).
2) A node from PDG_i is mapped to only one node from PDG_j in each path of the search tree (lines **12** and **13**).

Algorithm 3 builds recursively a search tree by visiting the pairs of mapped PDG nodes in depth-first order. Each node in the search tree is created when a new pair of PDG nodes is mapped and represents a state of the search space. Each state keeps track of all visited edges and mapped PDG nodes in its path starting from the root state (function *createState* copies the visited edges and mapped nodes from the parent state to the child state). Function *pruneBranch* (line **16**) examines the existence of other leaf states in the search tree that already contain the node mappings of the newly created state. In such a case, the branch starting from the newly created state is pruned (i.e., not further explored). The reason we added this condition is because we realized that in several cases the search algorithm was building branches containing exactly the same node mappings, but in different order. The leaf states in the deepest level of the search tree correspond to the maximum common subgraphs.

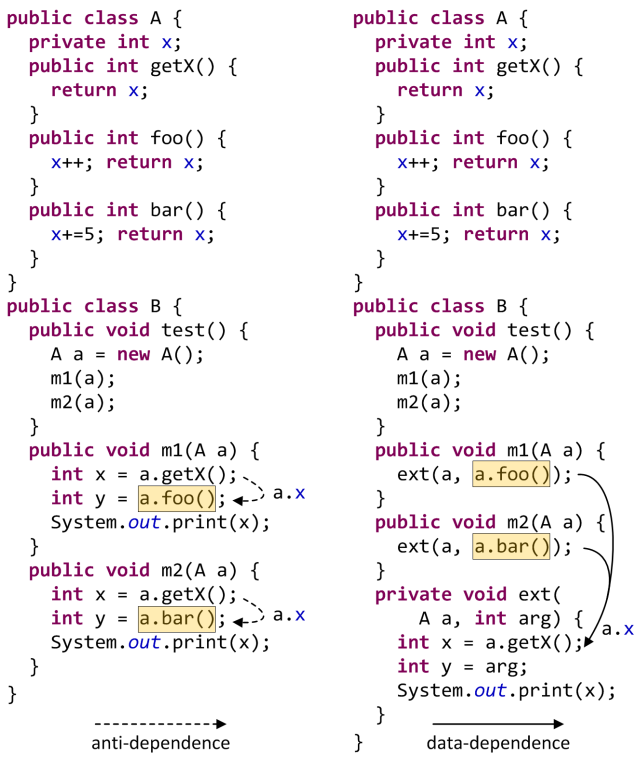
C. Examined Preconditions

After the completion of the matching process, we need to determine whether the duplicated code can be safely extracted into a common method. According to Opdyke [16], each refactoring should be accompanied with a set of *preconditions*, which ensure that the behavior of a program is preserved by the refactoring. If any of the preconditions is violated, then the refactoring is not applicable, or its application would cause a change in the program behavior. In this section, we define a set of preconditions that should be examined before the refactoring of duplicated code.

Precondition 1: The parameterization of differences between the matched statements should not break existing data-, anti-, and output-dependences.

In order to extract the duplicated code into a common method, the differences between the matched statements should be parameterized. Essentially, this means that the expressions being different should be passed as arguments to the extracted method call, and therefore these expressions will be evaluated (or executed) before the execution of the extracted duplicated code. Obviously, a change in the evaluation or execution order of the parameterized expressions could cause a change in the program behavior.

In Figure 4(a), methods `m1` and `m2` in class `B` contain exactly the same code with the exception of a difference in method calls `a.foo()` and `a.bar()`. In the first statement, both methods call `a.getX()` to read attribute `x` from object reference `a`. In the next statement, the value of attribute `x` is modified through method calls `a.foo()` and `a.bar()`, respectively. As a result, there exists an *anti-dependence* due to variable `a.x` from the first to the second statement of methods `m1` and `m2`, respectively. In order to merge the duplicated code, the common statements are extracted in method `ext()`, as shown in Figure 4(b), and expressions `a.foo()` and `a.bar()` are passed as arguments in the calls of the extracted method. This transformation breaks the previously existing anti-dependence, since after the



(a) Before refactoring (b) After refactoring

Fig. 4. Parameterization of a difference breaking an existing anti-dependence.

refactoring, variable `a.x` is first modified and then read. As a matter of fact, a new inter-procedural data-dependence due to variable `a.x` is introduced after refactoring. The breaking of the original anti-dependence is causing a change in the behavior of the program. In the original version in Figure 4(a), the execution of method `test` results in `m1` printing `0` and `m2` printing `1`, while in the refactored version in Figure 4(b) the execution of method `test` results in `m1` printing `1` and `m2` printing `6`. In a similar manner, the breaking of data-dependences or output-dependences could also cause a change in the program behavior.

