IMPROVING THE UNIFICATION OF SOFTWARE CLONES USING TREE AND GRAPH MATCHING ALGORITHMS

Giri Panamoottil Krishnan

A thesis

in

The Department

of

Computer Science

Presented in Partial Fulfillment of the Requirements

For the Degree of Master of Computer Science

Concordia University

Montréal, Québec, Canada

April 2014

© Giri Panamoottil Krishnan, 2014

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: **Giri Panamoottil Krishnan**

Entitled: **Improving the Unification of Software Clones using Tree and Graph Matching Algorithms**

and submitted in partial fulfillment of the requirements for the degree of

Master of Computer Science

complies with the regulations of the University and meets the accepted standards with respect to originality and quality.

Signed by the final examining commitee:

______________________	Chair
Dr. Joey Paquet ______________________	Examiner
Dr. Peter Rigby ______________________	Examiner
Dr. Nikolaos Tsantalis ______________________	Supervisor

Approved by ______________________

Chair of Department or Graduate Program Director

____

Dean of Faculty

Date ______________________

Abstract

Improving the Unification of Software Clones using Tree and Graph Matching Algorithms

Giri Panamoottil Krishnan

Code duplication is common in all kind of software systems and is one of the most troublesome hurdles in software maintenance and evolution activities. Even though these code clones are created for the reuse of some functionality, they usually go through several modifications after their initial introduction. This has a serious negative impact on the maintainability, comprehensibility, and evolution of software systems.

Existing code duplication can be eliminated by extracting the common functionality into a single module. In the past, several techniques have been developed for the detection and management of software clones. However, the unification and refactoring of software clones is still a challenging problem, since the existing tools are mostly focused on clone detection and there is no tool to find particularly refactoring-oriented clones. The programmers need to manually understand the clones returned by the clone detection tools, decide whether they should be refactored, and finally perform their refactoring. This obvious gap between the clone detection tools and the clone analysis tools, makes the refactoring tedious and the programmers reluctant towards refactoring duplicate codes.

In this thesis, an approach for the unification and refactoring of software clones that overcomes the limitations of previous approaches is presented. More specifically, the proposed technique is able to detect and parameterize non-trivial differences between the clones. Moreover, it can find a mapping between the statements of the clones that minimizes the number of differences. We have also defined preconditions in order to determine whether the duplicated code can be safely refactored to preserve the behavior of the existing code. 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.

Acknowledgments

Foremost, I would like to express my sincere gratitude to my advisor Dr. Nikolaos Tsantalis. His enthusiasm and motivation helped me throughout the research and writing of this thesis. His proper guidance helped me publish papers in major conferences. He continuously supported me in improving my knowledge and ability to do the research.

Besides my advisor, I would like to thank my thesis examiners: Dr. Joey Paquet and Dr. Peter C. Rigby. I would like to thank other faculty members of the Department of Computer Science and Software Engineering as well, for the necessary guidance. I also take this opportunity to thank Institute for Co-Operative Education, Concordia for giving me a chance to explore the professional world of Canada.

I thank my fellow lab mates and my friends in Concordia University for all their support towards the completion of this thesis. I express my gratitude to the staff members of our university for their help in providing a clean and safe environment for me to work. Also, I would like to thank my family and my roommates for supporting me throughout my endeavors.

Last but not the least, I would like to thank NSERC and the Faculty of Engineering and Computer Science at Concordia University for their generous financial support of this project.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter introduces the preliminary concepts and motivation behind the thesis. It also briefly discusses how we tackle the challenges involved.

1.1 Software Maintenance

Maintenance and enhancement activities of a software application constitute a major share of the software life cycle. According to McKee [McK84], at least two thirds of the software activities are attributable to maintaining existing applications rather than creating new applications. Also, various surveys show that application program maintenance expenditures represent about 40-75% of the total application program expenditures [Gui83]. Software maintenance deals with the modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the software to a changed environment. Followed by the initial definition of maintenance activities by Swanson [Swa76], four categories of maintenance activity are distinguished as follows [BS87].

1. **Corrective** - consists of activities of repairing faults to keep the system operational.

2. **Adaptive** - initiated as a result of changes in the environment in which the system must operate.

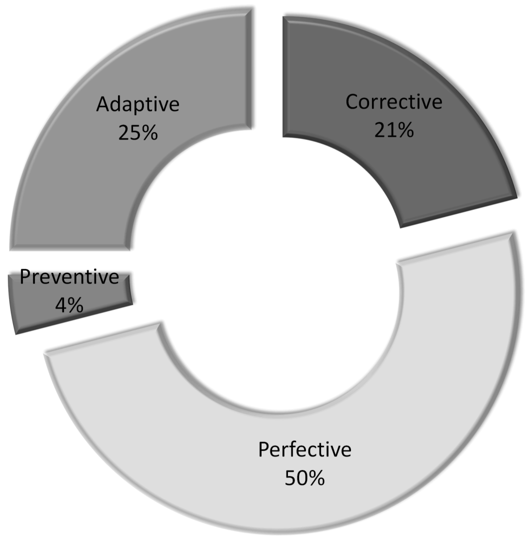


Figure 1: Maintenance activities

3. **Perfective -** includes all changes and enhancements which are made to the system to meet the new or changed user requirements.

4. **Preventive -** concerns activities aimed at improving the future maintainability and reliability of the software system.

Lientz and Swanson [LS80] did a survey on software maintenance practices at 487 companies and found that most maintenance is perfective. At the same time, many empirical studies show that preventive maintenance corresponds to less than 5% of the total maintenance costs. The distribution of maintenance activities is shown in the Figure 1 [MM83] [SPL03] [Vli08]. The chart clearly shows that software industries invest more on maintenance activities leading to immediate benefits rather than preventive maintenance which leads to long-term benefits. This may be due to the lack of availability of efficient and user friendly tools aiming preventive maintenance, in the market. This thesis work focuses on the preventive maintenance of software systems as it deals with the refactoring of existing code to improve the future maintainability of the software.

A major problem which can evolve from poor maintenance activities is *Legacy Crisis*. The design quality of a software system tends to deteriorate with software aging. As a result, the companies invest more on maintenance activities, especially in modernizing legacy systems. If this trend continues, there will be no resources left to develop new systems, and this reality is referred to as *Legacy Crisis* by Seacord et al. [SPL03]. One counteract to avoid this is improving the maintainability of the software through Preventive Maintenance.

1.2 Refactoring

"Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code, yet improves the internal structure" [Fow99]. In other words, refactoring is done to improve the design of the existing code. Refactoring is supported by different IDEs like Eclipse. Programmers use two types of tactics to apply refactoring - *floss refactoring* and *root-canal refactoring* [MHPB09]. *Floss refactoring* is frequent and it is intermingled with other kind of program changes in order to maintain the code healthy. *Root-canal refactoring* is an infrequent long activity done exclusively to fix unhealthy code. Murphy-Hill and Black [MHB08] suggest *floss refactoring* as the recommended tactic for the preventive maintenance of the code.

In refactoring process, the bad code is improved and thus the software quality is increased. Fowler [Fow99] calls the occurrences of substandard code quality as '**bad smells**'. Among the 22 bad smells identified [Fow99], he mentions '*Duplicated Code*' as "the number one in stink parade". We concentrate on this bad smell and here onwards, we call '*Duplicated Code*' as '*Code Clone*'. The next section explains why code clones are a major problem in software programs.

1.3 Software Code Clones

Code duplication happens when a code fragment has other code fragments identical or similar to it somewhere in the software system. Duplicated code or code clones are common in all kinds of software systems. Typically, 10-15% of the software comprises code clones [KG08]. This is mainly because programmers find it easier to use copy-paste technique during programming in order to reuse some functionality. But, this causes software to be less maintainable. There is empirical evidence that duplicated code makes the maintenance of software more difficult.

1. Duplicated code increases significantly the maintenance effort and cost [LW08]

2. It is associated with error-proneness due to the inconsistent changing of clones [JDHW09]

3. It is more unstable than non-duplicated code [MRS^{+}11] [MRS12] [MRS13].

Code duplication in general increases the code size and complexity, thus making the software maintenance difficult. This is often because any modification of original code should be applied to the code clones as well. For example, when enhancements or bug fixes are done on one copy of the duplicated code, we would need to find the other copies and fix them as well. This would be a major problem in software renovation projects and those projects where code changes are done quite often. If the developers forget to apply necessary changes in every instance of code clones, it will result in the inconsistent functioning of the software, which in turn causes expensive maintenance activities in future. The above mentioned reasons make code clones good candidates for software redesign. Therefore, it is of utmost importance to find code clones efficiently and remove them, i.e., to refactor the code such that only one copy of the code fragment is maintained.

Four types of code clones have been defined by the software clone community [DBFF95] [KFF06] [RC07].

1. **Type-1 clones** - Exactly identical code fragments except for differences in white spaces and comments.

2. **Type-2 clones** - Similar code snippets with variations in identifiers, literals and types.

3. **Type-3 clones** - *Gapped* code fragments in which statements(one or more) have been added/deleted or modified beyond syntactic similarity.

4. **Type-4 clones** - Code fragments that perform the same calculation(logic) but with different syntax.

This thesis focuses on refactoring of the first three types of clones.

1.4 Software Clone Management

Software clone management comprises all activities of looking after and making decisions about consequences of copying and pasting [Kos08]. According to Koschke [Kos08], we can distinguish three main lines of clone management:

1. *preventive*, which comprises activities to avoid new clones,
2. *compensative*, which encompasses activities aimed at limiting the negative impact of existing clones that are to be left in the system and
3. *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 [DER07], incremental clone detection and clone consistency analysis [NNP$^+$12], and much less on the corrective aspect of clone management.

The best solution to avoid the existing duplicates of code fragments is to find a way to unify them. 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. Both detection and elimination of code clones have been investigated before. Fowler presented a catalog of different techniques for refactoring, which have been widely followed by the refactoring community. Some of them which are relevant to this thesis are explained below.

1. *Extract Method* - When the clones belong to the same class, they are extracted into a new method and all the clone copies are replaced by calls to the new method.
2. *Pull Up Method* - This can be applied when the code clones (Type-1 and Type-2) lie in sibling subclasses. They are extracted into a new method created in their parent class.
3. *Form Template Method* - When the code clones are Type-3 and exist in sibling subclasses, duplicated code fragments are pulled up to the base class and the unique fragments remain in the derived classes.

1.5 Challenges Involved

Many code clone detection tools are available to identify all types of duplicated code. But they do not guarantee that all code clones identified by them are *refactoring-oriented* code clones. Refactoring-oriented code clones are more suitable for refactoring than general code clones returned by the clone detection tools [HKI08]. Figure 2 shows an example of general code clones returned by a token-based clone detection tool. These clones cannot be directly refactored as discussed in the following sub-sections. So, the first challenge is to find code clones that can be unified for the purpose of refactoring. The major problems with using the output directly from code clone detection tools are listed below.

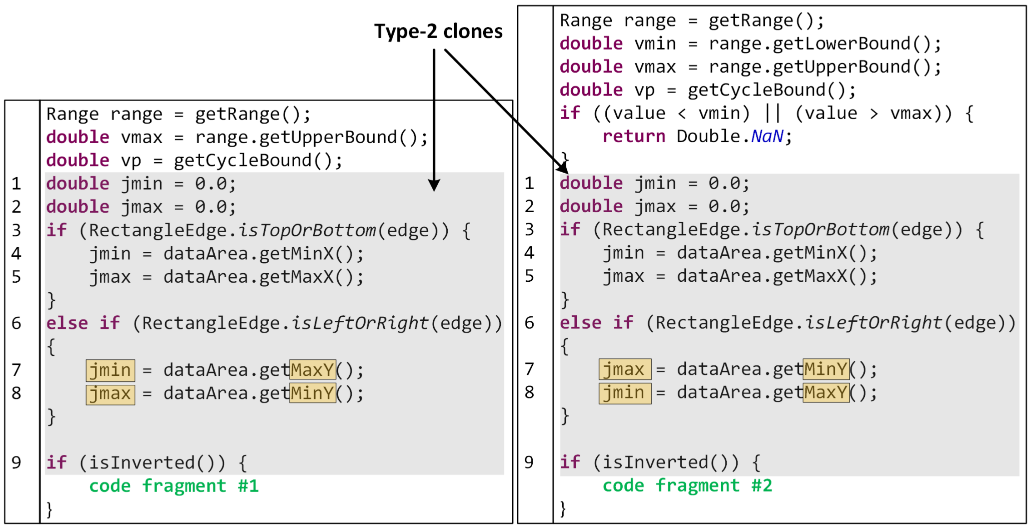
1.5.1 Determining Valid Clone Regions

Most existing clone refactoring techniques [TG12, HKI08] 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. The example shown in Figure 2 is a typical case of invalid clone regions (the last `if` statement is incomplete in both clones).

1.5.2 Optimal Statement Matching

The differences of mapped statements may not be minimal. For example in Figure 2, line number 7 in the first clone and line number 8 in the second clone are exactly the same. Likewise, line number 8 in the first clone and line number 7 in the second clone are exactly the same. But, due to the insufficient information from the results of clone detection tools, the refactoring tool tends to map the statements in order i.e., line 7 with line 7 and line 8 with line 8. This problem should be addressed before refactoring, in order to minimize the number of differences and thereby the number of required parameters while extracting the code fragment into a separate method.

Figure 3 shows another example where we need optimal matching of statements. The current

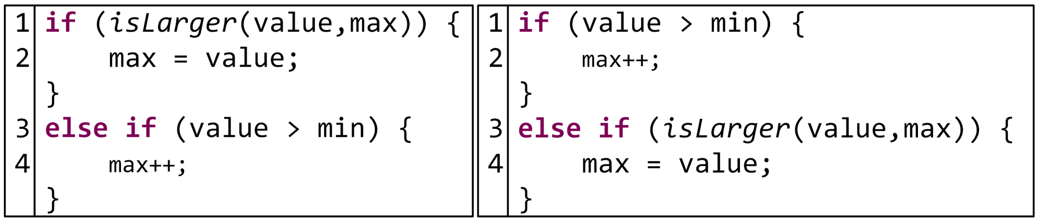


```
Range range = getRange();
double vmax = range.getUpperBound();
double vp = getCycleBound();
double jmin = 0.0;
double jmax = 0.0;
if (RectangleEdge.isTopOrBottom(edge)) {
    jmin = dataArea.getMinX();
    jmax = dataArea.getMaxX();
}
else if (RectangleEdge.isLeftOrRight(edge))
{
    jmin = dataArea.getMaxY();
    jmax = dataArea.getMinY();
}

if (isInverted()) {
    code fragment #1
}
```

```
Range range = getRange();
double vmin = range.getLowerBound();
double vmax = range.getUpperBound();
double vp = getCycleBound();
if ((value < vmin) || (value > vmax)) {
    return Double.NaN;
}
double jmin = 0.0;
double jmax = 0.0;
if (RectangleEdge.isTopOrBottom(edge)) {
    jmin = dataArea.getMinX();
    jmax = dataArea.getMaxX();
}
else if (RectangleEdge.isLeftOrRight(edge))
{
    jmax = dataArea.getMinY();
    jmin = dataArea.getMaxY();
}

if (isInverted()) {
    code fragment #2
}
```

Figure 2: Example of invalid clone regions (highlighted in gray)

tools do not support the matching of the two code fragments shown in Figure 3 as the conditional structures are reordered. Looking for an optimal mapping of statements in the entire clone fragments will help us to determine that these two code fragments can be mapped indeed.



