# Scalable Detection of Missed Cross-Function Refactorings


Narcisa Andreea Milea
School of Computing
National University of Singapore, Singapore
mileanar@comp.nus.edu.sg

Lingxiao Jiang
School of Information Systems
Singapore Management University, Singapore
lxjiang@smu.edu.sg

Siau-Cheng Khoo
School of Computing
National University of Singapore, Singapore
khoosc@nus.edu.sg



## ABSTRACT

Refactoring is an important way to improve the design of existing code. Identifying refactoring opportunities (i.e., code fragments that can be refactored) in large code bases is a challenging task. In this paper, we propose a novel, automated and scalable technique for identifying cross-function refactoring opportunities that span more than one function (e.g., Extract Method and Inline Method). The key of our technique is the design of efficient *vector inlining* operations that emulate the effect of method inlining among code fragments, so that the problem of identifying cross-function refactoring can be reduced to the problem of finding similar vectors before and after inlining. We have implemented our technique in a prototype tool named ReDex which encodes Java programs to particular vectors. We have applied the tool to a large code base, 4.5 million lines of code, comprising of 200 bundle projects in the Eclipse ecosystem (e.g., Eclipse JDT, Eclipse PDE, Apache Commons, Hamcrest, etc.). Also, different from many other studies on detecting refactoring, ReDex only searches for code fragments that can be, but not yet, refactored in a way similar to some refactoring that has happened in the code base. Our results show that ReDex can find 277 cross-function refactoring opportunities in 2 minutes, and 223 cases were labelled as true opportunities by users, and cover many categories of cross-function refactoring operations in classical refactoring books, such as Self Encapsulate Field, Decompose Conditional Expression, Hide Delegate, Preserve Whole Object, etc.



## Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and reengineering*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Program analysis*

## General Terms

Algorithms, Design and Experimentation, Reliability

## Keywords

Refactoring, Software Evolution, Vector-based representation




## 1. INTRODUCTION

Refactoring has long been recognized as an important way to improve the design of existing code [9]. Many modern development environments, such as Eclipse, Microsoft Visual Studio, have built-in support for various kinds of refactoring operations, such as Rename Variables, Encapsulate Field, Change Method Signature, Move Method, Extract Method, and Extract Interface [11].[1] These tools usually require *refactoring opportunities* (i.e., the parts of the code that can be, but not yet, refactored) to be identified first.

Identifying refactoring opportunities can be a challenging task for large code bases. *Scalability* of a refactoring detection technique is important to large-scale application. The detection problem is also compounded by *cross-function* refactoring opportunities that may involve moving code fragments across function boundaries across different software projects. Naively searching through all possible combinations of different code fragments is not likely scalable. As an example for illustration, we show in Figure 1 code snippets from two Eclipse projects.

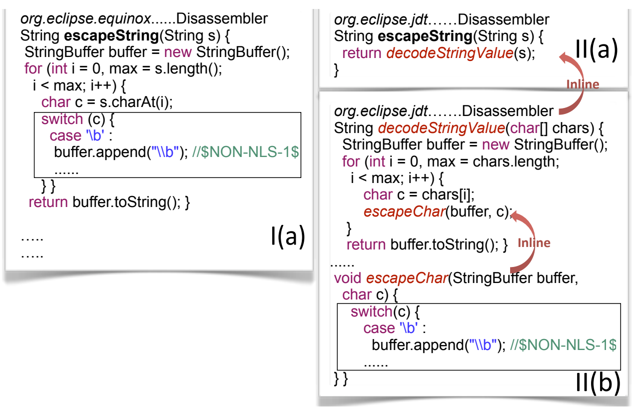



**Figure 1: Sample refactoring detected in Eclipse**

The snippet in I(a) is from a class named `Disassembler` in the `Equinox` project while the snippets in II ((a) and (b)) are from a class with the same name but in the different `JDT` project. While both `escapeString` methods have the same functionality, they are structurally different. The one in II(a) only contains a method call, and the callee `decodeStringValue` contains a call to another method `escapeChar` in II(b). The change history of Eclipse shows that the methods in II(a) and II(b) were refactored from an earlier version of the code, which was the same as the code in I(a) before Eclipse 3.5.2,

