Assessing the Refactorability of Software Clones

Nikolaos Tsantalis, *Member, IEEE*, Davood Mazinanian, and Giri Panamoottil Krishnan

Abstract—The presence of duplicated code in software systems is significant and several studies have shown that clones can be potentially harmful with respect to the maintainability and evolution of the source code. Despite the significance of the problem, there is still limited support for eliminating software clones through refactoring, because the unification and merging of duplicated code is a very challenging problem, especially when software clones have gone through several modifications after their initial introduction. In this work, we propose an approach for automatically assessing whether a pair of clones can be safely refactored without changing the behavior of the program. In particular, our approach examines if the differences present between the clones can be safely parameterized without causing any side-effects. The evaluation results have shown that the clones assessed as refactorable by our approach can be indeed refactored without causing any compile errors or test failures. Additionally, the computational cost of the proposed approach is negligible (less than a second) in the vast majority of the examined cases. Finally, we perform a large-scale empirical study on over a million clone pairs detected by four different clone detection tools in nine open-source projects to investigate how refactorability is affected by different clone properties and tool configuration options. Among the highlights of our conclusions, we found that a) clones in production code tend to be more refactorable than clones in test code, b) clones with a close relative location (i.e., same method, type, or file) tend to be more refactorable than clones in distant locations (i.e., same hierarchy, or unrelated types), c) *Type-1* clones tend to be more refactorable than the other clone types, and d) clones with a small size tend to be more refactorable than clones with a larger size.

Index Terms—Code duplication, software clone management, clone refactoring, refactorability assessment, empirical study

1 Introduction

Code duplication has been recognized as a potentially serious problem having a negative impact on the maintainability, comprehensibility, and evolution of software systems. Over the years, the software clone research community has developed several techniques for the detection and analysis of duplicated code [1], and more recently has focused on clone management activities [2], such as tracing clones in the history of a project, analyzing the consistency of modifications to clones [3], updating incrementally clone groups as the project evolves [4], and prioritizing the refactoring of clones [5], [6].

In addition to the development of tools and techniques for the detection and management of software clones, several researchers investigated empirically the effect of duplicated code on maintenance effort and cost [7], error-proneness due to inconsistent updates [8], [9], software defects [10], change-proneness [11], and change propagation [12]. However, to the best of our knowledge, there is no study investigating the refactorability of software clones. *What portion of the clones detected by tools can be actually refactored?* Additionally, there is a lack of tools that can automatically analyze software clones to determine whether they can be *safely* refactored without changing the program behavior. *Refactorability analysis* is an important missing feature from clone management, since it could be used to filter clones that can be directly refactored, when the developers are interested in finding refactoring opportunities for duplicated code. In this way, maintainers can focus their effort on parts of the code that can immediately benefit from refactoring, and thus expedite maintainability improvement.

In this paper, we present an approach that takes as input two clone fragments detected from any tool and applies three steps to determine whether they can be safely refactored (i.e., without any side effects). First, our approach finds code fragments with identical nesting structures within the input clones that could serve as *potential refactoring opportunities*. We consider that two code fragments can be unified, and therefore refactored, if they share a common nesting structure. In the second step, our approach finds a mapping between the statements of the code fragments that maximizes the number of mapped statements and minimizes the number of differences between the mapped statements by exploring the search space of alternative mapping solutions. This is generally an NP-hard problem [13], and since exhaustive search is impractical, our solution relies on heuristics to reduce the search space. From the refactoring point of view, we support that a mapping solution with a smaller number of differences between the mapped statements has a *higher refactorability* compared to an alternative mapping solution with a larger number of differences. The reason is that some differences cannot be safely parameterized, and thus a larger number of differences increases the probability of side effects from the parameterization of differences. Finally, in the last step, the differences between

• *The authors are with the Department of Computer Science and Software Engineering, Concordia University, Montreal, Quebec, Canada H3G 1M8. E-mail: {nikolaos.tsantalis, giri.krishnan}@concordia.ca, d_mazina@cse.concordia.ca.*

Manuscript received 8 Sept. 2014; revised 5 May 2015; accepted 16 June 2015. Date of publication 21 June 2015; date of current version 13 Nov. 2015. Recommended for acceptance by A. Hassan.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TSE.2015.2448531

the mapped statements detected in the previous step are examined against a set of preconditions to determine whether they can be parameterized without changing the program behavior. Additionally, the statements that have not been mapped in the previous step are examined against a set of preconditions to determine whether they can be safely moved before or after the execution of the extracted method containing the mapped statements.

This paper is an extension over our previous work [14], which contains the following improvements and additions:

1) We replace the *Control Dependence Tree* with the *Program Structure Tree* (PST) [15] for representing the nesting structure of the source code. The reason for this replacement is that control dependencies can form graphs in the case of unstructured control or exception flow, while the PST is defined for arbitrary flow graphs, even irreducible ones.
2) We support additional types of differences in the AST matching of statements (Section 3.1) to make more flexible the unification of statements with non-trivial differences and improve the quality of the refactoring solution.
3) We introduce conditions in our divide-and-conquer algorithm to guarantee that the resulting sub-solutions can be safely combined into a valid global solution respecting the initial nesting structure of the clone fragments (Section 3.3.2).
4) We introduce an additional decomposition phase of the statement mapping problem (Section 3.3.4) to deal better with the problem of combinatorial explosion.
5) We extend the list of examined preconditions with four new preconditions (Section 3.4). Additionally, we provide formal ways to detect precondition violations.
6) We provide tool support for the automatic refactorability analysis of software clones, the visual inspection of the differences and precondition violations found in a pair of clone fragments, and the refactoring of method-level clone fragments (Section 4).
7) We evaluate the correctness and efficiency of our approach (Sections 5.2 and 5.3).

Our technique supports the analysis of clones detected in Java programs, and therefore the defined preconditions are adjusted to the features/limitations of the Java programming language. For instance, the fact that Java does not support parameter passing *by reference*, makes necessary the definition of some preconditions that could be overcome by languages supporting this feature. The defined preconditions cover *Type-1* clones (i.e., identical code fragments except for variations in whitespace, layout, and comments [1]), *Type-2* clones (i.e., structurally/syntactically identical fragments except for variations in identifiers, literals and types in addition to *Type-1* differences [1]), and *Type-3* clones (i.e., copied fragments with statements changed, added or removed in addition to *Type-2* differences [1]).

Clone detection tools apply many different approaches for the detection of duplicated code fragments, including text-based, token-based, tree-based, metrics-based, and graph-based techniques [1] and have a variety of configuration options. Both of these factors (i.e., detection approach and configuration options) affect the quality of the detected clones with respect to refactorability. For instance, text-based and token-based approaches are more likely to return clone fragments having incomplete statements (i.e., partially matched statements) or different nesting structures compared to tree-based and graph-based approaches [16]. Therefore, it is very interesting to investigate the performance of different clone detection techniques with respect to the refactorability of the clones they detect, and how different configuration options may affect the quality of their results.

This work makes two main contributions in the area of software clone management:

1) We present a reliable and efficient approach to automatically assess the refactorability of software clones. To the best of our knowledge, this is the first approach to provide such an in-depth solution to this problem.
2) We conduct a large-scale empirical study (Section 5.4) on 1,150,967 clone pairs to investigate the refactorability of software clones taking into account different dimensions:
   - Source code type (production versus test code)
   - Clone location (same file versus different files)
   - Clone type (Type-1 versus Type-2 versus Type-3)
   - Clone size
   - Precondition violation types.

2 Preliminaries

In this section, we will briefly describe two core program structures that are used in our approach.

2.1 Program Structure Tree

The Program Structure Tree was introduced by Johnson et al. [15] as a hierarchical representation of program structure based on single-entry single-exit (SESE) regions of the control flow graph. Johnson et al. extended the notion of dominance and postdominance to control flow edges and defined SESE region as an ordered edge pair (a, b) of distinct control flow edges a (entry edge) and b (exit edge) where:

> "1) a dominates b
> 2) b postdominates a
> 3) every cycle containing a also contains b and vice versa."

The PST essentially captures the nesting relationship of SESE regions, as well as chains of sequentially composed SESE regions. Fig. 1c depicts the PST for the code example shown in Fig. 1a. Fig. 1b shows the control flow graph with its SESE regions marked in dotted rectangles. The nesting structure of the SESE regions in the control flow graph is used to generate the PST. Chains of sequentially composed SESE regions, such as regions A and D, are grouped within a dotted rectangle in the PST.

As it will be explained later in Section 3.2, the first step of our refactorability analysis approach involves the detection of matching nesting structures within two clone fragments. The nesting structure of a program is essentially captured by the control predicate nodes of the PST. For this purpose, we

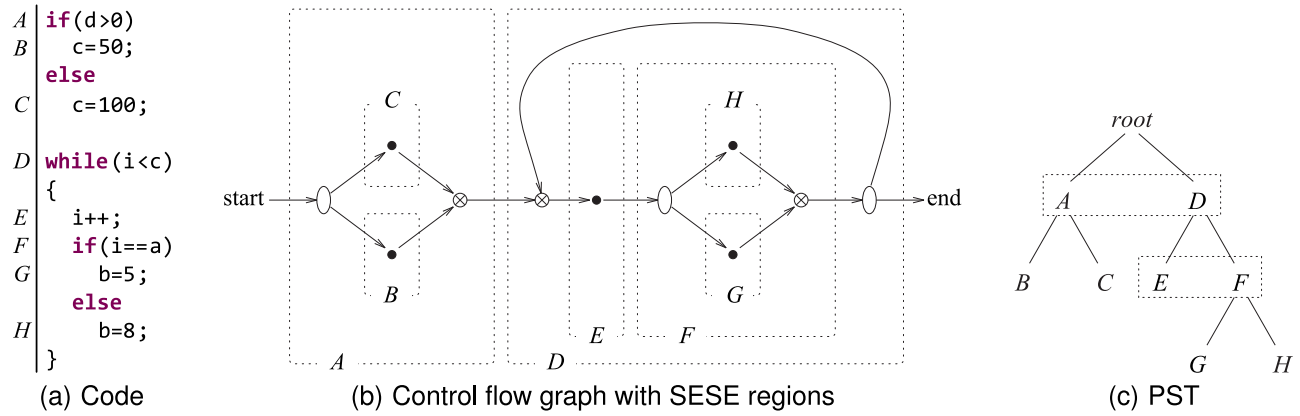


Fig. 1. Generating the program structure tree from a control flow graph with SESE regions.

define the *Nesting Structure Tree* (NST), as the tree that contains only the control predicate nodes (e.g., `if`, `for` statements) of the original PST. For example, the NST for the PST of Fig. 1c contains only nodes *root*, *A*, *D*, and *F*.

2.2 Program Dependence Graph

The Program Dependence Graph (PDG) [17] is a directed graph with multiple edge types, in which the nodes represent the statements of a function or method, and the edges represent control and data flow dependencies between statements. More specifically, we distinguish two kinds of statements, namely control predicate statements (i.e., statements with a body e.g., `if`, `for`) and non-predicate statements (i.e., leaf statements without a body). In the case of a control predicate statement, we consider only its conditional expression(s) (i.e., we ignore the statements inside its body) when computing dependencies from/to it. In the case of a leaf statement, we consider the entire statement (i.e., all expressions inside it) when computing dependencies from/to it.

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 at the start point of the edge. A *data dependence* edge is always labeled with a variable v and denotes that the statement at the end point of the edge is using the value of v, which has been previously modified by the statement at the start point of the edge. If the data dependence is carried through a loop node l, then it is considered as a *loop-carried* dependence.

The PDG representation used in this paper is extended in two ways. First, we introduce composite variables [18] representing the state of the objects being referenced within the body of a method, and create additional data dependencies for these variables by analyzing method calls that may modify or use the state of the referenced objects. Second, we add two more types of edges in the PDG, which are used in the examination of preconditions (Section 3.4). These edges are anti-dependencies and output-dependencies. An *anti-dependence* edge due to variable v denotes that the statement at the end point of the edge is modifying the value of v, which has been used by the statement at the start point of the edge (i.e., the opposite of a data dependence). An *output-dependence* edge due to variable v denotes that both statements at the start and end points of the edge modify the value of v.

3 Approach

Our approach is designed to process two different forms of input:

1) Two code fragments within the body of the same method, or different methods, reported as clones by a clone detection tool.
2) Two method declarations considered to be duplicated (i.e., method-level clones), or containing duplicate code fragments somewhere inside their bodies.

In a nutshell, our approach for assessing the refactorability of two clone fragments comprises three major steps, as shown in Fig. 2:

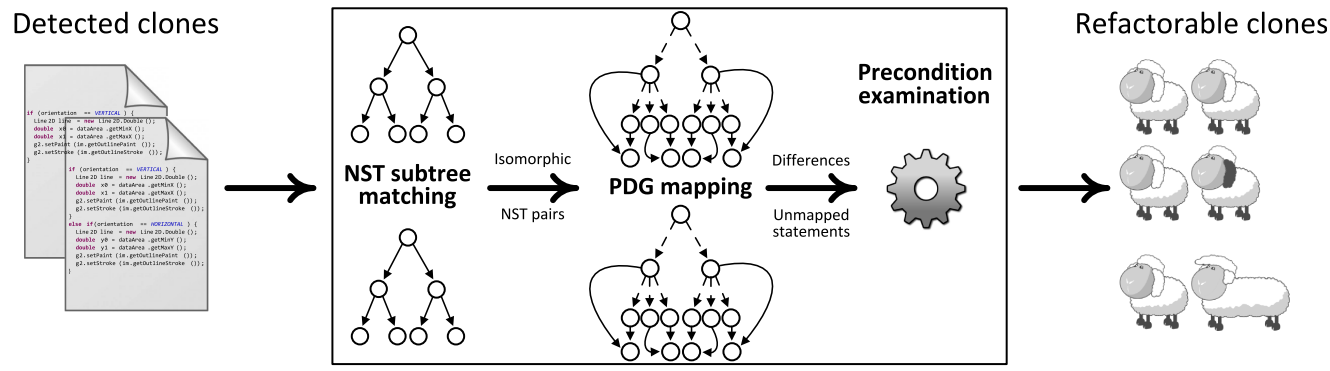


Fig. 2. An overview of the proposed refactorability analysis approach.

1) *Nesting structure matching*. The nesting structure of the input clone fragments is analyzed to find maximal isomorphic subtrees. The assumption is that two code fragments can be unified only if they have an identical nesting structure. Each pair of matched subtrees will be further investigated as a separate clone refactoring opportunity in the next steps.
2) *Statement mapping*. The statements within the subtree pairs extracted from the previous step are mapped in a divide-and-conquer fashion. Taking advantage of the identical nesting structure between the isomorphic subtrees, the global mapping problem is divided into smaller sub-problems (by mapping the statements nested under control predicate nodes at the same level of the subtrees). For each sub-problem the corresponding Program Dependence subgraphs are mapped by applying a *maximum common subgraph* (MCS) algorithm. At the end, the sub-solutions are combined to give the global mapping solution.
3) *Precondition examination*. Based on the differences between the mapped statements in the global solution, as well as the statements that may have not been mapped, a set of preconditions regarding the preservation of program behavior is examined. If none of the preconditions is violated, the clone fragments corresponding to the mapped statements can be safely refactored, and thus are considered *refactorable*.

Our idea was inspired by Johnson et al. [15] who supported that "any global analysis algorithm can be applied unchanged to each SESE region, and the partial results can be combined using the PST to give the global result. This (i.e., the PST) lets us apply analysis algorithms in a divide-and-conquer fashion to the program, which can be a win if the combining of partial results is not overly expensive." In our case, the statement mapping process is applied in a divide-and-conquer fashion, based on the nesting structure of the clone fragments as captured by their PSTs. The partial results (i.e., the mapping sub-solutions) can be combined as long as the global result (i.e., the final mapping solution) complies with the original nesting structure of the clone fragments.

3.1 Abstract Syntax Tree Compatibility

The first and second steps of our approach rely heavily on the matching of statements between the two examined clone fragments based on the analysis of their abstract syntax tree (AST) structure. In the first step (Section 3.2) the control predicate nodes (e.g., `if`, `for` statements) inside the clone fragments are matched by comparing the AST structure of their conditional expressions (i.e., the bodies of the control predicates are excluded from the comparison). In the second step (Section 3.3) the statements nested under the control predicate nodes (i.e., leaf statements without a body) are matched by comparing their entire AST structure.

In our approach, we consider two statements as compatible, if they correspond to the same AST statement type and have a homomorphic AST structure [19]. This means that we allow a subtree expression in the first AST (e.g., a method call) to be mapped to a leaf expression (e.g., a variable identifier) or another kind of subtree expression (e.g., a class instance creation) in the second AST, as long as this mapping respects the core structure of the ASTs. The only restriction in the mapping of sub-expressions within the two statements is that the mapped expressions should be evaluated to the same class/primitive type or types being subclasses of a common superclass. This restriction allows to extract differences between the clone fragments that can be potentially parameterized by introducing a parameter of the same type.

TABLE 1
Supported Expression Types In AST Matching

Expression Type	Example
Method Invocation	`expr.method(arg0, ...)`
Super Method Invocation	`super.method(arg0, ...)`
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, ...)`
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 provide a high degree of freedom in the mapping of expressions within the statements in order to make more flexible the unification of duplicated code with non-trivial differences. Table 1 contains the complete list of expression types that can be parameterized if found different between two given statements.

Our AST matching algorithm has been implemented by extending the `ASTMatcher` superclass provided in Eclipse JDT framework. The default implementation matches two ASTs only if they are structurally isomorphic (i.e., they have an identical tree structure and exactly the same node types/values). Our implementation adds a relaxation in the matching of AST nodes that may have different types or values, and additionally returns a list of the differences detected between the mapped statements that is used in the examination of preconditions (Section 3.4). Table 2 shows the difference types which are reported by our AST matching implementation. The last two difference types, namely operator and variable type mismatches (with the exception of *generic type parameters*, e.g., `<T>`) cannot be parameterized. In the cases where a difference refers to a property of a primary expression (e.g., two method calls having a different name or a different number of arguments), the entire primary expression (e.g., method invocation) is marked for parameterization.

3.2 Nesting Structure Matching

In the first step of the proposed approach, our goal is to find maximal isomorphic subtrees within the nesting structures (i.e., the NSTs) of the clone fragments given as input, since there is no guarantee that the input code fragments will

TABLE 2
Detected Differences between Matched Nodes

Difference Type	Example	
Variable Identifier	`int x = y;`	`int x = z;`
Literal Value	`int x = 0;`	`int x = 1;`
Method Name	`foo(arg);`	`bar(arg);`
Argument Number	`foo();`	`foo(arg0);`
Caller Expression	`expr.foo();`	`foo();`
Array Dimension	`x = a[i];`	`x = a[i][j];`
Array Initializer	`a[] = {0, 1};`	`a[] = {1};`
Infix Ext. Operands†	`x = 4 * a;`	`x = 3 * b * 2;`
Infix Left Operand	`4 * a = = 6 * c;`	`4 * a + 7 = = 6 * d;`
Infix Right Operand	`a + b = = 3 * c-d;`	`a + c = = 3 * d;`
Subclass Type *	`ArrayList x`	`Vector x`
AST Compatible	`int x = foo();`	`int x = 5;`
Field Access ↔ Getter call	`this.field`	`getField()`
Field Assignment ↔ Setter call	`this.field = a;`	`setField(b);`
Operator	`x = y + z;`	`x = y * z;`
Variable Type *	`int x = 5;`	`double x = 5;`

† *Extended infix operands is the way Eclipse JDT represents deeply nested infix expressions of the form* L *op* R *op* R_2 *op* R_3*... where the same operator appears between all the operands (the most common case being lengthy string concatenation expressions).*

* *A subclass type difference denotes that two variables have different types, which are subclasses of a common superclass (e.g.,* `AbstractList` *in the case of* `ArrayList` *and* `Vector`*). On the other hand, a variable type difference represents all other cases of variables having different types. We made this distinction, because the statements containing variables with different subclass types can be potentially unified by generalizing the variable types to the common superclass type.*

have an identical nesting structure. To this end, we developed an algorithm that takes as input two NSTs and finds all non-overlapping *largest common subtrees* [20]. Each resulting subtree match will be further investigated as a separate clone refactoring opportunity.

Initially, we collect from the two NSTs all leaf nodes, which either do not have siblings, or all of their siblings are also leaf nodes. We select these nodes in order to start the detection of isomorphic subtrees from the deepest levels of the NSTs in a bottom-up fashion. Next, we extract all matching (i.e., AST compatible) pairs between the collected leaf nodes from the two NSTs. In the case a leaf node from the first NST can be matched with multiple leaf nodes from the second NST, we keep only the best matching pair (i.e., the pair with the minimum number of differences). Each leaf node pair is given as input to Algorithm 1, which performs a combination of bottom-up and top-down tree matching techniques [20]. According to Valiente [20], for two unordered trees T_1 and T_2 with n_1 and n_2 nodes, respectively, the algorithm for bottom-up maximum common subtree isomorphism runs in the order of $(n_1 + n_2)^2$.

In a nutshell, the algorithm first compares the sibling nodes of the node pair given as input to find matching sibling pairs. For each matching sibling pair it performs a top-down tree match (line **11**) and examines if the resulting subtree match is "exactly paired" (line **12**). 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. The top-down tree match function is essentially a "fail-fast" mechanism that stops the main algorithm from exploring non-exactly paired subtree matches at an early stage. If all matching sibling pairs lead to exactly paired top-down subtree matches, then the parent nodes of the node pair given as input are visited. Finally, if the parent nodes match, then the function described in Algorithm 1 is recursively executed with the new parent node pair as input. The proposed algorithm returns the largest exactly paired subtree match starting from the given input node pair. We designed the algorithm to return only exactly paired subtree matches in order to avoid inconsistencies or gaps in the nesting structure of the matched subtrees.

Algorithm 1. Recursive Function Returning a Maximal Exactly Paired Subtree Match.