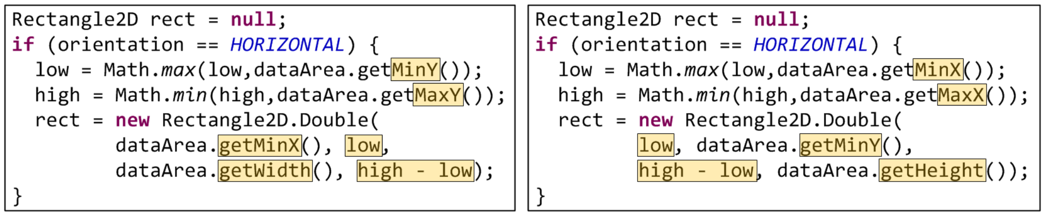
```
if (isLarger(value,max)) {
    max = value;
}
else if (value > min) {
    max++;
}
```

```
if (value > min) {
    max++;
}
else if (isLarger(value,max)) {
    max = value;
}
```

Figure 3: Example of reordered conditional structures

1.5.3 Non-trivial Differences

There are other challenges involved in unifying the code fragments. If *Type-1* clones are easy and straightforward to merge, *Type-2* clones often contain non-trivial 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 [TG12] introduced the parameterization of differences in field accesses, method calls and literals. This extended parameterization enables the refactoring of clones containing dissimilarities between different types of AST nodes (e.g., variables replaced with method calls). AST (Abstract Syntax Tree) is a tree-like representation of the source code where each node represents a code construct (Section 3.1.1). However, this approach would be ineffective in the example of Figure 4, 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).



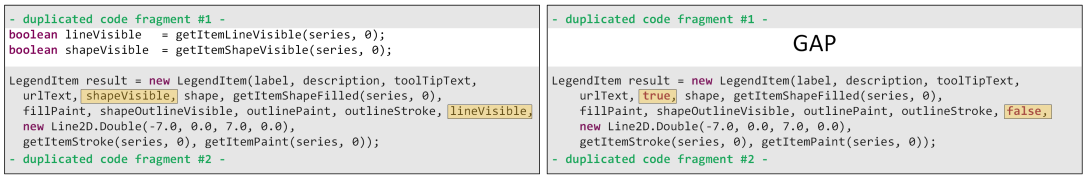
```
Rectangle2D rect = null;
if (orientation == HORIZONTAL) {
  low = Math.max(low,dataArea.getMinY());
  high = Math.min(high,dataArea.getMaxY());
  rect = new Rectangle2D.Double(
         dataArea.getMinX(), low,
         dataArea.getWidth(), high - low);
}
```

```
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());
}
```

Figure 4: Example requiring a more advanced parameterization of differences

1.5.4 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). An example for *Type-3* clones is shown in Figure 5. The clone on the left side of Figure 5 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. In order to extract the common statements, we need 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 data flow of the program. Another challenge with *Type-3* clones is that, if the clones lie in different classes, we need to apply the Form Template 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 -
```

Figure 5: Example of *Type-3* clones (highlighted in gray) with a gap

1.5.5 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 2, 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 [TC11]. 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 initially 8 statements before refactoring to just 2 statements after refactoring).

1.6 Motivation

There is a great potential for advancements in the research area of software clone refactoring. A recent study by Tairas and Gray [TG12] on the clones detected in 9 open-source Java projects using the Deckard [JMSG07] 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. This demands us to improve the algorithm behind the process. The main reason for the limited refactoring of both Eclipse IDE and CeDAR is they use the Eclipse JDT structure for performing the matching of Abstract Syntax Tree (3.1.1) of code fragments allowing only a small set of differences between them. Our approach improves this matching process by allowing non-trivial differences between the statements. Also, both Eclipse and CeDAR are not able to address the problems mentioned in Section 1.5.

Another aspect of clone refactoring which needs to be changed is the lack of tool support. After detecting refactoring-oriented clones, the next step is to aid the user in interpreting the clone information correctly. Even though IDEs like Eclipse support minimal refactoring, there is still no versatile tool available which helps the user to understand the similarities and differences of the clones precisely. The easier analysis of clones is relevant as the developers often find it time consuming and error prone to manually inspect the clones. Xing et al. [XXJ11] proposed a clone analysis tool called CloneDifferentiator which tries to characterize the clone based on the differences. This thesis aims to semi-automate the process of clone refactoring by giving the user a proper explanation of the clones differences, the reasons why clones are not refactorable and suggestions to make code refactorable.

1.7 Overview of the approach

This thesis presents a technique for refactoring of software clones in Java programs that tackles the limitations of the current state-of-the-art techniques. The proposed approach takes as input two code fragments or even entire methods that have been detected as clones by clone detection tools, and determines whether the clones or parts of them can be safely refactored. The three main steps involved in the process are the following. 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.

The proposed technique supports the refactoring of *Type-1* clones, *Type-2* clones and *Type-3* clones. The technique is compared with CeDAR [TG12], a state-of-the-art tool in the refactoring of *Type-2* clones. The same experiment that they performed on the clones detected by Deckard [JMSG07] is repeated in order to do a fair comparison. The results have shown that our approach is able to find 82% more refactorable clones than CeDAR in the 7 Java open-source projects examined. Also, a report on the additional refactorable clones found by the proposed technique is given.

1.8 Contributions

Clone refactorability analysis is assessing whether two input code fragments contain opportunities for refactoring. Our approach is the first of its kind that takes as input any code clone fragments detected by any clone detection approaches (e.g., *token-based* clone detection tools, *tree-based* clone detection tools) and finds refactoring opportunities inside them (code fragments having a similar control structures) and assesses if they can be safely refactored without altering the existing behavior of the program (i.e., the approach ensures that there are no side-effects on the program due to refactoring).

In summary, the contributions of the proposed technique are the following:

1. It supports the detection and parameterization of non-trivial differences between duplicated code fragments (Section 3.1).

2. 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 (Section 3.2).

3. 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 (Section 3.3).

4. It defines preconditions that can be used to determine whether a clone group is safe to be refactored (Section 3.4).

1.9 Thesis Outline

The remainder of this thesis is organized as follows. Chapter 2 provides the background and discusses the related research work. Section 3.1 describes our statement matching technique that is used in all our algorithms. The two major steps of the proposed clone unification technique are presented in Section 3.2 and Section 3.3. Section 3.4 explains the preconditions for refactoring process. In Chapter 4, we evaluate our technique by comparing it with CeDAR [TG12]. Finally, Chapter 5 contains the conclusions and future work.

Chapter 2

Literature Review

The extraction of code clone differences is an important step toward the process of refactoring code duplicates. The problem of source code matching or differencing has been investigated not only for software clone detection and refactoring, but also within the context of other applications such as change evolution analysis [FWPG07], plagiarism detection [LCHY06], source code retrieval [MdR04] and aspect mining [SGP04]. The first section explains the Program Dependence Graphs and their applications, the next two sections discuss the current approaches for code matching and the last section discusses the state-of-the-art techniques toward code clone refactoring. We will see that the existing mapping techniques either do not explore the entire search space of possible matches, and thus may return non-optimal solutions, or face scalability issues due to the problem of combinatorial explosion.

2.1 Program Dependence Graphs and their Applications

The core of our approach is built around the mapping of the *Program Dependence Graphs* (PDGs) corresponding to duplicated code fragments. A PDG [FOW87] is a directed graph with multiple types of edges that represents dependencies between the elements of a program. A node in a PDG represents a statement of a function or a control predicate, and an edge represents control or data flow dependences between the nodes. 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 it 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. Figure 6 shows an example code fragment and its corresponding PDG. In this example, node 0 represents the "*method entry*" node. There is a control dependence from the entry node to the loop (Node 1) and control dependences from Node 1 to all nodes that are directly contained by the loop (Node 2 and Node 3). There are data dependences from the entry node to all other nodes due to the variable *parent*. The other dependences are also due to the variable *parent*, since *parent* is defined in Node 3 and are used by Node 1, Node 2 and Node 3 because of the loop.

```
0 | public void printPathToRoot(Node parent) {
1 |   while(parent != null) {
2 |        System.out.println(parent.getId());
3 |        parent = parent.getParent();
  |   }
  | }
```

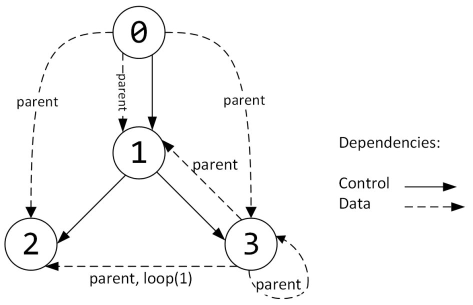


Figure 6: An example for PDG

Ottenstein and Ottenstein [OO84] mention PDG as an internal program representation which plays a major role in the system design. They suggest PDG as an effective tool in calculating program complexity metrics. Another main application of PDG is slicing [WRW03]. Slicing is the abstraction of program statements which affects the value of a variable in the code. The computation of slices is mostly used in debugging applications. PDG makes an ideal tool for constructing program slices. Horwitz and Reps [HR92] use PDGs in program differencing (finding differences between two programs) and program integration (integrating differences in one program onto another similar program). Software inspection is another area proposed by Walkinshaw et al. [WRW03] where PDGs can be used.

2.2 PDG-based Mapping Techniques

Komondoor and Horwitz [KH01] use Program Dependence Graphs(PDGs) and Program Slicing to find isomorphic Program Dependence Subgraphs that represent code clones. The main advantage of this approach is that it can detect non-contiguous clones (i.e., clones with gaps), clones in which matching statements have been re-ordered, and clones that are intertwined with each other. In their approach, two PDG nodes are matched if the corresponding statements or predicates are syntactically identical (i.e., their AST representation has the same structure) allowing only for differences in variable names and literal values. For each pair of matching nodes, *backward slicing* is performed in addition to *forward slicing* for matching predicates to construct a pair of isomorphic subgraphs (clones). A backward slice consists of all program points that affect a given point in the program whereas a forward slice consists of all program points that are affected by a given point in the program. The subsumed clone pairs are removed and the clone pairs are combined into larger groups wherever possible. Their method was implemented in a tool which finds duplicated code fragments in C programs and displays them to the programmer.

Krinke [Kri01] developed a method to identify code clones by computing the maximal similar subgraphs in fine-grained PDGs induced by k-limited paths starting from a pair of vertices. The method uses a fine-grained PDG (a specialization of the traditional PDG), in which there are special nodes for expressions, variable definitions, function calls etc. and special edges between expression components. In order 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 of this proposal is that the running time of the algorithm increases tremendously as k-limit increases. Also, many duplicated results are calculated again because the algorithm generates maximal similar graphs for every pair of predicate nodes. Another limitation is that the use of k-limit may lead to an incomplete solution (i.e., the selected k-limit is insufficient for detecting all possible matching vertices). Using only predicate nodes as starting vertices is also a drawback since it can result in not finding some clones. The proposed technique cannot guarantee an optimal result since it is a k-limited technique.

Shepherd et al. [SGP04] implemented an automated aspect mining technique exploiting the Program Dependence Graph and Abstract Syntax Tree representations of a program. The proposed method identifies initial refactoring candidates using a control-based comparison which is inspired by the algorithms [Kri01] and [KH01] and then filters out the undesirable refactoring candidates based on data dependence information. They use a source level PDG in which nodes that correspond to the same source line number are collapsed into a single node and the duplicate edges are avoided from or to the subsuming nodes. The mapping of the code fragments is perfomed by matching control dependence subgraphs of each PDG starting from the method entry nodes. The limitation of this approach is that since the algorithm starts from the method entry nodes, it will fail to match the control dependence subgraphs nested in different levels. An example case is when the control dependence subgraph of one method is directly nested under the method entry node and the matching control dependence subgraph of the other method is not directly nested under the method entry node. They consider all possible combinations of PDG nodes at the same level by breadth first traversal of the control dependence subgraphs of each PDG(i.e., when a single node can be mapped to multiple nodes), but they do not provide a consistent search space because of the extensive pruning performed for reducing the overhead of the algorithm.

Even though PDG-based code clone detection techniques have the advantage of finding non-contiguous code clones, they are time consuming compared to other techniques. Higo and Kusumoto [HK11] discuss PDG specializations and some heuristics for enhancing PDG-based code clone detection, thus improving Komondoor's technique [KH01]. The specializations are more specifically:

1. they introduced new edges called execution-next links, thereby expanding the range of program slicing in order to improve the ability to detect contiguous code, and

2. they merged multiple consecutive directly-connected equivalent nodes based on certain conditions thereby avoiding many node pairs as slice points resulting in false positives. This is done in order to reduce the computational cost of identifying isomorphic subgraphs.

The heuristics include two-way slicing (both backward and forward slicing), removing unnecessary slice points and neglecting small methods. They implemented the proposed technique in a tool called

Scorpio and investigated the effectiveness of each specialization and heuristic.

The common limitation of all aforementioned techniques is that they do not explore the entire search space of possible solutions and therefore they may return a non-optimal solution. 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.

Speicher and Bremm [SB12] view the problem of software code clone removal as a process of stepwise unification of *Type-3* clone instances using their Program Dependence Graph representations. They suggested additional data dependencies by taking into account the method invocations and object aliases that affect the state of an object. The PDGs of the clone candidates are compared and the nodes are matched depending on the refactorings that are considered. For example, in *RENAME* refactoring, the statements with different names of local variables, parameters, fields and methods are mapped, while in *REORDER PARAMETERS* refactoring, method signatures that differ just in the order of parameters are mapped. Along with the many other techniques suggested by them in accomplishing refactoring, they 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.

Liu et al. [LCHY06] developed a software plagiarism detection tool called GPLAG. They support that the PDG structures of the original and the plagiarized code remain invariant since the PDGs encode the program logic. Their technique exploits this invariance property of PDGs to find plagiarism through relaxed subgraph isomorphism testing, i.e., by checking if a PDG is γ-isomorphic to another, where γ is a relaxation parameter. In order to make the algorithm scalable to large programs, they prune the search space (i.e., they reduce the number of PDG pairs to be checked) by applying some filters. Even though they are able to detect five kinds of plagiarism disguises such as

‘Format alteration’, ‘Identifier renaming’, ‘Statement reordering’,‘Control replacement’ and ‘Code insertion’, the lossy filter applied to prune the search space may falsely exclude some interesting PDG pairs.

2.3 AST-based Mapping Techniques

Fluri et al. [FWPG07] describe an approach to extract the fine-grained changes that occur across different versions of a program. Their method is based on the tree alignment algorithm proposed by Chawathe et al. [CRGMW96], which takes as input two trees and produces a minimum edit script that can transform one tree into the other. They extended the original algorithm by applying the bigram string similarity measure for the matching of leaf nodes, and an enhanced subtree similarity criterion that takes into account the similarity of the nested nodes for the matching of control predicate structures. A limitation of the proposed approach is that string-based similarity matching is not resilient to extensive renaming of identifiers. In addition, the best match approach applied for leaf level nodes may match reoccurring statements that are not at the same position in the method body.

Cottrell et al. [CWD08] present an approach to help developers integrate reusable source code. Their algorithm takes as input two ASTs and tries to produce the best correspondences between the nodes. It applies a bottom-up comparison starting from leaf nodes (e.g., identifiers, types) and moving up to non-leaf nodes. The leaf identifiers are compared by means of the longest common substring measure. Non-leaf nodes are compared recursively taking into account the similarity of their children. The correspondences above a threshold value are finally used to identify the terms to be copied or transformed and integrated with the target system. A limitation is that the approach is only semi-automated, since user intervention is required to resolve the conflicts when multiple matches are found. Additionally, it tries to find a best fit in a greedy manner, which may lead to a non-optimal solution for the entire problem.

2.4 Clone Refactoring Techniques

Balazinska et al. [BMD$^+$00] focus on the extraction of code clone differences, perform advanced code clone analysis and provide the programmer with the information relevant to take a decision on the actual refactoring to be performed. Their technique to compare code fragments is based on the Dynamic Pattern Matching algorithm proposed by Kontogiannis et al. [KDM$^+$96]. The proposed algorithm aligns syntactically unstructured entities and finds an optimal distance between the two code fragments. The optimal distance is the minimum number of tokens to be inserted, deleted or substituted to transform one code fragment into another. However, this overall distance cannot be guaranteed as minimal as it tries to find optimal values at node level without considering the hierarchical structural differences at a higher level. The differences are expressed as programming language entities easily understandable by a programmer. This is done by projecting the tokens forming the differences onto the corresponding AST elements. The differences are also categorized based on the role in refactoring. The categories are:

1. superficial differences such as names of local variables which do not affect the behavior of methods

2. differences which affect the signature of methods: return value, access modifiers, thrown exceptions etc.

3. differences affecting the types of parameters

4. all other differences.

The distinction among the differences helps the programmers to make the right decision regarding refactoring. As part of the proposed approach, they implemented an automatic refactoring process which transforms code clones using the design patterns Strategy and Template Method [GHJV95].

Higo et al. [HKI08] proposed a set of metrics to suggest different refactoring opportunities such as extract method refactoring and pull-up method refactoring in order to remove the software clones. The proposed method was implemented in a tool called ARIES which builds the ASTs of code clones detected using an existing code clone detection tool called CCFinder [KKI02]. A minimum token length was used as the threshold in finding the refactoring oriented code clones. Some metric values were measured for these code clones which represent whether or not each of the code clones can be easily merged and how to merge them. It is up to the users to choose if refactoring should be performed. The following information is analyzed to characterize the clone sets:

1. how dispersed the code clones are across the class hierarchy and

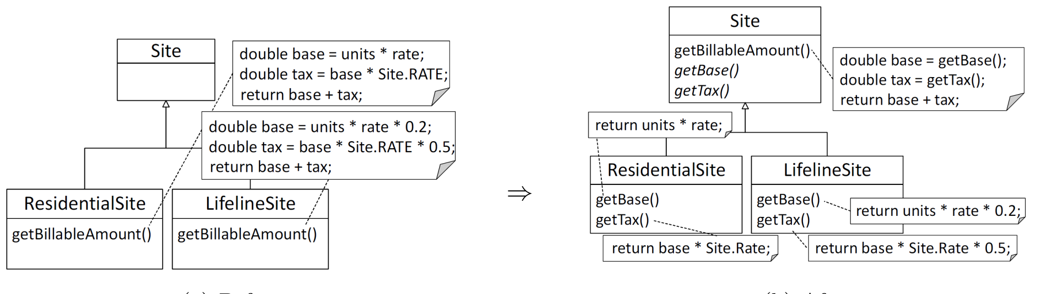
2. the coupling between the code clone and its surrounding code in terms of the number of referenced and assigned variables within the clones.

The above values were used to represent the possibility of different refactoring patterns such as *Pull Up Method*, *Form Template Method* etc. A small-scale case study was performed on the open source project Ant, and they concluded that the proposed method can efficiently merge code clones.

Choi et al. [CYI$^+$11] performed an industrial case study and concluded that clone sets extracted by combining multiple clone metrics constitute better refactoring opportunities than clone sets extracted by individual clone metrics. These metrics include the average length of token sequences of code clones within a clone set, ratio of length of the non-repeated token sequences to the length of the whole token sequences of code clones within a clone set, and the size of the code clone set. The analysis was performed on the clone sets detected by CCFinder [KKI02]. The effectiveness of their method was studied by conducting a case study on an industrial software. In this case study, they asked a software developer to fill out a questionnaire based on a list of selected code clones. The survey results were used as a basis to conclude that the developer found the clone sets detected by combining higher clone metrics as more desirable for refactoring. The survey conducted was very minimal as the survey included very small number of clone sets and was conducted on a single system and got feedback from one developer.

Tairas and Gray [TG12] developed an Eclipse plugin called CeDAR with the objective to unify the code clone maintenance activities by bridging the gap between the clone detection tools and the process of refactoring. They also extended the Eclipse refactoring engine to enable the processing of more types of differences among duplicated code fragments, such as differences in field accesses, and method calls without arguments, in addition to the differences in local variable identifiers which is supported by Eclipse refactoring engine. They managed to parse the output results from five clone detection tools and they were presented to the developer using their plugin, so that the user can attempt to refactor the clones within the Eclipse IDE. They performed an evaluation on Type-2 clones detected in 9 open-source projects using Deckard [JMSG07] clone detection tool, which revealed that the aforementioned enhancements in the matching of duplicated code increased the percentage of refactorable clones from 10.6% to 18.7%. As future work, they mentioned the inclusion of more parameterized differences (e.g., local variable identifiers replaced with method calls) and the support for additional types of refactorings of clones belonging to different classes.

Hotta et al. [HHK12] built upon the method proposed by Juillerat and Hirsbrunner [JH07], to refactor Type-3 clones by applying Form Template Method. *Form Template Method* is one of the refactoring patterns proposed by Fowler et al. [Fow99]. This pattern targets similar methods existing in derived classes that inherit the same parent class. The code clones found by this method can be pulled up into the base class as a common process. Figure 7 shows an example of application of Form Template Method [Fow99]. The two classes *ResidentialSite* and *LifelineSite* inherit from the parent class Site. The methods named *getBillableAmount()* in both child classes are similar to each other. Through Form Template Method, the common code fragments are pulled up into the base class and the unique code fragments in each method are extracted as new methods, *getBase()* and *getTax()*.