[1] This paper uses "method" and "function" interchangeably.

and the refactored `escapeChar` became used in a number of locations in the code. By simply looking at I(a) itself it may not be clear whether it is a refactoring opportunity. However, by looking at how the code in II is structured, a developer can easily see a missed cross-function refactoring opportunity for the method `escapeString` in I(a) as well.

The example also indicates that not all refactoring opportunities would be performed by developers at the same time. Refactoring *parts of a large code base* of related programs often causes initially similar code fragments in different projects to diverge. For code bases that have long evolution histories, such divergences can in time cause difficulties in finding those missed refactoring opportunities again. Usual refactoring detection based on clone detection (e.g., [2, 8, 18, 42, 45]) would not report two code fragments, one of which is a refactored copy of the other, as clones, and thus misses many partial refactoring opportunities. As for this example, the `switch` statement in I(a) may be detected as a clone of the body of `escapeChar` in II(b), but the `for` loops may not be detected, as the loop in II is in a separate method `decodeStringValue`. Thus, usual clone detection may fail to suggest a refactoring for code in I. Detecting such partially missed cross-function refactoring opportunities scalably is the goal of this paper.

In this paper, we provide a new scalable approach for identifying cross-function refactoring opportunities that may involve method extraction and/or inlining. The key of our technique is the design of efficient *vector inlining* operations that emulate the effect of method inlining, based on characteristic vector representations of code. Then, such inlined vectors naturally represent inlined code, taking method extraction and inlining into account. Thus, the problem of scalable identification of cross-function refactoring can be reduced to the scalable technique of identifying similar vectors.

Also, we take the intuition that *if two pieces of code become similar to each other, either syntactically or semantically, after the methods called in them are inlined, yet they were not similar before inlining, they are very likely to indicate a true cross-function refactoring opportunity.* This means that the two code pieces have structural differences involving method extraction and/or inlining, and implies that the refactoring operation applied to one of the code pieces, if any, may be applied to the other as well. Thus, our technique uses a special *vector query and filtering* strategy that first identifies pairs of similar vectors and then filters those pairs that do not satisfy the intuition above. This technique differs from traditional clone detection that would need to first identify simple fragments of methods as clone pairs (e.g the body of `escapeChar` and the the `for` in Figure 1), and then to determine if the fragments can be combined with any others to yield a higher similarity. This would require checking all combinations of code and an expensive analysis for each simple clone pair.

We have implemented our technique for Java in a prototype named ReDex.[2] The tool takes in the source code of a Java program, from which it first creates particular characteristic vectors for every Java method, and then generates inlined vectors by merging the vectors of the methods that have caller-callee relations to emulate the effect of method inlining. It then uses an efficient vector query technique, Locality Sensitive Hashing (LSH [14]), together with certain filters, to search for methods satisfying certain refactoring criteria and reports them as refactoring opportunities. We have applied the tool to a large code base comprising of 200 bundle projects in the Eclipse ecosystem (e.g., Eclipse JDT, Eclipse PDE, Apache Commons, Hamcrest, ObjectWeb ASM, etc.) containing 4.5 million lines of code. ReDex reported 277 refactoring opportunities, and with manual investigation done by 5 students, we found that the detected opportunities are of high accuracy at about 80%, and cover many categories of cross-function refactoring operations from classical collections of refactoring (e.g., [9, 20]), such as Self Encapsulate Field, Decompose Conditional, Preserve Whole Object, etc.