Input: a pair of matching NST nodes ($node_i$, $node_j$)
Output: a set of matching NST node pairs

```
 1: function BOTTOMUPMATCH(nodePair, solution)
 2:     solution = solution ∪ nodePair
 3:     siblings_i = nodePair.node_i.siblings
 4:     siblings_j = nodePair.node_j.siblings
 5:     mSiblings = ∅                          ▷ matched siblings
 6:     mPairs = ∅                             ▷ matched node pairs
 7:     for each sibling_i ∈ siblings_i do
 8:         for each sibling_j ∈ siblings_j do
 9:             if compatibleAST(sibling_i, sibling_j) and not
                alreadyMatched(sibling_j) then
10:                 pair = (sibling_i, sibling_j)
11:                 pairs = TOPDOWNMATCH(pair)
12:                 if exactlyPairedSubtrees(pairs) then
13:                     mSiblings = mSiblings ∪ pair
14:                     mPairs = mPairs ∪ pairs
15:                     break                  ▷ first-match
16:                 end if
17:             end if
18:         end for
19:     end for
20:     if |mSiblings| = |siblings_i| = |siblings_j| then
21:         solution = solution ∪ mPairs
22:         parent_i = nodePair.node_i.parent
23:         parent_j = nodePair.node_j.parent
24:         if compatibleAST(parent_i, parent_j) then
25:             pair = (parent_i, parent_j)
26:             BOTTOMUPMATCH(pair, solution)
27:         end if
28:     end if
29: end function
```

Algorithm 1 applies two heuristics to avoid the exploration of all possible matching pairs of NST nodes. At the leaf level of the NSTs, the algorithm selects only the best matching pairs, while in the other levels it always selects the first matching pair. We introduced these two heuristics to make more efficient the matching of long if-else-if chains, where all `if` statements in the chain have similar conditional expressions and can be matched with each other, thus leading to the problem of combinatorial explosion.

(a) Non-optimal mapping with 24 differences

(b) Optimal mapping with 2 differences

Fig. 3. Example motivating the need for optimal mapping.

3.3 Statement Mapping

In the previous step of our approach, we described an algorithm that extracts isomorphic subtrees from the NSTs of the clone fragments given as input. In this step, we present an approach for finding a *locally optimal* mapping (see Section 3.3.2) between the statements nested under the control predicate nodes of the NST subtrees. *Statement mapping* is an *injective* function that associates each statement from the first clone fragment with at most one statement from the second clone fragment. The statements that cannot be associated with any statement from the other clone fragment (due to AST incompatibility) are considered as *unmapped*.

To facilitate the refactoring of duplicated code, an optimal mapping should not only contain a *maximal* number of mapped statements, but also a *minimal* number of differences between them. The *minimization of 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. A large number of parameters makes more difficult the use/reuse of the extracted method, since calling such a method would require passing several arguments. Additionally, a large number of differences implies a higher probability for a precondition violation, since the parameterization of some differences could cause a change in the program behavior (as explained in Section 3.4).

3.3.1 Motivation for Optimizing Statement Mapping

Fig. 3 illustrates two alternative mappings for two code fragments found in methods `drawDomainMarker` and `drawRangeMarker`, respectively, within the class `AbstractXYItemRenderer` of the *JFreeChart* open-source project (version 1.0.14). These methods contain over 90 duplicated statements covering their entire body. However, for the sake of simplicity, we have included only a small portion of the duplicated code. The number next to each statement indicates the index of this statement in the ordered list of method statements.

Fig. 3a depicts the actual nesting structure of the two code fragments on the left and right hand side, along with a statement mapping as obtained from a matching approach selecting the *first* or *best* match in a top-down fashion. Two statements positioned on the same line, next to each other, are considered as mapped (e.g., statement 61 on the left hand side is mapped to statement 62 on the right hand side of Fig. 3a). A matching approach that does not explore the entire search space always selects the first or best match in the case of multiple possible matches (e.g., statement 61 on the left hand side can be mapped to either statement 62 or 74 on the right hand side). As a result, the mapping (61, 62) is the first match encountered in a top-down traversal, but also the best match in terms of similarity, since statement 61 on the left hand side is exactly the same with statement 62 on the right hand side. By matching statement 61 with 62, and 73 with 74, we finally obtain the mapping solution shown in Fig. 3a. This solution is maximal, since all 25 statements from each code fragment have been successfully mapped; however, it contains a large number of differences between the mapped statements.

Fig. 3b depicts an optimal statement mapping, which is again maximal in terms of the number of mapped statements, but also has the minimum number of differences between the mapped statements. By examining carefully the code fragments, one can observe that the code inside the body of statement 61 on the left hand side is exactly the same with the code inside the body of statement 74 on the right hand side, and the same holds for statement 73 on the left hand side with statement 62 on the right hand side. Consequently, by detecting the "symmetrical" structure of the two code fragments and parameterizing the differences in the conditional expressions of the respective `if` statements, we can obtain the optimal mapping shown in Fig. 3b. This alternative mapping makes feasible the refactoring of the clones and introduces significantly less parameters to the extracted method.

3.3.2 Decomposition of the Mapping Problem

The core of our statement mapping technique is a divide-and-conquer algorithm that breaks the initial mapping problem into smaller sub-problems based on the nesting structure of the isomorphic NST subtrees extracted in the previous step.

Let NST_i be the first subtree and NST_j the second one. In a nutshell, Algorithm 2 performs a bottom-up processing of every level in the subtrees. For each node of NST_i at a given level, it explores all possible pairs of matching control predicate nodes at the same level of NST_j. Each pair of matching control predicate nodes is used as a *starting point* for the application of a graph matching algorithm (Section 3.3.3), which matches the Program Dependence subgraphs containing only the non-predicate nodes nested under the predicate nodes of the starting point. After the examination of all possible matching combinations the best sub-solution (i.e., the solution with the largest number of mapped nodes and the smallest number of differences between them) is appended to the final solution. Algorithm 2 is essentially a *greedy* algorithm [21] that makes locally optimal choices at each level of the NST subtrees with the hope of finding a globally optimal solution.

Algorithm 2. A Divide-and-Conquer Statement Mapping Process Based on Nesting Structure.

Input: two isomorphic NSTs
Output: the final mapping solution

```
 1: function PDGMAPPING (NST_i, NST_j)
 2:     level = NST_i.maxLevel = NST_j.maxLevel
 3:     solution = ∅
 4:     while level ≥ 0 do
 5:         cpNodes_i = nodes at level of NST_i
 6:         cpNodes_j = nodes at level of NST_j
 7:         for each cp_i ∈ cpNodes_i do
 8:             states = ∅                                  ▷ MCS states
 9:             for each cp_j ∈ cpNodes_j do
10:                 if validNesting(cp_i, cp_j) then
11:                     mapping = (cp_i, cp_j)
12:                     root = createState(mapping)
13:                     SEARCH(root, mapping)
14:                     states = states ∪ findMCS(root)
15:                 end if
16:             end for
17:             solution = solution ∪ best(states)
18:         end for
19:         decrement level
20:     end while
21: end function
```

The examination of all possible pairs of matching control predicate nodes at every level of the subtrees, makes possible the matching of "symmetrical" structures, as the ones shown in Fig. 3, as well as control predicates placed in a different order within the clone fragments (i.e., *Type-3* clone differences).

Function *validNesting* (line **10**) ensures that the resulting sub-solutions can be combined to form a valid global solution (i.e., a solution that complies with the nesting structure of the NST subtrees). This function takes as input two control predicate nodes cp_i and cp_j in the NST subtrees and examines three conditions:

1) *Preservation of nesting structure:* all nodes in the path of cp_i to the root of NST_i should be compatible with the corresponding nodes in the path of cp_j to the root of NST_j. This condition ensures that the current mapping can lead to a final solution that covers the entire trees (i.e., there will always be compatible parents to be mapped until we reach the roots of the trees).
2) *Preservation of sibling relationships:* for all control predicate node mappings (n_i, n_j) created at the *currently* examined level of the subtrees as part of a best sub-solution, if cp_i is a sibling of n_i in NST_i (i.e., cp_i and n_i have the same parent node in NST_i), cp_j should be a sibling of n_j in NST_j and vice versa. This condition ensures that for all the predicate node mappings created in the current level, their actual siblings in the trees will be mapped in the current level.
3) *Preservation of parent-child relationships:* for all control predicate node mappings (n_i, n_j) created at the *previously* examined level of the subtrees as part of a best sub-solution, if cp_i is the parent of n_i in NST_i, cp_j should be the parent of n_j in NST_j and vice versa. This condition ensures that for all the predicate node mappings created in the previous level, only their actual parents in the trees will be mapped in the current level.

If these three conditions hold for every pair of control predicate nodes leading to a best sub-solution, then all resulting sub-solutions can be safely combined into a valid global solution.

Function *createState* (line **12**) creates an initial state of the search space containing only the node mapping passed as argument. Function *findMCS* (line **14**) returns the states corresponding to the maximum common subgraphs in the search tree. These states are the leaf nodes in the deepest level of the search tree. Finally, function *best* takes as input a set of states and applies a three-step elimination process to return the state with the largest number of mapped statements and the smallest number of differences between the mapped statements. In the first step, we keep only the states with the largest number of mapped statements (*max*) and eliminate the states having a number of mapped statements lower than *max*. In the second step, we keep only the states with the smallest number of distinct differences (*min*) and eliminate the states having a number of distinct differences greater than *min*. The reason we decided to compare the number of distinct differences (instead of the number of all differences) is to avoid penalizing states that include the same variable rename in many different syntactic positions. As a result, all *Variable Identifier* differences corresponding to the same pair of identifiers are considered as one distinct difference regardless of the number of times they are repeated in the clone fragments. In the third and final step, we select the state with the smallest number of non-distinct differences.

3.3.3 Program Dependence Subgraph Mapping

As explained in Section 3.3.2, the original statement mapping problem is decomposed into smaller sub-problems, i.e., mapping the sets of non-predicate statements $ncpNodes_i$ and $ncpNodes_j$ nested under two control predicate nodes from NST_i and NST_j, respectively. Each statement mapping subproblem is expressed as a graph matching problem by

extracting the Program Dependence subgraphs containing only the nodes in sets $ncpNodes_i$ and $ncpNodes_j$, respectively. The graph matching problem is solved by applying a maximum common subgraph algorithm.

The reason we expressed the statement mapping problem as a PDG matching problem is to better support the mapping of *Type-3* clone fragments. As explained by Komondoor and Horwitz [22], the PDG is the ideal structure to "find non-contiguous clones (i.e., clones whose statements do not occur as contiguous text in the program), and clones in which matching statements have been reordered". Therefore, a PDG mapping approach can reduce the ambiguity of statement matching, since the similarity of two statements can be assessed not only based on their textual or AST-structure similarity, but also based on the correspondence of incoming/outgoing data dependencies from/to other statements.

The detection of the *maximum common subgraph* is a well known NP-complete problem for which several optimal and suboptimal algorithms have been proposed in the literature. Conte et al. [23] compared the performance of the three most representative optimal algorithms, which are based on depth-first tree search:

1) the McGregor algorithm [24] 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 et al. algorithm [25] 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 [26] 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 [23]. 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 [23]. Conte et al. [23] 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).

For the implementation of our MCS search technique (Algorithm 3), we have adopted the McGregor algorithm [24], 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 [23]. 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 $O(N_1 \cdot N_2)$ [23]. Given two PDG subgraphs, namely PDG_i and PDG_j, Algorithm 3 applies the following constraints:

1) An edge of PDG_i is traversed only once in each path of the search tree (line **6**).
2) A node from PDG_i is mapped to at most one node from PDG_j (and vice versa) in each path of the search tree (line **12**).
3) Two edges $edge_i$ and $edge_j$ are considered compatible (line **9**) if they connect nodes which are compatible (i.e., the nodes in the starting and ending points of the edges, respectively, should be compatible with each other) and they have the same dependence type (i.e., they are both control or data flow dependencies). In the case of control dependencies, both should have the same control attribute (i.e., True or False). In the case of data dependencies, the data attributes should correspond to variables having the same name, or to variables detected as renamed during the AST compatibility analysis of the attached nodes. Finally, if both data dependencies are *loop-carried*, then the loop nodes through which they are carried should be compatible too.

Algorithm 3. Recursive Function Building a Search Tree.

Input: a parent state in the search tree, a pair of mapped PDG nodes ($node_i$, $node_j$)
Output: a search tree

```
 1: function SEARCH(pState, nodeMapping)
 2:     edges_i = nodeMapping.node_i.edges
 3:     edges_j = nodeMapping.node_j.edges
 4:     for each edge_i ∈ edges_i do
 5:         visited = pState.visitedEdges
 6:         if edge_i ∉ visited then
 7:             visited = visited ∪ edge_i
 8:             for each edge_j ∈ edges_j do
 9:                 if compatible(edge_i, edge_j) then
10:                     vN_i = edge_i.otherEndPoint
11:                     vN_j = edge_j.otherEndPoint
12:                     if not mapped(vN_i) and not mapped(vN_j)
                        then
13:                         mapping = (vN_i, vN_j)
14:                         state = createState(mapping)
15:                         if not prune (state) then
16:                             state --add--> pState.children
17:                             SEARCH(state, mapping)
18:                         end if
19:                     end if
20:                 end if
21:             end for
22:         end if
23:     end for
24: end function
```

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* (line **14**) copies the visited edges and mapped nodes from the parent state to the child state. Function *prune* (line **15**) 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

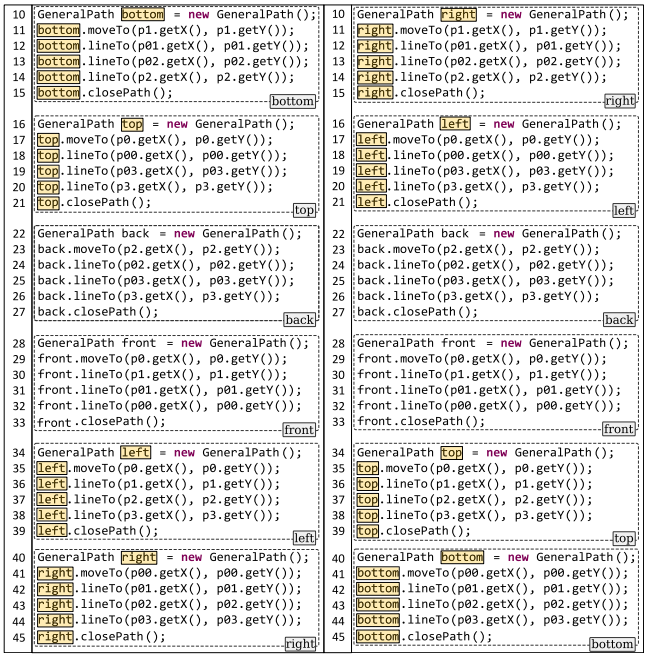


Fig. 4. Decomposing code fragments into subsets of statements to face the problem of combinatorial explosion in the MCS algorithm.

(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.

3.3.4 The Problem of Combinatorial Explosion

The reason we proposed the divide-and-conquer approach (Section 3.3.2) is that a direct application of Algorithm 3 on the original problem (i.e., the complete PDGs) 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 nesting structure of the clone fragments and break the original mapping problem into smaller sub-problems. However, there are still some cases that could cause a combinatorial explosion.

Fig. 4 shows two code fragments found in methods `createHorizontalBlock` (left hand side) and `createVerticalBlock` (right hand side), respectively, within the class `StackedBarRenderer3D` of the *JFreeChart* open-source project (version 1.0.14). As it can be observed all `GeneralPath` object creations (statements 10, 16, 22, 28, 34, 40) can be mapped with each other, since the only difference is the name of the variable (`bottom`, `top`, `back`, `front`, `left`, and `right`). Additionally, all statements following the object creations (i.e., method calls `moveTo`, `lineTo`, and `closePath`) can be mapped with each other due to the degree of freedom we allow in the matching of expressions within the statements (e.g., differences in method call names, number of arguments, and variable identifiers). This situation leads to a very large number of matching statement combinations (i.e., a search space explosion) that deteriorates dramatically the performance of the MCS algorithm.

In order to tackle this problem, we perform an additional decomposition of the mapping problem into subproblems by finding subsets of related statements within the original set of statements whenever it is possible. We consider a subset of statements as related to each other, if they modify the state of an object referenced by the same variable. For example, statements 10-15 (on the left hand side of Fig. 4) modify the state of the object referenced by `bottom`, statements 16-21 modify the state of the object referenced by `top` and so on. The detection of statements modifying the state of an object referenced by `ref` is performed by examining the presence of composite variables (Section 2.2) in the form of `ref.field` inside the set of Defined variables of each statement. After extracting the subsets of statements in both code fragments, we apply Algorithm 3 on the Program Dependence subgraphs corresponding to each subset of statements. Again, each subgraph from the first code fragment is matched with all subgraphs from the second code fragment and the best sub-solution (i.e., the solution with the largest number of mapped nodes and the smallest number of differences between them) is appended to the final solution.

Fig. 4 shows in dashed rectangles the subsets that were detected in the two code fragments, as well as the mapping solution resulting after the decomposition of statements into subsets. In this example, we can see that the developers essentially renamed variables `bottom` to `right`, `top` to `left`, `left` to `top`, and `right` to `bottom`. It should be emphasized that the same mapping solution would be achieved even if the developers had reordered the subsets of statements in the second code fragment, in addition to the renaming of variables.

3.4 Preconditions

After the completion of the statement mapping process, we need to determine whether the clone fragments can be safely extracted into a common method by parameterizing all existing differences between the mapped statements and moving the unmapped statements before or after the execution of the common statements.

According to Opdyke [27], each refactoring should be accompanied with a set of *preconditions*, which ensure that the behavior of a program is preserved by the refactoring. If any of the preconditions is violated, then the refactoring is not applicable, or its application would cause a change in the program behavior. In this section, we define a set of preconditions that should be examined before the refactoring of duplicated code.

3.4.1 Preconditions Handling Differences between Mapped Statements

In order to extract the duplicated code into a common method, the differences between the mapped 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

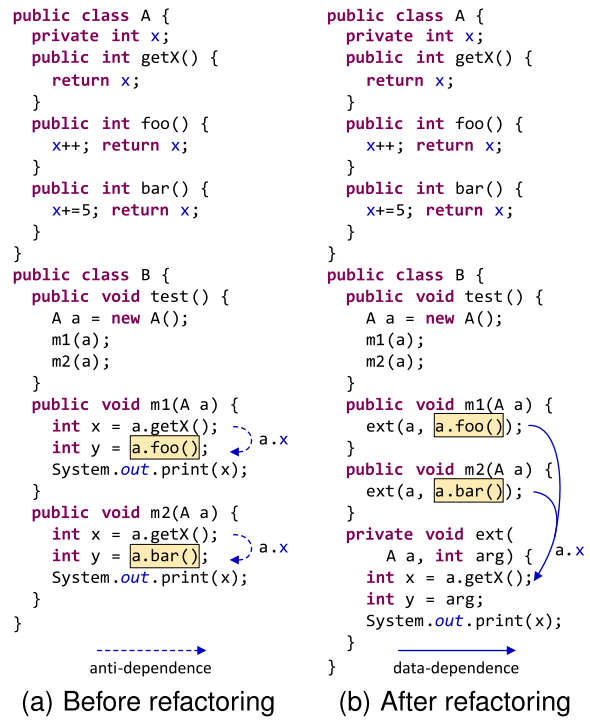


Fig. 5. Parameterization of a difference breaking two existing anti-dependencies.

extracted common statements. Obviously, a change in the evaluation or execution order of the parameterized expressions could cause a change in the program behavior.

In Fig. 5a, methods `m1` and `m2` in class `B` contain exactly the same code with the exception of a difference in method calls `a.foo()` and `a.bar()`. In the first statement, both methods call `a.getX()` to read attribute `x` from object reference `a`. In the next statement, the value of attribute `x` is modified through method calls `a.foo()` and `a.bar()`, respectively. As a result, there exists an *anti-dependence* due to variable `a.x` from the first to the second statement of methods `m1` and `m2`, respectively. In order to merge the duplicated code, the common statements are extracted in method `ext()`, as shown in Fig. 5b, 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 used. As a matter of fact, a new inter-procedural data-dependence due to variable `a.x` is introduced after refactoring. The breaking of the original anti-dependence is causing a change in the behavior of the program. In the original version in Fig. 5a, the execution of method `test` results in `m1` printing `0` and `m2` printing `1`, while in the refactored version in Fig. 5b the execution of method `test` results in `m1` printing `1` and `m2` printing `6`. In a similar manner, the breaking of data-dependencies or output-dependencies could also cause a change in the program behavior.

Therefore, we propose the following precondition to handle the differences existing between mapped statements. This precondition excludes the differences corresponding to variables that have been renamed between the two clone fragments, because such differences do not require parameterization.

```
if (dataset != null) {
    ...
    Range r = DatasetUtilities.findDomainBounds(dataset, false);
    ...
}
if (column > 0) {
    ...
    Number previousValue = dataset.getValue(row,column - 1);
    ...
}
```

Fig. 6. Examples of use-constrained control dependencies due to variables examined in predicate statements.

> *Precondition 1.* The parameterization of the differences between the mapped statements should not break any existing control, data, anti, and output dependencies.

In order to detect such a precondition violation for the expressions e_i and e_j being different between two mapped statements s_i and s_j, we first have to find the sets of variables V_i and V_j (including also composite variables) that are modified or used by e_i and e_j. Let M_i and M_j be the sets of statements that have been mapped from the first and the second clone fragment in methods m_i and m_j, respectively. Precondition #1 is violated if either statement s_i or s_j has an incoming dependence from a mapped statement due to a variable in V_i and V_j, respectively. Formally,

$$\{p_i \xrightarrow{v} s_i \in D(m_i) \mid p_i \in M_i \wedge s_i \in M_i \wedge v \in V_i\} \neq \varnothing \text{ or}$$
$$\{p_j \xrightarrow{v} s_j \in D(m_j) \mid p_j \in M_j \wedge s_j \in M_j \wedge v \in V_j\} \neq \varnothing,$$

where $p \xrightarrow{v} s$ denotes a control, data, anti, or output dependence from statement p to statement s due to variable v, and $D(m)$ denotes the set of dependencies in the PDG of method m. If statement s_i or s_j has a self-loop dependence (i.e., a dependence starting from and ending to the same statement) due to a variable belonging to sets V_i and V_j, respectively, which is carried through a loop statement l, then Precondition #1 is violated as long as l belongs to the mapped statements.

In our approach, we consider a more strict version of the original control dependence definition, which we call "use-constrained control dependence". Statement s is control dependent to predicate statement p due to variable v, if s is nested (directly or indirectly) under p, p examines the value of v in its condition, and s uses the value of v. We consider that a variable is examined in a predicate statement, if the variable appears somewhere in its condition; for example, in the left or right operand of a relational operator (==, !=, >, <, >=, <=), or as the left operand of an `instanceof` expression. Fig. 6 shows two examples of use-constrained control dependencies containing variables that are examined in predicate statements and are used by control dependent statements. The reason we adopted this stricter version of control dependence is that we found out many cases of differences that were rejected from being parameterized due to the existence of a control dependence (in its original definition), but their parameterization was actually feasible without changing the program behavior.

Finally, we also consider an "exception-constrained control dependence" as the case where statement s is nested

under a `try` block p, and s contains a method call, or a `throw` statement throwing an exception handled by p.