Precondition 2: The unmatched statements should be movable before or after the matched statements without breaking existing data-, anti-, and output-dependences.

Obviously, the statements within the duplicated code fragments that have not been matched cannot be extracted along with the matched statements. As a result, they have to be moved either before or after the execution of the extracted method. The move of unmatched statements should not break existing data-, anti-, and output dependences, as in the case of Precondition #1. Under certain circumstances, the unmatched statements could remain in their original position by applying the Template Method design pattern [17]. However, this is applicable only when the duplicated code fragments belong to subclasses of the same superclass, the unmatched statements are "symmetrically" placed in the same level within the control structures of the duplicated

code fragments, and the unmatched statements can form a method returning at most one variable of the same type [18].

Precondition 3: The duplicated code fragments should return at most one variable of the same type.

In Java a method can return the value of one variable at most, since all method arguments are passed by value. As a result, the duplicated code fragments should return at most one variable to the original method from which they are extracted. In the case of multiple variables being returned by the duplicated code fragments, an alternative approach for refactoring could be to decompose the original clones into sub-clones having a distinct functionality [12] (i.e., each sub-clone contains the statements required for the computation of a single variable). However, this kind of decomposition might not eliminate completely the duplication, since some statements may be required for the computation of multiple variables.

Precondition 4: Matched branching (`break`, `continue`) statements should be accompanied with corresponding matched loop statements.

In Java the unlabeled `break` statement is used to terminate the innermost `for`, `while`, or `do-while` loop. The unlabeled `continue` statement is used to skip the current iteration of the innermost `for`, `while`, or `do-while` loop. As a result, when two branching statements are matched the corresponding loops should be also matched. Otherwise the extraction of a branching statement without the corresponding loop would cause a compilation error.

V. Evaluation

To evaluate the effectiveness of our approach on the unification and refactoring of clones, we compared it against CeDAR [9], a state-of-the-art clone refactoring tool. Tairas and Gray [9] created a benchmark containing the *Type-2* clones detected in 9 open-source projects by Deckard [10]. In their evaluation, they compared CeDAR against Eclipse IDE on a subset of the clone groups detected by Deckard. More specifically, they considered only the clone groups in which all clones of the group belong to the same Java file. The reason behind the exclusion of the clone groups in which the clones are scattered in different classes is that CeDAR essentially extends the refactoring engine of Eclipse, which supports only the basic Extract Method refactoring (i.e., the extraction of a duplicated code fragment from methods in the same class) and not more advanced refactorings, such as the extraction of a duplicated code fragment from methods in different subclasses and its move to a new method in the common superclass (Extract and Pull Up Method refactoring).

For the sake of a fair comparison, we will divide our evaluation in two parts. In the first part, we will focus only on the clone groups in which all clones belong to the same file, to make possible a direct comparison with CeDAR. In the second part, we will report the results of our approach for the clone groups in which the clones are scattered in different files.

Within the context of the experiment we applied the following work flow. For each clone group reported by Deckard and for each clone within the group we generated the CDT representing its control dependence structure. The first step of our approach (i.e., finding isomorphic subtrees within the original CDTs given as input) was skipped, since Deckard always returns clones having an identical control dependence structure. Next, we applied the two subsequent steps of our approach, namely the mapping of the Program Dependence Subgraphs corresponding to the clones (Section IV-B), and the examination of preconditions (Section IV-C). In the case where a clone group had more than two clones, then the aforementioned work flow was applied to all possible pairs of clones within the group. A clone group was considered refactorable if the total list of violated preconditions resulting from all possible pairs of clones within the group was empty.