Our study differs from many other studies on refactoring. Some focus on the detection of refactoring operations that have happened and are recorded in the version histories of a project (e.g., [4, 25, 41, 43, 47]), so as to reconstruct those operations. Other studies focus on formal definitions of refactoring operations (e.g., [38, 39, 44]), so as to help ensure semantic equivalence or correctness of code refactoring. Some tools, such as LambdaFicator and Concurrencer, can automatically perform certain refactoring operations (e.g., converting sequential code to use `java.util.concurrent`, replacing certain `for` loops with functional operations, etc.). Other tools only perform a refactoring operation if the code that needs the operation is identified first with sufficient relevant information (e.g., [11, 18]).

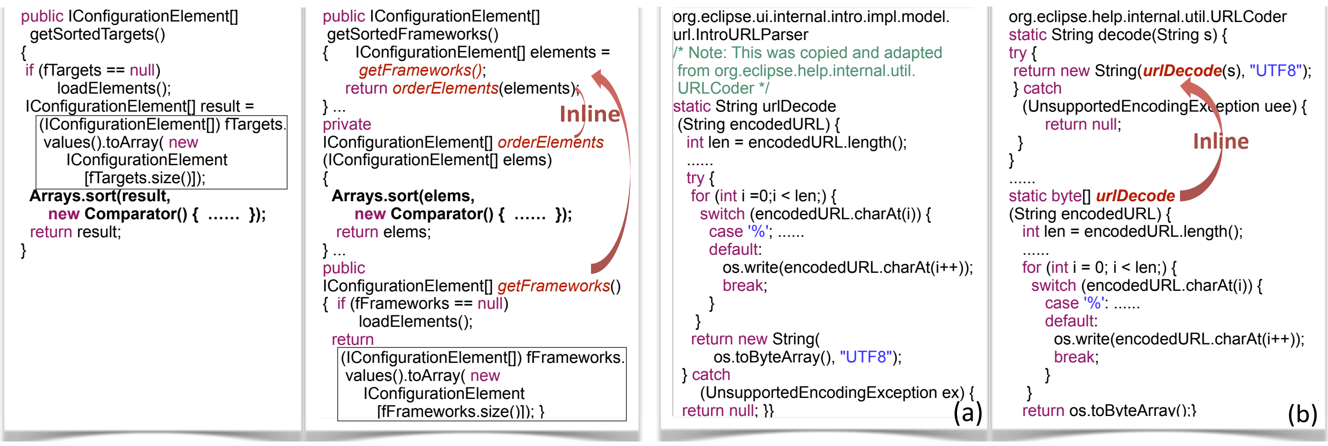
This paper addresses a different problem of scalable identification of missed cross-function refactoring opportunities that have yet to happen; results from our tool can be used to facilitate other tools in performing and validating refactoring. Similar to our work, Cider [40] is a recent study that can detect code clones that may have diverged due to refactoring. However, Cider's detection algorithm works on a graph representation of a program, which is less efficient than ReDex's vector representation and has limited ability in detecting cross-function refactoring. Also, Cider requires initial seeds for its search algorithm, while ReDex works automatically without seeds. Another study by Meng et al. [32] can also detect refactoring opportunities. They create context-aware edit scripts from two or more examples and use the scripts to identify edit locations and transform the code. However, edit-scripts are also limited within a function, and are not yet scalable to identify cross-function changes.

Our main contributions in this paper are as follows:

- We design a new technique based on vector inlining to emulate the effect of method inlining, which enables scalable detection of cross-function refactoring opportunities;
- We have evaluated a prototype of our technique on a code base containing 200 projects (4.5M lines of code) from the Eclipse ecosystem, and results show that our prototype can efficiently detect more than 200 missed refactoring opportunities with an accuracy of 80%.

The rest of the paper is organized as follows. Section 2 describes more cross-function refactoring examples that can be detected by our technique. Section 3 presents our technique in detail. Section 4 presents the results of our empirical evaluation and discusses threats to validity. Section 5 presents related work. Section 6 concludes with future work.