In the example shown in Fig. 5a, we would find that variable `a.x` is modified through method calls `a.foo()` and `a.bar()` (i.e., the expressions being different between the mapped statements). Therefore, $V_i = \{\texttt{a.x}\}$ and $V_j = \{\texttt{a.x}\}$. The parameterization of expressions `a.foo()` and `a.bar()` would eliminate the original anti-dependencies due to variable `a.x` from the mapped statements `int x = a.getX();`.

It should be noted that we exclude the dependencies from unmapped statements, because these statements will be moved in the original methods either before or after the execution of the extracted method containing the mapped statements (assuming that their move does not violate Precondition #5 that will be explained in Section 3.4.2), and therefore any existing dependencies to/from the parameterized expressions will be preserved.

In the case where some mapped statements in the clone fragments contain variables having different subclass types of a common superclass, our statement matching approach reports a *Subclass Type* difference (Section 3.1, Table 2). In order to unify properly the statements containing such differences into a single statement, we should generalize the different variable types to the common superclass type [28]. Practically, this can be achieved by finding the declaration of the variable in the unified code or in the parameters of the extracted method and setting the type to the common superclass type. This unification mechanism can be applied as long as the mapped statements in the clone fragments are not calling methods declared in the subclasses (excluding calls to overridden methods) through these variables.

Therefore, we propose the following precondition to handle the *Subclass Type* differences existing between mapped statements.

> *Precondition 2.* Matched variables having different subclass types should call only methods that are declared in the common superclass or are being overridden in the respective subclasses.

Let s_i and s_j be a pair of mapped statements that use variables v_i and v_j having the subclass types t_i and t_j, where $t_i \neq t_j$. In order to detect such a precondition violation, we first have to find the sets of methods MC_i in class t_i and MC_j in class t_j called through v_i and v_j, respectively. Formally,

$$MC_i = \{mc.\mathsf{declaration} \mid mc.\mathsf{class} = t_i \wedge mc.\mathsf{invoker} = v_i\}$$
$$MC_j = \{mc.\mathsf{declaration} \mid mc.\mathsf{class} = t_j \wedge mc.\mathsf{invoker} = v_j\}$$

where mc denotes a method call inside statement s.

Next, for each pair of method declarations in sets MC_i and MC_j having the same signature, we examine whether the common superclass of t_i and t_j declares or inherits a method with the same signature (i.e., the subclasses override a method of the common superclass). If no such method is found in the common superclass, then Precondition #2 is violated. From our analysis, we exclude the method declaration signatures that are not common in sets MC_i and MC_j, because the corresponding method calls in the mapped statements will have to be parameterized, since they refer to different methods. Therefore, these cases will be examined with Precondition #1.

```
public void setBackgroundPaint(Paint paint) {
1 if (paint == null) {
2   if (this.backgroundPaint != null) {
3     this.backgroundPaint = null;
4     fireChangeEvent();
    }
  }
  else {
5   if (this.backgroundPaint != null) {
6     if (this.backgroundPaint.equals(paint)) {
7       return;  // nothing to do
      }
    }
8   this.backgroundPaint = paint;
9   fireChangeEvent();
  }
}
```

```
public void setOutlinePaint(Paint paint) {
1 if (paint == null) {
2   if (this.outlinePaint != null) {
3     this.outlinePaint = null;
4     fireChangeEvent();
    }
  }
  else {
5   if (this.outlinePaint != null) {
6     if (this.outlinePaint.equals(paint)) {
7       return;  // nothing to do
      }
    }
8   this.outlinePaint = paint;
9   fireChangeEvent();
  }
}
```

Fig. 7. Example of clone fragments modifying fields.

Another category of differences whose parameterization would cause a change in the behavior of the program is related to fields (instance variables) being modified in the clone fragments. Fig. 7 shows a case of two clone fragments, found in class `Plot` of the JFreeChart project, that modify the value of fields `backgroundPaint` and `outlinePaint`, respectively, in lines 3 and 8. These two clone fragments can be unified by introducing a parameter for the fields being different in the extracted method. However, the extracted method will update only the value of the local parameter and will not update the values of the original fields passed as arguments, since all parameters are passed by value in Java.

Therefore, we propose the following precondition to handle differences in fields being modified by assignment statements or increment/decrement operators. We restrict the effect of this precondition only to modified fields and not local variables, because the extracted method should return the values of the local variables being modified within clone fragments (Precondition #6 that will be explained in Section 3.4.3 handles the case of multiple returned variables).

> *Precondition 3.* The parameterization of fields belonging to differences between the mapped statements is possible only if they are not modified.

The final precondition is related to the return type of the method calls found in the differences between the mapped statements. The corresponding method declarations should not return the `void` type, since it is not possible to introduce a parameter of `void` type in the extracted method.

> *Precondition 4.* The parameterization of method calls belonging to differences between the mapped statements is possible only if they do not return a `void` type.

3.4.2 Preconditions Handling Unmapped Statements

The statement mapping process may result in *unmapped* statements. These statements could not be mapped with any statement from the other clone fragment because either there is no corresponding statement (i.e., statements that exist in only one of the clone fragments), or there exists a corresponding statement, but it has been so extensively modified that its AST structure is no longer compatible (Section 3.1) (i.e., statements that cannot be

unified due to incompatible AST structure). Assuming that the mapping process resulted in a set of unmapped statements from the first and the second clone fragment in methods m_i and m_j, respectively, these statements should be moved either before or after the call of the extracted method containing the mapped statements inside m_i and m_j, respectively. As in the case of the parameterization of differences, the move of the unmapped statements could break existing data-, anti-, and output-dependencies from/to mapped statements. Therefore, we propose the following precondition to handle the unmapped statements.

> *Precondition 5*. The unmapped statements should be movable before or after the mapped statements without breaking existing control, data, anti, and output dependencies.

In order to detect such a precondition violation, we developed two rules that examine whether moving an unmapped statement before the first or after the last mapped statement, respectively, would break existing dependencies. Let M_i and M_j be the sets of statements that have been mapped from the first and the second clone fragment in methods m_i and m_j, respectively. Let U_i and U_j be the sets of statements that have not been mapped from the first and the second clone fragment in methods m_i and m_j, respectively.

Moving a statement before the mapped statements. An unmapped statement belonging to U_i or U_j cannot be moved before the mapped statements if it has an incoming dependence from a statement in M_i and M_j, respectively. Formally,

$$\{p_i \xrightarrow{v} s_i \in D(m_i) \mid p_i \in M_i \wedge s_i \in U_i\} \neq \varnothing \text{ or}$$
$$\{p_j \xrightarrow{v} s_j \in D(m_j) \mid p_j \in M_j \wedge s_j \in U_j\} \neq \varnothing$$

where $p \xrightarrow{v} s$ denotes a control, data, anti, or output dependence from statement p to statement s due to variable v, and $D(m)$ denotes the set of dependencies in the PDG of method m. If statement s_i or s_j has a self-loop dependence (i.e., a dependence starting from and ending to the same statement), which is carried through a loop statement l, then it can still not be moved before the mapped statements as long as l belongs to the mapped statements.

Moving a statement after the mapped statements. An unmapped statement belonging to U_i or U_j cannot be moved after the mapped statements if it has an outgoing dependence to a statement in M_i and M_j, respectively. Formally,

$$\{p_i \xrightarrow{v} s_i \in D(m_i) \mid p_i \in U_i \wedge s_i \in M_i\} \neq \varnothing \text{ or}$$
$$\{p_j \xrightarrow{v} s_j \in D(m_j) \mid p_j \in U_j \wedge s_j \in M_j\} \neq \varnothing$$

where $p \xrightarrow{v} s$ denotes a data-, anti-, or output-dependence from statement p to statement s due to variable v, and $D(m)$ denotes the set of dependencies in the PDG of method m. Additionally, if an unmapped statement is using a local variable that is modified by the mapped statements, then the extracted method should return the value of this variable.

3.4.3 Preconditions Related to Method Extraction

Murphy-Hill and Black [29] 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. Their list [29] includes the following preconditions:

> "1) The selected code must be a list of statements.
> 2) Within the selection, there must be no assignments to variables that might be used later in the flow of execution. For Java, this can be relaxed to allow assignment to one variable, the value of which can be returned from the new method.
> 3) Within the selection, there must be no conditional returns. In other words, the code in the selection must either always return, or always flow beginning to end.
> 4) Within the selection, there must be no branches to code outside of the selection. For Java, this means no `break` or `continue` statements, unless the selection also contains their corresponding targets."

In this section, we will adjust these preconditions to the Extract Clone refactoring (i.e., extracting two duplicated code fragments into a separate method).

Regarding the first precondition in the list of Murphy-Hill and Black, our approach guarantees that the mapped statements will always form a complete syntactic unit in two ways. First, in the implementation of Algorithm 1 (i.e., the algorithm that finds exactly paired subtrees within the nesting structures of the clone fragments), we take special care in the matching of `if/else` and `switch case` structures. Two `if` statements can be matched only if they have matching if-else-if chain structures. Additionally, two `switch` statements can be matched only if the case statements nested inside them are exactly paired. In this way, we make sure that the list of mapped statements will not contain any partially matched control predicate structures. Our approach for matching nested control structures has several similarities with the way that the clone detection algorithm proposed by Koschke et al. [30], which is based on *abstract syntax suffix trees*, handles nested sequences (i.e., blocks). Second, our AST comparison mechanism (Section 3.1) examines the entire AST structure of the non-predicate statements, and therefore by definition there are no partially matched non-predicate statements. As a result, the mapped statements to be extracted constitute a list of syntactically complete statements.

The second precondition in the list of Murphy-Hill and Black can be adjusted as follows:

> *Precondition* 6. The mapped statements within the clone fragments should return at most one variable of the same type to the original methods from which they are extracted.

In order to detect such a precondition violation, we first have to determine the sets of variables RV_i and RV_j that should be returned by the mapped statements to the original methods m_i and m_j, respectively. We take into account

only local variables that are declared within the bodies of m_i and m_j, as well as the parameters of m_i and m_j. We exclude instance variables (i.e., class fields) that may be modified within the clone fragments, because they have a *class scope*, and thus it is redundant to be returned from the extracted method. Let M_i and M_j be the sets of statements that have been mapped from the first and the second clone fragment, respectively. Let R_i and R_j be the sets of statements that will remain in m_i and m_j, respectively, after the extraction of the mapped statements. Formally, $R_i = A_i \setminus M_i$ and $R_j = A_j \setminus M_j$, where sets A_i and A_j include all statements within the bodies of m_i and m_j, respectively. In general, when extracting a code fragment from a method, the variables that should be returned to the original method are those for which a data dependence exists from the set of extracted statements to the set of remaining statements. In the case of Extract Clone refactoring, we formally define

$$RV_i = \{v \in V(m_i) \mid p \xrightarrow{v} q \in D(m_i) \wedge p \in M_i \wedge q \in R_i\}$$
$$RV_j = \{v \in V(m_j) \mid p \xrightarrow{v} q \in D(m_j) \wedge p \in M_j \wedge q \in R_j\}$$

where $p \xrightarrow{v} q$ denotes a data-dependence from statement p to statement q due to variable v, $V(m)$ denotes the set of variables which are declared within the body of method m (including the parameters of m), and $D(m)$ denotes the set of dependencies in the PDG of method m.

Precondition #6 is violated if $|RV_i| > 1$, or $|RV_j| > 1$, or $|RV_i| \neq |RV_j|$. If $|RV_i| = |RV_j| = 1$, then the variable in RV_i should have the same type with the variable in RV_j.

One way to overcome the problem of multiple returned variables would be to make the extracted method return a bean object in which the returned variable values are assigned to appropriate fields. Although this is a feasible solution, it requires the introduction of a new class, which will be instantiated only by the extracted method (i.e., a class instantiated only once in the entire system). The bean object solution would make more sense, if there were several clone fragments in different clone groups returning the same set of variables. In that case, each extracted method would instantiate a bean object, thus making the newly introduced class more reusable.

The third precondition in the list of Murphy-Hill and Black can be adjusted as follows:

> *Precondition* 7. The mapped statements within the clone fragments should not contain any conditional `return` statements.

A conditional return statement, as the one shown below, can be used to branch out of a control flow block and directly exit a method.

```
public void originalMethod(){
  ...
  if (condition)
   return;
  ...
}
```

Extracting a piece of code containing a conditional return statement, and then simply calling the extracted method, would make the original method not to exit in the same way as it did before. One way to overcome this problem would be to make the extracted method return a boolean *flag* that is set to `true` when the conditional return statement is reached in the extracted code, and is set to `false` otherwise.

```
private boolean extractedMethod()  {
  boolean flag = false;
  if (condition){
    flag = true;
    return flag;
  }
  ...
  return flag;
}
```

If the value of the returned flag is `true`, then the original method should exit directly after the execution of the extracted method as shown below.

```
public void originalMethod(){
  ...
  boolean flag = extractedMethod();
  if (flag)
    return;
  ...
}
```

However, this solution requires to insert additional conditional code in both the original and the extracted methods, thus increasing the initial complexity of the code.

The fourth precondition in the list of Murphy-Hill and Black can be adjusted as follows:

> *Precondition 8*. The mapped branching statements (`break`, `continue`) should be accompanied with the corresponding mapped loop statements.

In Java the unlabeled `break` statement is used to terminate the innermost `for`, `while`, or `do-while` loop, or the enclosing `switch` statements. 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 mapped the corresponding loops should be also mapped. Otherwise the extraction of a branching statement without the corresponding loop would cause a compilation error. In the same manner, the labeled `break` statement (i.e., `break label;`) terminates the outer loop marked with the specified label, and the labeled `continue` statement (i.e., `continue label;`) skips the current iteration of the outer loop marked with the specified label. When two labeled branching statements are mapped, the corresponding loops marked with the specified labels should be also mapped. It should be noted that our approach supports consistently renamed labels between the clone fragments (i.e., when the label used in the branching statements is renamed in the same way with the label marking the outer loop).

3.5 Limitations

A major limitation of the proposed approach is that it does not support the analysis and refactoring of clone groups

(also known as clone classes) containing more than two clone fragments. In order to support the analysis of clone groups, the proposed approach should be extended to find largest common subtrees in the nesting structures of multiple clone fragments. Then, the differences among the clone fragments of the group could be extracted by applying our statement mapping approach on every possible combination of clone pairs in the group, and summarizing the differences corresponding to the same syntactic positions (i.e., a single parameter should be introduced in the extracted method for all the differences found in the same syntactic position). Finally, by examining the same set of preconditions on all differences extracted in the previous step, we could determine whether the entire clone group is refactorable (i.e., a clone group is refactorable if there are no precondition violations). Obviously, adapting the current clone-pair-based approach to support the analysis of clone groups might cause serious scalability problems, especially when the size of a clone group is large.

Recently, Lin et al. [31] proposed an approach for detecting differences across multiple clone instances. In their approach, *MCIDiff* (Multi-Clone-Instances Differencing), each clone instance in a clone group is parsed into a sequence of tokens. Then, *MCIDiff* computes the Longest Common Subsequence (LCS) of all clone instances in the clone group, and analyzes the LCS to determine differential ranges across the clone instances. Finally, it produces as output a list of differential multisets of tokens that summarize differences that can be found in parameterized and gapped clones. Finding the global optimal alignment for N sequences is an NP-complete problem. Therefore, *MCIDiff* adopts a progressive alignment approach to compute an approximate solution. The preconditions defined in this paper can be certainly examined on the differences extracted by *MCIDiff* to assess whether a clone group can be safely refactored. However, this would first require to map the tokens returned by *MCIDiff* to appropriate AST nodes in order to determine the expressions to be parameterized (e.g., if a token corresponds to a partial AST expression, then the entire AST expression should be parameterized). This is a general limitation of token-based differencing approaches, because they do not consider the syntactic structure of the program [31]. Next, the PDGs of all the methods involved in the clone group will have to be generated in order extract the dependencies required for the examination of the preconditions. This additional cost of AST and PDG generation could perhaps justify to build a solution that operates directly on ASTs and PDGs, like our approach.

4 Tool Support

The proposed technique for assessing the refactorability of software clones has been implemented as an Eclipse plug-in, which is part of the JDeodorant[1] code smell detection and refactoring suite, and can be used in two ways:

1) In the *GUI mode* the user interacts directly with the Eclipse IDE by selecting pairs of methods containing duplicated code fragments to be analyzed for potential refactoring opportunities.
2) In the *batch processing mode* the user specifies as input files containing results from clone detection tools, and our Eclipse plug-in is executed in headless mode analyzing all clone pairs found in the input files and generating a report.

In the following sections, we present the features offered by each execution mode.

4.1 GUI Mode

In this mode, the user has to select two methods containing duplicated code fragments either in the *Package Explorer* or the *Outline* view of Eclipse. The *Package Explorer* view allows the selection of methods belonging to different Java files, while the *Outline* view allows the selection of methods only from the same Java file. By right-clicking on the selected methods and selecting "Refactor Duplicated Code..." from the context menu, the dialog shown in Fig. 8 appears after the analysis of the methods. In this dialog the user can inspect the refactoring opportunities detected in the selected methods. If more than one refactoring opportunity is present (i.e., multiple isomorphic subtrees have been detected in the nesting structures of the methods), the user can inspect each one of them separately by selecting the corresponding option in the "Select Refactoring Opportunity" combo box (Fig. 8, point 1).

4.1.1 Clone Visualization

The mapped and unmapped statements corresponding to each refactoring opportunity selected by the user are visualized in two side-by-side tree structures representing the nesting structures of the duplicated code fragments, as shown in Fig. 8. Each node in the tree structures represents a mapped or unmapped statement. The unmapped statements are highlighted in red color, while the differences between the mapped statements are highlighted in green color. The two tree structures are synchronized in the sense that collapsing/expanding a node in the first tree will automatically collapse/expand the corresponding node in the second tree and vice versa. Additionally, the vertical and horizontal scrollbars surrounding the tree structures are synchronized, so that the same code area of the clone fragments is always displayed when scrolling.

When the user hovers over the mapped/unmapped statements a tooltip appears with the following information:

1) *Semantic differences*. As it has been shown in Table 2 (Section 3.1), our approach can detect various types of differences between the mapped statements. As in the case of *CloneDifferentiator* [32], our approach is aware of the program elements in which the differences occur, and thus it can provide a more meaningful explanation of the differences (Fig. 8, point 2) compared to text differencing techniques that ignore semantic information. In addition, the detected semantic differences are used to improve the quality of the applied refactoring transformation by avoiding redundant parameterizations. For example, a difference regarding a field access that is replaced with the corresponding getter method call should not be parameterized, since the involved expressions are

1. http://www.jdeodorant.com

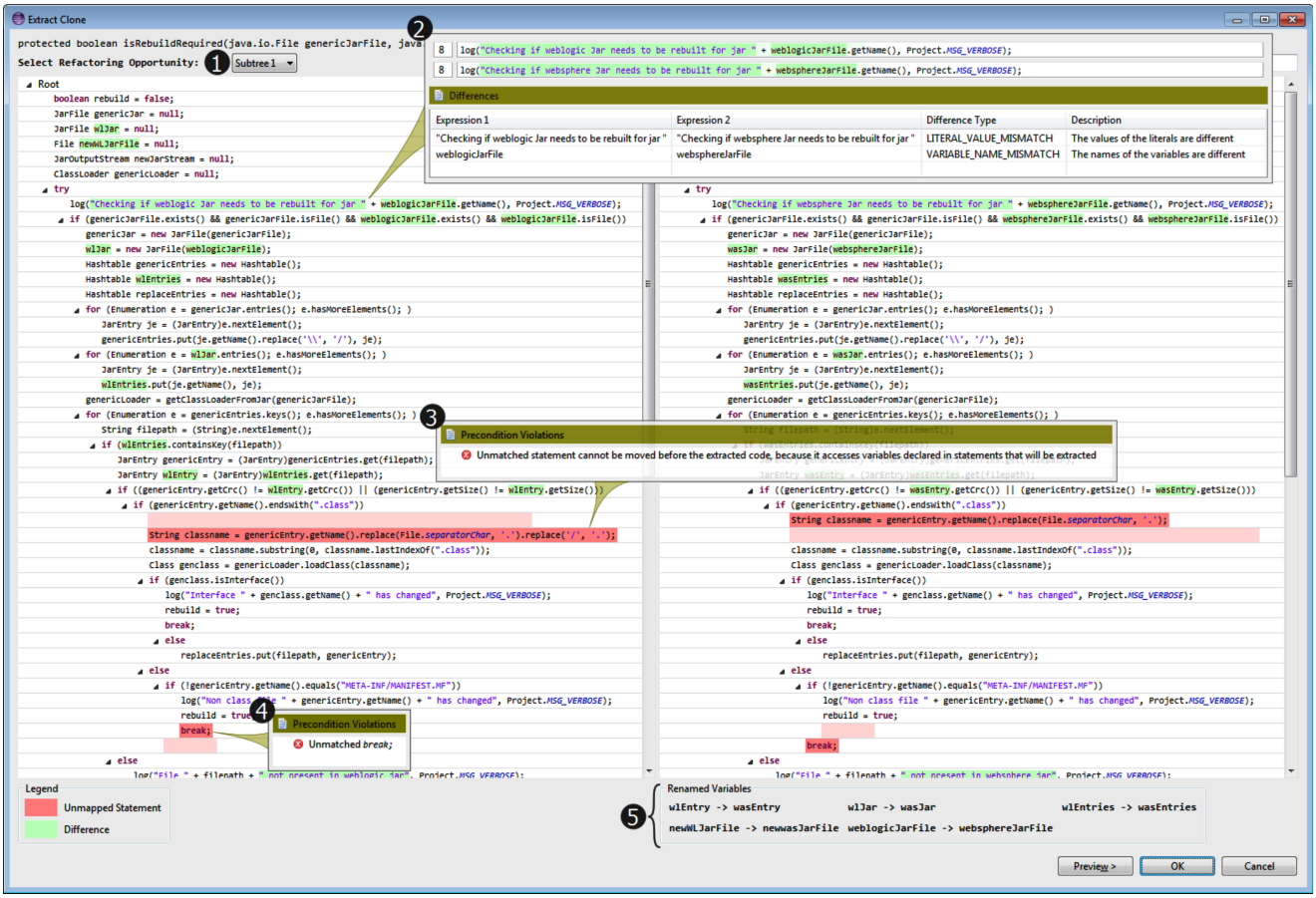


Fig. 8. A visualization of the differences and precondition violations detected in method `isRebuildRequired` of the classes `WeblogicDeploymentTool` (left) and `WebsphereDeploymentTool` (right) in Apache Ant 1.9.0 project.

semantically equivalent and thus one of them can be used in the unified code.

2) *Precondition violations*. Our approach examines all preconditions described in Section 3.4 and presents on each (mapped or unmapped) statement the corresponding precondition violations (Fig. 8, points 3 and 4).