```
double base = units * rate;
double tax = base * Site.RATE;
return base + tax;

double base = units * rate * 0.2;
double tax = base * Site.RATE * 0.5;
return base + tax;

double base = getBase();
double tax = getTax();
return base + tax;

return units * rate;
return base * Site.Rate;
return units * rate * 0.2;
return base * Site.Rate * 0.5;
```

(a) Before (b) After

Figure 7: An example of application of Form Template Method

The technique proposed by Hotta et al. detects isomorphic subgraphs in the PDGs of two similar methods, which are considered to be the clone pair. Their PDG-based detection technique can detect code clones containing different order of statements, and different implementation styles such as differences in for and while loops. The proposed method suggests program statements that can be merged into the base class as the common process and the program statements that should remain in the derived classes as the unique processes.

Bian et al. [BKSM13] presented a semantic-preserving amorphous procedure extraction algorithm (SPAPE) to extract the near-miss cloned code fragments (Type-2 and Type-3 code clones). The algorithm analyzes the two PDGs of the clone code fragments and uses a set of amorphous transformation rules to make the cloned code statements suitable for extraction. The transformations are applied in order to replicate predicate statements and partition loop structures. In addition, the differing statements are identified and combined by inserting control variables and control statements. Finally, the clone sequences are extracted into a procedure. Currently, SPAPE supports procedural code written in the C programming language.

Goto et al. [GYI$^+$13] proposed a method which detects 'Extract Method' candidate sets among a pair of similar Java methods and are ranked according to cohesiveness. AST differencing is used to detect the syntactic differences between the input pair of methods and slice-based cohesion metrics such as *Tightness*, *Coverage* and *Overlap* are used to rank the obtained refactoring candidates. The proposed approach is developed to help those programmers who need to refactor similar methods into cohesive methods. A case study performed by the authors indicated that the refactorings suggested by their method helped to increase two out of the three slice based cohesion metrics.

Chapter 3

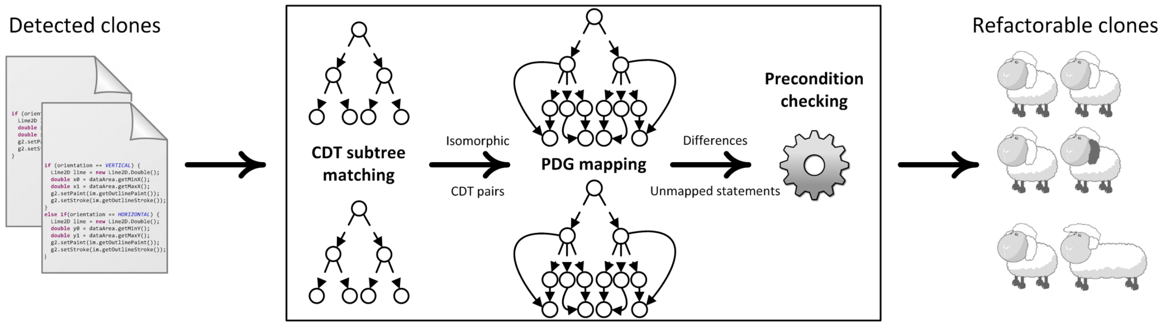
Clone Unification

The proposed technique for the unification of clones in order to refactor them comprises three major steps as follows:

1. **Control Structure Matching**: The control structure of the code fragments is extracted into trees called Control Dependence Trees and they are matched for identifying potential refactoring candidates as well as to determine valid clone regions.

2. **Program Dependence Graph Matching**: The output of this phase is an optimal match of the PDGs corresponding to the matched subtrees from the previous step.

3. **Checking Preconditions**: A check is done against a set of predetermined conditions to ensure that the code behavior is preserved and to determine whether it is safe to refactor.



```
if (orientation == VERTICAL) {
  Line2D line = new Line2D.Double();
  double x0 = dataArea.getMinX();
  double x1 = dataArea.getMaxX();
  g2.setPaint(im.getOutlinePaint());
  g2.setStroke(im.getOutlineStroke());
}
else if(orientation == HORIZONTAL) {
  Line2D line = new Line2D.Double();
  double y0 = dataArea.getMinY();
  double y1 = dataArea.getMaxY();
  g2.setPaint(im.getOutlinePaint());
  g2.setStroke(im.getOutlineStroke());
}
```

Figure 8: An overview of the proposed technique

An overview of the process flow is shown in Figure 8. The input to the process can be either clone fragments detected by a clone detection tool or two methods manually selected by the programmer. The technique determines the Control Dependence Trees (CDTs) of the input code fragments. The matching pairs of subtrees of the CDTs are found and passed to the PDG mapping phase. The PDGs undergo the mapping process where two nodes are matched if they are within the control structure of the matched CDT pairs. The output of the PDG mapping phase is a set of mapped and unmapped statements with the differences between the mapped statements. The differences and the unmapped statements are checked against preconditions to evaluate if the mapped statements can be refactored.

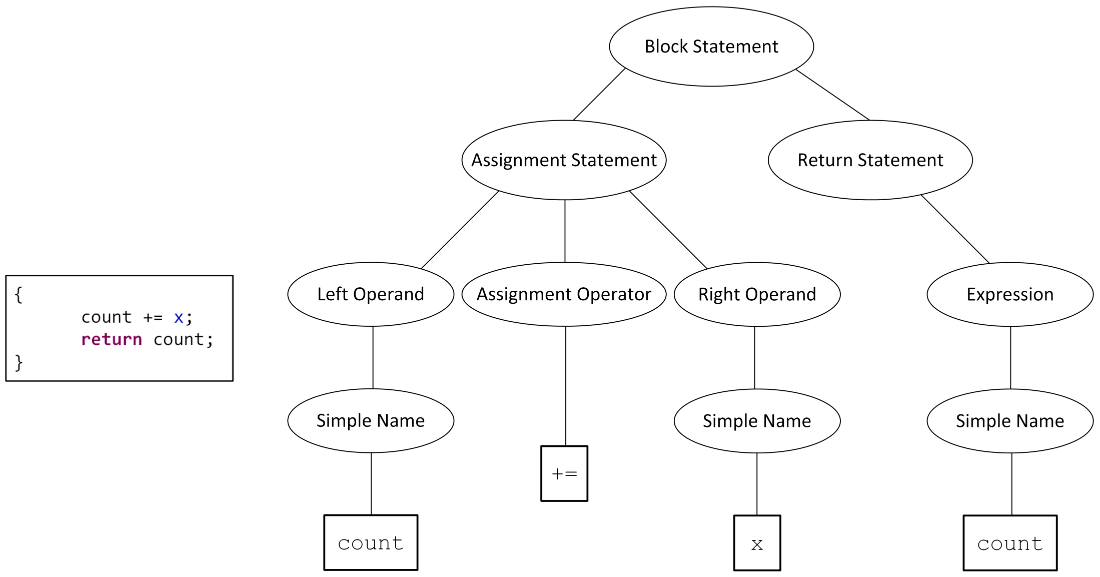
The section 3.1 in this chapter explains the statement matching process used in the proposed technique and the rest of the chapter discusses each of the above steps in detail.

3.1 Statement Matching

This section deals with our statement matching process. Statement matching is a core functionality used throughout our algorithms in order to examine if two statements are compatible and to find the exact differences between them, if any. Two statements are said to be compatible if they can be matched by parameterizing their differences.

3.1.1 Abstract Syntax Tree

In our statement matching process, we analyze the Abstract Syntax Tree (AST) structure of the statements. AST is a tree-like representation of the source code. Each Java source file is represented as a `CompilationUnit`, which is the root of the corresponding AST tree. Each node of the Abstract Syntax Tree denotes a construct in the code and provides specific information about the object it represents. For example, a method is represented as a `MethodDeclaration` AST node and any string that is not a Java keyword is represented as a `SimpleName` node. An example of an Abstract Syntax Tree is given in the Figure 9.