[2] "ReDex" means refactoring detection in this paper. We use the name since refactoring operations, especially method extraction/inlining, bear similarity to "reducible expressions" in lambda calculus.




(a) Sample Refactoring from Eclipse (b) Sample Extract Service Method from Eclipse

**Figure 2: Examples of Refactoring Oportunities**

## 2. CROSS-FUNCTION REFACTORING OPPORTUNITIES

Fowler et al. provide a catalog of refactoring operations [9] and the code changes induced by these operations. Many of the code changes affect methods in a program and may be classified into multiple refactoring categories. Specifically, at a coarse granularity, a particular code change may be classified as Extract Method or Inline Method, while based on the semantic purposes of the refactoring operation, it may be classified into finer-grained categories, such as Replace Temp with Query, Remove Middleman, Encapsulate Field, Separate Query from Modifier, and Form Template Method.

In this paper, we define *refactoring opportunities* as potential code changes that can be fit into classical refactoring categories (i.e., Fowler's categorization [9]) with small variants from recent resources (e.g., [20]). To increase the chance that a detected refactoring indeed has the potential to improve the design of existing code, our tool looks for *missed* opportunities *which are similar to some refactoring that may have happened*. In addition, *cross-function* refactoring opportunities in this paper refer to those categories in Fowler's list that involve method extraction/inlining. In this paper, all refactoring opportunities mentioned are cross-function.

As the first example, the method `escapeString` in Figure I(a) is a missed refactoring opportunity as it can be refactored in the same way as the code in Figure II, or even be replaced by the method in II(b), which can help reduce duplication. According to Fowler's catalog, the example in II(a) is the result of Extract Method. However, we may also classify it as Replace Duplicated Functionality by Existing Method since `escapeString` in II(a) was refactored to reuse the functionality of an existing method `decodeStringValue`. We can also say that the method `decodeStringValue` is a *1-way extraction* since compared with `escapeString` in I(a), it has one method extracted from its body.

Another example is shown in Figure 2(a). The single method `getSortedTargets` on the left is from a class `TargetDefinitionManager` that implements `IRegistryChangeListener`. It gets an array of configuration elements and sorts them. The three methods on the right are from a different class `OSGiFrameworkManager` that also implements `IRegistryChangeListener`. Although the three methods are spatially apart from each other, they together perform the same functionality as `getSortedTargets`. Based on what has been done for the code on the right, a developer may easily see a refactoring opportunity for `getSortedTargets` as well. On the reverse, for reasons such as performance, a developer may also choose, in reference to `getSortedTargets`, to refactor `getSortedFrameworks` by inlining the methods used in it. We call this example a *2-way extraction* since compared with `getSortedTargets`, `getSortedFrameworks` has two methods `getFrameworks` and `orderElements` extracted from its body. In general, we could have cross-function refactoring opportunities that are *n-way extraction*.

In Figure 2(b)(a), the method `urlDecode` was copied from an earlier version of `urlDecode` in Figure 2(b)(b) according to the comments in the code. However, the code in Figure 2(b)(b) has gone through refactoring: the `decode` method was introduced to invoke the local method `urlDecode` and the `try-catch` statement was moved from `urlDecode` into `decode`. This indicates the method in (a) is missed a refactoring opportunity. Such a refactoring operation can be classified as Extract Service Method. Similar to the example in Figure 1, usual clone detection tools may be able to detect parts of the body of both `urlDecode` as clones, but they would *not* be able to link the clones to the additional `decode` method or suggest a concrete way to refactor the code in (a).

Overall, our technique aims to scalably detect missed cross-function refactoring opportunities based on actual refactoring operations that have occurred. ReDex achieves this aim by relying on efficient vector inlining: for every method $m$ in a code base, one or more than one vector is generated to represent $m$; then ReDex searches for another method $m'$ whose vector(s) can become similar to $m$'s vector(s) if all vectors are inlined according to call relations. The needed similarity search is carried out in the form of a vector query with automated filtering of the results. The results, if any, are presented as a set of pairs of code fragments including the query $m$ and its counter-part, indicating possible ways to refactor $m$. Section 3 has more details.