3) *Suggestions*. In some cases of precondition violations, our approach makes suggestions that could make the examined clone fragments refactorable. For example, if there is a difference involving a private method call that cannot be parameterized due to a precondition violation, our tool suggests to inline the called method. Additionally, if there are statements that cannot be unified, because they access variables having different class types, our tool suggests to make these class types extend a common superclass.

Finally, our approach detects the variables that have been consistently renamed in the clone fragments and presents them to the user (Fig. 8, point 5). Differences involving renamed variables should not be parameterized, and therefore are not examined against preconditions. The algorithm for the detection of consistently renamed variables works as follows:

Let set D include all distinct *Variable Identifier* differences (see Table 2 in Section 3.1) that were detected in the examined clone fragments. For each $d \in D$, where d is a pair of variable identifiers in the form (v_i, v_j), we examine two conditions.

1) If D contains a pair of variable identifiers in the form (v_i, y), where $y \neq v_j$ or (x, v_j), where $x \neq v_i$, then d is not considered as a consistent variable rename. In that case, either v_i or v_j has been replaced with two or more different identifiers in the other clone fragment.

2) Let s_i and s_j be a pair of mapped statements and D_s the set of *Variable Identifier* differences detected between these two statements. If s_i uses variable v_i or s_j uses variable v_j, and $(v_i, v_j) \notin D_s$, then d is not considered as a consistent variable rename. In that case, either v_i or v_j has not been replaced with any identifier in the other clone statement.

The second condition is examined on every pair of mapped statements. The proposed algorithm for the detection of renamed variables is greatly affected by the quality of the statement mapping solution produced by our approach.

4.1.2 Clone Refactoring

If there are no precondition violations the user can proceed to refactor the clone fragments. By clicking on the "Preview" button, the user can have a detailed preview of all the changes that will take place in the code after the application of the refactoring. Our tool supports the following refactoring scenarios at the moment:

1) *Extract Method* is applied when the clone fragments are located in methods that belong to the same class.

Clone Group ID	Package	Class	Method	Start Line	End Line	Start Offset	End Offset	#PDG Nodes	#State-ments	Clone Group Size	#Refactor-able Pairs	Details				
762	org.apache.tools.ant.taskdefs.optional.ejb	BorlandDeploymentTool	getBorlandDescriptorHandler	228	231	8201	8444	6	3	6	10	762-1-2	762-1-3	762-1-4	762-1-5	762-1-6
762	org.apache.tools.ant.taskdefs.optional.ejb	GenericDeploymentTool	getDescriptorHandler	377	380	12294	12531	6	3			762-2-3	762-2-4	762-2-5	762-2-6	
762	org.apache.tools.ant.taskdefs.optional.ejb	WeblogicDeploymentTool	getWeblogicDescriptorHandler	443	447	15668	15912	11	3			762-3-4	762-3-5	762-3-6		
762	org.apache.tools.ant.taskdefs.optional.ejb	WebsphereDeploymentTool	getDescriptorHandler	372	376	11229	11473	6	3			762-4-5	762-4-6			
762	org.apache.tools.ant.taskdefs.optional.ejb	WebsphereDeploymentTool	getWebsphereDescriptorHandler	394	398	11894	12138	5	3			762-5-6				
762	org.apache.tools.ant.types	CommandlineJava	setSystem	146	150	5224	5480	12	4							

Fig. 9. A spreadsheet containing the refactorability analysis results for a clone group.

In this scenario the unified code is extracted in a new private method within the same class.

2) *Extract and pull up method* is applied when the clone fragments are located in methods that belong to different subclasses of the same superclass. If the superclass is extended by only these two subclasses, then the unified code is placed in a new protected method within the superclass. If there are more subclasses extending the superclass, then the unified code is placed in a protected method within a new intermediate class extending the common superclass and being inherited by the two subclasses. Fields and methods declared in the subclasses that are commonly accessed in the clone fragments and have an identical structure (i.e., fields having the same type and name, and methods being *Type-1* clones or *Type-2* clones with only local variable renames as differences) are also pulled up.
3) *Introduce template method* is a special case of the previous refactoring. If the methods being commonly accessed in the clone fragments do not belong to the aforementioned clone types, but have an identical signature and the same return type, then an abstract method with the same signature is created in the superclass where the unified code is pulled up. After the application of the refactoring, the unified code will call the newly introduced abstract method that is overridden in the two subclasses. Therefore, this refactoring introduces an instance of the Template Method design pattern [33].
4) *Introduce utility method* is applied when the clone fragments are located in methods of unrelated classes (not being part of the same inheritance hierarchy), and the fragments do not access any instance variables or methods. Then, the unified code can be extracted into a static method placed in a utility class.

4.2 Batch Processing Mode

This mode is suitable for large-scale refactorability analysis of the clones detected in an entire Java project. To enable this kind of analysis, we created a separate Eclipse command-line application that executes the JDeodorant plug-in in headless mode for each clone pair reported by a clone detection tool. The user has to provide the following input:

1) The path to the file/folder containing the clone detection results.
2) The name of the clone detection tool. Currently, CCFinder, Deckard, and NiCad clone detection tools are supported.
3) The name of the Eclipse Java project in which the clones were detected. This project should be open in the Eclipse workspace.
4) The path to the output file of the refactorability analysis report.

The tool performs the analysis in two steps. In the first step, it parses the clone detection results and generates a spreadsheet containing some basic information about the detected clones (Fig. 9). Each row in the spreadsheet corresponds to a clone instance and contains information such as the *clone group id* of the clone instance, the *class* and *method* that the clone belongs to, the start/end *line* and *offset* of the clone in the Java file it belongs to. Additionally, based on the recorded location and start/end offset of the clone instances, the tool determines whether some instances have a sub-clone relationship and records this information in a column of the spreadsheet. Clone y is a sub-clone of x, if x and y belong to the same class and method, and *start-offset*$(y) \geq$ *start-offset*(x) and *end-offset*$(y) \leq$ *end-offset*(x).

In the second step, the tool parses the source code of the specified Java project in which the clones were detected. Next, it processes the clone instances of each clone group by examining all possible combinations of clone pairs in the group. For each clone instance (i.e., row in the spreadsheet) in the examined group, the tool locates the method in which the clone fragment belongs to and generates the method's PDG. At this point, the corresponding row in the spreadsheet is updated with some additional source code analysis information extracted from the PDG, such as the total number of statements in the PDG and the clone fragment, respectively. If a clone instance extends beyond the boundaries of a method (i.e., class-level clone), then the entire clone group is excluded from the analysis.

For each examined clone pair the tool applies the proposed refactorability analysis approach, which results in two pieces of information. The first piece is the statement mapping information, which includes the statements that have been mapped, the differences between the mapped statements, and the unmapped statements from each clone fragment. The second piece is the precondition violation information, which includes all examined preconditions that failed. Both pieces of information are combined into an HTML report as the one shown in Fig. 10. This report makes use of advanced JavaScript and cascading style sheets (CSS) features to give an experience to the user similar to the GUI mode (Section 4.1) experience during the inspection of the

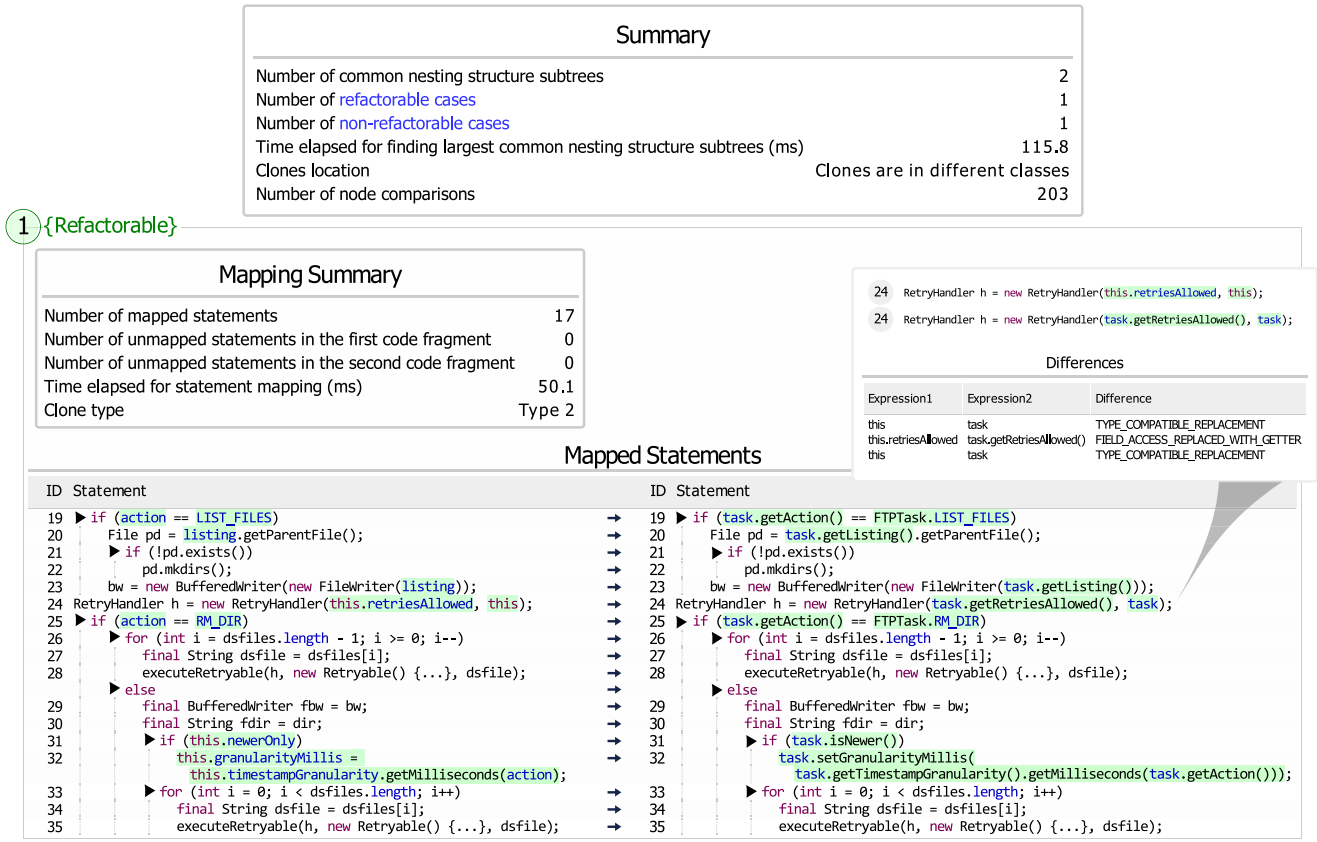


Fig. 10. A dynamic HTML report with the refactoring opportunities detected in method `transferFiles` of the classes `FTP` (left) and `FTPTaskMirrorImpl` (right) in Apache Ant 1.9.0 project.

refactorability analysis results. In the same manner, the user can dynamically interact with the report by collapsing/expanding nodes in the side-by-side tree clone structure visualization, and hovering over mapped and unmapped statements to get more information about the differences and violated preconditions in the form of tooltips. Fig. 10 shows an example of two clone fragments having multiple (often combined) advanced differences, such as the replacement of field accesses with getter method calls, the replacement of field assignments with setter method calls, and the replacement of `this` reference with a variable (i.e., `task`).

After the generation of the HTML report, a cell with a hyperlink to the report is inserted into the spreadsheet as shown in the "Details" area on the right hand side of Fig. 9. The *id* of the cell and the corresponding HTML report file name has the format a-b-c, where a is the clone group id, b is the position of the first clone instance in the group, and c is the position of the second clone instance in the group. The cell gets a green background color if the clone pair is refactorable (i.e., the number of precondition violations is equal to zero), and a red background color in the opposite case. The coloring of the cells allows to easily distinguish the refactorable from the non-refactorable clone pairs, and also discover some interesting patterns among the examined clone pairs. For instance, Fig. 9 shows a clone group consisting of six clone instances detected by Deckard in Apache Ant 1.7.0 project. The "Details" area presents the results from the analysis of all possible clone pair combinations (15 in total). We can easily observe that the sixth (last) clone instance was assessed as non-refactorable in all clone pair combinations in which it is involved. This is a clear indication that the last clone instance has a relatively weak similarity with the rest of the instances in the group, and therefore should be excluded from this particular clone group in the refactoring process.

The reason we selected the combination of a spreadsheet with links to HTML documents for reporting the refactorability analysis results is twofold. First, spreadsheets allow the application of various column-filters in order to filter out undesired clone groups, such as groups without any refactorable clone pairs, and groups containing class-level clone instances, or sub-clone instances. Second, these document types are supported in all operating systems by standard applications (e.g., spreadsheet processors and web browsers), and thus there is no need to install additional software in order to inspect the results.

5 Evaluation

The evaluation section is organized into four parts. In the first part (Section 5.1), we describe the process that we followed for collecting our experimental data.

In the second part (Section 5.2), we evaluate the correctness of the proposed refactorability analysis approach. To achieve this goal, we refactored 610 clone pairs that have been assessed as refactorable and were completely or partially covered by unit tests. We consider a positive refactorability assessment as correct, if the corresponding refactoring is applicable in practice without introducing compile errors and there are no unit test failures after the application of the refactoring.

TABLE 3
Examined Projects

Project	Domain	Age†	Size*
Apache Ant 1.7.0	Java application build tool	$6^{1/2}$	67
Columba 1.4	email client	$1^{1/2}$	75
EMF 2.4.1	modeling framework	$5^{1/2}$	118
JMeter 2.3.2	server performance testing tool	$7^{1/4}$	54
JEdit 4.2	text editor	5	51
JFreeChart 1.0.10	chart library	$7^{1/2}$	76
JRuby 1.4.0	programming language	$3^{1/2}$	101
Hibernate 3.3.2	Java persistence framework	$7^{1/2}$	209
SQuirreL SQL 3.0.3	universal SQL client	8	141

†*years of development from the initial release to the examined release of the project*
** in KLoC*

In the third part (Section 5.3), we evaluate the performance of our approach. For each examined clone pair we collected the execution times corresponding to all three phases of our technique (i.e., the detection of common nesting structures within the clone fragments, the mapping of the statements within the common nesting structures, and the examination of preconditions). In addition, we collected the total number of distinct statement comparisons performed by our technique for each clone pair, and made a comparison with a hypothetical exhaustive search approach that does not take into account the nesting structure of the clone fragments.

In the fourth part (Section 5.4), we perform a large-scale empirical study on the clones detected by four different state-of-the-art clone detection tools in nine Java open-source systems to investigate whether and how the refactorability of software clones is affected from various clone properties, such as the clone source code nature (production versus test code), the relative clone location, the clone type, and the clone size.

5.1 Experiment Setup

In this section, we provide information about the selection of the subject systems and clone detection tools used in the study, as well as the process we followed for collecting the experimental data.[2]

5.1.1 Subject Selection

In order to avoid bias in the selection of projects, we adopted the systems used in the study conducted by Tairas and Gray [34]. As shown in Table 3, the list includes nine Java open-source projects coming from different application domains and having a different development history, ranging from 2 to 8 years. These two variation points certainly affect the characteristics of the detected clones with respect to their domain-specificity and the maturity of the involved code, thus allowing for more generalizable results. Additionally, the projects vary in size ranging from 50 to 200 KLoC.

2. http://tiny.cc/TSE15

5.1.2 Clone Detector Selection

As it is evident from the qualitative study performed by Roy et al. [1], there is a large number of available clone detection tools (over 40), which makes more difficult the selection of tools for the context of our study. Roy et al. [1] categorized the clone detection approaches into five categories, namely text-based, token-based, tree-based, metrics-based, and graph-based. According to this categorization the text-based, token-based, and tree-based techniques are the most dominant (i.e., these three categories have a comparatively larger number of available tools).

Therefore, for our experiment we considered the three most dominant categories of clone detection techniques (i.e., text-based, token-based, and tree-based approaches), and set the following criteria for the selection of a representative tool from each category of clone detection approaches:

1) Public availability for download.
2) Support for the detection of clones in Java programs.
3) Support for the detection of *Type-2* and *Type-3* clones.
4) Extensive use in past empirical studies.

After the examination of the aforementioned criteria, we selected the following clone detection tools.

CCFinder [35] is a popular token-based clone detection technique. According to the quantitative comparison of 6 clone detection tools performed by Bellon et al. [36], CCFinder had the highest recall and a precision comparable to the other techniques (72 percent), and was one of the most efficient tools with respect to execution time and memory requirements.

Deckard [37] is a tree-based clone detection technique, which uses a characteristics vector to approximate the structural information from ASTs in the Euclidean space, and then adapts the locality sensitive hashing (LSH) scheme to efficiently cluster similar vectors using the Euclidean distance. Jiang et al. [37] compared Deckard with CloneDR (a tree-based technique) and CP-Miner (a token-based technique). They concluded that Deckard is faster than CloneDR when the similarity parameter is less than 0.999, and it has a comparable performance to CP-Miner when the similarity parameter is larger than 0.95.

CloneDR [38] is another tree-based clone detection technique, which partitions subtrees of the abstract syntax tree of a program based on a hash function and then compares subtrees in the same partition through tree matching allowing for some divergences [36]. In the quantitative comparison performed by Bellon et al. [36], CloneDR had the highest precision (100 percent), but the lowest recall (9 percent). We included CloneDR in our experiments, because it computes the similarity of the clones based on the number of shared and different AST nodes in their AST representation, and thus can be considered as a "pure" AST-based approach.

NiCad [39] is a text-based clone detection technique, but exploits the benefits of tree-based structural analysis based on lightweight parsing to implement flexible pretty-printing, code normalization, source transformation and code filtering. Roy and Cordy [40] created an automatic framework for evaluating code clone detection tools that generates randomly mutated clone fragments from the original code base, and injects them randomly into the code base to get mutated code bases. Using this evaluation framework, they found

TABLE 4
Configuration of Clone Detection Tools

Tool	Configuration option	value
CCFinderX 10.2.7.4 [35]	Minimum clone length*	**50**
	Minimum TKS (# distinct token types)	**12**
	Block shaper level	**2-Soft**
	Parameterized match	**true**
Deckard 1.3 [37]	Minimum size*	**50**
	Stride (sliding window size)	**0**
	Similarity	**0.95**
CloneDR 2.2.12 [38]	Similarity threshold	**0.95**
	Max clone parameters	**65,535**
	Min clone mass†	**6**
	Characters per node	**16**
	Starting depth	**2**
NiCad 3.5 [39]	Minimum size‡	**5**
	Maximum size‡	**2,500**
	Maximum difference threshold	**0.2**
	Renaming	**blind**
	Clone granularity	**function**

* *number of tokens*
† *number of AST nodes*
‡ *number of pretty printed lines.*

out that Full NiCad (a version of NiCad that utilizes the entire range of the aforementioned NiCad capabilities) had 100 percent recall and over 96 percent precision for *Type-2* and *Type-3* generated clones.

Table 4 shows the configuration we used for each clone detection tool. For NiCad we used the standard settings for detecting all three type of clones as provided by its authors [12]. For CCFinder, Deckard, and CloneDR we used the default settings coming with the tools. Since none of the tools has a restriction on the number of differences that are allowed in the detected clones, we configured the maximum number of clone parameters for CloneDR to 65,535 (i.e., a number that is large enough to be considered as a restriction waiver). The same configuration for the number of clone parameters has been used by [37].

Each of the aforementioned tools has a limitation in the reporting of the detected clones that we had to tackle to ensure that our approach takes accurate and valid clone fragments as input. CCFinder reports sequences of tokens as clones using a start and an end offset. However, in several cases the reported sequence may contain incomplete statements (e.g., a method call where only the invocation expression is within the token sequence). We have removed all incomplete statements in the reported clone fragments. Deckard reports the starting line of the clone fragments and the number of subsequent lines that should be included in the clone fragment. However, in several cases the lines containing the closing brackets of conditional and loop statements are omitted. To avoid considering these statements as incomplete, we modified our clone fragment extraction algorithm to tolerate any missing closing bracket or parenthesis characters. NiCad reports the start and end lines of the clone fragments. However, in several cases the lines containing the signature of the method declaration are also included in the reported fragments. Since our approach is able to handle only code within method bodies, we modified our clone fragment extraction algorithm to set the start offset of the clone fragment to the start offset of the first statement being present in the body of the method. Finally, CloneDR in several occasions reports lists of method parameter declarations as clones. We have excluded all these cases from our analysis.

5.1.3 Data Collection

In order to collect the data required for our experiment, we followed the process shown in Fig. 11 for the clones detected from each clone detection tool in each examined project (i.e., 9 projects $\times$ 4 clone detectors $=$ 36 datasets). We have excluded the generated code found in some projects (see webpage[2]) from the clone detection process, since generated code is never modified manually [41], and thus it is excluded from analysis in most clone-related studies [42].

The collected data is organized in two levels, namely clone pair related data, and statement mapping related data. For each clone pair we record the following information.

1) The number of statements in the first and second clone fragment, respectively.

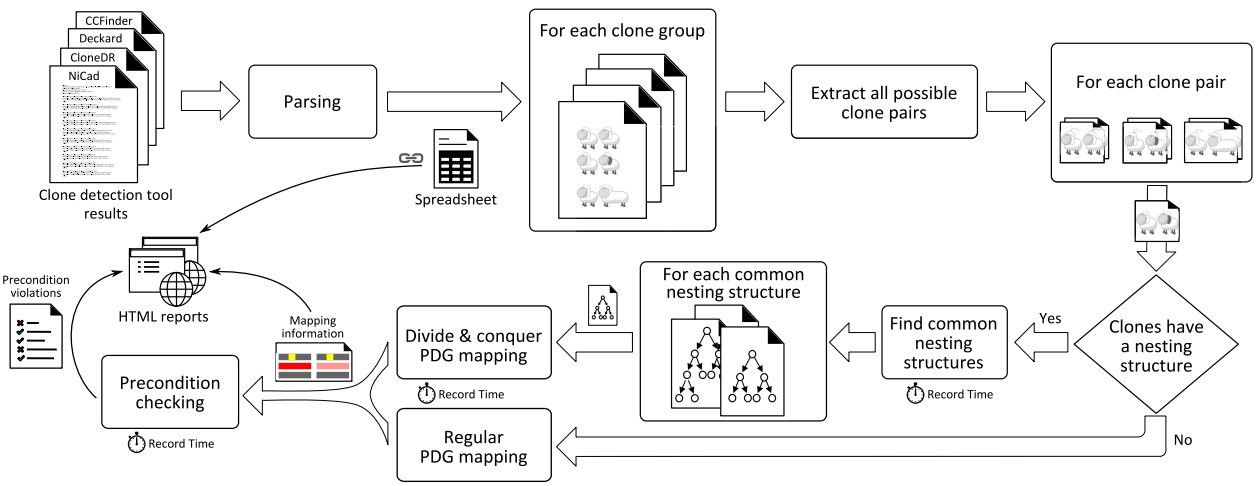


Fig. 11. The workflow applied for the collection of the experimental data.