```
{
        count += x;
        return count;
}
```

(a) Example code (b) AST form

Figure 9: A block of source code and the corresponding Abstract Syntax Tree

3.1.2 AST Matching

The clone unification method in this thesis is basically a process of matching the PDGs of two methods. Each AST statement forms a PDG node where AST statement is the AST representation of a program statement. There are many different types of AST statements such as `ContinueStatement`, `WhileStatement` etc. We consider two PDG nodes as compatible, if they correspond to the same AST statement type and have a matching AST structure. However, a high degree of freedom is required in the matching of expressions within the statements in order to make flexible the unification of duplicated code with non-trivial differences. In the past, Tairas and Gray [TG12] extended the Eclipse IDE refactoring engine for duplicated code, which supports only the parameterization of differences in local variable identifiers, by additionally allowing the matching of differences in field accesses, string literals, and method calls without arguments. But in general, the differences are not always trivial as mentioned before in Chapter 1. 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 1 contains the complete list of supported expression types and shows the degree of freedom we allow in matching two expressions. 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 `java.lang.Object`, because `Object` is the implicit superclass of every Java class, unless it explicitly extends another class).

Table 1: 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)`.

3.1.3 Implementation Details

Our AST matching algorithm has been implemented by extending the `ASTMatcher` superclass provided in Eclipse JDT framework. The default implementation of `ASTMatcher` provided by Eclipse computes whether two ASTs subtrees are structurally isomorphic. Our implementation overrides some of the `match` methods in order to define more relaxed subtree matchers. For example, in the case of control predicate nodes (e.g., `if`, `for` statements), the AST structure contains the conditional expression(s) and their bodies. Our overridden `match` implementation will compare only their conditional expression(s) and ignores the bodies, because the statements inside them will be subsequently compared if the control predicate statements are found compatible. In addition, the

AST matching algorithm returns a list of differences detected between the matched AST statements. These differences are essential for the procedure of optimization and the examination of preconditions, which will be explained in the next chapter.

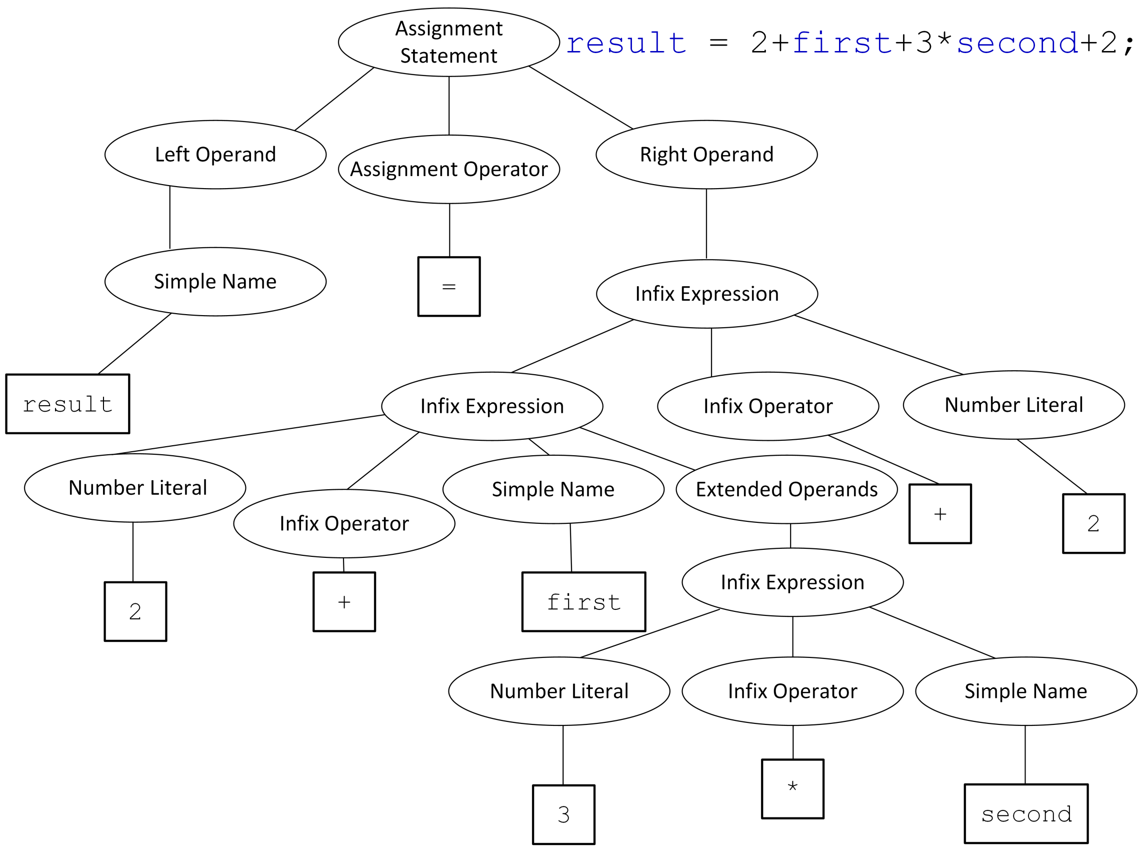
Table 2 shows the difference types which are reported by our AST matching algorithm. The last two difference types, namely operator and variable type mismatches are not parameterized in our approach. However, lambda expressions (a feature of Java 8.0) can be used to unify operator mismatches, whereas the concept of generics can be applied to unify variable type mismatches. Though these two scenarios are not explored as part of this thesis, they can be used to advance the refactoring technique in the future. 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 2: Detected Differences Between Matched Nodes

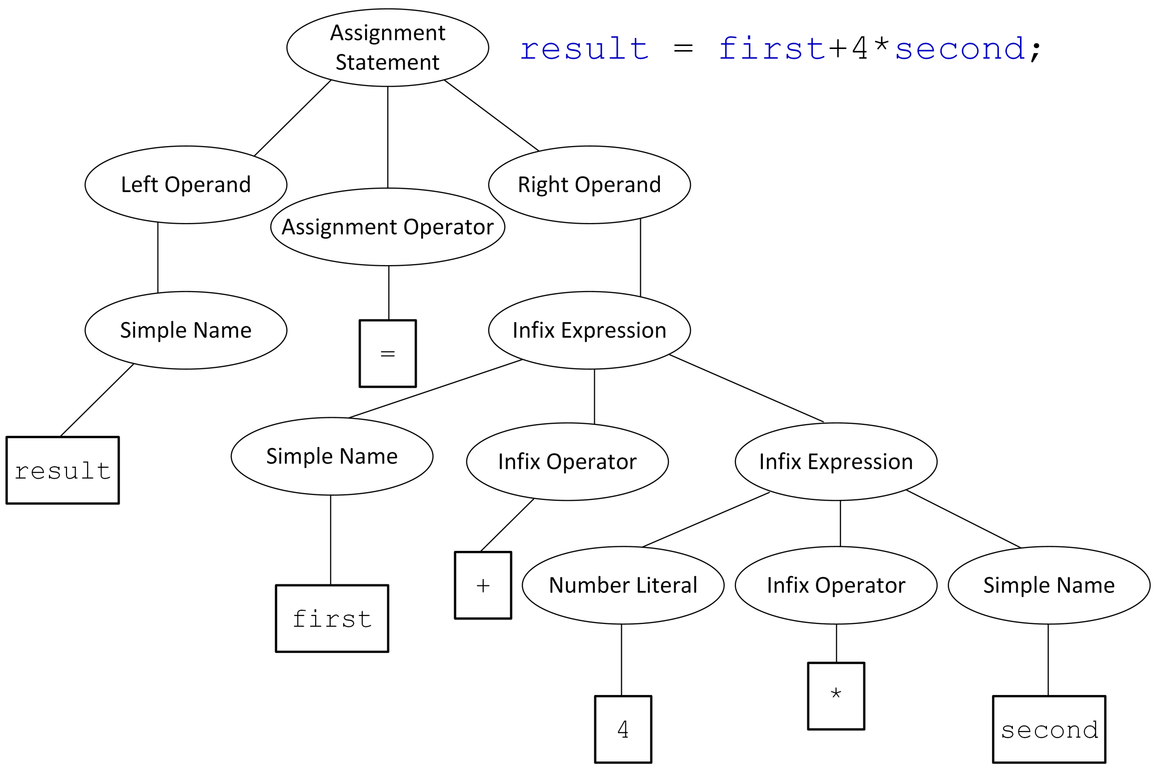
Difference Type	**Example**	
Variable Identifier Mismatch	`int x = y;`	`int x = z;`
Literal Value Mismatch	`String s = "s1";`	`String s = "s2";`
Method Name Mismatch	`expr.foo(arg);`	`expr.bar(arg);`
Argument Number Mismatch	`foo(arg0, arg1);`	`foo(arg0);`
Missing Caller Expression	`expr.foo(arg);`	`foo(arg);`
Array Dimension Mismatch	`int x = a[i];`	`int x = a[i][j];`
Infix Operand Number Mismatch[†]	`int x = 4*a;`	`int x = 3*b*2;`
Infix Left Operand Mismatch	`boolean x = 4*a∧6*c;`	`boolean x = 4*a+7*b∧6*d;`
Infix Right Operand Mismatch	`boolean x = a+b∧3*c-d;`	`boolean x = a+c∧3*d;`
AST Compatible Change	`int x = foo();`	`int x = 5;`
Operator Mismatch	`int x = y + z;`	`int x = y - z;`
Variable Type Mismatch	`int x = 5;`	`double x = 5;`

[†] Infix operand number mismatch refers to the number of extended operands. The extended operands is the preferred way of representing deeply nested expressions of the form L op R op R2 op R3... where the same operator appears between all the operands (the most common case being lengthy string concatenation expressions).

An example of our AST matching process is explained below. Let's assume we have the statements shown in Figure 10 along with their AST representations. Our AST matching algorithm applies the steps as described in Table 3.



(a) Statement 1



(b) Statement 2

Figure 10: Abstract Syntax Trees of two assignment statements

Table 3: Execution Steps In AST Matching Process

Step	AST Node 1	AST Node 2	Action
1	`result=2+first+3*second+2;`	`result=first+4*second;`	Both are ExpressionStatements. Proceeding to match expressions.
2	`result=2+first+3*second+2`	`result=first+4*second`	Both are Assignments. Proceeding to match assignment operators.
3	`=`	`=`	Both operators are same. Proceeding to match left hand side operands.
4	`result`	`result`	No difference. Proceeding to match right hand side operands.
5	`2+first+3*second+2`	`first+4*second`	Both are Infix Expressions. Proceeding to match extended operands.
6			No extended operands. Proceeding to match left hand side operands.
7	`2+first+3*second`	`first`	No matching AST structure. The first node is Infix Expression and the second node is Simple Name. Reporting INFIX LEFT OPERAND MISMATCH. However, they have the same type. Proceeding to match right hand side operands.
8	`2`	`4*second`	No matching AST structure. The first node is Number Literal and the second node is Infix Expression. Reporting INFIX RIGHT OPERAND MISMATCH. However, they have the same type.
9			Both left hand side and right hand side operands did not match. Decision to parameterize the entire infix expression.

3.2 Control Structure Matching

Control structure matching is the first major step in our clone unification algorithm. We have made the assumption that in order to extract pieces of duplicated code, they should have exactly the same structure of control. In other words, only complete AST-subtrees having the same structure can be valid candidates for refactoring. To achieve this condition, we find the common control structures within the input clones. Finding control structures also helps us to tackle the challenge of getting valid clone regions as mentioned in Section 1.5. Experience has shown that text-based and token-based clone detection tools may return invalid clone regions [TG12, HKI08]. The control structure matching phase addresses this problem, since the control structures always form valid clone regions. Another advantage of comparing control structures beforehand is that it helps to improve the performance of PDG matching algorithm. The PDG matching process is subject to the combinatorial explosion effect since there would be many nodes to be matched. As the number of possible matches for the nodes increases, the width of the search tree grows rapidly as a result of the numerous combinatorial considerations to be explored. The risk of combinatorial explosion is reduced by taking advantage of the control dependence structure of the two compared PDGs and matching them first. By taking advantage of the control structure, we can avoid the unnecessary comparison of nodes nested at different levels. This is explained in Section 3.3.

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 definitions considered to be duplicated or containing duplicate code fragments, usually selected by the user.

Since control structure essentially forms a tree, we can conclude that our input constitutes two trees corresponding to the control structure of the input clones. Tree matching algorithms are used to find the largest common, non-overlapping subtrees within the input trees. In the book "Algorithms on Trees and Graphs", Valiente [Val02] describes bottom-up and top-down algorithms for finding the common subtrees. A hybrid algorithm which combines both bottom-up and top-down approaches is developed for our matching technique.

3.2.1 Control Dependence Tree Representation

The control dependence structure of an input clone is represented as a *Control Dependence Tree* (CDT). More specifically, we first build the Control Dependence Tree of each input clone. A CDT has the same structure as the *Control Dependence Graph* (CDG) [FOW87] 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. Basically, a CDT represents the nesting of control structure of the code fragment.

For example, consider the two candidate methods for clone refactoring given in Figure 11 and the corresponding Control Dependence Trees in Figure 12. The code example is taken from the book "Refactoring: Improving the Design of Existing Code" [Fow99]. The CDTs of the methods do not form isomorphic trees. Two trees are said to be *isomorphic* when there is a bijective correspondence between their node sets which preserves the structure of the trees. The formal definition of tree isomorphism [Val02] is given below.

Two unordered trees $T_1 = (V_1,E_1)$ *and* $T_2 = (V_2,E_2)$ *are* ***isomorphic****, if there is a bijection* $M \subseteq V_1 \times V_2$ *such that* $(root[T_1],root[T_2]) \in M$ *and the following condition*

- $\mathrm{parent}[v] = \mathrm{parent}[w]$ *for all nonroots* $v \in V_1$ *and* $w \in V_2$ *with* $(v,w) \in M$

is satisfied.

In this thesis, we are interested in finding isomorphic subtrees within the CDTs of the clone fragments, where the bijective correspondence between two nodes exists when they are AST compatible. AST compatibility is checked using the techniques mentioned in Section 3.1. We have to find the largest possible common subtrees [Val02] in the CDTs of the clones to find valid refactoring opportunities. In the cases, where the input CDTs are isomorphic, the input pair of CDTs itself forms a refactoring opportunity.

```
0    private double getCharge(Rental rental) {
1          double result = 0;
2          switch (rental.getMovie().getPriceCode()) {
3          case Movie.REGULAR:
4                result += 2;
5                if (rental.getDaysRented() > 2)
6                      result += (rental.getDaysRented() - 2) * 1.5;
7                break;
8          case Movie.NEW_RELEASE:
9                result += rental.getDaysRented() * 3;
10               break;
11         case Movie.CHILDRENS:
12               result += 1.5;
13               if (rental.getDaysRented() > 3)
14                     result += (rental.getDaysRented() - 3) * 1.5;
15               break;
           }
16         return result;
     }
```

(a)

```
0    public String statement() {
1          double totalAmount = 0;
2          int frequentRenterPoints = 0;
3          Enumeration rentals = _rentals.elements();
4          String result = "Rental Record for " + getName() + "\n";
5          while (rentals.hasMoreElements()) {
6                Rental each = (Rental) rentals.nextElement();
7                double thisAmount = 0;
8                switch (each.getMovie().getPriceCode()) {
9                case Movie.REGULAR:
10                     thisAmount += 2;
11                     if (each.getDaysRented() > 2)
12                           thisAmount += (each.getDaysRented() - 2) * 1.5;
13                     break;
14               case Movie.NEW_RELEASE:
15                     thisAmount += each.getDaysRented() * 3;
16                     break;
17               case Movie.CHILDRENS:
18                     thisAmount += 1.5;
19                     if (each.getDaysRented() > 3)
20                           thisAmount += (each.getDaysRented() - 3) * 1.5;
21                     break;
                 }
22               frequentRenterPoints++;
23               if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE) && each.getDaysRented() > 1)
24                     frequentRenterPoints++;
25               result += "\t" + each.getMovie().getTitle() + "\t" + String.valueOf(thisAmount) + "\n";
26               totalAmount += thisAmount;
           }
27         result += "Amount owed is " + String.valueOf(totalAmount) + "\n";
28         result += "You earned " + String.valueOf(frequentRenterPoints) + " frequent renter points";
29         return result;
     }
```

(b)

Figure 11: Example clone candidates

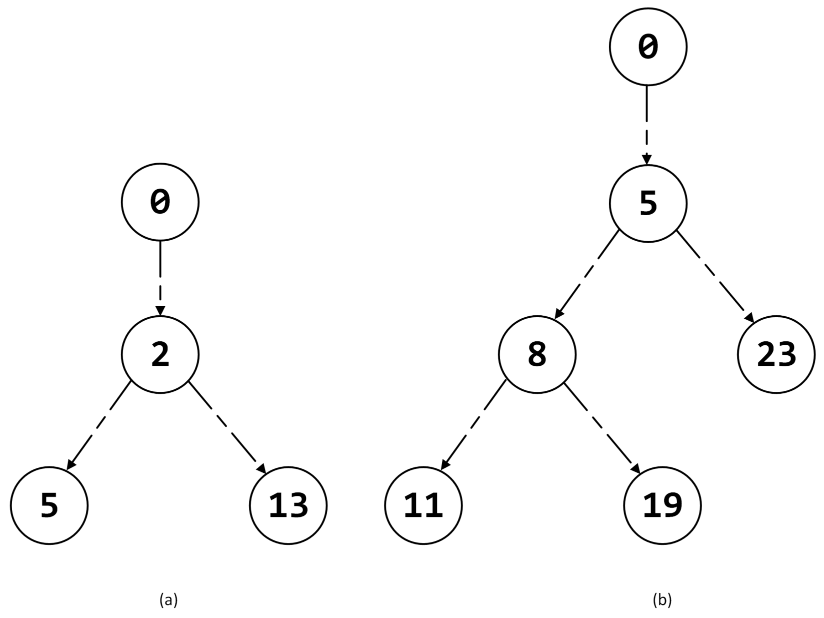


Figure 12: The Control Dependence Trees for the code fragments of Figure 11

3.2.2 Algorithm

Our goal is to find *isomorphic* CDTs within the duplicated code fragments. In other words, we have to find the largest common subtrees [Val02] in the input CDTs. We construct the control dependence subtrees corresponding to each code fragment or the entire method. An algorithm is developed that takes as input two CDTs (CDT_1,CDT_2) and finds all non-overlapping *largest common subtrees* [Val02] within the CDTs. Each resulting subtree match will be further investigated as a separate clone refactoring opportunity. In the previous example shown in Figure 12, the largest common subtrees will be **{2,5,13}** and **{8,11,19}** where **(2,8)**, **(5,11)** and **(13,19)** are the AST compatible node pairs. The numbers are used only for representing each statement in the example and have no relevance to the implementation of the process.

Step 1: Selecting leaf nodes

Initially, we start in a bottom up fashion i.e., we start from the leaf nodes of the input Control Dependence Trees. We collect from the two CDTs all leaf nodes, which either do not have siblings, or all of their siblings are also leaf nodes. This is done for improving the efficiency of the algorithm by avoiding repeated analysis of the nodes. The reason is explained with an example as follows. In the beginning, we do not need to consider the leaf nodes which have any non-leaf siblings, since they will be explored in the later recursions of the algorithm when we consider their siblings which are non-leaf nodes. For example, in the Control Dependence Tree shown in Figure 12(b), the leaf nodes are **11**, **19** and **23**. But only the nodes **11** and **19** are considered initially, as **23** would be visited later as part of the siblings of **8**.

Step 2: Making match pairs of leaf nodes

The next step is to extract all matching pairs i.e., AST compatible nodes (the compatibility is checked using the statement matching process in Section 3.1.2), among the collected leaf node sets from the two CDTs. Each extracted leaf node pair is represented as ($node_i$,$node_j$), where $node_i$ and $node_j$ are leaf nodes of CDT_1 and CDT_2 respectively. In the CDT pair in Figure 12, the possible matchings of leaf nodes are **(5,11)**, **(5,19)**, **(13,11)** and **(13,19)**. We keep the best matching node pairs (i.e., node pairs with minimum number of differences) and therefore the extracted leaf node pairs are **(5,11)** and **(13,19)** (these are exactly matched node pairs with no differences).

Step 3: Filtering match pairs of leaf nodes

Instead of processing every extracted leaf node pairs, a filtering (i.e., removing some node pairs based on certain conditions as explained below) is done as another step to improve the efficiency of the algorithm. When sibling node pairs exist in the extracted leaf node pairs, only one of them is taken for further processing, since the common subtrees found from any of the sibling node pairs using Algorithm 1 will exactly be the same. In other words, when ($node_i$,$node_j$) and ($node_x$,$node_y$) exist in the extracted leaf node pairs, and $node_x$ and $node_y$ are siblings of $node_i$ and $node_j$ respectively, ($node_i$,$node_j$) and ($node_x$,$node_y$) are called sibling pairs. Only one of the sibling pairs, say ($node_i$,$node_j$) is added to the filtered leaf node pairs list for further processing. In the example given in Figure 12, **(5,11)** is the only pair we need to process as **(13,19)** is a sibling pair of **(5,11)**.

Step 4: Forming common subtree pairs

Each leaf node pair ($node_i$,$node_j$) in the filtered leaf node pairs list obtained from Step 3, is given as input to Algorithm 1, which returns a subtree match (a pair of *isomorphic* subtrees) as a solution.