## 3. METHODOLOGY

Figure 3 illustrates the main steps of our approach. Given a code base, we construct its abstract syntax trees (ASTs), program dependence graphs (PDGs), and call graphs (CGs). The ASTs and PDGs are used in a way similar to previous studies [12, 21] in order to generate characteristic vectors

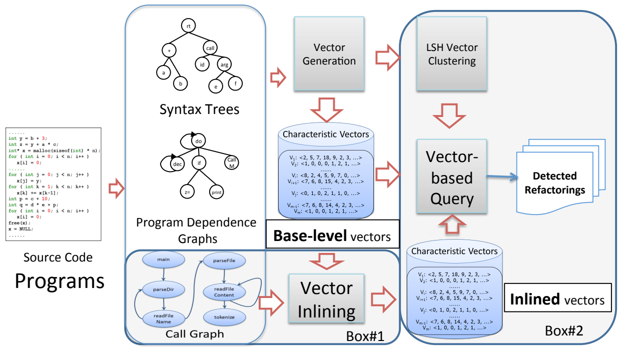



**Figure 3: Approach Overview.**

for code fragments from the code base. The PDGs allow for flexibility in generating vectors for particular data or control flows in a method, as well as the data dependency information needed for accurate method call resolution during for the construction of the call graphs. Our tailored vector generation is recapped in Section 3.1. These vectors only capture characteristics of the code inside the same function: if a method is invoked in a code fragment, the vector for the code fragment does not capture any characteristic of the code inside the invoked method, except the method invocation expression and actual parameters. Thus, we call these vectors *base-level* characteristic vectors in this paper.

The key novelty of our approach is the use of CGs to merge vectors from different functions together according to call relations, so that the merged vectors are able to capture cross-function, semantically related code fragments. The merge operation of vectors is in spirit similar to method inlining, and thus we call it *vector inlining*, which is the main subject of Section 3.2, and we collectively call such merged vectors *inlined* characteristic vectors.

After vector inlining, Locality-Sensitive Hashing (LSH) [14] is adapted to return vectors similar to a vector used as a *query*. Last but not least, all query results are then filtered to identify refactoring opportunities. More details about the query and filtering component are presented in Section 3.3.

In comparison with our previous studies [12,21], the components in the shaded boxes in Figure 3 are new developments of this paper. The components inside the shaded box #1 correspond to vector inlining. The components inside the shaded box #2 correspond to vector querying and filtering tailored for cross-function refactoring opportunities.

## 3.1 Characteristic Vectors for Code

The key idea for efficient code clone detection in our previous studies [12,21] is to represent code fragments as high dimensional vectors in the form of $v = \langle v_1, v_2, ..., v_n \rangle$, where $v_i$ represents the number of occurrences of a particular kind of program element. Then, efficient near-neighbour search algorithms from the database area, such as locality-sensitive hashing [14] can be used to find similar vectors quickly.

In this work we use characteristic vectors for the purpose of refactoring detection. Vectors can be generated directly from the abstract syntax tree of a code fragment to represent the *syntactic* characteristics of the code [21]. They can also be generated from certain parts of the abstract syntax tree of the code that match slices of the program dependence graph of the code [12]. In principle, vectors can be generated from arbitrary combinations of parts of the trees and graphs.