2) The total number of statements inside the methods where the first and second clone fragment are located, respectively.
3) The number of common nesting structures found within the clone fragments.
4) The time elapsed for the detection of the common nesting structures.
5) The number of nesting structures that were eventually assessed as refactorable.
6) The total number of statement comparisons performed for the examined clone pairs.
7) The relative location of the clone fragments, which takes five possible values:
   a) Same method declaration.
   b) Different method declarations within the same type declaration.
   c) Different type declarations within the same compilation unit (i.e., Java file).
   d) Different type declarations within the same inheritance hierarchy.
   e) Unrelated compilation units.

For each pair of common nesting subtrees found in a clone pair we record the following information.

1) The number of mapped statements.
2) The number of unmapped statements from the first and second clone fragment, respectively.
3) The clone type (*Type-1*, *Type-2*, and *Type-3*).
4) The time elapsed for the mapping of the statements within the common nesting subtrees.
5) The number of differences between the mapped statements.
6) The list of violated preconditions.
7) The time elapsed for the examination of preconditions.

For determining the clone type we have defined our own rules, since none of the clone detection tools used in the study reports the types of the detected clones. We consider an instance as a *Type-1* clone, if the number of differences between the mapped statements is equal to zero, and the number of unmapped statements in both clone fragments is equal to zero. We consider an instance as a *Type-2* clone, if the number of differences between the mapped statements is larger than zero and the number of gaps is equal to zero. A gap exists, when there is a different number of unmapped statements nested under a mapped control statement. All remaining cases are classified as *Type-3* clones.

5.2 Correctness

In this section, our goal is to evaluate the correctness of our Extract Clone Refactoring implementation with respect to the examined preconditions and the applied source code transformation. In the recent years, there has been a significant progress in the development of techniques for the automated testing of refactoring engines.

Daniel et al. [43] proposed a technique for automated testing of refactoring engines, which generates abstract syntax trees representing complex Java programs as test inputs for refactoring engines. They also created oracles to automatically check if the refactoring engine transforms the generated program correctly. The oracles check that a) the refactoring engine does not crash, b) the refactored program compiles, c) the refactoring can be inverted, and d) the refactored programs returned by different engines have equivalent ASTs. This approach also requires the implementation of refactoring-specific oracles, which are aware of the structural changes applied by their corresponding refactorings and thus check whether the refactored program exhibits the expected changes. Testing our Extract Clone Refactoring engine with this approach would require the implementation of custom oracles that examine changes related to the extraction of a method, the introduction of a new superclass, and the parameterization of all kinds of differences that could be potentially found between the clone fragments.

Soares et al. [44] proposed a technique for automated behavioral testing of refactoring engines, which also generates programs for a given scope of Java declarations (packages, classes, fields, and methods) as test inputs, but uses SafeRefactor [45] as an oracle to evaluate the correctness of refactorings. SafeRefactor first finds the methods with matching signatures in the programs before and after the refactoring transformation. Next, it applies a Java unit test generator to produce a test suite for those methods. In the final step, it executes the tests on the programs before and after the refactoring transformation and reports a behavioral change if the test results are different. Their technique detects behavioral changes related to method overriding and overloading, field hiding, class variable shadowing, moving a super reference up or down the hierarchy, and changing the accessibility of a field or method. Although the supported behavioral changes can cover a part of the Extract Clone Refactoring transformation dealing with pulling up methods and fields to an existing superclass, they certainly cannot cover the more advanced parts of the transformation dealing with the parameterization of the differences found between the clone fragments, and the move of the unmapped statements in the case of *Type-3* clones.

Since the aforementioned approaches cannot adequately support the detection of behavioral changes in all the steps of the Extract Clone Refactoring transformation, we decided to evaluate the correctness of the proposed refactoring implementation based on the following hypothesis:

> A clone pair that has been assessed as refactorable should be possible to be refactored without causing any compile errors, and all unit tests of the project should pass after the application of the refactoring.

The testing condition makes sense if at least one of the clone fragments in the examined clone pair is covered by an existing test. As a result, we focused our evaluation on the clone pairs that have been assessed as refactorable by our approach and at least one of the clone fragments is partially or fully covered by a unit test.

The workflow we applied for this experiment involves the following steps:

1) For each clone pair assessed as refactorable, examine if at least one of the clone fragments is covered by a test. We used the EclEmma[3] Java Code Coverage

3. http://www.eclemma.org/

Eclipse plug-in to get the source code that is covered by the tests of the examined project. EclEmma highlights directly in the editor the statements that are fully covered in green color, the statements that are partially covered in yellow color, and the statements that are not covered in red color.

2) If the clone pair is covered by a test, then first extract the clone fragments indicated by the clone detection tool into two separate methods using the Eclipse Extract Method refactoring. Next, using the Clone Refactoring tool support described in Section 4.1, refactor the two extracted methods containing the clone fragments.
3) If the clone refactoring is successful (i.e., there are no compile errors after its application), execute the entire test suite of the project and record the tests that failed.
4) Undo the sequence of refactorings applied in step 2 to bring the system to its original state.

Since the aforementioned workflow is quite time consuming (even with the provided clone refactoring tool support), we decided to evaluate our approach on a single dataset out of the 36 datasets (9 projects $\times$ 4 clone detectors) collected in total. We set the following criteria for selecting an appropriate project for this experiment.

1) The project should contain a significant number of clone pairs assessed as refactorable by our approach.
2) The project should have a test suite with a relatively high source code coverage percentage to increase the probability of having clone pairs covered by a test.
3) The test suite should execute without any test failures before the application of refactorings in the project.
4) The project should contain clones in different classes of the same inheritance hierarchy to enable the testing of more advanced clone refactorings supported by our tool, such as Pull Up Method, and Introduce Template Method.

The project we selected based on the aforementioned criteria is *JFreeChart* (version 1.0.10). We focused our analysis only on the production code present in the `src` directory, excluding the `experimental`, `swt`, and `test` source directories. The examined version of JFreeChart has 1,091 refactorable clone pairs in total (the total number of method-level clone pairs reported by Deckard is 2,306), out of which 610 are covered by a test. The code coverage for the `src` directory is 54.4 percent based on EclEmma. Out of the 1091 refactorable clone pairs in total, 489 (45 percent) of them belong to different classes of the same inheritance hierarchy, while 360 (33 percent) belong to the same method and 185 (17 percent) belong to different methods of the same class. The remaining 57 clone pairs (5 percent) belong to different classes that are not part of the same inheritance hierarchy, and thus can be refactored either by introducing a new common superclass (if there are instance variables and methods commonly accessed in the clone fragments), or creating a new utility class (if only static fields and methods are commonly accessed in the clone fragments). This diversity in the relative location of the clone fragments certainly allows us to test all clone refactoring scenarios supported by our approach (Section 4.1).

The experiment was conducted by the last author of the paper, Giri Krishnan, in consultation with the first author of the paper. Krishnan was a software engineer at Infosys for three years prior his graduate studies at Concordia University, and at the time of the experiment, he had over five years of experience in Java programming. The experiment lasted for two months, and by examining carefully some failed refactorings and tests, we were able to extend the list of preconditions with some new ones (e.g., Preconditions #3 and #4) that have not been originally considered.

We refactored in total 610 clone pairs that were covered by a test and were assessed as refactorable by our approach. All 610 refactorings were applied without causing any compile errors. However, 13 refactorings led to a test failure related to serialization. The involved clone fragments belong to classes `CombinedDomainXYPlot` and `CombinedRangeXYPlot` (both having class `XYPlot` as a superclass), and access identical fields (i.e., fields having the same type and name) in the subclasses. As a result, the applied refactorings pulled up the commonly accessed fields in the `XYPlot` superclass. However, the deletion of the commonly accessed fields from the subclasses caused a failure in the deserialization of existing serialized objects, because field deletion is an incompatible change with respect to maintaining the contract of a class for the purpose of serialization.

To overcome this limitation, we adjusted the mechanics of our refactoring implementation to avoid pulling up the non-transient commonly accessed fields (the `transient` keyword excludes a field from being serialized), if the clone fragments accessing those fields belong to classes implementing the `Serializable` interface. Instead, we introduce a parameter in the extracted method for each pair of commonly accessed fields (as if there was a difference in the names of the accessed fields). We refactored again the 13 clone pairs for which the serialization test failed using the new refactoring mechanics, and this time all tests were successful.

In conclusion, all 610 clone pairs assessed as refactorable by our approach were successfully refactored without causing any compile errors and test failures. The detailed results of this experiment can be found online[2] under the section titled "Testing Results". Based on these results, we can conclude, to a large extent, that the clone pairs that are assessed as refactorable by our approach can be indeed safely refactored.

5.3 Performance

To evaluate the performance of the proposed refactorability analysis approach we collected the execution time for every step, namely a) the detection of common nesting subtrees in the input clone fragments, b) the mapping of the statements nested under each matched nesting subtree, and c) the examination of preconditions. Table 5 shows some descriptive statistics for the total execution time (i.e., the sum of the execution times for all three steps of our approach) collected over all clone pairs in the 36 datasets (9 projects $\times$ 4 clone detectors).

TABLE 5
Execution Times in Seconds

Execution Time (x)	# Cases	%
$x \leq 1$	1,149,274	99.853
$1 < x \leq 10$	1,460	0.127
$10 < x \leq 100$	177	0.015
$x > 100$	56	0.005
Total	1,150,967	100

** Measurements performed on Intel Core i7-3770 3.4 GHz with 8 GB DDR3 RAM.*

As it can be observed from Table 5, in 99.85 percent of the cases the total execution time of our approach is less than one second. In 0.127 percent of the cases the total execution time is between one and 10 seconds. In 0.015 percent of the cases the total execution time is between 10 and one hundred seconds. Finally, in only 56 cases (0.005 percent) the execution time was over one hundred seconds with a maximum value of 199.4 seconds. We examined carefully all cases with an execution time over one hundred seconds and we found out that the largest portion of the time was spent on the examination of preconditions. In particular, Precondition #1 that examines the differences between mapped statements, analyzes the dependencies between the expressions in the differences that should be parameterized and the rest of the mapped statements. If the expressions in the differences contain a method call, then its call graph is analyzed to find whether the state of instance variables (i.e., class fields), or the object references being present in the method call (i.e., arguments and invocation expression) is modified/accessed in subsequent methods of the call graph. In all examined cases, we discovered that they had differences containing method calls, and the corresponding call graphs were exceptionally large and required significantly more time to be traversed. Large call graphs are often caused by polymorphic method calls (i.e., abstract method calls), which are implemented in a large number of subclasses. Our approach examines all subclass implementations of the abstract method calls to extract all possible dependencies.

To evaluate the performance of our greedy statement mapping algorithm, we compare it with a hypothetical exhaustive search approach. We consider that an exhaustive search approach would perform all possible comparisons between the statements of the clone fragments. Assuming that the first clone fragment has n_1 statements and the second one has n_2 statements, the maximum number of distinct statement comparisons is equal to $n_1 \times n_2$. Fig. 12 shows the violin plots [46] for the number of distinct node comparisons performed by our approach and a hypothetical exhaustive search approach over all clone pairs in the 36 examined datasets. The median value of distinct node comparisons performed by our approach is 1.33 times smaller than the median value in the exhaustive search approach (12 versus 16 comparisons), while the mean value of distinct node comparisons performed by our approach is 1.64 times smaller than the mean value in the exhaustive search approach (24 versus 40 comparisons).

We also applied the one-tailed variant of the Wilcoxon signed-rank test on the paired samples of the node comparisons performed by our approach and the exhaustive search approach with the following null hypothesis: "the number of node comparisons performed by our approach is larger than the number of node comparisons performed by the exhaustive search approach". The null hypothesis was rejected with significance at 95 percent confidence level ($p\text{-value} < 2.2 \times 10^{-16}$), and thus we can conclude that the number of node comparisons performed by our approach is smaller than the number of node comparisons performed by the exhaustive search approach.

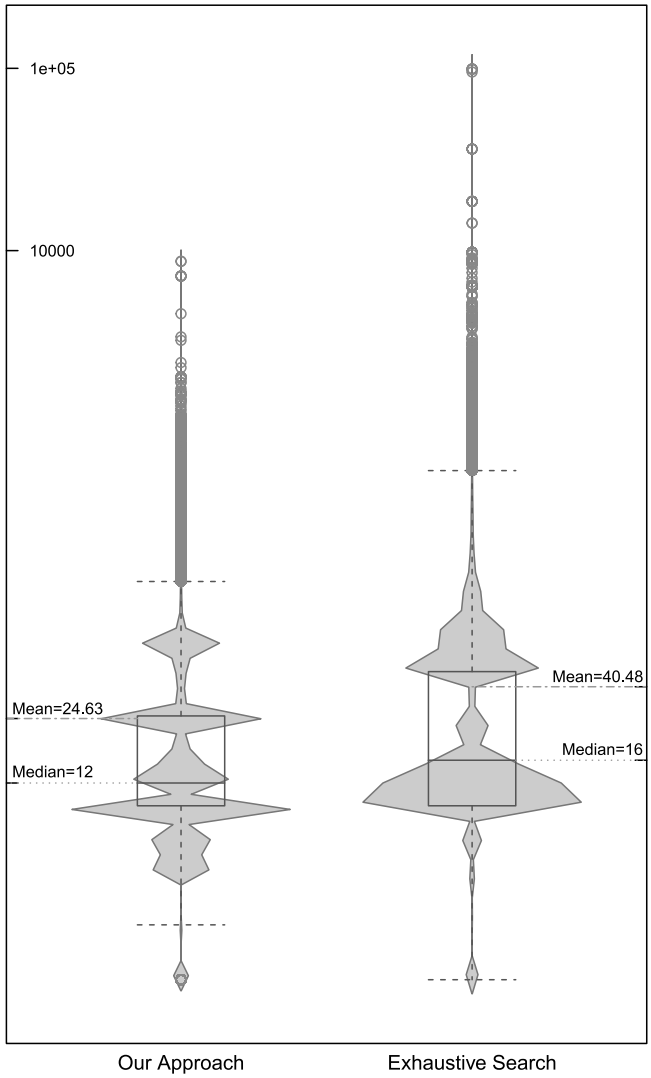


Fig. 12. Number of distinct node comparisons.

In conclusion, the computational cost introduced by our approach for determining the refactorability of clones is negligible (less than a second) in the vast majority of the examined clone pairs (99.85 percent). Additionally, our statement mapping algorithm takes advantage of the nesting structure of the clone fragments to decompose the original mapping problem, thus reducing drastically the number of distinct node comparisons required to find a solution.

5.4 Empirical Study

In this section, our goal is to explain how the refactorability of software clones is affected by different clone properties and tool configurations. To this end, we conduct an empirical study by analyzing over a million clone pairs detected

TABLE 6
Overview of Refactorability Analysis Results

Project Name	CCFinder			Deckard			CloneDR			NiCad		
	Pairs	Refactorable	%	Pairs	Refactorable	%	Pairs	Refactorable	%	Pairs	Refactorable	%
Apache Ant	3062	1749	57.1	1132	511	45.1	3371	2735	81.1	84165	4041	4.8
Columba	2665	1064	39.9	2048	838	40.9	5846	3737	63.9	57948	2210	3.8
EMF	5289	1459	27.6	2911	606	20.8	5675	3733	65.8	52831	2536	4.8
JMeter	1285	669	52.1	1532	742	48.4	803	500	62.3	58598	3870	6.6
JEdit	414	208	50.2	334	125	37.4	744	457	61.4	758	174	23.0
JFreeChart	188085	31734	16.9	79429	4716	5.9	65840	7768	11.8	189894	7530	4.0
JRuby	2360	1254	53.1	1316	523	39.7	2981	1926	64.6	39073	3689	9.4
Hibernate	4288	1438	33.5	1880	551	29.3	4050	2213	54.6	61376	2177	3.5
SQuirreL SQL	5600	2762	49.3	2523	1849	73.3	13894	11886	85.5	196967	19947	10.1
Total	213048	42337	19.9	93105	10461	11.2	103204	34955	33.9	741610	46174	6.2

by four different clone detection tools in nine open-source projects to answer the following research questions:

RQ1: How does the source code type (production versus test) of software clones affect their refactorability?

RQ2: How does the relative location of software clones affect their refactorability?

RQ3: How does the clone type of software clones affect their refactorability?

RQ4: How does the size of software clones affect their refactorability?

RQ5: What are the most frequent reasons (precondition violations) that hinder the refactoring of software clones?

In all investigated research questions, we present and discuss the results separately for each clone detection tool to explore the effect of different clone detection approaches and tool configurations.

Table 6 presents an overview of the refactorability analysis results per examined project and clone detection tool. The first observation that we can make is that Deckard and CloneDR (i.e., the AST-based clone detectors) detect a considerably smaller number of clone pairs compared to the other two clone detection tools (two times less than CCFinder, and seven times less than NiCad). This could be perhaps explained by the fact that Deckard and CloneDR were configured with a comparatively higher similarity threshold (0.95) that does not allow the matching of code fragments with extensive differences. In terms of the percentage of refactorable clone pairs detected by each tool, we can observe that CloneDR clearly outperforms the other tools with 33.9 percent, while CCFinder, Deckard and NiCad have 19.9, 11.2, and 6.2 percent, respectively.

Table 7 shows the refactorability analysis results per clone detection tool focusing only on method-level clone pairs (i.e., clones extending over the entire body of the methods they have been found in). The first observation that we can make is that NiCad detects a considerably larger percentage of method-level clone pairs (60 percent of the detected clones are method-level clones) compared to the other clone detection tools, while CCFinder detects a considerably smaller percentage of method-level clone pairs (only 5.3 percent of the detected clones are method-level clones). However, NiCad has the lowest percentage of refactorable method-level clone pairs with 7.7 percent, while CloneDR has the highest percentage of refactorable method-level clone pairs with 79.8 percent.

In the sections that follow, we will shed more light on these results by investigating the aforementioned research questions.

5.4.1 Production versus Test Code

In this section, we examine whether clone pairs found in production code exhibit a different refactorability compared to clone pairs found in test code. Table 8 shows the refactorability analysis results per examined project, clone detection tool and source type (i.e., production and test code). Projects EMF and JEdit have not been included in the table, because they do not contain test code.

In general, all clone detection tools detect more clone pairs in test code (CCFinder: 79.4 percent, Deckard: 81.9 percent, CloneDR: 64.7 percent), with the exception of NiCad that detects more clone pairs in production code (85.7 percent). Depending on the size of the test code base in each project, we have cases where the percentage of clone pairs detected in the test code is considerably larger than the production code (e.g., JFreeChart), or exactly the opposite (e.g., SQuirreL SQL, JRuby, and Columba).

CloneDR is able to detect considerably more refactorable clone pairs in production code (with 70.5 percent), and is followed by Deckard (with 36.1 percent) and CCFinder (with 23.6 percent), while NiCad has the smallest percentage of refactorable clone pairs in production code (with 6.5 percent). From these results, we can conclude that the

TABLE 7
Method Clone Pairs and Refactorability

Tool	Method Clone Pairs (%)	Rafactorable (%)
CCFinder	11281 (5.3)	4313 (38.2)
Deckard	26514 (28.5)	5844 (22.0)
CloneDR	19900 (19.3)	15874 (79.8)
NiCad	445156 (60.0)	34218 (7.7)
Total	502851 (43.7)	60249 (12.0)

TABLE 8
Production versus Test Code and Refactorability

Project Name	Source Type	CCFinder		Deckard		CloneDR		NiCad	
		Pairs (%)	Ref. (%)	Pairs (%)	Ref. (%)	Pairs (%)	Ref. (%)	Pairs (%)	Ref. (%)
Apache Ant	Production	1453 (47.5)	765 (52.6)	566 (50.0)	201 (35.5)	2196 (65.1)	1718 (78.2)	82939 (98.5)	3859 (4.7)
	Test	1609 (52.5)	984 (61.2)	566 (50.0)	310 (54.8)	1175 (34.9)	1017 (86.6)	1226 (1.5)	182 (14.8)
Columba	Production	2458 (92.2)	928 (37.8)	1511 (73.8)	475 (31.4)	4414 (75.5)	2484 (56.3)	57762 (99.7)	2123 (3.7)
	Test	207 (7.8)	136 (65.7)	537 (26.2)	363 (67.6)	1432 (24.5)	1253 (87.5)	186 (0.3)	87 (46.8)
JMeter	Production	647 (50.4)	386 (59.7)	1136 (74.2)	495 (43.6)	406 (50.6)	209 (51.5)	58364 (99.6)	3760 (6.4)
	Test	638 (49.6)	283 (44.4)	396 (25.8)	247 (62.4)	397 (49.4)	291 (73.3)	234 (0.4)	110 (47.0)
JFreeChart	Production	23668 (12.6)	2307 (9.7)	5456 (6.9)	1526 (28.0)	5209 (7.9)	2583 (49.6)	97208 (51.2)	4230 (4.4)
	Test	164417 (87.4)	29427 (17.9)	73973 (93.1)	3190 (4.3)	60631 (92.1)	5185 (8.6)	92686 (48.8)	3300 (3.6)
JRuby	Production	2347 (99.4)	1249 (53.2)	1298 (98.6)	510 (39.3)	2949 (98.9)	1907 (64.7)	39043 (99.9)	3673 (9.4)
	Test	13 (0.6)	5 (38.5)	18 (1.4)	13 (72.2)	32 (1.1)	19 (59.4)	30 (0.1)	16 (53.3)
Hibernate	Production	2192 (51.1)	358 (16.3)	1166 (62.0)	322 (27.6)	1291 (31.9)	869 (67.3)	57139 (93.1)	1749 (3.1)
	Test	2096 (48.9)	1080 (51.5)	714 (38.0)	229 (32.1)	2759 (68.1)	1344 (48.7)	4237 (6.9)	428 (10.1)
SQuirreL SQL	Production	5477 (97.8)	2701 (49.3)	2472 (98.0)	1823 (73.7)	13503 (97.2)	11702 (86.7)	189412 (96.2)	19232 (10.2)
	Test	123 (2.2)	61 (49.6)	51 (2.0)	26 (51.0)	391 (2.8)	184 (47.1)	7555 (3.8)	715 (9.5)
Total	Production	43945 (20.6)	10361 (23.6)	16850 (18.1)	6083 (36.1)	36387 (35.3)	25662 (70.5)	635456 (85.7)	41336 (6.5)
	Test	169103 (79.4)	31976 (18.9)	76255 (81.9)	4378 (5.7)	66817 (64.7)	9293 (13.9)	106154 (14.3)	4838 (4.6)

* Projects EMF and JEdit are excluded from this table, because they do not have any test code.