```
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      $siblings_i \leftarrow$ nodePair.$node_i$.siblings
4      $siblings_j \leftarrow$ nodePair.$node_j$.siblings
5      $matchedSiblings \leftarrow \varnothing$
6      tempSolution $\leftarrow \varnothing$
7      foreach $sibling_i \in siblings_i$ do
8          foreach $sibling_j \in siblings_j$ do
9              if compatibleAST($sibling_i$, $sibling_j$) and
10             not alreadyMatched($sibling_j$) then
11                 pair $\leftarrow$ ($sibling_i$, $sibling_j$)
12                 $pairs \leftarrow$ topDownMatch(pair)
13                 if exactlyPairedSubtrees(pairs) then
14                     add pair $\rightarrow$ $matchedSiblings$
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         $parent_i \leftarrow$ nodePair.$node_i$.parent
24         $parent_j \leftarrow$ nodePair.$node_j$.parent
25         if compatibleAST($parent_i$, $parent_j$) then
26             pair $\leftarrow$ ($parent_i$, $parent_j$)
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.

Explanation of the Algorithm

The algorithm first compares sibling nodes of the nodes in the input node pair to find matching sibling pairs. The siblings are matched by checking if they have compatible AST structure (line **9**) as explained in Section 3.1. For each matching sibling pair it performs a top-down tree match (line **12**) and examines if the resulting subtree match is "exactly paired" (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 (line **25**), 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 [Val02] and guarantees that the returned subtree match will be exactly paired. The algorithm is designed to return only exactly paired subtree matches in order to avoid inconsistencies or gaps in the control dependence structure of the matched subtrees. CDT subtrees without inconsistencies or gaps in their control structure make better candidates for clone refactoring, since the possibility of having unmatched statements is lower.

3.2.3 Working Example of `bottomUpMatch` Algorithm

In the example given in Figure 12, **(5,11)** is given as input to `bottomUpMatch` algorithm. **(5,11)** is added to solution. Siblings of **5** and **11** form a pair **(13,19)**, since **13** and **19** are AST compatible. As **13** and **19** have no children, they will satisfy the condition of "exactly paired top-down subtree matches". **(13,19)** is added to the solution and parents of **5** and **11** are visited since all siblings are covered. The parents of **5** and **11** are AST compatible and they form a nodePair **(2,8)**. The nodePair **(2,8)** is input to the recursive call of `bottomUpMatch` algorithm. The nodePair **(2,8)** is appended to the solution. Since no sibling pairs are found for **(2,8)** and the counts of siblings of nodes **2** and **8** do not match, algorithm does not continue. The resulting solution is therefore **{(2,8),(5,11),(13,19)}**.

3.3 PDG Matching

In the previous section, we described the algorithm that extracts isomorphic subtrees from the CDTs of the clones given as input. The obtained isomorphic subtrees are used in matching the subgraphs of Program Dependence Graphs such that only the non-control nodes within the matched control structures need to be analyzed as the control nodes are already matched. This avoids the exhaustive comparison of all nodes, since the algorithm will try to match only those nodes nested under matched control structures. In this section, an approach for an “optimal” mapping of the PDG subgraphs corresponding to the extracted CDT subtree match pairs is presented.

3.3.1 Advantages of PDG Matching

Since clones may be identical (Type-1) or may be with some differences (Type-2 and Type-3), PDG is the most appropriate representation of the clone instances to identify the statements that are equal or unifiable. This is because of two reasons:

1. PDG 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 (Chapter 2).
2. 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 data dependencies of the graph [TC11].

3.3.2 Why do we need the PDG Mapping to be Improved?

With each statement considered as a node in PDG, one thing to remember is that there can be more than one common subgraph with the same number of nodes for the given PDGs. Therefore, we need to find an optimal mapping of PDG subgraphs. It is defined as a problem in which we find the common subgraph that satisfies the following conditions:

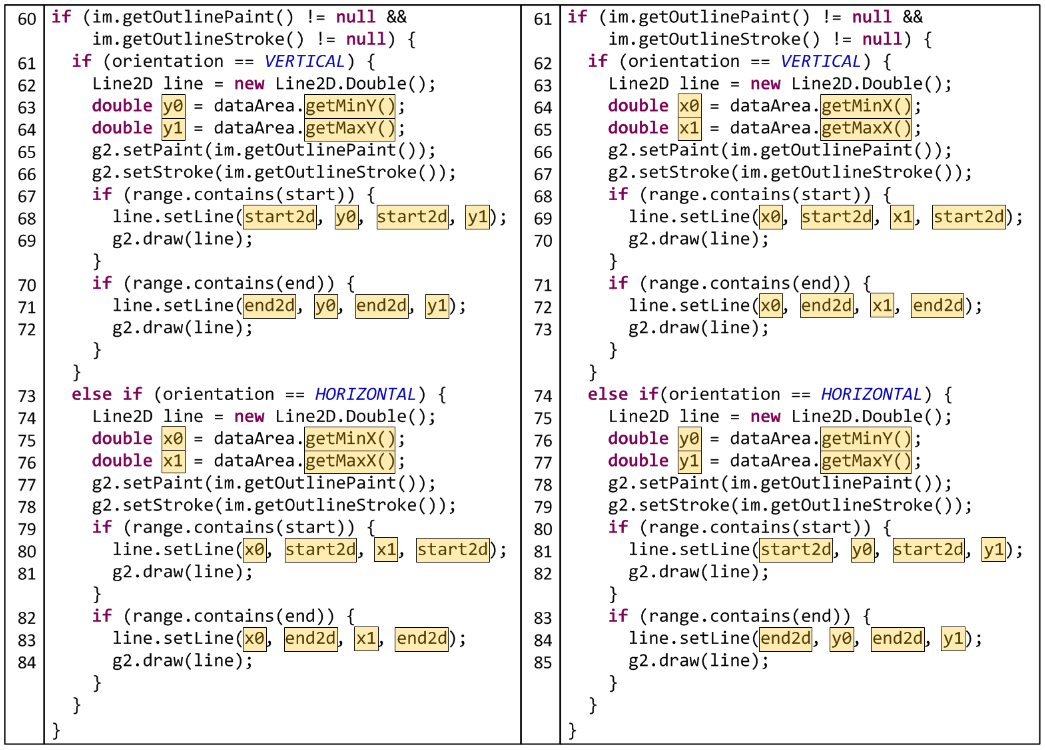
1. It has the maximum number of matched nodes.
2. The matched nodes have the minimum number of differences.

The reason why we assume the above conditions to obtain an optimal mapping is explained below.

Let us see 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 13(a) illustrates two code fragments taken from methods `drawDomainMarker` and `drawRangeMarker`, respectively, found in class `AbstractXYItemRenderer` of the open-source project *JFreeChart* (version 1.0.14). These two methods contain over 90 duplicated statements extending through their entire body. However, only a small portion of the duplicated code is shown in the figure for the sake of simplicity.

Figure 13(a) depicts a possible mapping of the statements as obtained from the PDG-based clone detection approaches discussed in Section 2.2. 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 13(a), coincides with the 'first' match according to the actual order of the statements. As it can be observed from Figure 13(a), the mapping is maximum, since all 25 statements have been successfully mapped; however, it contains a large number of differences between the mapped statements.

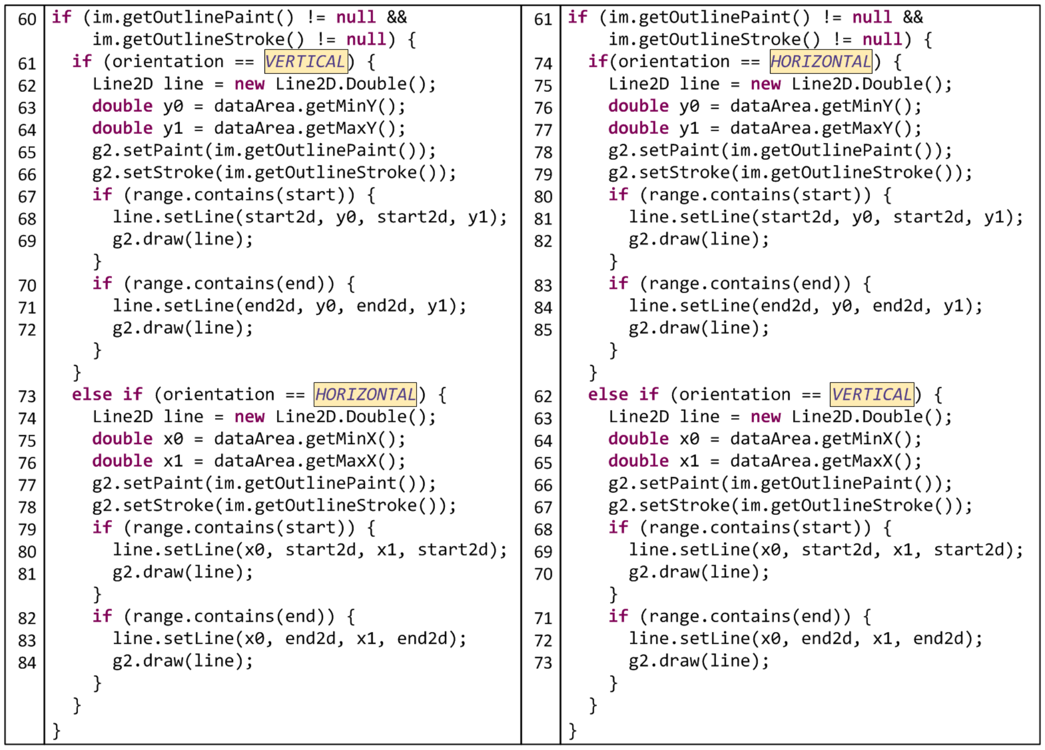
The minimization of the differences is of key importance for the refactoring of clones, since it directly affects the number of parameters that have to be introduced in the extracted method containing the common functionality, as well as the feasibility of the refactoring transformation. Figure 13(b) depicts the optimal mapping, which is again maximum in terms of the number 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 13(b) 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 above example motivates us to design an algorithm to find the largest code segment possible (i.e., with the maximum number of mappings) for extraction and the mapping of statements should be such that we get the lowest possible number of differences between the mapped statements.



```java
if (im.getOutlinePaint() != null &&
    im.getOutlineStroke() != null) {
  if (orientation == VERTICAL) {
    Line2D line = new Line2D.Double();
    double y0 = dataArea.getMinY();
    double y1 = dataArea.getMaxY();
    g2.setPaint(im.getOutlinePaint());
    g2.setStroke(im.getOutlineStroke());
    if (range.contains(start)) {
      line.setLine(start2d, y0, start2d, y1);
      g2.draw(line);
    }
    if (range.contains(end)) {
      line.setLine(end2d, y0, end2d, y1);
      g2.draw(line);
    }
  }
  else if (orientation == HORIZONTAL) {
    Line2D line = new Line2D.Double();
    double x0 = dataArea.getMinX();
    double x1 = dataArea.getMaxX();
    g2.setPaint(im.getOutlinePaint());
    g2.setStroke(im.getOutlineStroke());
    if (range.contains(start)) {
      line.setLine(x0, start2d, x1, start2d);
      g2.draw(line);
    }
    if (range.contains(end)) {
      line.setLine(x0, end2d, x1, end2d);
      g2.draw(line);
    }
  }
}
```

```java
if (im.getOutlinePaint() != null &&
    im.getOutlineStroke() != null) {
  if (orientation == VERTICAL) {
    Line2D line = new Line2D.Double();
    double x0 = dataArea.getMinX();
    double x1 = dataArea.getMaxX();
    g2.setPaint(im.getOutlinePaint());
    g2.setStroke(im.getOutlineStroke());
    if (range.contains(start)) {
      line.setLine(x0, start2d, x1, start2d);
      g2.draw(line);
    }
    if (range.contains(end)) {
      line.setLine(x0, end2d, x1, end2d);
      g2.draw(line);
    }
  }
  else if(orientation == HORIZONTAL) {
    Line2D line = new Line2D.Double();
    double y0 = dataArea.getMinY();
    double y1 = dataArea.getMaxY();
    g2.setPaint(im.getOutlinePaint());
    g2.setStroke(im.getOutlineStroke());
    if (range.contains(start)) {
      line.setLine(start2d, y0, start2d, y1);
      g2.draw(line);
    }
    if (range.contains(end)) {
      line.setLine(end2d, y0, end2d, y1);
      g2.draw(line);
    }
  }
}
```

(a) Non-optimal mapping with 25 mapped nodes and 24 differences.



```java
if (im.getOutlinePaint() != null &&
    im.getOutlineStroke() != null) {
  if (orientation == VERTICAL) {
    Line2D line = new Line2D.Double();
    double y0 = dataArea.getMinY();
    double y1 = dataArea.getMaxY();
    g2.setPaint(im.getOutlinePaint());
    g2.setStroke(im.getOutlineStroke());
    if (range.contains(start)) {
      line.setLine(start2d, y0, start2d, y1);
      g2.draw(line);
    }
    if (range.contains(end)) {
      line.setLine(end2d, y0, end2d, y1);
      g2.draw(line);
    }
  }
  else if (orientation == HORIZONTAL) {
    Line2D line = new Line2D.Double();
    double x0 = dataArea.getMinX();
    double x1 = dataArea.getMaxX();
    g2.setPaint(im.getOutlinePaint());
    g2.setStroke(im.getOutlineStroke());
    if (range.contains(start)) {
      line.setLine(x0, start2d, x1, start2d);
      g2.draw(line);
    }
    if (range.contains(end)) {
      line.setLine(x0, end2d, x1, end2d);
      g2.draw(line);
    }
  }
}
```

```java
if (im.getOutlinePaint() != null &&
    im.getOutlineStroke() != null) {
  if(orientation == HORIZONTAL) {
    Line2D line = new Line2D.Double();
    double y0 = dataArea.getMinY();
    double y1 = dataArea.getMaxY();
    g2.setPaint(im.getOutlinePaint());
    g2.setStroke(im.getOutlineStroke());
    if (range.contains(start)) {
      line.setLine(start2d, y0, start2d, y1);
      g2.draw(line);
    }
    if (range.contains(end)) {
      line.setLine(end2d, y0, end2d, y1);
      g2.draw(line);
    }
  }
  else if (orientation == VERTICAL) {
    Line2D line = new Line2D.Double();
    double x0 = dataArea.getMinX();
    double x1 = dataArea.getMaxX();
    g2.setPaint(im.getOutlinePaint());
    g2.setStroke(im.getOutlineStroke());
    if (range.contains(start)) {
      line.setLine(x0, start2d, x1, start2d);
      g2.draw(line);
    }
    if (range.contains(end)) {
      line.setLine(x0, end2d, x1, end2d);
      g2.draw(line);
    }
  }
}
```

(b) Optimal mapping with 25 mapped nodes and 2 differences.