As an illustrating example, Figure 4 shows partial ASTs

**Algorithm 1** Vector Inlining with Depth 1: Inline direct callees' vectors into caller's

**Input**: $v_c$: a vector for a code fragment $c$ that requires inlining
**Input**: $V$: a set of candidate vectors that may be inlined into $v_c$
**Input**: $G$: a call graph of all code involved
**Output**: $v_{in}$: an inlined vector for $v_c$

Let $M_{called}$ be the *set* of functions invoked by $c$, which can be obtained from $G$
$V_{M_{called}} := \emptyset$
**for all** $m \in M_{called}$ **do**
  Let $V_m$ be the vector set for $m$, obtained from $V$
  $V_{M_{called}} := V_{M_{called}} \cup V_m$
**end for**
**if** $V_{M_{called}} = \emptyset$ **then**
  **return none**
**else**
  $v_{in}$ := `inlineVector`($v$, $V_{M_{called}}$)
**end if**

**Method** `inlineVector`
**Input**: $v_c$: a vector from a caller $c$
**Input**: $V_{M_c}$: a set of vectors from $c$'s callees
**Output**: $v$: an inlined vector

$v := v_c$; $flag := false$; $u := \vec{0}$;
**for all** $v_m \in V_{M_c}$ and $v_m$ is a vector from the method $m$
  and $\neg$`isAPI`($m, config$) and `isInlinable`($v_m, config$) **do**
  Let $v'$ be the version of $v$ that excludes the call and the actual arguments to $m$
  $v := v'$; $flag := true$
  Let $v'_m$ be the version of $v_m$ that excludes "return"s
  $u := u + v'_m$
**end for**
**if** $flag = false$ **then**
  **return none**
**else**
  $v := v + u$
**end if**

and characteristic vectors for the code fragments in Figure 5 that will be used explain our key technique—*vector inlining*—in Section 3.2.1. The vector along with the top "block"-node in Figure 4(a) is the vector for the whole tree shown in 4(a). The elements of this vector indicate the occurrences of nodes of the following types: ⟨`return`, `if`, `for`, `assign`, `init`, `new`, `type`, `funcall`, `.`, `<`, `!=`, `++`, `[]`, `id`, `param`, `const`⟩. Program elements, such as "block" and "parameter" in the boxes with dashed borders in 4(a), are often used to facilitate parsing and considered irrelevant for code semantics, and thus not counted in the vectors. The vectors can be easily generated by traversing the tree from bottom to top and by accumulating counters for various node types. We can also remove certain functionally non-essential code (e.g., simple error-handling code, null-check, assertions, throws, try-catch, etc.) when generating vectors.

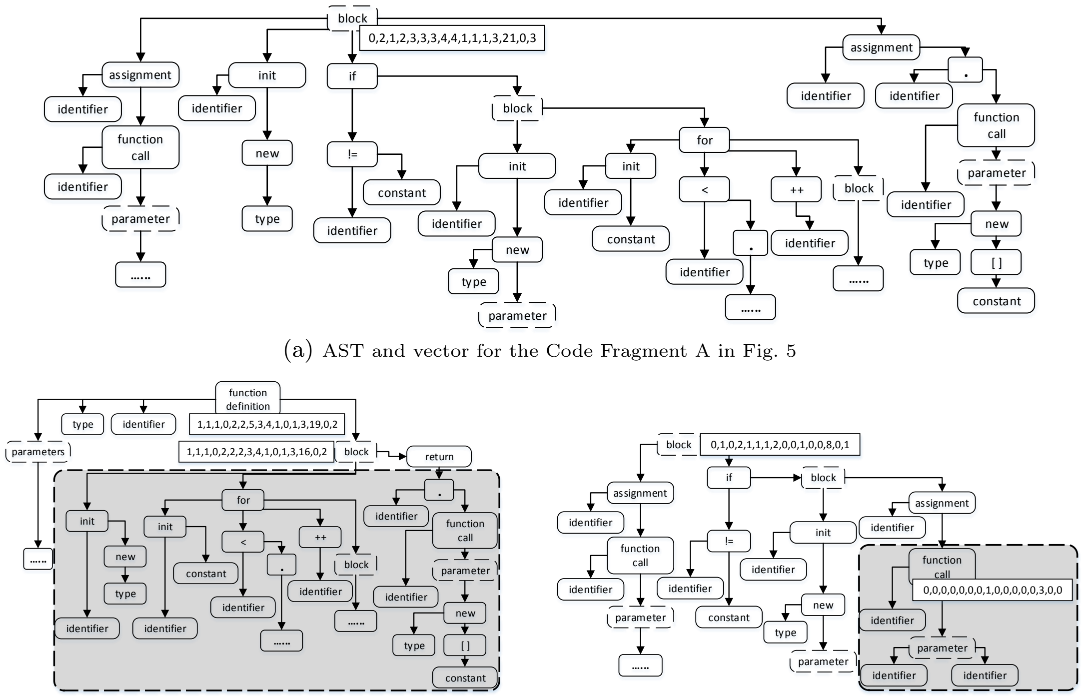
Each vector also comes with various meta data (not shown in the figures), such as the name of the method and the corresponding file, line ranges of the code, number of tokens, etc., to facilitate various postprocessing when needed.