Table III presents the number of clone groups assessed as refactorable by CeDAR and JDeodorant (the tool implementing the proposed technique), respectively. Projects Hibernate 3.3.2 and Squirrel-SQL 3.0.3 have been excluded from the analysis, because the Deckard clone detection results provided by [9] did not correspond to the source code of the aforementioned versions. We have contacted the authors about this issue, but they were not able to provide the actual source code versions from which the clone detection results were derived. Table IV presents additional clone groups (in which clones belong to different Java files) assessed as refactorable by JDeodorant.

TABLE III
Refactorable Clone Groups Found by CeDAR, and JDeodorant

Project	**CG_1** *	CeDAR	JDeodorant	Δ
Apache Ant 1.7.0	120	28 (23%)	50 (42%)	+79%
Columba 1.4	88	30 (34%)	41 (47%)	+37%
EMF 2.4.1	149	14 (9%)	54 (36%)	+286%
JMeter 2.3.2	68	11 (16%)	20 (29%)	+82%
JEdit 4.2	157	20 (13%)	57 (36%)	+185%
JFreeChart 1.0.10	291	62 (21%)	87 (30%)	+40%
JRuby 1.4.0	81	23 (28%)	35 (43%)	+52%
Total	954	188 (20%)	344 (36%)	+83%

* column CG_1 refers to the total number of clone groups in which all clones belong to the same Java file.

TABLE IV
Additional Refactorable Clone Groups Found by JDeodorant

Project	**CG_2** *	JDeodorant
Apache Ant 1.7.0	211	42 (20%)
Columba 1.4	275	66 (24%)
EMF 2.4.1	58	12 (21%)
JMeter 2.3.2	225	68 (30%)
JEdit 4.2	101	21 (21%)
JFreeChart 1.0.10	337	121 (36%)
JRuby 1.4.0	181	43 (24%)
Total	1388	373 (27%)

* column CG_2 refers to the total number of clone groups in which clones belong to different Java files.

A. Discussion

As it can be observed in Table III, JDeodorant was able to find a significantly larger number of refactorable clone groups compared to CeDAR in all examined projects. More specifically, on average JDeodorant found as refactorable 36% of the clone groups in which clones belong to the same file, while CeDAR found only 19.7%. This corresponds to an overall increase of 83% over CeDAR. In particular projects, such as EMF, JDeodorant found almost 3 times more refactorable clones than CeDAR. It should be noted that there was no case found by CeDAR that could not be found by JDeodorant. This significant increase in the percentage of refactorable clone groups can be mainly attributed to the applied AST matching mechanism, which enabled a more flexible unification of duplicated statements with non-trivial differences. Additionally, in some cases the minimization of differences in the mapping of the duplicated statements led to the elimination of precondition violations.

Table IV presents the number of additional clone groups that were found as refactorable by JDeodorant. These cases refer to clone groups in which clones belong to different Java files. On average, JDeodorant assessed as refactorable 27% of these clones groups. The difference in the percentage of clones within the same and different files found as refactorable (36% vs. 27%) probably indicates that the clones within the same files have a stronger similarity and their unification is easier.

Another interesting insight, presented in Table V, is that 7% of the total non-refactorable clone groups violate only precondition #3 (i.e., the duplicated code fragments return more than one variable). All these cases can be actually refactored by decomposing the original clones into sub-clones [12], where each sub-clone contains the statements required for the computation of a single returned variable. If these cases were considered as refactorable, the total percentage of refactorable clone groups found by JDeodorant would be equal to 36% (837 out of 2342 clone groups in total).

TABLE V
NON-REFACTORABLE CLONE GROUPS VIOLATING ONLY PRECOND. #3

Project	**NR***	Violations
Apache Ant 1.7.0	239	27 (11%)
Columba 1.4	256	7 (3%)
EMF 2.4.1	141	6 (4%)
JMeter 2.3.2	205	6 (3%)
JEdit 4.2	180	14 (8%)
JFreeChart 1.0.10	420	55 (13%)
JRuby 1.4.0	184	5 (3%)
Total	1625	120 (7%)

* column NR refers to the total number of non-refactorable clone groups, which is equal to (CG_1 + CG_2) minus the total number of refactorable clone groups found by JDeodorant.