AST-based clone detectors used in the experiment (i.e., CloneDR and Deckard) tend to detect more refactorable clones in production code. On the other hand, CCFinder tends to detect more refactorable clone pairs in test code (with 18.9 percent), and is followed by CloneDR (with 13.9 percent), while Deckard and NiCad have the smallest percentages of refactorable clone pairs in test code (with 5.7 percent and 4.6 percent, respectively).

By examining the percentages of refactorable clone pairs in the last row of Table 8, which contains all clone pairs detected in all examined projects by each tool, we can see that all tools have detected a higher percentage of refactorable clone pairs in production code compared to the percentage of refactorable clone pairs in test code. This could be a possible explanation about the reason developers perform a considerably smaller number of clone-related refactorings (e.g., Extract Method, Pull Up Method, and Extract Superclass) in test code compared to production code [47].

RQ1 conclusion. Clones in production code tend to be more refactorable than clones in test code. AST-based clone detection tools (i.e., CloneDR and Deckard) tend to detect more refactorable clones in production code. CCFinder tends to detect more refactorable clones in test code.

5.4.2 Relative Location

In this section, we examine how the relative location of the clone pair fragments affects the refactorability. Table 9 presents the refactorability analysis results per relative location and clone detection tool, presented separately for a) all clone pairs, b) the clone pairs found in production code, and c) the clone pairs found in test code. As it can be observed in the top part of Table 9 containing the results for all clone pairs, all tools (with the exception of NiCad) detect the majority of the clone pairs in classes belonging to the same inheritance hierarchy. This is even

TABLE 9
Relative Clone Location and Refactorability

Clone location		CCFinder		Deckard		CloneDR		NiCad	
		Pairs (%)	Ref. (%)	Pairs (%)	Ref. (%)	Pairs (%)	Ref. (%)	Pairs (%)	Ref. (%)
Same Method	All	9584 (4.5)	4008 (41.8)	1961 (2.1)	1167 (59.5)	4955 (4.8)	3792 (76.5)	3347 (0.5)	1228 (36.7)
Same Type	All	9652 (4.5)	3936 (40.8)	6892 (7.4)	2168 (31.5)	13337 (12.9)	8458 (63.4)	23071 (3.1)	5519 (23.9)
Same Java File	All	116 (0.1)	83 (71.6)	45 (0.0)	22 (48.9)	358 (0.3)	183 (51.1)	1110 (0.1)	308 (27.7)
Same Hierarchy	All	173728 (81.5)	33226 (19.1)	79594 (85.5)	6245 (7.8)	78944 (76.5)	19395 (24.6)	188937 (25.5)	20877 (11.0)
Unrelated Types	All	19968 (9.4)	1084 (5.4)	4613 (5.0)	859 (18.6)	5610 (5.4)	3127 (55.7)	525145 (70.8)	18242 (3.5)
Same Method	Production	7508 (17.1)	2617 (34.9)	1317 (7.8)	854 (64.8)	2629 (7.2)	1871 (71.2)	3253 (0.5)	1166 (35.8)
Same Type	Production	5443 (12.4)	1467 (27.0)	4772 (28.3)	931 (19.5)	8288 (22.8)	5158 (62.2)	20885 (3.3)	4547 (21.8)
Same Java File	Production	113 (0.3)	80 (70.8)	35 (0.2)	21 (60.0)	351 (1.0)	176 (50.1)	903 (0.1)	274 (30.3)
Same Hierarchy	Production	11008 (25.0)	5148 (46.8)	6225 (36.9)	3505 (56.3)	19604 (53.9)	15399 (78.6)	92579 (14.6)	17245 (18.6)
Unrelated Types	Production	19873 (45.2)	1049 (5.3)	4501 (26.7)	772 (17.2)	5515 (15.2)	3058 (55.4)	517836 (81.5)	18104 (3.5)
Same Method	Test	2076 (1.2)	1391 (67.0)	644 (0.8)	313 (48.6)	2326 (3.5)	1921 (82.6)	94 (0.1)	62 (66.0)
Same Type	Test	4209 (2.5)	2469 (58.7)	2120 (2.8)	1237 (58.3)	5049 (7.6)	3300 (65.4)	2186 (2.1)	972 (44.5)
Same Java File	Test	3 (0.0)	3 (100.0)	10 (0.0)	1 (10.0)	7 (0.0)	7 (100.0)	207 (0.2)	34 (16.4)
Same Hierarchy	Test	162720 (96.2)	28078 (17.3)	73369 (96.2)	2740 (3.7)	59340 (88.8)	3996 (6.7)	96358 (90.8)	3632 (3.8)
Unrelated Types	Test	95 (0.1)	35 (36.8)	112 (0.1)	87 (77.7)	95 (0.1)	69 (72.6)	7309 (6.9)	138 (1.9)

TABLE 10
Clone Type and Refactorability

Clone type		CCFinder		Deckard		CloneDR		NiCad	
		Pairs (%)	Ref. (%)	Pairs (%)	Ref. (%)	Pairs (%)	Ref. (%)	Pairs (%)	Ref. (%)
Type I	All	8029 (3.8)	5566 (69.3)	4165 (4.5)	3298 (79.2)	17910 (17.4)	14291 (79.8)	5705 (0.8)	4881 (85.6)
Type II	All	200396 (94.1)	36113 (18.0)	84585 (90.8)	6597 (7.8)	83934 (81.3)	20594 (24.5)	287010 (38.7)	29288 (10.2)
Type III	All	4340 (2.0)	658 (15.2)	3837 (4.1)	566 (14.8)	238 (0.2)	70 (29.4)	182751 (24.6)	12005 (6.6)
Unknown[†]	All	283 (0.1)	0 (0.0)	518 (0.6)	0 (0.0)	1122 (1.1)	0 (0.0)	266144 (35.9)	0 (0.0)
Type I	Production	3402 (7.7)	2609 (76.7)	2119 (12.6)	1942 (91.6)	12357 (34.0)	10707 (86.6)	4091 (0.6)	3738 (91.4)
Type II	Production	37764 (85.9)	7465 (19.8)	12163 (72.2)	3649 (30.0)	22989 (63.2)	14926 (64.9)	212656 (33.5)	25977 (12.2)
Type III	Production	2536 (5.8)	287 (11.3)	2095 (12.4)	492 (23.5)	131 (0.4)	29 (22.1)	157306 (24.8)	11621 (7.4)
Unknown	Production	243 (0.6)	0 (0.0)	473 (2.8)	0 (0.0)	910 (2.5)	0 (0.0)	261403 (41.1)	0 (0.0)
Type I	Test	4627 (2.7)	2957 (63.9)	2046 (2.7)	1356 (66.3)	5553 (8.3)	3584 (64.5)	1614 (1.5)	1143 (70.8)
Type II	Test	162632 (96.2)	28648 (17.6)	72422 (95.0)	2948 (4.1)	60945 (91.2)	5668 (9.3)	74354 (70.0)	3311 (4.5)
Type III	Test	1804 (1.1)	371 (20.6)	1742 (2.3)	74 (4.2)	107 (0.2)	41 (38.3)	25445 (24.0)	384 (1.5)
Unknown	Test	40 (0.0)	0 (0.0)	45 (0.1)	0 (0.0)	212 (0.3)	0 (0.0)	4741 (4.5)	0 (0.0)

[†] The Unknown clone type refers to the cases where the number of statements mapped by our approach is equal to zero.

more evident when considering only the clone pairs detected in test code (bottom part of Table 9), since most test classes extend class `TestCase` from the JUnit testing framework. On the other hand, NiCad detects a very large percentage of clone pairs (70.8 percent), in which the clone fragments belong to unrelated types.

In general, the closer the relative location of the clones (i.e., same method, type, or file), the higher the percentage of refactorable clone pairs detected by the tools. This probably means that the clones being located closer to each other have a higher similarity by nature, while clones in distant locations (i.e., same hierarchy, or unrelated types) tend to diverge more [48]).

By analyzing the results for all clone pairs (top part of Table 9), we can observe that CloneDR is able to detect the highest percentage of refactorable clone pairs in every location category, with the exception of the "Same Java File" category where CCFinder has the highest percentage. Another finding is that CloneDR is able to detect a considerably larger percentage of refactorable clone pairs in the location category of "Unrelated Types" (with 55.7 percent), while Deckard, CCFinder and NiCad have a much poorer performance (with 18.6, 5.4 and 3.5 percent, respectively) in that category.

By analyzing separately the results for the clone pairs in production code (middle part of Table 9), we can see that CloneDR again detects the highest percentage of refactorable clone pairs in every location category, with the exception of the "Same Java File" category where CCFinder has the highest percentage. When considering only the clone pairs in test code (bottom part of Table 9), CCFinder clearly outperforms the other tools in the detection of refactorable clone pairs in the same inheritance hierarchy, which constitute 96 percent of all clone pairs found in test code.

RQ2 conclusion. Clones with a close relative location (i.e., same method, type, or file) tend to be more refactorable than clones in distant locations (i.e., same hierarchy, or unrelated types). CloneDR tends to detect more refactorable clones located in the same method and type, as well as in unrelated types.

5.4.3 Clone Type

In this section, we examine how the type of the clone pairs affects the refactorability. Table 10 shows the refactorability analysis results per clone type and clone detection tool, presented separately for a) all clone pairs, b) the clone pairs found in production code, and c) the clone pairs found in test code.

Type-2 is by far the most frequent clone type detected by all four clone detection tools. However, NiCad detects a considerably larger number of *Type-3* clones (with 24.6 percent) compared to CloneDR, CCFinder and Deckard (with 0.2, 2 and 4.1 percent, respectively). This could be perhaps explained by the fact that NiCad was configured with a comparatively higher difference threshold (0.2) that allows the matching of code fragments with gaps or extensive differences. In addition, NiCad has a considerably higher percentage of clone pairs in the *Unknown* type category (with 35.9 percent) compared to CCFinder, Deckard and CloneDR (with 0.1, 0.6 and 1.1 percent, respectively). The *Unknown* type refers to the cases where the clone type cannot be determined, because the number of statements mapped by our approach is equal to zero. Our approach cannot map statements having a different AST statement type (e.g., variable assignment against variable declaration statement, or `for` against `while` statement), or statements having a non-homomorphic AST structure. The large number of problematic clones, from the refactoring point of view, detected by NiCad can be attributed to the fact that NiCad treats code as sequences of pretty-printed lines, and based on the value of the difference threshold allows a number of lines to be textually different, which in combination with the *blind* renaming normalization applied by NiCad can lead to the detection of clones containing statements with incompatible AST types or structure. In fact, by simply changing the renaming option from *blind* to *consistent*, the numbers of *Type-3* and *Unknown* type clones detected by NiCad have been considerably reduced. The effect of the renaming option on NiCad results is extensively discussed in Section 5.4.6.

By analyzing the results for all clone pairs (top part of Table 10), we can observe that all tools are able to detect a very high percentage of refactorable *Type-1* clone pairs,

```
3 if (result == INVOKE_IMPLEMENTATION)
4   Object target = getImplementation();
5   try
6     final Object returnValue;
7     if (ReflectHelper.isPublic(persistentClass, method))
8       if (!method.getDeclaringClass().isInstance(target))
9         throw new ClassCastException(target.getClass().getName());
10      returnValue = method.invoke(target, args);
        else
11        if (!method.isAccessible())
12          method.setAccessible(true);
13        returnValue = method.invoke(target, args);
14    return returnValue == target ? proxy : returnValue;
    else
15    return result;
```

```
5 if (result == INVOKE_IMPLEMENTATION)
6   Object target = getImplementation();
7   final Object returnValue;
8   try

9     if (ReflectHelper.isPublic(persistentClass, thisMethod))
10      if (!thisMethod.getDeclaringClass().isInstance(target))
11        throw new ClassCastException(target.getClass().getName());
12      returnValue = thisMethod.invoke(target, args);
        else
13        if (!thisMethod.isAccessible())
14          thisMethod.setAccessible(true);
15        returnValue = thisMethod.invoke(target, args);
16    return returnValue == target ? proxy : returnValue;
    else
17    return result;
```

(a) Hibernate Production Code

```
1 boolean success = false;
2 try
3   DefaultBoxAndWhiskerCategoryDataset dataset = new
    DefaultBoxAndWhiskerCategoryDataset();
    dataset.add(new BoxAndWhiskerItem(new Double(1.0), new Double(2.0),
4     new Double(0.0), new Double(4.0), new Double(0.5), new Double(4.5),
      new Double(-0.5), new Double(5.5), null), "S1", "C1");
5   CategoryPlot plot = new CategoryPlot(dataset  new CategoryAxis(
      "Category"),new NumberAxis("Value"), new BoxAndWhiskerRenderer());

6   JFreeChart chart = new JFreeChart(plot);
7   chart.createBufferedImage(300, 200, null);
8   success = true;
9 assertTrue(success);
```

```
1 boolean success = false;
2 try
3   DefaultBoxAndWhiskerCategoryDataset dataset = new
    DefaultBoxAndWhiskerCategoryDataset();
    dataset.add(new BoxAndWhiskerItem(null, new Double(2.0),
4     new Double(0.0), new Double(4.0), new Double(0.5), new Double(4.5),
      new Double(-0.5), new Double(5.5), null), "S1", "C1");
5   CategoryPlot plot = new CategoryPlot(dataset, new CategoryAxis(
      "Category"), new NumberAxis("Value"), new BoxAndWhiskerRenderer());
6   ChartRenderingInfo info = new ChartRenderingInfo();
7   JFreeChart chart = new JFreeChart(plot);
8   chart.createBufferedImage(300, 200, info);
9   success = true;
10 assertTrue(success);
```

(b) JFreeChart Test Code

```
52 if (state.getInfo() != null)
53   EntityCollection entities = state.getEntityCollection();
54   if (entities != null)
55     String tip = null;

56     if (getToolTipGenerator(row, column) != null)
57       tip = getToolTipGenerator(row, column).generateToolTip(dataset,
           row, column);
58     String url = null;
59     if (getItemURLGenerator(row, column) != null)
60       url = getItemURLGenerator(row, column).generateURL(dataset, row,
         column);
61     CategoryItemEntity entity = new CategoryItemEntity(bar, tip, url,
         dataset, dataset.getRowKey(row), dataset.getColumnKey(column));
62     entities.add(entity);
```

```
58 if (state.getInfo() != null)
59   EntityCollection entities = state.getEntityCollection();
60   if (entities != null)
61     String tip = null;
62     CategoryToolTipGenerator tipster = getToolTipGenerator(row, column);
63     if (tipster != null)
64       tip = tipster.generateToolTip(dataset, row, column);
65     String url = null;
66     if (getItemURLGenerator(row, column) != null)
67       url = getItemURLGenerator(row, column).generateURL(dataset, row,
         column);
68     CategoryItemEntity entity = new CategoryItemEntity(bar, tip, url,
         dataset, dataset.getRowKey(row), dataset.getColumnKey(column));
69     entities.add(entity);
```

(c) JFreeChart Production Code

Fig. 13. Examples of Refactorable Type-3 Clones.

with NiCad having the highest percentage (85.6 percent) followed by CloneDR, Deckard and CCFinder (with 79.8, 79.2 and 69.3 percent, respectively). The reasons that prevent *Type-1* clones from being refactorable are:

1) The clones return more than one variable (49 percent)
2) The clones return variables with different types (30 percent)
3) The clones contain conditional return statements (16 percent)
4) The clones contain branching statements (e.g., `break`, `continue`) without the corresponding loop (5 percent)

CloneDR has the highest percentage of refactorable clone pairs in the *Type-2* category (with 24.5 percent), which constitutes the most dominant clone type category, as well as in the *Type-3* category (with 29.4 percent). We found that there is a considerable number of *Type-3* clones that can be refactored as *Type-1* or *Type-2* clones just by moving the unmapped statements before or after the mapped ones (i.e., the percentage of refactorable *Type-3* clone pairs in CCFinder is 15.2 percent, in Deckard is 14.8 percent, in CloneDR is 29.4 percent, and in NiCad is 6.6 percent).

Fig. 13 shows three examples of refactorable *Type-3* clones detected by Deckard. All three cases contain unmapped variable declaration statements, which are either nested in different levels (Fig. 13a), do not have a corresponding statement in the other clone fragment (Fig. 13b), or serve as temporary variables (Fig. 13c). The refactoring of these cases can be achieved by moving the unmapped statements before the common code shared between the clone fragments. The move of the unmapped statements preserves the behavior of the program, since it does not break existing data dependencies. Our approach also analyzes the bodies of all called methods inside the clones to find whether the state of instance variables (i.e., class fields), or the object references being present in a method call (i.e., arguments and invocation expression) is modified/accessed in subsequent methods of the call graph (inter-procedural data flow analysis). The move of the unmapped statements is feasible due to the stricter definition of control dependencies that we adopted in our approach, which we call "use-constrained" and "exception-constrained" control dependencies and are defined in Section 3.4.1.

For the example shown in Fig. 13c, somebody could argue that a better refactoring strategy is to first eliminate the unmapped statement by performing an Inline refactoring to remove the temporary variable `tipster`, or by introducing

a new temporary variable named `tipster` in the clone fragment having the gap, and then refactor the resulting *Type-1* clones. This alternative solution has the advantage of avoiding the move of statements, and leads perhaps to code with a lower computational cost, since method `getToolTipGenerator()` is executed only if the conditions of the preceding control predicate statements are true.

By analyzing separately the results for the clone pairs in production code (middle part of Table 10), we can see that CloneDR is able to detect a considerably higher percentage of refactorable clone pairs in the *Type-2* category (with 64.9 percent), while Deckard and CloneDR perform equally well in the *Type-3* category (with 23.5 and 22.1 percent, respectively). The picture is different when considering only the clone pairs in test code (bottom part of Table 10). CCFinder clearly outperforms the other tools in detecting refactorable *Type-2* clone pairs in test code.

RQ3 conclusion. *Type-1* clones tend to be more refactorable than *Type-2* and *Type-3* clones. There is a considerable number of *Type-3* clones that can be refactored as *Type-2* clones just by moving the unmapped statements before or after the mapped ones. AST-based clone detection tools (i.e., CloneDR and Deckard) tend to detect more *Type-2* and *Type-3* refactorable clones in production code.

5.4.4 Size

In this section, we examine whether the size of the clone fragments affects the refactorability. We consider that the size of a clone pair is equal to the average number of statements included in the two clone fragments. Fig. 14 shows the violin plots for the size of all clone pairs detected by each tool. As it is evident from the plots, NiCad and CloneDR are detecting smaller clone fragments (with an average size of 4.5 and 5.1, respectively) compared to CCFinder and Deckard (with an average size of 7.4 and 8, respectively). The median value (equal to 3.5 statements) for NiCad, indicates that the minimum size configuration option (set to five pretty-printed lines) actually returns clone fragments with four statements or even less (i.e., there is a large density of clone fragments ranging from three to four statements). CloneDR is the only tool that detects a considerable number of clone fragments having only one statement. This indicates that the minimum clone mass configuration option (set to six AST nodes) can actually return clone fragments with a single statement.

Fig. 15 shows the violin plots for the size of the refactorable versus the non-refactorable clone pairs separately for each clone detection tool. It is common among all tools that the refactorable clone pairs tend to have a smaller average size compared to the non-refactorable clone pairs. For each clone detection tool, we applied the one-tailed variant of the Mann-Whitney U test on the sizes of the refactorable and non-refactorable clone pairs with the following null hypothesis: "the number of statements in the refactorable clone pairs is larger than the number of statements in the non-refactorable clone pairs". In all four cases, the null hypothesis was rejected with significance at 95 percent confidence level (*p-value* $< 2.2 \times 10^{-16}$), and thus we can conclude that the number of statements in the refactorable clone pairs is smaller than the number of statements in the non-refactorable clone pairs.

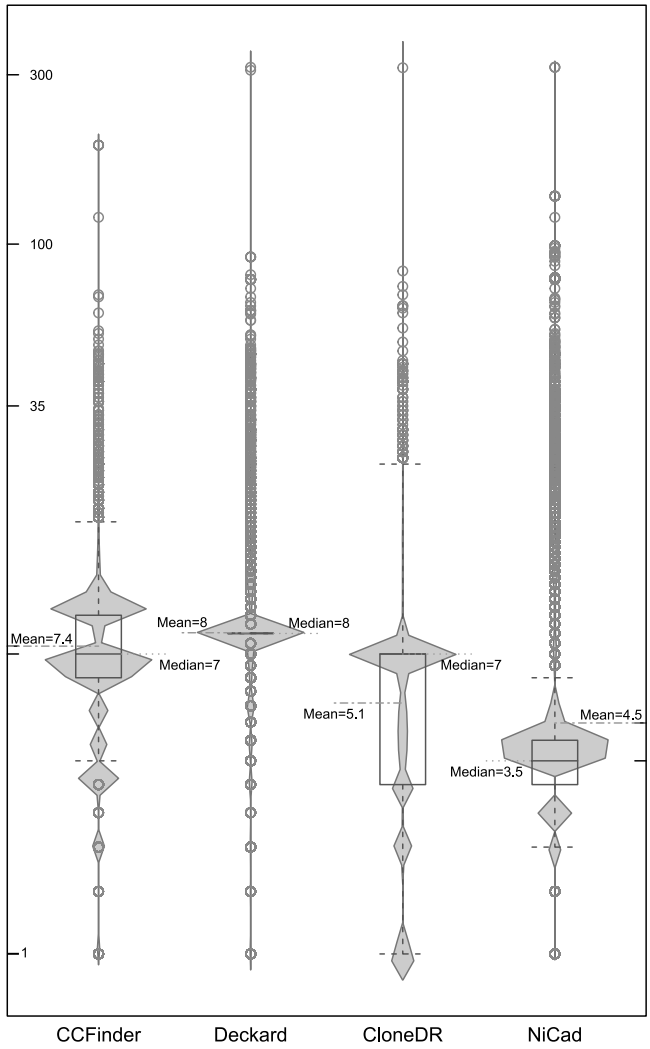


Fig. 14. Size of all detected clone pairs.

However, for NiCad there is a large density of relatively small clone fragments with 3-5 statements, which are not refactorable. This anomaly can be explained by the fact that NiCad detects a considerably larger number of *Type-3* clones compared to the other tools, even in relatively small duplicate code fragments. More specifically, 40 percent of the non-refactorable clone pairs detected by NiCad are *Type-3* clones, while the remaining 60 percent consists of *Type-2* clones. This 60-40 percentage ratio between *Type-2* and *Type-3* clones remains the same even for the subset of non-refactorable clones having 3-5 statements. On the other hand, the other three tools have a percentage of non-refactorable *Type-3* clones ranging from 0.3 to 4 percent.