## 3.2 Vector Inlining based on Call Relations

The key challenge for detecting cross-function refactoring is to efficiently capture the call relations among code and to efficiently search for code having similar functionality in the presence of method calls. Our solution is to use vector inlining to emulate the effect of method inlining and extraction.

### 3.2.1 Inlining for One Vector

Given a piece of code $c$ and its corresponding characteristic vector $v_c$, if $c$ contains a call to a function $f$, method inlining




(a) AST and vector for the Code Fragment A in Fig. 5

(b) AST and vectors for `filter` method in Fig. 5

(c) AST and vectors for code C in Fig. 5

**Figure 4: Partial, Illustrative Abstract Syntax Trees Used for Vector Generation and Inlining.**

would replace the function call with the code $b$ in $f$'s body. Intuitively, vector inlining emulating method inlining would replace the parts of $v_c$ corresponding to the call to $f$ with the characteristic vector(s) for the code $b$ in $f$'s body. After the replacement, the changed vector $v'$ would approximately represent the code $c$ inlined with $b$, taking away the code related to the function call expression.

Algorithm 1 implements the above idea and accounts for the situations when $c$ may contain zero or more calls. It takes as input a vector $v_c$, a call graph $G$, and a set of vectors that can be inlined into others (this could simply be all available vectors or chosen by users). It identifies all callees of $c$ based on $G$ and collects all vectors for the callees (Lines 7–11), and then calls the `inlineVector` method to inline those callee vectors $V_{M_{called}}$ into $v_c$ (Line 15). The `inlineVector` method transforms the caller vector $v_c$ and all callee vectors in $V_{M_c}$ for inlining: The caller vector is transformed by subtracting the parts from it that represent method invocations and the actual parameters used in the invocations (Line 27); each callee vector is transformed by subtracting the parts from it that represent the return statements (but retaining the expressions actually returned) (Line 29). Then the transformed caller and callee vectors are summed to produce the inlined vector for $v_c$ (Lines 30 and 35).

As an illustrating example, let us consider inlining the method call to `filter` in Figure 4(c) into its caller vector. The method call to `filter` is shown by the shaded part in Figure 4(c). By following the algorithm `inlineVector`, the inlining proceeds by first subtracting the vector for the call expression from the vector for the caller, which gives us a new vector: $\langle 0,1,0,2,1,1,1,1,0,0,1,0,0,5,0,1\rangle$. Then "return"s should be subtracted from the vector for the function body of `filter` (i.e., the vector along with "block" in Figure 4(b)), which results in a new vector $\langle 0,1,1,0,2,2,2,3,4,1,0,1,3,16,0,2\rangle$. The last step of the algorithm sums the

```
members = getMemberList(...);
ArrayList f = new ArrayList<Member>();
if ( members!=null ) {
    IFilter filter = new Filter(...);
    for(int i=0; i<members.length; i++) {
        if ( filter.satisfy(members[i]) )
            f.add(members[i]);
    }
}
members = f.toArray(new Member[0]);
......
```