B. Threats to Validity

A major threat to the external validity of the study is related to the use of a single clone detection tool (i.e., Deckard) for the collection of clones. It is reasonable to expect that other clone detection tools might be able to detect more advanced clones, which are possibly harder to refactor. Additionally, we cannot assume that the clones reported by Deckard constitute representative cases that would be detected by the majority of the clone detection tools. However, the reason we selected this particular tool was to make possible a direct comparison with a competitive clone refactoring tool (i.e., CeDAR) on the same dataset that was used for its evaluation [9].

Another possible threat to the external validity of the study is the inability to generalize our findings beyond the examined open-source systems. The systems used in the evaluation of our technique exhibit a variation in both their size, ranging from 50 to 120 KLOC, as well as in their application domain, including a build tool (Ant), an email client (Columba), a modeling framework (EMF), a server performance testing tool (JMeter), a text editor (JEdit), a chart library (JFreeChart), and a programming language (JRuby). These two variation points certainly affect the characteristics of the detected clones with respect to their size and domain-specificity, allowing for more generalizable results.

VI. RELATED WORK

We organized related work in two groups, namely PDG mapping techniques and clone refactoring techniques, since the proposed approach deals with both research problems.

A. PDG Mapping Techniques

Komondoor and Horwitz [19] apply slicing on PDGs to find isomorphic subgraphs that represent code clones. The advantage of this approach is the detection of non-contiguous clones (i.e., clones with gaps), clones with re-ordered statements, and clones intertwined with each other. Two nodes are matched if the corresponding statements are syntactically identical (i.e., their AST representation has the same structure) allowing only differences in variable names and literal values.

Krinke [20] proposed an approach to identify code clones by finding the maximal similar subgraphs in two PDGs by induction from a pair of starting vertices. To reduce the complexity of the algorithm, he considers only a subset of vertices (i.e., predicate vertices) as starting points, and restricts the maximum length of the explored paths using a *k*-limit. One important limitation is that the running time of the algorithm explodes as *k*-limit increases. Another limitation is that the use of *k*-limit may lead to an incomplete solution (i.e., the selected *k*-limit may be insufficient for detecting all possible matching vertices).

Shepherd et al. [21] implemented an automated aspect mining technique exploiting the PDG and AST representations of a program. The proposed algorithm, inspired by [20] and [19], starts by matching the control dependence subgraphs of two compared PDGs to extract all possible matching solutions. Next, it filters out the undesirable matching solutions based on data dependence information. A limitation is that the algorithm always starts from the method entry nodes, and thus will fail to match control dependence subgraphs nested in different levels.

Higo and Kusumoto [22] improved Komondoor's technique [19] by extending the PDG representation and introducing

some heuristics to enhance code clone detection. More specifically, they introduced new edges called *execution-next links* to improve the ability to detect contiguous code, and merged the directly-connected equivalence nodes in order to reduce the computational cost of identifying isomorphic subgraphs.

Speicher and Bremm [23] propose a stepwise unification process of *Type-3* clones based on PDG mapping. They also suggest that differences in expression operators can be parameterized using lambda expressions (a feature introduced in Java 8). The process of statement unification allows for differences in the identifiers of local variables, parameters, fields, and method calls, differences in literals, differences in the types of declared objects, and finally, differences in the order of parameters in method calls.

The common limitation of all aforementioned techniques is that they do not explore the entire search space of possible solutions and therefore may return a non-optimal solution. In contrast to MCS approach that builds a search tree examining all possible combinations in the case of multiple node matches, the aforementioned techniques always select one match for each node, essentially exploring only a single path of the entire search tree. In addition, the applied node matching process allows only for differences in variable names and literal values, thus missing potential node matches that would lead to a better solution.

B. Clone Refactoring Techniques

Higo et al. [24] proposed a metric-based approach to suggest refactoring opportunities for software clones. The proposed metrics take as input a set of clones and quantify properties such as the degree of coupling in terms of the number of referenced and assigned variables within the clones, and the dispersion of the clones in the class hierarchy. The refactoring opportunities (Extract or Pull Up Method) are suggested by checking some conditions, which compare the metric values with some fixed thresholds.

Choi et al. [25] proposed a method combining clone metrics to extract code clones for the purpose of refactoring. These metrics take as input a set of clones and compute the average length of token sequences within the clones, the size of the clone set, and the ratio of non-repeated token sequences within the clones. The metric values are combined in order to rank the clone sets detected in a system. The top ranked clone sets constitute more interesting refactoring opportunities according to a case study.