Another important conclusion is that Deckard exhibits a more uniform distribution in the size of the refactorable clone pairs it detects (i.e., the size of the refactorable clone pairs takes a wide range of values with a median value equal to 6), while NiCad and especially CCFinder concentrate the majority of the refactorable clone pairs around a single and smaller size value (with a median value equal to 3). Finally, CloneDR concentrates the majority of the

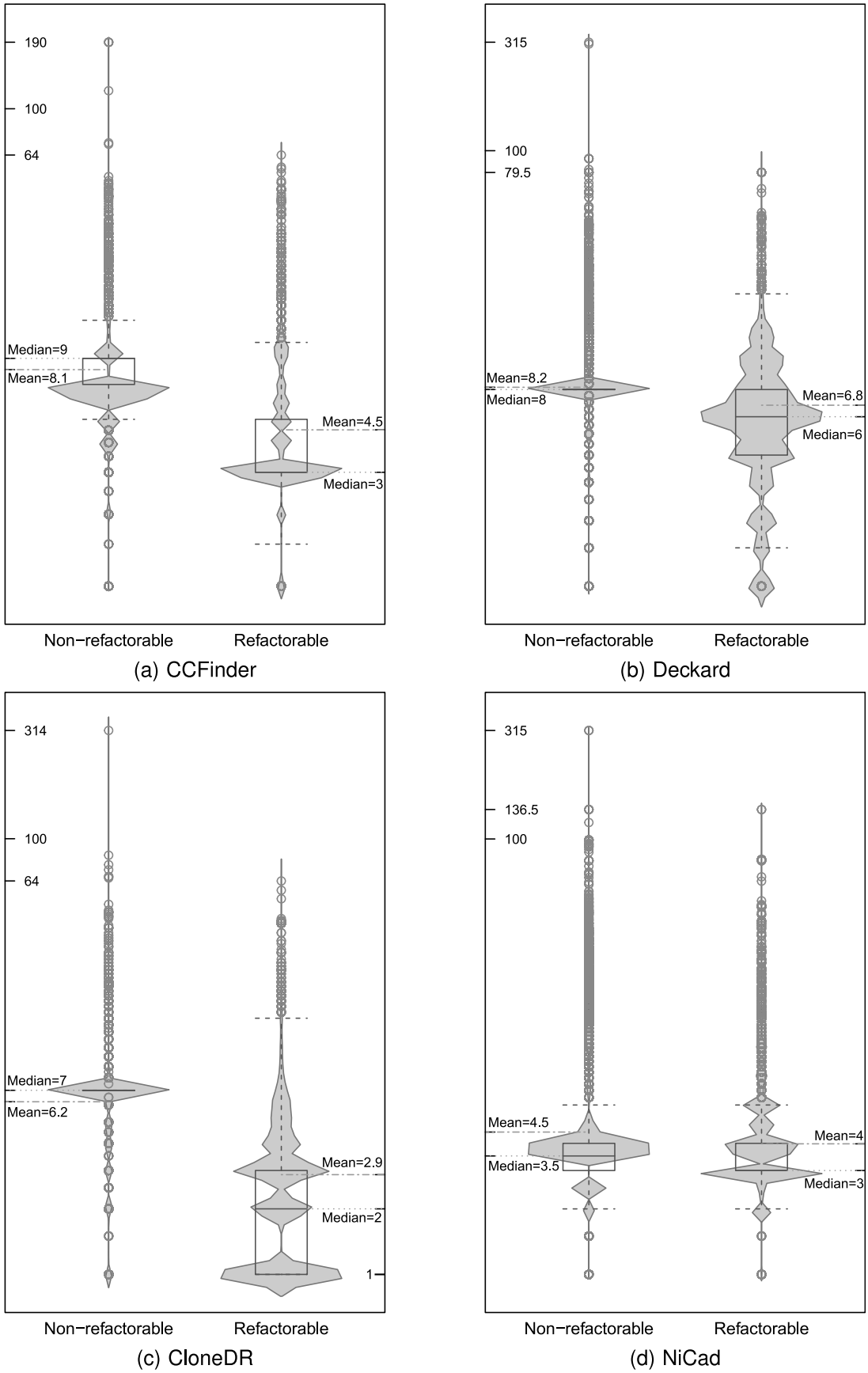


Fig. 15. Size of refactorable and non-refactorable clone pairs per clone detection tool.

TABLE 11
Results of the the Mann-Whitney U Test on the Sizes of the Refactorable Clones Detected by Each Tool

X \ Y	CloneDR	NiCad	CCFinder
Deckard	$>$	$>$	$>$
CCFinder	$>$	$>$	
Nicad	$>$		

* in all cases $p\text{-}value < 2.2 \times 10^{-16}$

refactorable clone pairs around three size values, namely 1, 2, and 3 statements (with a median value equal to 2).

For each pair of clone detection tools, we applied the one-tailed variant of the Mann-Whitney U test on the sizes of the refactorable clone pairs with the following null hypothesis: "the number of statements in the refactorable clone pairs detected by tool X is smaller than the number of statements in the refactorable clone pairs detected by tool Y". Table 11 shows all six pair combinations of clone detection tools that the statistical test was applied on. The rows of the table represent tool X, while the columns of the table represent tool Y in the statistical test. The symbol $>$ indicates that the null hypothesis was rejected with significance at 95 percent confidence level, and thus we can conclude that the number of statements in the refactorable clone pairs detected by tool X is larger than the number of statements in the refactorable clone pairs detected by tool Y. Based on the results of the statistical tests, we can conclude that Deckard tends to detect larger refactorable clones compared to the other three tools, and that all tools tend to detect larger refactorable clones when compared to CloneDR.

RQ4 conclusion. Clones with a small size tend to be more refactorable than clones with a larger size. Deckard tends to detect larger and more uniformly distributed (in terms of size) refactorable clones. CloneDR tends to detect smaller refactorable clones.

5.4.5 Precondition Violations

In this section, we examine the reasons that hinder the refactoring of software clones based on the number of precondition violation type instances collected from all clone pairs assessed as non-refactorable by our approach. Preconditions 1-4 refer to differences between mapped statements (Section 3.4.1), preconditions 5A-5C refer to unmapped statements (Section 3.4.2), and preconditions 6-8 refer to method extraction (Section 3.4.3).

For preconditions #2 and #5, we decided to give a more fine-grained view of the results by dividing them into subcategories. Precondition 2A includes the cases where variables in mapped statements have different types, and thus their types cannot be generalized. Precondition 2B is a special case of 2A that includes the cases where variables in mapped statements have different subclass types of a common superclass; however, there are methods called through these variables which are not declared in the common superclass, and thus the types of the variables cannot be generalized. Precondition 5A includes the cases of unmapped statements that cannot be moved before or after the common mapped statements due to existing dependencies. Precondition 5B includes the cases of unmapped branching statements (i.e., `break`, `continue`, `return`, and `throw` statements). Finally, precondition 5C includes the cases of unmapped statements inside `try` blocks that contain method calls throwing exceptions.

TABLE 12
Precondition Violations per Clone Detection Tool

Type	CCFinder Instances (%)	Deckard Instances (%)	CloneDR Instances (%)	NiCad Instances (%)
1	34727 (4.8)	28519 (6.6)	22011 (21.8)	94313 (5.8)
2A	236804 (32.8)	76910 (17.9)	53388 (52.8)	246744 (15.0)
2B	413 (0.1)	434 (0.1)	332 (0.3)	556 (0.0)
3	6816 (0.9)	4793 (1.1)	3105 (3.1)	89626 (5.5)
4	18542 (2.6)	52332 (12.2)	4922 (4.9)	694930 (42.4)
5A	150361 (20.8)	124310 (28.9)	1634 (1.6)	101891 (6.2)
5B	7026 (1.0)	3354 (0.8)	448 (0.4)	195170 (11.9)
5C	143190 (19.8)	121782 (28.3)	1159 (1.1)	71418 (4.4)
6	16339 (2.3)	8639 (2.0)	12054 (11.9)	23090 (1.4)
7	107493 (14.9)	8396 (2.0)	1800 (1.8)	121069 (7.4)
8	532 (0.1)	354 (0.1)	172 (0.2)	854 (0.1)
Total	722243 (100)	429823 (100)	101025 (100)	1639661 (100)

Table 12 shows the number of instances for each precondition violation type per clone detection tool. Among the preconditions related to *Type-2* differences (i.e., Preconditions 1-4), 2A is the most dominant precondition violation type. In particular, the percentage of violations belonging to type 2A is very high for CloneDR and CCFinder (with 52.8 and 32.8 percent, respectively), and relatively high for Deckard and NiCad (with 17.9 and 15 percent, respectively). Precondition violation type #4, which includes the cases where method calls in mapped statements are different and return a `void` type, is particularly dominant for NiCad (with 42.4 percent).

Regarding the precondition violations related to unmapped statements (i.e., preconditions 5A-5C), CloneDR is the least affected tool, which can be explained by the fact that it detects a very small number of *Type-3* clones. On the other hand, Deckard and CCFinder are greatly affected by precondition types 5A (with 28.9 and 20.8 percent, respectively) and 5C (with 28.3 and 19.8 percent, respectively), while NiCad is considerably affected by precondition type 5B (with 11.9 percent).

Finally, regarding the preconditions related to method extraction (i.e., Preconditions 6-8), CCFinder and NiCad are considerably affected by precondition type #7, which includes the cases where the clone fragments contain conditional `return` statements, while CloneDR is considerably affected by precondition type #6, which includes the cases where the clone fragments return more than one variable or variables having different types.

The dominance of precondition violation type 2A can be explained from the fact that all clone detection tools ignore the types of the variables when computing the textual or syntactical similarity of two code fragments. This leads to a large number of clones, which although they appear to have a strong textual and syntactical similarity, they cannot be unified, because they use variables having different types. These cases may constitute opportunities to generalize certain types by making them extend a common superclass or implement a common interface. Regarding the second most dominant reason hindering the refactoring of clones, i.e.,

the presence of unmapped statements that cannot be moved before or after the common code due to existing dependencies, more advanced solutions that do not require the move of the unmapped statements could be explored, such as the introduction of the Template Method design pattern or Lambda expressions (a feature added recently in Java 8).

RQ5 conclusion. The two most dominant reasons hindering the refactoring of clones are a) the presence of variables in mapped statements having different types, and b) the presence of unmapped statements that cannot be moved before or after the common code due to existing dependencies.

5.4.6 Threats to Validity

An important threat to the internal validity of the study is related to the configuration of the clone detection tools. First, it is very difficult to configure the tools with similar settings, because each one of them is using parameters of different nature. For instance, the minimum clone size can be either specified in lines of code (NiCad), tokens (CCFinder and Deckard), or AST nodes (CloneDR). Regarding the definition of similarity, each tool is using a different notion of similarity. Deckard computes the Euclidean distance of the *characteristic vectors* extracted from AST subtrees. NiCad uses a Longest Common Subsequence algorithm to compute the similarity of sequences of text lines. CloneDR computes the similarity between AST subtrees based on the number of shared and different AST nodes. Finally, CCFinder detects equivalent substrings in sequences of source code tokens. Regarding the normalization of identifiers, NiCad offers two options, namely *consistent* renaming, where the same identifiers are replaced with a single pseudo-variable (X_{index}) using an order-sensitive indexing scheme, and *blind* renaming, where all identifiers are replaced with a single pseudo-variable (X). CCFinder applies *parameter replacement*, where all identifiers related to types, variables, and constants are replaced with the same special token ($p). CloneDR uses a hash function that ignores the identifier names (i.e., the leaves in the AST). Finally, Deckard uses *characteristic vectors* that simply contain counts of the identifiers being declared, used, or modified inside an AST subtree. Since all tools perform some kind of *blind* renaming normalization, we configured NiCad with the *blind* renaming setting.

Second, it is definitely expected that different configuration options would lead to differences in the detected clones, and possibly to different conclusions. To overcome this threat we have used the default or recommended configuration options for each tool. These options have been consistently used in a large number of empirical studies and can be considered as standards, thus allowing for a direct comparison with the results of other studies. As an example showing the sensitivity of the configuration with respect to the obtained results, we executed NiCad on all projects after changing only the renaming option from *blind* to *consistent*, and run our refactorability analysis approach on the detected clone pairs. This configuration change decreased the number of the detected clone pairs by three times (741,610 with blind renaming versus 234,845 with consistent renaming), and almost doubled the percentage of the refactorable clone pairs (6.2 percent with blind renaming versus 10.8 percent with consistent renaming). In addition, as it can be observed from Table 13, the numbers of *Type-3* and *Unknown* type clones have been considerably reduced with consistent renaming; however, the percentages of the refactorable clone pairs remained almost the same in every clone type. Finding the tool configurations that optimize the percentage of refactorable clone pairs is out of the scope of this paper, although it is a very interesting direction for future research.

TABLE 13
The Effect of Blind and Consistent Renaming Option on NiCad Results

Clone type	NiCad-blind		NiCad-consistent	
	Pairs (%)	Ref. (%)	Pairs (%)	Ref. (%)
Type I	5705 (0.8)	4881 (85.6)	5771 (2.5)	4948 (85.7)
Type II	287010 (38.7)	29288 (10.2)	167163 (71.2)	17711 (10.6)
Type III	182751 (24.6)	12005 (6.6)	48726 (20.7)	2800 (5.7)
Unknown	266144 (35.9)	0 (0.0)	13185 (5.6)	0 (0.0)
Total	741,610	46174 (6.2)	234,845	25459 (10.8)

Another threat to the internal validity of the study is having an incomplete list of examined preconditions to assess whether a pair of clone fragments can be safely refactored. An incomplete list of preconditions would result in some clone pairs being erroneously assessed as refactorable, while they cannot be safely refactored in practice. To mitigate the risk of having an incomplete list of preconditions, we tested the proposed preconditions on 610 clone pairs found in project *JFreeChart* that were assessed as refactorable by our approach and were covered by unit tests (Section 5.2). All clone pairs were successfully refactored without causing any compile errors or test failures. To the best of our knowledge, this is the most extensive testing of preconditions performed in the related literature, and makes us confident that the list is complete. In this experiment, we used as a test suite the manually written test cases by the developers of *JFreeChart*. According to Fraser et al. [49], although automated test generation is effective in producing test suites with high code coverage compared to those constructed by humans, there is no measurable improvement in the number of bugs actually found by developers when writing tests with the aid of an automated unit test generation tool. Therefore, we believe that the use of automatically generated tests would not affect the results of our experiment (i.e., lead to test failures due to missing preconditions); however, it would possibly increase the number of clone pairs being covered by unit tests, and thus the number of clone refactorings being examined.

It should be noted though that the proposed preconditions have not been tested for concurrent Java programs making use of `synchronized` methods and statements. However, the preconditions examining the preservation of dependencies could be extended to include synchronization and communication dependencies between threads [50].

The external validity of the study is threatened by the inability to generalize our findings beyond the examined open-source projects and the selected clone detection tools. To face the first threat, we used nine open-source systems coming from different domains and having a diverse

development history (Table 3). Furthermore, we adopted the exact same systems used in the study conducted by Tairas and Gray [34] to avoid bias in the selection of subjects. To face the second threat, we selected a representative tool from each category of the three most dominant clone detection approaches (i.e., text-based, token-based, and tree-based). Finally, using a large number of clone pairs (over a million) as a sample allows to increase the significance level of the findings.

Regarding the reliability of the study, we have provided online[2] all the artifacts required to replicate the experiment, including the source code of our tools, the results of the clone detection tools used in the experiment, and the R scripts we developed to get the results of the experiment.

6 Related Work

We have organized the related work into three sections, namely clone refactoring techniques, clone ranking techniques, and program dependence graph matching techniques.

6.1 Clone Refactoring Techniques

Higo et al. [16] proposed an approach for merging software clones. In the first phase, they detect *refactoring-oriented* code clones within general clones detected by token-based or text-based clone detection tools. Essentially, a refactoring-oriented clone is a code fragment within a general clone that corresponds to a complete AST (i.e., it does not contain partial statements in terms of their AST structure). In the second phase, they compute some clone-related metrics to determine whether the refactoring-oriented clones extracted from the previous phase can be merged or not and how they will be merged (i.e., what refactoring should be applied). These metrics quantify properties such as the degree of coupling in terms of the number of referenced and assigned variables within the clones, and the dispersion of the clones in the class hierarchy. This work can be considered as a pioneer attempt to assess the refactorability of software clones. However, it is exposed to some serious limitations that are addressed by our approach.

1) It does not guarantee that the refactoring-oriented clones extracted from the first phase will have an identical nesting structure that enables their merging.
2) It supports only trivial differences between the clone fragments (e.g., different variable identifiers and literal values), which can be always parameterized. Therefore, it does not take into account any side effects that may result from the parameterization of more advanced differences (e.g., method calls).
3) It does not support the merging of clone fragments with unmatched statements (i.e., gaps).

Goto et al. [51] proposed an approach based on slice-based cohesion metrics for merging software clones. Their approach takes as input a pair of similar methods and first detects syntactic differences between them through AST differencing. Next, it finds pairs of code fragments within the methods that constitute valid Extract Method candidates by examining a set of preconditions (the same preconditions presented in Section 3.4.3). Finally, the detected candidates are ranked based on slice-based cohesion metrics [52]. The authors consider that highly cohesive candidates constitute better cases for Extract Method refactoring, because they are more likely to implement a single feature. Similarly to the work by Higo et al. [16], this work does not examine whether the parameterization of advanced differences between the clone fragments (e.g., method calls) could cause any side effects, and it does not handle unmatched statements that may exist within the clone fragments. Finally, their approach has been evaluated on only a single open-source project, namely Apache Ant.

Tairas and Gray [34] extended the Eclipse refactoring engine to enable the processing of more types of differences among duplicated code fragments, such as differences in field accesses, string literals, and method calls without arguments, in addition to differences in local variable identifiers. The evaluation on the *Type-2* clones detected in 9 open-source projects using the Deckard clone detection tool [37] revealed that the aforementioned enhancements in the matching of duplicated code increased the percentage of refactorable clones from 10.6 to 18.7 percent on average. In a recent work [14], we have shown that by providing a higher degree of freedom in the matching of expressions within the statements of the clone fragments (as described in Section 3.1), we were able to increase the percentage of refactorable clones to 36 percent on the same clone dataset used by Tairas and Gray [34]. As a result, restricting the parameterization of clone differences to specific types of AST expressions, reduces drastically the chances of finding clones that can be actually refactored.

Hotta et al. [53] proposed an approach to refactor *Type-3* clones by introducing an instance of the Template Method design pattern [33]. Their technique detects isomorphic subgraphs in the PDGs of two methods containing the clones. Next, the isomorphic subgraphs that constitute the *common process* of the examined methods are pulled up in a method of the base class. The remaining code fragments that do not belong to the detected isomorphic subgraphs (i.e., unmatched statements) constitute the *unique processes* and are extracted into methods within each derived class. For each pair of *unique processes* introduced in the derived classes an abstract method is created in the base class that is called from the *common process* at the point where the corresponding unmatched statements were found. This is an interesting alternative approach to handle unmatched statements inside the clone fragments (it should be reminded that in our approach we examine whether the unmatched statements can be moved before or after the call of the extracted method containing the matched statements). However, the introduction of a Template Method is applicable only under the following conditions:

1) the duplicated code fragments should belong to subclasses of the same superclass.
2) the unmatched statements from each clone fragment should be placed at exactly the same position in the nesting structures of the clone fragments.
3) the unmatched statements from each clone fragment should form a method returning at most one variable of the same type.

Obviously, meeting all these conditions is not always feasible, and therefore the introduction of Template Method design pattern is applicable in exceptional cases. Additionally, some of the open-source case studies presented in [53] can be actually refactored by supporting the parameterization of non-trivial differences (e.g., method calls), thus avoiding the Template Method solution that makes the design more difficult to understand due to the use of polymorphism.

Bian et al. [54] proposed SPAPE, a semantic-preserving amorphous procedure extraction technique, for the automatic refactoring of *Type-3* (near-miss) clones. SPAPE applies amorphous transformation on the PDGs of the clones in order to replicate `if` predicates and partition loop structures. In this way, it makes possible the merging of control structures having unmatched statements (i.e., gaps) within their bodies. Additionally, the unmatched statements from each clone fragment are placed inside the branches of `if/else if` conditionals in the AST of the extracted procedure, which are executed based on the value of a control variable (i.e., flag) passed as argument in the extracted procedure. Although this approach is able to handle the merging of clones having a significant number of unmatched statements or even a different nesting structure, it increases the complexity of the extracted code by introducing additional conditional logic and flag parameters.

6.2 Clone Ranking Techniques for the Purpose of Refactoring

Choi et al. [55] proposed a method combining clone metrics to extract code clones for the purpose of refactoring. Their approach takes as input a set of clones and computes the average length of token sequences within the clones, the size of the clone set, and the ratio of non-repeated token sequences within the clones. Then, the metric values are combined in order to rank the clone sets detected in a system. The authors analyzed an industrial software project and based on the feedback received by professional developers they concluded that the top ranked clone sets extracted by combining all the aforementioned clone metrics are more appropriate for refactoring compared to the top ranked clone sets extracted by using the same clone metrics individually.

Zibran and Roy [5], [56] proposed a model for automatically estimating the effort required to understand the source code context of a clone group, as well as the effort required to refactor a clone group. Additionally, they used the QMOOD model to estimate the effect of the refactoring of a clone group on software quality. Finally, the proposed a constraint programming approach to compute an optimal refactoring schedule of clone groups that maximizes code quality and minimizes effort, taking into account any conflicts and dependencies that might exist among the available clone group refactoring opportunities.

Mondal et al. [6] defined a particular clone change pattern, namely the Similarity Preserving Change Pattern (SPCP), and supported that clone fragments that change following this pattern are better candidates for refactoring. The SPCP essentially captures clones that are frequently and consistently modified during the evolution of a software system. In order to detect clones following the SPCP pattern, Mondal et al. extract association rules in the form $CF_1 \Rightarrow CF_2$, where CF_1 and CF_2 represent clone fragments from the same clone group. The support for each rule is the number of similarity preserving co-changes that occurred in the respective clone fragments during the history of the project. The underlying assumption is that the higher the support value of an association rule, the higher the likelihood that the involved clone fragments have changed consistently during the project evolution.