Figure 13: Optimizing statement mapping

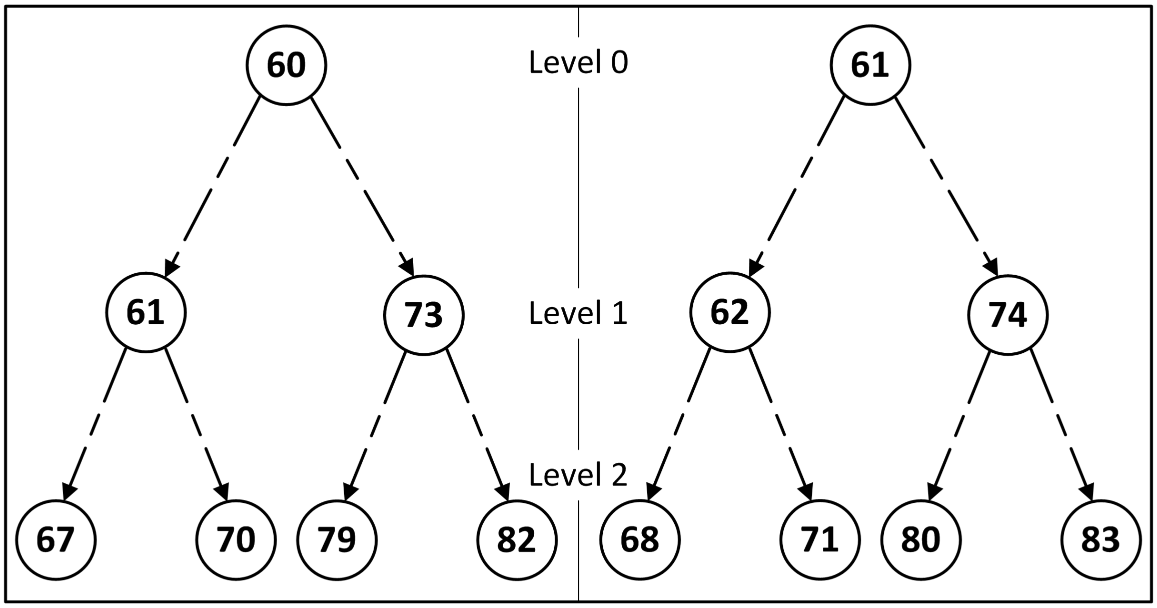


Figure 14: The Control Dependence Trees for the code fragments of Figure 13

3.3.3 Maximum Common Subgraph

Taking the two graphs (PDGs corresponding to the matched CDT pairs) as inputs, we want to find the largest common subgraph within the two graphs. This is called the Maximum Common Subgraph (MCS) problem. The maximum common subgraph can be defined for both disconnected and connected graphs. MCS problem has application in many areas such as bioinformatics [YAM04], chemistry [MW81] [RW02], video indexing [SBV01] and pattern recognition [CFSV04]. Finding MCS is an NP-complete problem [GJ79]. Therefore, many approximate algorithms have been developed. The worst case time complexity of the solution is exponential (more precisely, factorial) with respect to the number of nodes in the graphs. But in our case, due to the diversity in statement types and AST structure, we have a relatively limited number of node mappings (Section 4.1), which means that the search tree will not be very wide and in turn makes our problem smaller in size.

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. An algorithm for finding maximal common subgraphs of two given graphs is explained in [Lev72]. Conte et al. [CFV07] compared the performance of the three most representative optimal algorithms, which are based on depth-first tree search, namely:

1. the McGregor algorithm that searches for the maximum common subgraph by finding all common subgraphs of the two given graphs and choosing the largest one,
2. the Durand-Pasari algorithm that builds the *association graph* between the two given graphs and then searches for the maximum clique of the latter graph,
3. the Balas Yu algorithm that also searches for the maximum clique, but uses more sophisticated graph theory concepts for determining upper and lower bounds during the search process.

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 [CFV07]. 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 [CFV07]. Conte et al. [CFV07] concluded that the McGregor algorithm is more suitable for the applications that use regular graphs (i.e., graphs where each vertex has the same number of neighbors)

3.3.4 Divide-and-Conquer Algorithm

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 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. Keeping the best sub-solution for a particular set of control predicate nodes at each level is a greedy approach.

A greedy algorithm always makes the choice that looks best at the moment i.e., it makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution [CSRL09].

```
1  Function PDGMapping($CDT_i$, $CDT_j$)
       Data: Two isomorphic CDTs
       Result: The final mapping solution as finalSolution
2      $level_i \leftarrow CDT_i$.maxLevel
3      $level_j \leftarrow CDT_j$.maxLevel
       /* an initially empty solution                                  */
4      finalSolution $\leftarrow \varnothing$
5      while $level_i \geq 0$ and $level_j \geq 0$ do
6          $cpNodes_i \leftarrow$ nodes at $level_i$ of $CDT_i$
7          $cpNodes_j \leftarrow$ nodes at $level_j$ of $CDT_j$
8          foreach $cp_i \in cpNodes_i$ do
9              $mcsStates \leftarrow \varnothing$
10             foreach $cp_j \in cpNodes_j$ do
11                 if compatibleAST($cp_i$.parent, $cp_j$.parent)
12                 and compatibleAST($cp_i$, $cp_j$) then
13                     mapping $\leftarrow (cp_i, cp_j)$
14                     root $\leftarrow$ 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.

The reason why 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 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 14 shows the CDTs for the duplicated code fragments of Figure 13(a) which are isomorphic. 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).

```
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                                  */
    $Edges_i \leftarrow node_i$.inEdges $\cup$ $node_i$.outEdges
    $Edges_j \leftarrow node_j$.inEdges $\cup$ $node_j$.outEdges
    foreach $edge_i \in Edges_i$ do
        if $edge_i \notin$ pState.visitedEdges then
            add $edge_i \rightarrow$ pState.visitedEdges
            foreach $edge_j \in Edges_j$ do
                if compatibleEdges($edge_i$, $edge_j$) then
                    $vN_i \leftarrow edge_i$.otherEndPoint
                    $vN_j \leftarrow edge_j$.otherEndPoint
                    if compatibleAST($vN_i$, $vN_j$) and
                    mappedCtrlParents($vN_i$, $vN_j$) and
                    not alreadyMapped($vN_i$) and
                    not alreadyMapped($vN_j$) then
                        mapping $\leftarrow$ ($vN_i$, $vN_j$)
                        state $\leftarrow$ createState(mapping)
                        if not pruneBranch(state) then
                            add state $\rightarrow$ pState.children
                            search(state, mapping)
                        end if
                    end if
                end if
            end foreach
        end if
    end foreach
end
```

Algorithm 3: Recursive function building a search tree.

For the implementation of our MCS search technique (Algorithm 3), we have adopted the McGregor algorithm [McG82], 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 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 (and vice versa) in each path of the search tree(lines **13** and **14**).

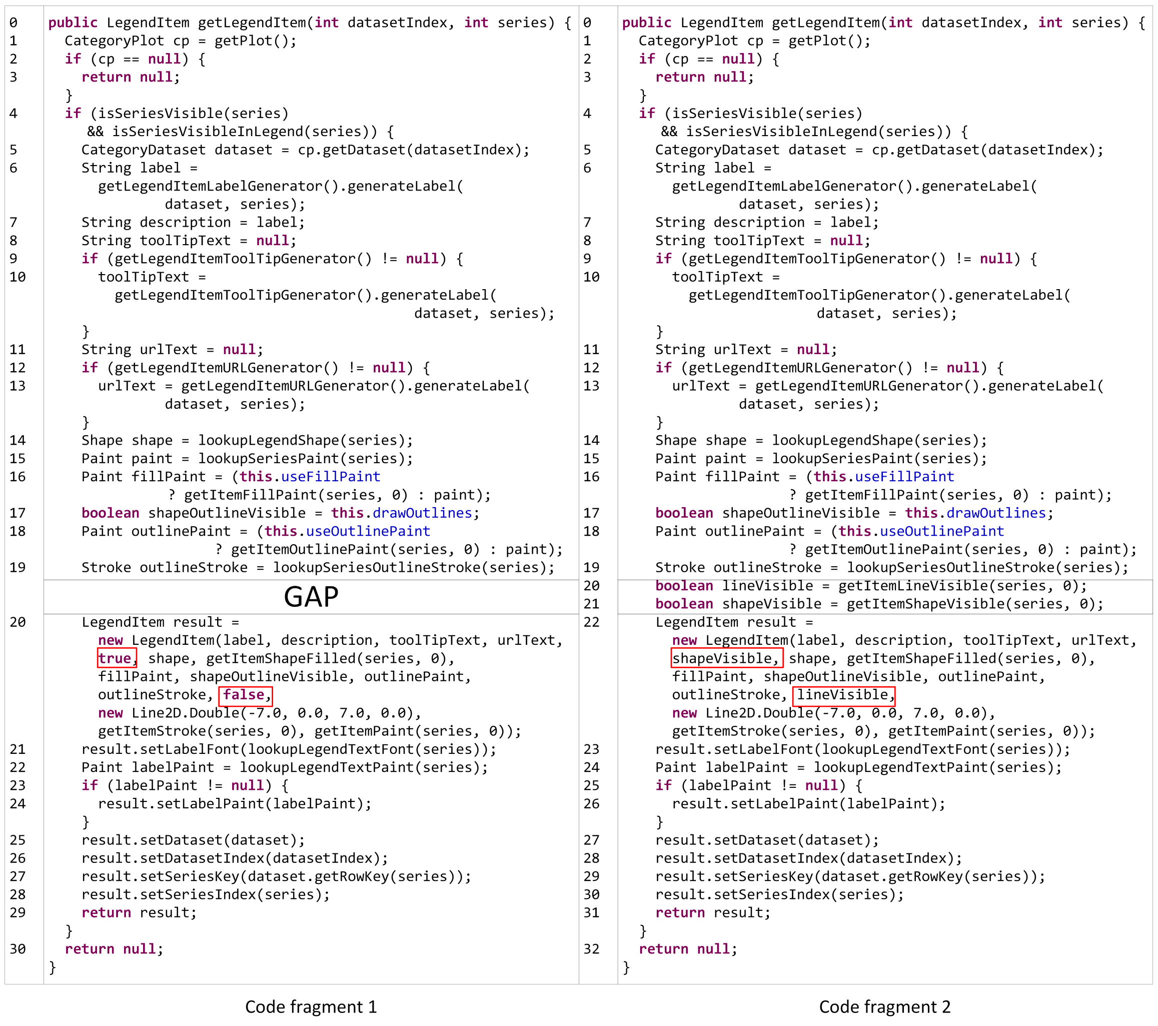
3. The control dependence structure of PDG_i and PDG_j is preserved throughout the mapping process. This means that if two control predicate nodes cp_i and cp_j have been mapped at a given stage of the search process, then a node nested under cp_i can only be mapped to nodes nested under cp_j (and vice versa) at later stages of the search process (line **12**).

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 **17**) 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.

Regarding PDG edge compatibility (line **8**), two PDG edges are considered compatible if they connect nodes which are AST 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.

3.3.5 Example for Divide-and-Conquer Algorithm

In this subsection, we will demonstrate a step-by-step execution of the `PDGMapping` algorithm (Algorithm 2). Figure 15 illustrates the code fragments of two methods named `getLegendItem` found in two different classes of the open-source project *JFreeChart* (version 1.0.14). The CDTs of the clone fragments are shown in Figure 16. The maximum common subgraph is built starting from the deepest level of the CDTs in a bottom-up fashion.



```
0   public LegendItem getLegendItem(int datasetIndex, int series) {
1     CategoryPlot cp = getPlot();
2     if (cp == null) {
3       return null;
      }
4     if (isSeriesVisible(series)
         && isSeriesVisibleInLegend(series)) {
5       CategoryDataset dataset = cp.getDataset(datasetIndex);
6       String label =
          getLegendItemLabelGenerator().generateLabel(
                  dataset, series);
7       String description = label;
8       String toolTipText = null;
9       if (getLegendItemToolTipGenerator() != null) {
10        toolTipText =
            getLegendItemToolTipGenerator().generateLabel(
                                                    dataset, series);
        }
11      String urlText = null;
12      if (getLegendItemURLGenerator() != null) {
13        urlText = getLegendItemURLGenerator().generateLabel(
                  dataset, series);
        }
14      Shape shape = lookupLegendShape(series);
15      Paint paint = lookupSeriesPaint(series);
16      Paint fillPaint = (this.useFillPaint
                  ? getItemFillPaint(series, 0) : paint);
17      boolean shapeOutlineVisible = this.drawOutlines;
18      Paint outlinePaint = (this.useOutlinePaint
                        ? getItemOutlinePaint(series, 0) : paint);
19      Stroke outlineStroke = lookupSeriesOutlineStroke(series);
                             GAP
20      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));
21      result.setLabelFont(lookupLegendTextFont(series));
22      Paint labelPaint = lookupLegendTextPaint(series);
23      if (labelPaint != null) {
24        result.setLabelPaint(labelPaint);
        }
25      result.setDataset(dataset);
26      result.setDatasetIndex(datasetIndex);
27      result.setSeriesKey(dataset.getRowKey(series));
28      result.setSeriesIndex(series);
29      return result;
      }
30    return null;
    }
```

Code fragment 1