Tairas and Gray [9] extended the Eclipse refactoring engine to enable the processing of more types of differences among duplicated code fragments, such as differences in field accesses, string literals, and method calls without arguments, in addition to differences in local variable identifiers. The evaluation on the *Type-2* clones detected in 9 open-source projects using the Deckard clone detection tool [10] revealed that the aforementioned enhancements in the matching of duplicated code increased the percentage of refactorable clones from 10.6% to 18.7% on average. The authors mention as future extensions the addition of more parameterized differences (e.g., local variable identifiers replaced with method calls), the support of additional types of refactorings for clones belonging in different classes, and the creation of a mechanism for the refactoring of sub-clones.

Hotta et al. [18] proposed an approach to refactor *Type-3* clones by introducing an instance of the Template Method design pattern [17]. Their technique detects isomorphic subgraphs in the PDGs of two methods containing the clones. Next, the isomorphic subgraphs that constitute the *common process* of the examined methods are pulled up in a method of the base class. The remaining code fragments that do not belong to the detected isomorphic subgraphs (i.e., unmatched statements) constitute the *unique processes* and are extracted into methods within each derived class. For each pair of *unique processes* introduced in the derived classes an abstract method is created in the base class that is called from the *common process* at the point where the corresponding unmatched statements were found.

Bian et al. [26] proposed SPAPE, a semantic-preserving amorphous procedure extraction technique, for the automatic refactoring of *Type-3* (near-miss) clones. SPAPE applies amorphous transformation on the PDGs of the clones in order to replicate `if` predicates and partition loop structures. The unmatched statements are merged by inserting control variables and conditional statements in the AST of the extracted procedure. SPAPE supports procedural code written in the *C* programming language.

Goto et al. [27] proposed an approach based on slice-based cohesion metrics for the merging of clones. Their approach takes as input a pair of similar methods and first detects syntactic differences between them through AST differencing. Next, it finds pairs of code fragments within the methods that constitute valid Extract Method candidates (the validation is performed by examining a set of preconditions). Finally, the detected candidates are ranked based on slice-based cohesion metrics. Highly cohesive candidates can be extracted as modules implementing a single feature.

VII. Conclusions and Future Work

This work is the first step towards a broader research goal, namely assisting developers in the refactoring of clones. To this end, we developed a clone unification and refactoring technique that overcomes the limitations of previous approaches. More specifically, the proposed technique can detect and parameterize a larger set of non-trivial differences between the clones, it can process clones that do not have an identical control dependence structure or do not expand over a valid block region, it can find an optimal mapping between the statements of the clones that minimizes the number of differences in the mapped statements, and finally it examines a set of preconditions to determine whether a clone group can be safely refactored without altering program behavior. Currently, we are working on an interactive visualization of clone pairs, where the developer can be informed about the differences between the clones causing precondition violations (i.e., the differences hindering the safe refactoring of the clones), and

also get suggestions about changes that could be applied in order to make the clones refactorable.

In the evaluation of our approach, we performed a direct comparison with CeDAR [9], a tool specialized in the refactoring of *Type-2* clones, on a set of 2342 clone groups detected in 7 open-source projects. Our technique managed to find 83% more refactorable clone groups than CeDAR, and additionally assessed as refactorable 27% of the clone groups in which clones belong to different Java files (a feature not supported by CeDAR). Furthermore, the study revealed that 36% of the clone groups in the examined projects can be refactored **directly** or in the form of **sub-clones** (Section V-A). This result is an encouraging starting point for further research developments in the refactoring of clones.

As future work we are planning to extend the evaluation of the proposed technique on more challenging cases of *Type-3* clones. To achieve this goal, we will first create a refactoring benchmark of *Type-3* clones in open-source projects using state-of-the-art tools specialized in the detection of *Type-3* clones [28]. Next, we will develop a decision tree to determine the most appropriate refactoring strategy based on the particular characteristics of the unmatched statements in gaps. For example, if the statements in the gaps cannot be moved before or after the clones, then we should consider more complex refactoring transformations, such as the introduction of the Template Method design pattern. Additionally, we are planning to extend our AST matching mechanism in order to support the matching of different types of control predicate statements. For example, there may exist clones in which a traditional `for` loop has been replaced with an equivalent enhanced `for` or a `while` loop, or a chain of `if`/`else if` conditional statements has been replaced with an equivalent `switch` statement. Finally, we are also planning to apply expression transformation techniques [29] in order to support the unification of semantically equivalent expressions that have a different syntax. In this way, we could further reduce the number of expressions that require parameterization.