In a follow-up work, Mondal et al. [57] investigated the evolutionary coupling relationships of clone fragments following the SPCP pattern. Their idea is that an SPCP clone fragment that has change coupling relationships with code beyond the boundaries of the clone group it belongs to, should not be considered for removal through refactoring, because its removal might cause change ripple effects. Therefore, they consider that SPCP clone fragments that co-change with code beyond the boundaries of their clone group should be important candidates for tracking, while those that co-change only with code within the boundaries of their clone group should be candidates for refactoring. By analyzing the evolution history of six open-source systems developed in C and Java, respectively, they found that overall 13.20 percent of all clones in a software system are important candidates for refactoring, and overall 10.27 percent of all clones are important candidates for tracking.

Wang and Godfrey [58] constructed a decision tree-based classifier to recommend clones for refactoring. They analyzed 323 refactored and 323 non-refactored clone instances found in the history of three open-source projects, and collected15 features that are associated with the source code, the context, and the history of clones. Within-project testing, using 10-fold validation to test the classifier, resulted a precision ranging from 78.3 to 87.9 percent for the refactored group, and from 77.3 to 90.8 percent for the non-refactored group. Cross-project testing resulted also a good precision ranging from 73.2 to 88.5 percent. Further analysis showed that the subset of features related to cloned code snippet information has the closest performance when compared with the classifier that is trained with the entire set of features. Adding the features related to clone context and cloning relation information can further improve the performance of the classifier.

Our work complements existing clone ranking/prioritization techniques, since it comes into play after the ranking/prioritization phase to help the developers in refactoring the clones considered harmful in three different ways:

1) It analyzes the differences between the clones and displays the reasons that the clones might not be refactorable (i.e., precondition violations) in a comprehensive visualization.
2) In some cases, it provides suggestions of changes that the developers could do to make the clones refactorable (e.g., inlining methods called within the clones, generalizing different types used within the clones to extend a common superclass).
3) It automates the application of the refactoring, when there are no precondition violations detected. It supports the refactoring of clones located in methods of

the same class, different classes in the same inheritance hierarchy, and unrelated classes.

Alternatively, our tool can be executed in batch mode on the results of clone detection tools to filter out the clones that can be directly refactored (i.e., have no precondition violations), or that can be refactored after some changes (i.e., have specific precondition violations that can be easily overcome following the suggestions of our tool). We strongly believe that clone ranking techniques could take into account the *clone refactorability* aspect, in order to produce rankings of clones that can be actually refactored without any side effects by filtering out the non-refactorable cases.

6.3 PDG Matching Techniques

Komondoor and Horwitz [22] applied slicing techniques on PDGs to find isomorphic subgraphs that represent code clones. The advantage of this approach is the detection of non-contiguous clones (i.e., clones with gaps), clones with re-ordered statements, and clones intertwined with each other. In their approach, each pair of matching nodes is used as a starting point for backward slicing. In addition, whenever a pair of matching control predicate nodes is visited, forward slicing is applied to visit the nodes nested under the control predicate nodes. Two PDG nodes are matched if the corresponding statements are syntactically identical (i.e., their AST representation has the same structure) allowing only for differences in variable names and literal values.

Krinke [59] proposed an approach to identify code clones by finding the maximal similar subgraphs in two PDGs by induction starting from a pair of vertices. To reduce the complexity of the algorithm, he considers only a subset of vertices (i.e., predicate vertices) as starting points, and restricts the maximum length of the explored paths using a k-limit. A limitation of the approach is that the running time of the algorithm explodes as k-limit increases. Another limitation is that the use of k-limit may lead to an incomplete solution (i.e., the selected k-limit may be insufficient for detecting all possible matching vertices).

Shepherd et al. [60] implemented an automated aspect mining technique exploiting the PDG and AST representations of a program. The proposed algorithm, inspired by [59] and [22], starts by matching the control dependence subgraphs of two compared PDGs to extract all possible matching solutions. Next, it filters out the undesirable matching solutions based on data dependence information. A limitation is that the algorithm always starts from the method entry nodes, and thus will fail to match control dependence subgraphs nested in different levels.

Liu et al. [61] developed a software plagiarism detection tool called GPLAG. They support that the PDG structures of the original and the plagiarized code remain invariant and exploit this property 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. Their approach is able to detect five kinds of plagiarism disguises, such as format alteration, identifier renaming, statement reordering, control replacement and code insertion. To increase the efficiency of the algorithm, they prune the search space (i.e., reduce the number of PDG pairs to be checked) by applying some filters. For instance, the *lossy* filter generates a vertex histogram as a summarized representation of each examined PDG. A PDG pair is excluded from isomorphism testing, if the corresponding vertex histograms have a dissimilar distribution.

Higo and Kusumoto [62] improved Komondoor's technique [22] by extending the PDG representation and introducing some heuristics to enhance code clone detection. The PDG representation was extended by introducing new edges called *execution-next links* to improve the ability to detect contiguous code, and by merging the directly-connected equivalence nodes to reduce the computational cost of identifying isomorphic subgraphs. The proposed heuristics include the application of two-way slicing (both backward and forward slicing), removing unnecessary slice points, and neglecting small methods.

Speicher and Bremm [63] proposed a theoretical stepwise unification process of *Type-3* clones based on PDG mapping. In the first step, they suggest potential refactoring operations to make statements with differences unifiable. For instance, the Rename refactoring can be used to consistently change the names of local variables and parameters and eliminate identifier differences between the clone fragments. The Generalize Types refactoring [28] allows to unify statements that are using object references instantiated from different subclass types. In the second step, they suggest alternative ways to unify the control flow of the clone fragments. For instance, statement reordering could be used to separate non-unifiable from unifiable statements and form a group of contiguous statements that can be extracted to a new method. Finally, they also suggest that differences in expression operators can be parameterized using Lambda expressions (a feature introduced recently in Java 8).

Krishnan and Tsantalis [14] expressed the problem of finding isomorphic subgraphs within the PDGs of software clones as an optimization problem with two objectives, namely maximizing the number of matched statements and at the same time minimizing the number of differences between the matched statements. They used a maximum common subgraph algorithm to find a solution to this problem. To deal with the problem of combinatorial explosion, they applied the MCS algorithm in a divide-and-conquer fashion by breaking the initial matching problem to smaller sub-problems based on the control dependence structure of the software clones.

7 Conclusion

This work introduces *refactorability analysis* as an important (previously unsupported) feature in clone management to help the developers in assessing whether a pair of clone fragments can be safely refactored. To achieve this goal, we developed a technique that first matches the statements of the clones in a way the minimizes the number of differences between them, and then examines these differences against a set of preconditions to determine whether they can be parameterized without changing the program behavior. If no precondition violations are found, we provide tool support for the automatic refactoring of the clones.

To evaluate the correctness of the proposed technique, we refactored 610 clone pairs covered by tests and assessed as refactorable by our approach. All refactorings were successfully performed without causing any compile errors or

test failures. Additionally, we conducted a performance analysis showing that all the steps of the proposed technique can be executed in less than a second for the vast majority (99.85 percent) of the analyzed clone pairs. Therefore, we can conclude that the computational cost introduced by our approach is negligible, and refactorability analysis can be incorporated in clone management practices with a limited cost.

Using our refactorability analysis technique, we conducted a large-scale empirical study involving over a million clone pairs detected by four different clone detection tools, namely CCFinder, Deckard, CloneDR, and NiCad, in nine open-source projects. In this study, we investigated how refactorability is affected by various clone properties, such as the nature of the source code (production versus test code) that the clones were found in, the relative location of the clones in the examined projects, the types of the clones, and the size of the clones. The highlights of our findings are:

1) Clones in production code tend to be more refactorable than clones in test code.
2) Clones with a close relative location (i.e., same method, type, or file) tend to be more refactorable than clones in distant locations (i.e., same hierarchy, or unrelated types).
3) *Type-1* clones tend to be more refactorable than the other clone types. However, there is also a considerable number of *Type-3* clones that can be refactored as *Type-2* clones just by moving the unmapped statements before or after the mapped ones.
4) Clones with a small size tend to be more refactorable than clones with a larger size.

Regarding the clone detection tools used in the study our findings are:

1) AST-based clone detection tools (i.e., CloneDR and Deckard) tend to detect more refactorable clones in production code.
2) CCFinder (token-based detector) tends to detect more refactorable clones in test code.
3) CloneDR tends to detect more refactorable clones located in the same method and type, as well as in unrelated types.
4) AST-based clone detection tools (i.e., CloneDR and Deckard) tend to detect more *Type-2* and *Type-3* refactorable clones in production code.
5) Deckard tends to detect larger and more uniformly distributed (in terms of size) refactorable clones.
6) CloneDR tends to detect smaller refactorable clones.

Finally, our findings with respect to the configuration of the clone detection tools used in the study are:

1) For NiCad (text-based detector), the *consistent* renaming option is more suitable for detecting clones meant for refactoring.
2) For CloneDR, the default *minimum clone mass* setting (set to six AST nodes) results in a considerable number of clone fragments having a single statement.

As future work, we believe that the detected precondition violations could serve as starting points for the application of more advanced clone unification and refactoring scenarios. For instance, the precondition violations related to unmapped statements or differences in method calls with a `void` return type could be possibly surpassed by applying the Template Method design pattern or by introducing Lambda expressions. Our empirical results show that the aforementioned precondition violations are frequent, and thus research towards this direction could drastically increase the number of refactorable clones. Additionally, although we provide some basic suggestions to overcome precondition violations by making changes facilitating the successful unification of the clones, we believe that developers could be assisted with a more thorough and advanced automated guidance driven by the detected precondition violations. Another interesting research direction is to apply search-based techniques in order to find tool configurations that maximize the refactorability of the detected clones (i.e., configurations that can lead to a higher percentage of refactorable clones).

Acknowledgments

The authors would like to thank the Natural Sciences and Engineering Council of Canada (NSERC) and the Faculty of Engineering and Computer Science at Concordia University for their generous support. Also, they would like to thank Raphael Stein (funded with an Undergraduate Student Research Award from NSERC) for his contribution in the graphical user interface of the tool presented in this paper. Nikolaos Tsantalis has been also partially funded by the European Union (European Social Fund—ESF) and Greek national funds through the Operational Program "Education and Lifelong Learning" of the National Strategic Reference Framework (NSRF)—Research Funding Program: Thalis—Athens University of Economics and Business—Software Engineering Research Platform.

References

[1] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Sci. Comput. Programm.*, vol. 74, no. 7, pp. 470–495, 2009.

[2] C. Roy, M. Zibran, and R. Koschke, "The vision of software clone management: Past, present, and future (keynote paper)," in *Proc. IEEE Conf. Softw. Maintenance, Reeng. Reverse Eng., Softw. Evol. Week*, 2014, pp. 18–33.

[3] H. A. Nguyen, T. T. Nguyen, N. H. Pham, J. Al-Kofahi, and T. N. Nguyen, "Clone management for evolving software," *IEEE Trans. Softw. Eng.*, vol. 38, no. 5, pp. 1008–1026, Sep.-Oct. 2012.

[4] N. Göde and R. Koschke, "Studying clone evolution using incremental clone detection," *J. Softw.: Evol. Process*, vol. 25, no. 2, pp. 165–192, 2013.

[5] M. Zibran and C. Roy, "A constraint programming approach to conflict-aware optimal scheduling of prioritized code clone refactoring," in *Proc. 11th IEEE Int. Working Conf. Source Code Anal. Manipulation*, 2011, pp. 105–114.

[6] M. Mondal, C. Roy, and K. Schneider, "Automatic ranking of clones for refactoring through mining association rules," in *Proc. IEEE Conf. Softw. Maintenance, Reeng. Reverse Eng. Softw. Evol. Week*, 2014, pp. 114–123.

[7] A. Lozano and M. Wermelinger, "Assessing the effect of clones on changeability," in *Proc. 24th IEEE Int. Conf. Softw. Maintenance*, 2008, pp. 227–236.

[8] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner, "Do code clones matter?" in *Proc. 31st Int. Conf. Softw. Eng.*, 2009, pp. 485–495.

[9] N. Bettenburg, W. Shang, W. M. Ibrahim, B. Adams, Y. Zou, and A. E. Hassan, "An empirical study on inconsistent changes to code clones at the release level," *Sci. Comput. Programm.*, vol. 77, no. 6, pp. 760–776, 2012.

[10] G. Selim, L. Barbour, W. Shang, B. Adams, A. Hassan, and Y. Zou, "Studying the impact of clones on software defects," in *Proc. 17th Working Conf. Reverse Eng.*, 2010, pp. 13–21.

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

[12] M. Mondal, C. K. Roy, and K. A. Schneider, "An insight into the dispersion of changes in cloned and non-cloned code: A genealogy based empirical study," *Sci. Comput. Programm.*, vol. 95, pp. 445–468, 2014.

[13] J. van Leeuwen, Ed., *Handbook of Theoretical Computer Science (Vol. A): Algorithms and Complexity*. Cambridge, MA, USA: MIT Press, 1994.

[14] G. P. Krishnan and N. Tsantalis, "Unification and refactoring of clones," in *Proc. IEEE Conf. Softw. Maintenance, Reeng. Reverse Eng. Softw. Evol. Week*, 2014, pp. 104–113.

[15] R. Johnson, D. Pearson, and K. Pingali, "The program structure tree: Computing control regions in linear time," in *Proc. ACM SIGPLAN Conf. Programm. Lang. Des. Implementation*, 1994, pp. 171–185.

[16] Y. Higo, S. Kusumoto, and K. Inoue, "A metric-based approach to identifying refactoring opportunities for merging code clones in a Java software system," *J. Softw. Maintenance Evol.: Res. Practice*, vol. 20, no. 6, pp. 435–461, 2008.

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

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

[19] P. Hell and J. Nesetril, *Graphs and Homomorphisms* (series Oxford Lecture Series in Mathematics and Its Applications). London, U.K.: Oxford Univ. Press, 2004.

[20] G. Valiente, *Algorithms on Trees and Graphs*. New York, NY, USA: Springer-Verlag, 2002.

[21] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. Cambridge, MA, USA: MIT Press, 2009.

[22] R. Komondoor and S. Horwitz, "Using slicing to identify duplication in source code," in *Proc. 8th Int. Symp. Static Anal.*, 2001, pp. 40–56.

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

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

[25] P. J. Durand, R. Pasari, J. W. Baker, and C.-C. Tsai, "An efficient algorithm for similarity analysis of molecules," *Internet J. Chemistry*, vol. 2, article 17, pp. 1–12, 1999.

[26] E. Balas and C. S. Yu, "Finding a maximum clique in an arbitrary graph," *SIAM J. Comput.*, vol. 15, no. 4, pp. 1054–1068, Nov. 1986.

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

[28] F. Tip, A. Kiezun, and D. Bäumer, "Refactoring for generalization using type constraints," in *Proc. 18th Annu. ACM SIGPLAN Conf. Object-Oriented Programm, Syst., Lang. Appl.*, 2003, pp. 13–26.

[29] E. Murphy-Hill and A. P. Black, "Breaking the barriers to successful refactoring: Observations and tools for extract method," in *Proc. 30th Int. Conf. Softw. Eng.*, 2008, pp. 421–430.

[30] R. Koschke, R. Falke, and P. Frenzel, "Clone detection using abstract syntax suffix trees," in *Proc. 13th Working Conf. Reverse Eng.*, 2006, pp. 253–262.

[31] Y. Lin, Z. Xing, Y. Xue, Y. Liu, X. Peng, J. Sun, and W. Zhao, "Detecting differences across multiple instances of code clones," in *Proc. 36th Int. Conf. Softw. Eng.*, 2014, pp. 164–174.

[32] Z. Xing, Y. Xue, and S. Jarzabek, "CloneDifferentiator: Analyzing clones by differentiation," in *Proc. 26th IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2011, pp. 576–579.

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

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

[35] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: A multilinguistic token-based code clone detection system for large scale source code," *IEEE Trans. Software Eng.*, vol. 28, no. 7, pp. 654–670, Jul. 2002.

[36] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and evaluation of clone detection tools," *IEEE Trans. Softw. Eng.*, vol. 33, no. 9, pp. 577–591, Sep. 2007.

[37] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "DECKARD: Scalable and accurate tree-based detection of code clones," in *Proc. 29th Int. Conf. Softw. Eng.*, 2007, pp. 96–105.

[38] I. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," in *Proc. Int. Conf. Softw. Maintenance*, 1998, pp. 368–377.

[39] C. K. Roy and J. R. Cordy, "NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization," in *Proc. 16th IEEE Int. Conf. Program Comprehension*, 2008, pp. 172–181.

[40] C. K. Roy and J. R. Cordy, "A mutation/injection-based automatic framework for evaluating code clone detection tools," in *Proc. IEEE Int. Conf. Softw. Testing, Verification, Validation Workshops*, 2009, pp. 157–166.

[41] F. Deissenboeck, B. Hummel, and E. Juergens, "Code clone detection in practice," in *Proc. 32nd ACM/IEEE Int. Conf. Softw. Eng.-Volume 2*, 2010, pp. 499–500.

[42] N. Göde, *Clone Evolution*. Berlin, Germany, Logos Verlag Berlin GmbH, 2011.

[43] B. Daniel, D. Dig, K. Garcia, and D. Marinov, "Automated testing of refactoring engines," in *Proc. 6th Joint Meeting Eur. Softw. Eng. Conf. ACM SIGSOFT Symp. Found. Softw. Eng.*, 2007, pp. 185–194.

[44] G. Soares, R. Gheyi, and T. Massoni, "Automated behavioral testing of refactoring engines," *IEEE Trans. Softw. Eng.*, vol. 39, no. 2, pp. 147–162, Feb. 2013.

[45] G. Soares, R. Gheyi, D. Serey, and T. Massoni, "Making program refactoring safer," *IEEE Softw.*, vol. 27, no. 4, pp. 52–57, Jul./Aug. 2010.

[46] J. L. Hintze and R. D. Nelson, "Violin plots: A box plot-density trace synergism," *The Am. Statist.*, vol. 52, no. 2, pp. 181–184, 1998.

[47] N. Tsantalis, V. Guana, E. Stroulia, and A. Hindle, "A multidimensional empirical study on refactoring activity," in *Proc. Conf. Center Adv. Studies Collaborative Res.*, 2013, pp. 132–146.

[48] L. Barbour, F. Khomh, and Y. Zou, "Late propagation in software clones," in *Proc. 27th IEEE Int. Conf. Softw. Maintenance*, 2011, pp. 273–282.

[49] G. Fraser, M. Staats, P. McMinn, A. Arcuri, and F. Padberg, "Does automated unit test generation really help software testers? A controlled empirical study," *ACM Trans. Softw. Eng. Methodology*, To Appear.

[50] J. Zhao, "Multithreaded dependence graphs for concurrent Java programs," in *Proc. Int. Symp. Softw. Eng. Parallel Distrib. Syst.*, 1999, pp. 13–23.

[51] A. Goto, N. Yoshida, M. Ioka, E. Choi, and K. Inoue, "How to extract differences from similar programs? A cohesion metric approach," in *Proc. 7th Int. Workshop. Softw. Clones*, 2013, pp. 23–29.

[52] L. M. Ott and J. J. Thuss, "Slice based metrics for estimating cohesion," in *Proc. 1st Int. Softw. Metrics Symp.*, 1993, pp. 71–81.

[53] K. Hotta, Y. Higo, and S. Kusumoto, "Identifying, tailoring, and suggesting form template method refactoring opportunities with program dependence graph," in *Proc. 16th Eur. Conf. Softw. Maintenance Reeng.*, 2012, pp. 53–62.

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

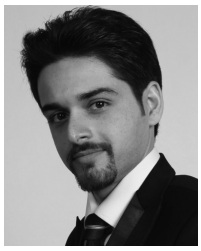
[55] E. Choi, N. Yoshida, T. Ishio, K. Inoue, and T. Sano, "Extracting code clones for refactoring using combinations of clone metrics," in *Proc. 5th Int. Workshop Softw. Clones*, 2011, pp. 7–13.

[56] M. F. Zibran and C. K. Roy, "Conflict-aware optimal scheduling of prioritised code clone refactoring," *IET Softw.*, vol. 7, no. 3, pp. 167–186, 2013.

[57] M. Mondal, C. K. Roy, and K. A. Schneider, "Automatic identification of important clones for refactoring and tracking," in *Proc. 14th IEEE Int. Working Conf. Source Code Anal. Manipulation*, 2014, pp. 11–20.

[58] W. Wang and M. W. Godfrey, "Recommending clones for refactoring using design, context, and history," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, 2014, pp. 331–340.
[59] J. Krinke, "Identifying similar code with program dependence graphs," in *Proc. 8th Working Conf. Reverse Eng.*, 2001, pp. 301–307.
[60] D. Shepherd, E. Gibson, and L. L. Pollock, "Design and evaluation of an automated aspect mining tool," in *Proc. Int. Conf. Softw. Eng. Res. Practice*, 2004, pp. 601–607.
[61] C. Liu, C. Chen, J. Han, and P. S. Yu, "GPLAG: Detection of software plagiarism by program dependence graph analysis," in *Proc. 12th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2006, pp. 872–881.
[62] Y. Higo and S. Kusumoto, "Code clone detection on specialized PDGs with heuristics," in *Proc. 15th Eur. Conf. Softw. Maintenance Reeng.*, 2011, pp. 75–84.
[63] D. Speicher and A. Bremm, "Clone removal in Java programs as a process of stepwise unification," in *Proc. 26th Workshop Logic Programm.*, 2012, pp. 75–80.

Nikolaos Tsantalis received the PhD degree in computer science from the University of Macedonia, Greece, in 2010. He is an assistant professor at the Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, and holds a Concordia University Research Chair (New Scholar) in Web Software Technologies. From January 2011 until May 2012, he was a postdoctoral fellow at the Department of Computing Science, University of Alberta, Edmonton, Canada. His research interests include software maintenance, empirical software engineering, refactoring recommendation systems, and software quality assurance. He serves and has served as a program committee member of international conferences in the field of software engineering, such as ICSE, ICSME, SANER, ICPC, and SCAM. He is a member of the IEEE and ACM.



Davood Mazinanian received the BS degree (Suma Cum Laude) in information technology engineering from the Shahrood University of Technology, Iran, in 2009, and the MS degree in information technology engineering with specialization in e-commerce from Shiraz University, Iran, in 2012. He is currently working towards the PhD degree in computer science at the Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada. His research interests include source code analysis and refactoring (focusing on client-side web code), and empirical software engineering.

Giri Panamoottil Krishnan received the BS degree in computer science and engineering from the University of Kerala, India, in 2008, and the MS degree in computer science from Concordia University, Canada, in 2014. Soon after his bachelor studies, he joined Infosys, India, where he worked as a software engineer for three years. He is currently working as a technology associate for Morgan Stanley Financial Services. In 2015, he was the recipient of the F.A. Gerard Prize awarded annually to the most deserving graduate of the master in applied science and computer science programs (thesis) of the Faculty of Engineering and Computer Science, Concordia University.