```
0   public LegendItem getLegendItem(int datasetIndex, int series) {
1     CategoryPlot cp = getPlot();
2     if (cp == null) {
3       return null;
      }
4     if (isSeriesVisible(series)
         && isSeriesVisibleInLegend(series)) {
5       CategoryDataset dataset = cp.getDataset(datasetIndex);
6       String label =
          getLegendItemLabelGenerator().generateLabel(
                  dataset, series);
7       String description = label;
8       String toolTipText = null;
9       if (getLegendItemToolTipGenerator() != null) {
10        toolTipText =
            getLegendItemToolTipGenerator().generateLabel(
                        dataset, series);
        }
11      String urlText = null;
12      if (getLegendItemURLGenerator() != null) {
13        urlText = getLegendItemURLGenerator().generateLabel(
                  dataset, series);
        }
14      Shape shape = lookupLegendShape(series);
15      Paint paint = lookupSeriesPaint(series);
16      Paint fillPaint = (this.useFillPaint
                        ? getItemFillPaint(series, 0) : paint);
17      boolean shapeOutlineVisible = this.drawOutlines;
18      Paint outlinePaint = (this.useOutlinePaint
                        ? getItemOutlinePaint(series, 0) : paint);
19      Stroke outlineStroke = lookupSeriesOutlineStroke(series);
20      boolean lineVisible = getItemLineVisible(series, 0);
21      boolean shapeVisible = getItemShapeVisible(series, 0);
22      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));
23      result.setLabelFont(lookupLegendTextFont(series));
24      Paint labelPaint = lookupLegendTextPaint(series);
25      if (labelPaint != null) {
26        result.setLabelPaint(labelPaint);
        }
27      result.setDataset(dataset);
28      result.setDatasetIndex(datasetIndex);
29      result.setSeriesKey(dataset.getRowKey(series));
30      result.setSeriesIndex(series);
31      return result;
      }
32    return null;
    }
```

Code fragment 2

Figure 15: Example clone fragments from the open-source project *JFreeChart*

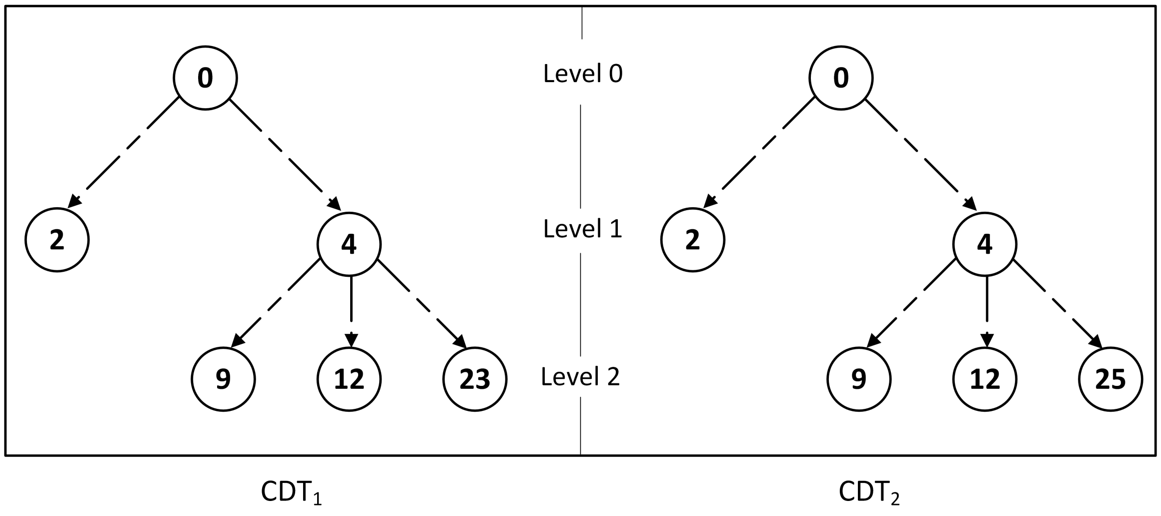


Figure 16: The Control Dependence Trees for the code fragments of Figure 15

Level 2

As we can see from the Figure 16, the nodes at Level 2 of CDT_1 are **{9,12,23}** and the nodes at Level 2 of CDT_2 are **{9,12,25}**. Table 4 shows the best mapping solutions found by considering the nodes at Level 2 of the CDTs. The column cp_i,cp_j refers to the control predicate nodes we consider at each iteration (Line **10** of Algorithm 2). As soon as we finish the inner loop (Line **18** of Algorithm 2), we select the `bestState` which is shown in the table. After each iteration, `bestState` will be augmented with the mappings from the current iteration.

Table 4: Mappings Found At Level 2

Iteration	cp_i,cp_j	***mcsStates***	**# differences**	*bestState*
#1	Node **9**, Node **9**	**{(9,9),(10,10)}**	0	**{(9,9),(10,10)}**
	Node **9**, Node **12**	**{(9,12),(10,13)}**	3	
	Node **9**, Node **25**	**{}**	N/A	
#2	Node **12**, Node **12**	*bestState* ∪ **{(12,12),(13,13)}**	0	**{(9,9),(10,10), (12,12),(13,13)}**
	Node **12**, Node **25**	**{}**	N/A	
#3	Node **23**, Node **25**	*bestState* ∪ **{(23,25),(24,26)}**	0	**{(9,9),(10,10), (12,12),(13,13), (23,25),(24,26)}**

The final solution after processing the control dependence nodes at Level 2 contains the best maximum common subgraph **{(9,9),(10,10),(12,12),(13,13),(23,25),(24,26)}**.

Level 1

From the Figure 16, we can see that the nodes at Level 1 of CDT_1 are **{2,4}** and the nodes at Level 1 of CDT_2 are **{2,4}**. Table 5 shows the best mapping solutions found by considering the nodes at Level 1 of the CDTs. These solutions are built on the maximum common subgraph obtained from the previous level of CDTs.

Table 5: Mappings Found At Level 1

Iteration	cp_i,cp_j	***mcsStates***	**# differences**	*bestState*
#1	Node **2**, Node **2**	**{(9,9),(10,10),(12,12),(13,13),(23,25),(24,26),(2,2),(3,3)}**	0	**{(9,9),(10,10),(12,12),(13,13),(23,25),(24,26),(2,2),(3,3)}**
	Node **2**, Node **4**	**{(9,9),(10,10),(12,12),(13,13),(23,25),(24,26),(3,31)}**	1	
#2	Node **4**, Node **4**	*bestState* ∪ **{(4,4),(5,5),(6,6),(7,7),(8,8),(11,11),(14,14),(15,15),(16,16),(17,17),(18,18),(19,19),(20,22),(21,23),(22,24),(25,27),(26,28),(27,29),(28,30),(29,31)}**	2	**{(9,9),(10,10),(12,12),(13,13),(23,25),(24,26),(2,2),(3,3),(4,4),(5,5),(6,6),(7,7),(8,8),(11,11),(14,14),(15,15),(16,16),(17,17),(18,18),(19,19),(20,22),(21,23),(22,24),(25,27),(26,28),(27,29),(28,30),(29,31)}**

The final solution contains the maximum common subgraph **{(9,9),(10,10),(12,12),(13,13),(23,25),(24,26),(2,2),(3,3),(4,4),(5,5),(6,6),(7,7),(8,8),(11,11),(14,14),(15,15),(16,16),(17,17),(18,18),(19,19),(20,22),(21,23),(22,24),(25,27),(26,28),(27,29),(28,30),(29,31)}**.

Level 0

The nodes at Level 0 of both CDTs in Figure 16 are only the method entry nodes **{0}** and **{0}**. There will be only one maximum common subgraph obtained at this level which is the final solution. At this level, the maximum common subgraph obtained from the previous level is augmented with the mappings **(1,1)** and **(30,32)**. The final solution contains the maximum common subgraph **{(9,9),(10,10),(12,12),(13,13),(23,25),(24,26),(2,2),(3,3),(4,4),(5,5),(6,6),(7,7),(8,8),(11,11),(14,14),(15,15),(16,16),(17,17),(18,18),(19,19),(20,22),(21,23),(22,24),(25,27),(26,28),(27,29),(28,30),(29,31),(1,1),(30,32)}**.

Thus we can see that by selecting the maximum common subgraph with the minimum number of differences at every level, and by combining the resulting sub-solutions, we get a final solution with reduced number of differences.

3.4 Clone Refactoring

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 [Opd92], each refactoring should be accompanied with a set of *preconditions*, which ensures 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.

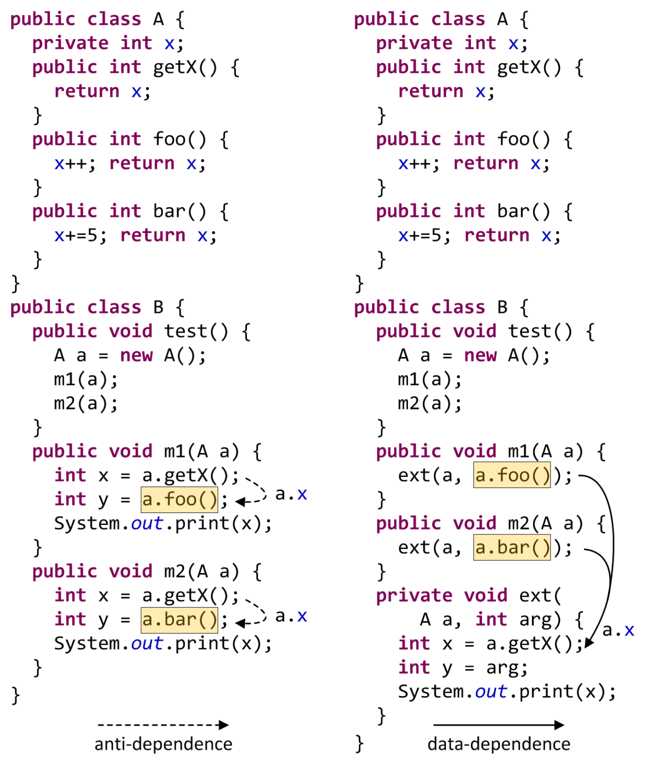
3.4.1 Preconditions Related to Clone Differences

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. Moreover, when there is a gap between the clone fragments, the unmapped statements should be moved before or after the extracted method which could alter the exisiting program behavior. The preconditions related to these clone differences are discussed in this section.

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

An anti-dependence [KH00] exists from statement p to statement q due to variable x, when there is a control flow path starting from statement p that uses the value of x and ending at statement q that modifies the value of x. Just like data flow dependences, anti-dependences can be either loop carried or loop independent. An output-dependence [KH00] exists from statement p to statement q due to variable x, when there is a control flow path starting from statement p that modifies the value of x and ending at statement q that also modifies the value of x.

In Figure 17(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 the accessor method `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 17(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 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 causes a change in the behavior of the program. In the original version in Figure 17(a), the execution of method `test` results in `m1` printing



```
public class A {
  private int x;
  public int getX() {
    return x;
  }
  public int foo() {
    x++; return x;
  }
  public int bar() {
    x+=5; return x;
  }
}
public class B {
  public void test() {
    A a = new A();
    m1(a);
    m2(a);
  }
  public void m1(A a) {
    int x = a.getX();
    int y = a.foo();
    System.out.print(x);
  }
  public void m2(A a) {
    int x = a.getX();
    int y = a.bar();
    System.out.print(x);
  }
}
```