ACKNOWLEDGMENT

The authors would like to thank NSERC and the Faculty of Engineering and Computer Science at Concordia University for their generous support.

REFERENCES

[1] C. K. Roy and J. R. Cordy, "A survey on software clone detection research," School of Computing, Queens University, Tech. Rep., 2007.

[2] A. Lozano and M. Wermelinger, "Assessing the effect of clones on changeability," in *Proceedings of the 24th IEEE International Conference on Software Maintenance*, 2008, pp. 227–236.

[3] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner, "Do code clones matter?" in *Proceedings of the 31st International Conference on Software Engineering*, 2009, pp. 485–495.

[4] M. Mondal, C. K. Roy, and K. A. Schneider, "An empirical study on clone stability," *SIGAPP Appl. Comput. Rev.*, vol. 12, no. 3, pp. 20–36, Sep. 2012.

[5] 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.

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

[7] N. Tsantalis and G. Panamoottil Krishnan, "Refactoring clones: A new perspective," in *Proceedings of the 7th International Workshop on Software Clones*, 2013, pp. 12–13.

[8] G. Panamoottil Krishnan and N. Tsantalis, "Refactoring clones: An optimization problem," in *Proceedings of the 29th IEEE International Conference on Software Maintenance ERA Track*, 2013.

[9] 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.

[10] 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*, 2007, pp. 96–105.

[11] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Trans. Program. Lang. Syst.*, vol. 9, no. 3, pp. 319–349, Jul. 1987.

[12] 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.

[13] G. Valiente, *Algorithms on Trees and Graphs*. Springer-Verlag, 2002.

[14] D. Conte, P. Foggia, and M. Vento, "Challenging complexity of maximum common subgraph detection algorithms: A performance analysis of three algorithms on a wide database of graphs," *J. Graph Algorithms Appl.*, vol. 11, no. 1, pp. 99–143, 2007.

[15] J. J. McGregor, "Backtrack search algorithms and the maximal common subgraph problem," *Software: Practice and Experience*, vol. 12, no. 1, pp. 23–34, 1982.

[16] W. F. Opdyke, "Refactoring object-oriented frameworks," Ph.D. dissertation, University of Illinois at Urbana-Champaign, IL, USA, 1992.

[17] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[18] 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.

[19] 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.

[20] J. Krinke, "Identifying similar code with program dependence graphs," in *Proceedings of the Eighth Working Conference on Reverse Engineering*, 2001, pp. 301–307.

[21] D. Shepherd, E. Gibson, and L. L. Pollock, "Design and evaluation of an automated aspect mining tool," in *Proceedings of the International Conference on Software Engineering Research and Practice*, 2004, pp. 601–607.

[22] 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.

[23] D. Speicher and A. Bremm, "Clone removal in java programs as a process of stepwise unification," in *Proceedings of the 26th Workshop on Logic Programming*, 2012.

[24] 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.

[25] 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.

[26] Y. Bian, G. Koru, X. Su, and P. Ma, "SPAPE: A semantic-preserving amorphous procedure extraction method for near-miss clones," *Journal of Systems and Software*, vol. 86, no. 8, pp. 2077–2093, 2013.

[27] A. Goto, N. Yoshida, M. Ioka, E. Choi, and K. Inoue, "How to extract differences from similar programs? A cohesion metric approach," in *Proceedings of the 7th International Workshop on Software Clones*, 2013.

[28] R. Tiarks, R. Koschke, and R. Falke, "An extended assessment of type-3 clones as detected by state-of-the-art tools," *Software Quality Journal*, vol. 19, no. 2, pp. 295–331, 2011.

[29] M. Harman, L. Hu, M. Munro, X. Zhang, D. Binkley, S. Danicic, M. Daoudi, and L. Ouarbya, "Syntax-directed amorphous slicing," *Automated Software Engineering*, vol. 11, no. 1, pp. 27–61, 2004.