```
public class A {
  private int x;
  public int getX() {
    return x;
  }
  public int foo() {
    x++; return x;
  }
  public int bar() {
    x+=5; return x;
  }
}
public class B {
  public void test() {
    A a = new A();
    m1(a);
    m2(a);
  }
  public void m1(A a) {
    ext(a, a.foo());
  }
  public void m2(A a) {
    ext(a, a.bar());
  }
  private void ext(
      A a, int arg) {
    int x = a.getX();
    int y = arg;
    System.out.print(x);
  }
}
```

(a) Before refactoring (b) After refactoring

Figure 17: Parameterization of a difference breaking an existing anti-dependence

0 and m2 printing 1, while in the refactored version in Figure 17(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 [GHJV95]. 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 [HHK12].

3.4.2 Preconditions Related to Method Extraction

Murphy-Hill and Black [MHB08] have recorded the most common preconditions for the Extract Method refactoring that were encountered during a formative study, in which they observed 11 programmers performing a number of Extract Method refactoring operations using the Eclipse refactoring tool. The following preconditions used in our approach guarantee that the conditions mentioned by Murphy-Hill and Black are preserved while extracting two duplicated code fragments into a separate method.

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 [TC11] (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.

Precondition 5: If two clones belong to different classes, these classes should extend a common superclass.

This precondition concerns clones belonging to different classes, and thus cannot be refactored by applying the *Extract Method* refactoring. If the classes, in which the clones are placed, extend a common superclass, and this superclass is a system class (i.e., not an external class from a library), then the clones could be extracted and pulled up to a new method in the common superclass. Pairs of commonly accessed instance variables from the subclasses (i.e., fields having the same type and being accessed at the same positions within the clones) should be also pulled up to the superclass. For each pair of commonly accessed instance methods from the subclasses (i.e., methods having the same signature and being called at the same positions within the clones), an abstract method having the same signature should be added in the superclass if they have different implementations. If they have the same implementation (i.e., they are also clones), they should be pulled up too.

If the classes, in which the clones are placed, do not extend a common superclass, and the clones do not access any instance variables or methods, then the duplicated code could be extracted as a new static method in a utility class.

Chapter 4

Evaluation

This section details the experiment we performed on source code clones found in different open source projects. The proposed technique is developed as part of an Eclipse plugin, which is named as JDeodorant[1]. To evaluate the effectiveness of our approach on the unification and refactoring of clones, the presented technique (JDeodorant) is compared against CeDAR [TG12], a state-of-the-art clone refactoring tool for Type-2 clones. Tairas and Gray [TG12] created a benchmark containing the *Type-2* clones detected in 9 open-source Java projects by the Deckard [JMSG07] clone detection tool. In their evaluation, they compared CeDAR against Eclipse IDE on a subset of the clone groups detected by the Deckard [JMSG07] clone detection tool. 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).

In order to perform a fair comparison, the evaluation of the technique is divided into two parts.

[1]http://www.jdeodorant.com/

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 closer 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.

4.1 Experiment

Within the context of the experiment we applied the following workflow. 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 3.3), and the examination of preconditions (Section 3.4). 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 6: Refactorable Clone Groups Found By Eclipse, CeDAR And JDeodorant

Project	**CG_1**[*]	Eclipse	CeDAR	JDeodorant	Δ
Apache Ant 1.7.0	120	14 (12%)	28 (23%)	50 (42%)	+79%
Columba 1.4	88	13 (15%)	30 (34%)	41 (47%)	+37%
EMF 2.4.1	149	8 (5%)	14 (9%)	54 (36%)	+286%
Jakarta-JMeter 2.3.2	68	3 (4%)	11 (16%)	20 (29%)	+82%
JEdit 4.2	157	15 (10%)	20 (13%)	57 (36%)	+185%
JFreeChart 1.0.10	291	29 (10%)	62 (21%)	87 (30%)	+40%
JRuby 1.4.0	81	23 (28%)	23 (28%)	33 (41%)	+43%
Total	954	105 (11%)	188 (20%)	342 (36%)	+82%

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

Table 6 presents the number of clone groups assessed as refactorable by Eclipse, CeDAR, and JDeodorant (the tool implementing the proposed technique), respectively. The results for Eclipse and CeDAR were taken from [TG12]. 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 [TG12] did not correspond to the source code of the aforementioned versions. The authors were contacted about this issue, but they were not able to provide the actual source code versions from which the clone detection results were derived. Table 7 presents additional clone groups (in which clones belong to different Java files) assessed as refactorable by JDeodorant.

Table 7: Additional Refactorable Clone Groups Found By JDeodorant

Project	**CG_2**[*]	JDeodorant
Apache Ant 1.7.0	211	40 (19%)
Columba 1.4	275	66 (24%)
EMF 2.4.1	58	17 (29%)
Jakarta-JMeter 2.3.2	225	68 (30%)
JEdit 4.2	101	35 (35%)
JFreeChart 1.0.10	337	121 (36%)
JRuby 1.4.0	181	46 (25%)
Total	1388	393 (28.3%)

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

4.2 Discussion

As it can be observed in Table 6, our proposed technique 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 82% 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 or Eclipse 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 7 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 28.3% of these clones groups. The difference in the percentage of clones within the same and different files found as refactorable (36% vs. 28.3%) probably indicates that the clones within the same files have a stronger similarity and their unification is easier.

Another interesting insight, presented in Table 8, is that 8.4% 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 [TC11], 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 37% (870 out of 2342 clone groups in total).

Table 8: Non-Refactorable Clone Groups Violating Only Precondition #3

Project	**Non-Refactorable**[*]	Violations
Apache Ant 1.7.0	239	27 (11%)
Columba 1.4	256	7 (3%)
EMF 2.4.1	141	6 (4%)
Jakarta-JMeter 2.3.2	205	6 (3%)
JEdit 4.2	180	14 (8%)
JFreeChart 1.0.10	420	57 (14%)
JRuby 1.4.0	184	18 (10%)
Total	1607	135 (8.4%)

[*] column Non-Refactorable 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.

4.3 Performance Analysis

In this section, we analyze the performance of our matching technique by comparing it with a brute-force matching approach. We consider as brute-force an approach that makes all possible comparisons between the statements of two clones. Assuming that the first clone has n_1 statements and the second clone has n_2 statements, the maximum number of statement comparisons is equal to $n_1 \times n_2$. In order to perform this comparison, we calculated the number of node comparisons performed by our PDG mapping approach and the brute-force matching approach for each of the clone groups in the examined projects (Section 4.1). If a clone group has more than two clones, we take the sum of the node comparisons resulting from the matching of every possible clone pair.

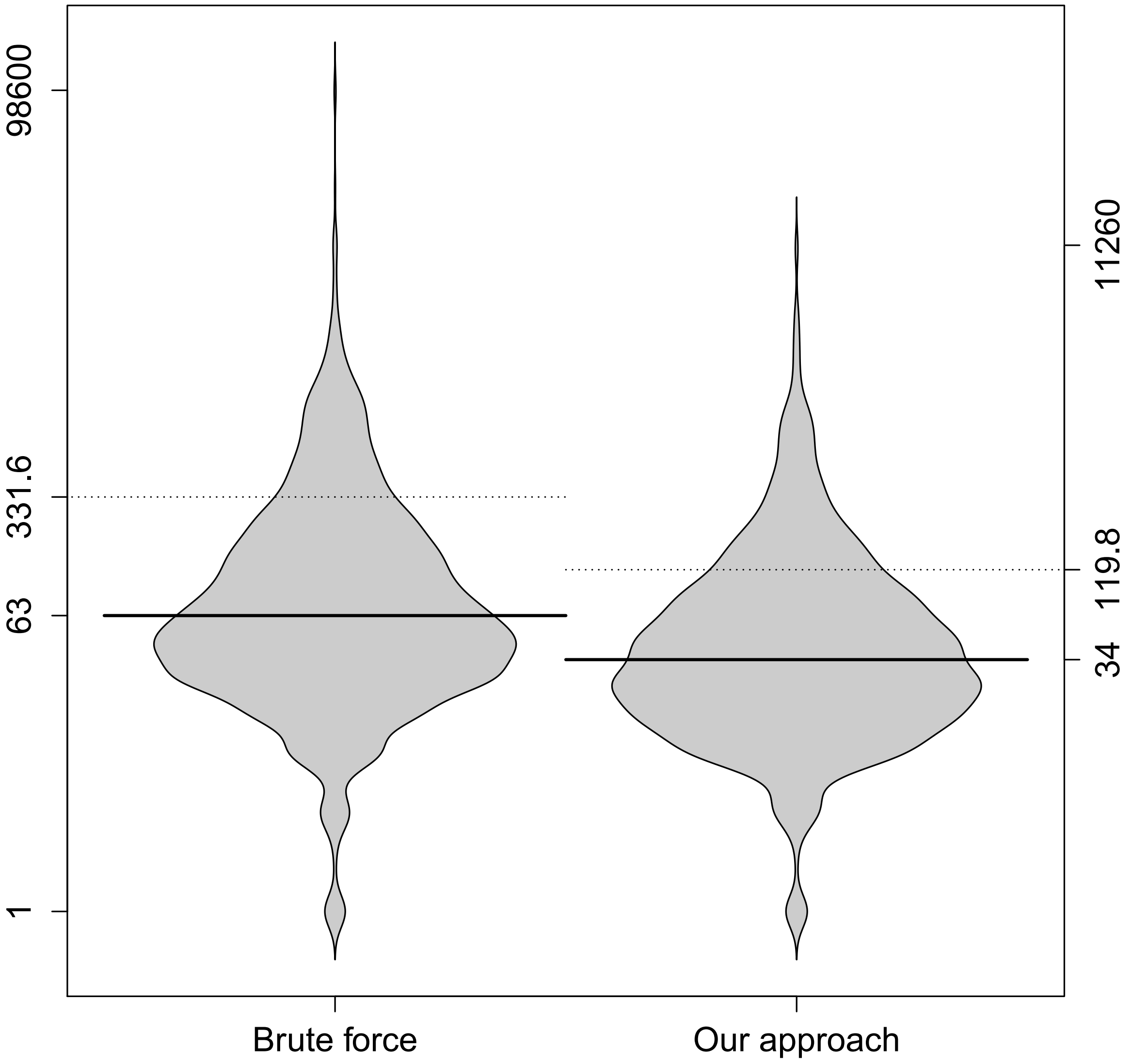


Figure 18: Beanplots for the numbers of node comparisons using a brute-force approach and our mapping approach

Figure 18 shows the beanplots [Kam08] for the number of node comparisons performed by the brute-force approach and our mapping approach. A beanplot is an alternative to the boxplot for visual comparison of univariate data between groups, visualizing the estimated density of the distributions. As you can see in the figure, the median values are 63 and 34 for the brute-force approach and our mapping approach, while the mean values are 331.6 and 119.8, respectively. From these results, we can conclude that our approach requires only three times less node comparisons on average, compared to the brute-force approach in order to determine the refactorability of clones.

Finally, we computed the CPU time taken for the execution of the PDG mapping process for each of the clone groups in our experiment (Section 4.1). The mean value is found to be 354.1 milliseconds. These results show that our approach has a negligible computation cost and could be easily integrated into the clone management practice to assess whether clones can be safely refactored in a way that minimizes the number of required parameters.

4.4 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 [TG12].

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.

Chapter 5

Conclusion and Future Work

This work is the first step towards a broader research goal, namely assisting developers in the refactoring of software clones. To this end, we developed a clone unification and refactoring technique that overcomes some of 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 a 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 extensive empirical study on the refactorability of clones detected from different clone detection tools such as ConQat [DPS05], NiCad [CR11] and CCFinder [KKI02] in addition to Deckard.

In the evaluation of our approach, we performed a direct comparison with CeDAR [TG12], 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 82% more refactorable clone groups than CeDAR, and additionally assessed as refactorable 28.3% of the clone groups in which clones belong to different Java files (a feature not supported by CeDAR). Furthermore, the study revealed that 37% of the clone groups in the examined projects can be refactored **directly** or in the form of **sub-clones**

(Section 4.2). 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 [TKF11]. 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 [HHM$^+$04] 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.

Bibliography

[BKSM13] Yixin Bian, Gunes Koru, Xiaohong Su, and Peijun Ma. SPAPE: A semantic-preserving amorphous procedure extraction method for near-miss clones. *Journal of Systems and Software*, 86(8):2077–2093, 2013.

[BMD$^+$00] Magdalena Balazinska, Ettore Merlo, Michel Dagenais, Bruno Lagüe, and Kostas Kontogiannis. Advanced clone-analysis to support object-oriented system refactoring. In *Proceedings of the Seventh Working Conference on Reverse Engineering*, pages 98–107, 2000.

[BS87] Salah Bendifallah and Walt Scacchi. Understanding software maintenance work. *IEEE Transactions on Software Engineering*, 13:311–323, 1987.

[CFSV04] Donatello Conte, Pasquale Foggia, Carlo Sansone, and Mario Vento. Thirty years of graph matching in pattern recognition. *International Journal of Pattern Recognition and Artificial Intelligence*, 18(3):265–298, 2004.

[CFV07] Donatello Conte, Pasquale Foggia, and Mario 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.*, 11(1):99–143, 2007.

[CR11] James R. Cordy and Chanchal K. Roy. The NiCad clone detector. In *Proceedings of the 2011 IEEE 19th International Conference on Program Comprehension*, pages 219–220, 2011.

[CRGMW96] Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. Change detection in hierarchically structured information. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 493–504, 1996.

[CSRL09] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.

[CWD08] Rylan Cottrell, Robert J. Walker, and Jörg Denzinger. Semi-automating small-scale source code reuse via structural correspondence. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 214–225, 2008.

[CYI$^+$11] Eunjong Choi, Norihiro Yoshida, Takashi Ishio, Katsuro Inoue, and Tateki Sano. Extracting code clones for refactoring using combinations of clone metrics. In *Proceedings of the 5th International Workshop on Software Clones*, pages 7–13, 2011.

[DBFF95] Neil Davey, Paul Barson, Simon Field, and Ray J Frank. The development of a software clone detector. *International Journal of Applied Software Technology*, 1995.

[DER07] Ekwa Duala-Ekoko and Martin P. Robillard. Tracking code clones in evolving software. In *Proceedings of the 29th International Conference on Software Engineering*, pages 158–167, 2007.

[DPS05] Florian Deissenboeck, Markus Pizka, and Tilmann Seifert. Tool support for continuous quality assessment. In *Proceedings of the International Workshop on Software Technology and Engineering Practice*, pages 127–136. IEEE CS Press, 2005.

[FOW87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, July 1987.

[Fow99] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA, 1999.

[FWPG07] Beat Fluri, Michael Wuersch, Martin PInzger, and Harald Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Trans. Softw. Eng.*, 33(11):725–743, November 2007.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

[Gui83] Tor Guimaraes. Managing application program maintenance expenditures. *Commun. ACM*, 26(10):739–746, October 1983.

[GYI$^+$13] Akira Goto, Norihiro Yoshida, Masakazu Ioka, Eunjong Choi, and Katsuro Inoue. How to extract differences from similar programs? A cohesion metric approach. In *Proceedings of the 7th International Workshop on Software Clones*, 2013.

[HHK12] Keisuke Hotta, Yoshiki Higo, and Shinji 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*, pages 53–62, 2012.

[HHM$^+$04] Mark Harman, Lin Hu, Malcolm Munro, Xingyuan Zhang, Dave Binkley, Sebastian Danicic, Mohammed Daoudi, and Lahcen Ouarbya. Syntax-directed amorphous slicing. *Automated Software Engineering*, 11(1):27–61, 2004.

[HK11] Yoshiki Higo and Shinji Kusumoto. Code clone detection on specialized PDGs with heuristics. In *Proceedings of the 15th European Conference on Software Maintenance and Reengineering*, pages 75–84, 2011.

[HKI08] Yoshiki Higo, Shinji Kusumoto, and Katsuro 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.

[HR92] S. Horwitz and T. Reps. The use of program dependence graphs in software engineering. In *Software Engineering, 1992. International Conference on*, pages 392–411, 1992.

[JDHW09] Elmar Juergens, Florian Deissenboeck, Benjamin Hummel, and Stefan Wagner. Do code clones matter? In *Proceedings of the 31st International Conference on Software Engineering*, pages 485–495, 2009.

[JH07] Nicolas Juillerat and Béat Hirsbrunner. Toward an implementation of the "Form Template Method" refactoring. In *Proceedings of the 7th IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2007)*, pages 81–90, Paris, France, 2007.

[JMSG07] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. DECKARD: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th International Conference on Software Engineering*, pages 96–105, 2007.

[Kam08] Peter Kampstra. Beanplot: A boxplot alternative for visual comparison of distributions. *Journal of Statistical Software, Code Snippets*, 28(1):1–9, October 2008.

[KDM$^+$96] K. A. Kontogiannis, R. Demori, E. Merlo, M. Galler, and M. Bernstein. Reverse engineering. chapter Pattern matching for clone and concept detection, pages 77–108. Kluwer Academic Publishers, 1996.

[KFF06] Rainer Koschke, Raimar Falke, and Pierre Frenzel. Clone detection using abstract syntax suffix trees. In *Proceedings of the 13th Working Conference on Reverse Engineering*, WCRE '06, pages 253–262, Washington, DC, USA, 2006. IEEE Computer

Society.

[KG08] Cory J. Kapser and Michael W. Godfrey. "Cloning considered harmful" considered harmful: Patterns of cloning in software. *Empirical Softw. Engg.*, 13(6):645–692, December 2008.

[KH00] Raghavan Komondoor and Susan Horwitz. Semantics-preserving procedure extraction. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '00, pages 155–169, New York, NY, USA, 2000. ACM.

[KH01] Raghavan Komondoor and Susan Horwitz. Using slicing to identify duplication in source code. In *Proceedings of the 8th International Symposium on Static Analysis*, pages 40–56, 2001.

[KKI02] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Ccfinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Software Eng.*, 28(7):654–670, 2002.

[Kos08] Rainer Koschke. Frontiers of software clone management. In *Frontiers of Software Maintenance*, pages 119–128, 2008.

[Kri01] Jens Krinke. Identifying similar code with program dependence graphs. In *Proceedings of the Eighth Working Conference on Reverse Engineering*, pages 301–307, 2001.

[LCHY06] Chao Liu, Chen Chen, Jiawei Han, and Philip S. Yu. GPLAG: Detection of software plagiarism by program dependence graph analysis. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 872–881, 2006.

[Lev72] G. Levi. A note on the derivation of maximal common subgraphs of two directed or undirected graphs. *Calcolo*, 9, 1972.

[LS80] Bennett P. Lientz and E. Burton Swanson. *Software Maintenance Management.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1980.

[LW08] Angela Lozano and Michel Wermelinger. Assessing the effect of clones on changeability. In *Proceedings of the 24th IEEE International Conference on Software Maintenance*, pages 227–236, 2008.

[McG82] James J. McGregor. Backtrack search algorithms and the maximal common subgraph problem. *Software: Practice and Experience*, 12(1):23–34, 1982.

[McK84] James R. McKee. Maintenance as a function of design. In *Proceedings of the July 9-12, 1984, National Computer Conference and Exposition*, American Federation of Information Processing Societies '84, pages 187–193, New York, NY, USA, 1984. ACM.

[MdR04] Gilad Mishne and Maarten de Rijke. Source code retrieval using conceptual similarity. In *Proceedings of the 2004 Conf. Computer Assisted Information Retrieval (RIAO 04)*, pages 539–554, 2004.

[MHB08] E. Murphy-Hill and A.P. Black. Refactoring tools: Fitness for purpose. *Software, IEEE*, 25(5):38–44, 2008.

[MHPB09] Emerson Murphy-Hill, Chris Parnin, and Andrew P. Black. How we refactor, and how we know it. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 287–297, Washington, DC, USA, 2009. IEEE Computer Society.

[MM83] J. Martin and C. McClure. *Software Maintenance: The Problem and its Solutions.* Prentice Hall, London, 1983.

[MRS$^+$11] Manishankar Mondal, Md. Saidur Rahman, Ripon K. Saha, Chanchal K. Roy, Jens Krinke, and Kevin A. Schneider. An empirical study of the impacts of clones in software maintenance. In *Proceedings of the 2011 IEEE 19th International Conference on Program Comprehension*, ICPC '11, pages 242–245, Washington, DC, USA, 2011. IEEE Computer Society.

[MRS12] Manishankar Mondal, Chanchal K. Roy, and Kevin A. Schneider. An empirical study on clone stability. *ACM Special Interest Group on Applied Computing (SIGAPP) Review*, 12(3):20–36, September 2012.

[MRS13] Manishankar Mondal, Chanchal K. Roy, and Kevin A. Schneider. Insight into a method co-change pattern to identify highly coupled methods: An empirical study. In *2013 IEEE 21st International Conference on Program Comprehension*, pages 103–112, 2013.

[MW81] James J. McGregor and Peter Willett. Use of a maximum common subgraph algorithm in the automatic identification of ostensible bond changes occurring in chemical reactions. *Journal of Chemical Information and Computer Sciences*, 21(3):137–140, 1981.

[NNP$^+$12] Hoan Anh Nguyen, Tung Thanh Nguyen, Nam H. Pham, Jafar Al-Kofahi, and Tien N. Nguyen. Clone management for evolving software. *IEEE Transactions on Software Engineering*, 38(5):1008–1026, 2012.

[OO84] Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in a software development environment. *SIGPLAN Not.*, 19(5):177–184, April 1984.

[Opd92] William F. Opdyke. *Refactoring object-oriented frameworks.* PhD thesis, University of Illinois at Urbana-Champaign, IL, USA, 1992.

[RC07] Chanchal Kumar Roy and James R. Cordy. A survey on software clone detection research. *School Of Computing TR 2007-541, Queens University*, 115, 2007.

[RW02] John W. Raymond and Peter Willett. Maximum common subgraph isomorphism algorithms for the matching of chemical structures. *Journal of Computer-Aided Molecular Design*, 16(7):521–533, 2002.

[SB12] Daniel Speicher and Andri Bremm. Clone removal in Java programs as a process of stepwise unification. In *Proceedings of the 26th Workshop on Logic Programming*, 2012.

[SBV01] Kim Shearer, Horst Bunke, and Svetha Venkatesh. Video indexing and similarity retrieval by largest common subgraph detection using decision trees. *Pattern Recognition*, 34(5):1075–1091, 2001.

[SGP04] David Shepherd, Emily Gibson, and Lori L. Pollock. Design and evaluation of an automated aspect mining tool. In *Software Engineering Research and Practice*, pages 601–607, 2004.

[SPL03] Robert C. Seacord, Daniel Plakosh, and Grace A. Lewis. *Modernizing Legacy Systems: Software Technologies, Engineering Process and Business Practices.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[Swa76] E. Burton Swanson. The dimensions of maintenance. In *Proceedings of the 2nd International Conference on Software Engineering*, ICSE '76, pages 492–497, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[TC11] Nikolaos Tsantalis and Alexander Chatzigeorgiou. Identification of extract method refactoring opportunities for the decomposition of methods. *J. Syst. Softw.*, 84(10):1757–1782, October 2011.

[TG12] Robert Tairas and Jeff Gray. Increasing clone maintenance support by unifying clone detection and refactoring activities. *Inf. Softw. Technol.*, 54(12):1297–1307, December 2012.

[TKF11] Rebecca Tiarks, Rainer Koschke, and Raimar Falke. An extended assessment of type-3 clones as detected by state-of-the-art tools. *Software Quality Journal*, 19(2):295–331, 2011.

[Val02] Gabriel Valiente. *Algorithms on Trees and Graphs.* Springer-Verlag, 2002.

[Vli08] Hans van Vliet. *Software Engineering: Principles and Practice.* Wiley Publishing, 3rd edition, 2008.

[WRW03] N. Walkinshaw, M. Roper, and M. Wood. The Java System Dependence Graph. In *Proceedings of the Third IEEE International Workshop on Source Code Analysis and Manipulation, 2003*, pages 55–64, 2003.

[XXJ11] Zhenchang Xing, Yinxing Xue, and S. Jarzabek. Clonedifferentiator: Analyzing clones by differentiation. In *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*, pages 576–579, 2011.

[YAM04] Atsuko Yamaguchi, Kiyoko F. Aoki, and Hiroshi Mamitsuka. Finding the maximum common subgraph of a partial k-tree and a graph with a polynomially bounded number of spanning trees. *Inf. Process. Lett.*, 92(2):57–63, October 2004.