(a) Code fragment A

```
members = getMemberList(...);
if ( members!=null ) {
    IFilter f = new Filter(...);
    members = filter(members, f);
} .....
                                 //Code fragment C
```

```
Member[] filter(Member[] array, IFilter f)
{
 ArrayList r = new ArrayList<Member>();
 for(int i=0; i<array.length; i++)
    if ( f.satisfy(array[i]) )
       r.add(array[i]);
 return r.toArray(new Member[0]);
}
```

(b) Code fragment B

**Figure 5: Sample code: (a) may be refactored as (b)**

modified caller and callee vector together thus obtaining $\langle 0,2,1,2,3,3,3,4,4,1,1,1,3,21,0,3\rangle$ which is obviously the same as the vector in Figure 4(a).

### 3.2.2 *Inlinable Vectors*

Algorithm 1 also allows skipping certain vectors based on project-specific or user-specific preferences (`isAPI` and `isInlinable` used at Line 26). For example, whether the method is a third-party library code that the developers of the current project do not care about; whether the code corresponding to the vector is not big enough; or, whether the code does not contain relevant program elements interesting to the users. Such criteria can be stored in a global configuration file $config$, used to decide whether a vector from a method can be inlined into a vector from the method's caller. In the implementation of ReDex, we heuristically checked the fully qualified names of each Java method call and if they belong to certain packages (such as `java.*`), then we treated them as APIs and did not use them for inlining. Also, if some called methods are interface methods or if some call sites cannot be statically resolved to a unique target method, or if some methods are abstract or refer to native code, we treat them as not inlinable and skipped.

```
String getSectionName() {
    return "EXPRESSION_INPUT_DIALOG"; }                (b)
```

```
IDialogSettings section =
    master.getSection("pluginsView");
if (section == null) {
    section = master.
      addNewSection("pluginsView");
......                                     (a)
```

```
IDialogSettings section =
    settings.getSection(getSectionName());
if (section == null) {
    section = settings.
      addNewSection(getSectionName());
......                                     (c)
```

**Figure 6: Replace Constants with Methods**

#### 3.2.3 Multiple Calls to the Same Function

Each code fragment may contain multiple calls to the same function. For method inlining, the same method is usually inlined multiple times. However, detecting a refactoring may require inlining the same function either once or multiple times. For example, the code in Figure 6 needs inlining of the same method `getSectionName` twice to be detected. In ReDex, we make it an option for users to choose, and by default we inline the same function multiple times.

#### 3.2.4 Handling Recursive Function Calls

There are pros and cons for inlining the same function into itself or for inlining another function that directly or indirectly calls itself. Recursive inlining may be too expensive, but it may help to capture more "semantic" characteristics of code into the same function, and the following analysis may be more convenient and "accurate." Our vector inlining algorithm provides two capabilities for users to decide how to inline recursive functions.

First, it relies on a control parameter called *inlining depth* ($d$ in Algorithm 2 in Section 3.2.5) to let a user provide a suitable depth of inlining so that we can terminate vector inlining when the depth of inlining is reached. This is an experience-based way to balance the costs and accuracy of cross-function refactoring detection.

Second, it relies on the structure of the given call graphs (used in Algorithm 1) to avoid potential non-terminating inlining. When cycles exist in a call graph and it is requested by a user, we break cyclic call relations in the call graph: Starting from an entry node or a random node when there is no obvious main entry in the call graph, we traverse the call graph in a depth-first fashion, and remove every back edge found during the traversal. This back edge removal process is repeated until every node in the graph is traversed. Then, the normal vector inlining is applied.

#### 3.2.5 Depth of Inlining

In method inlining, we can choose the inlining depth, from 0, 1, 2, to infinity. Depth 0 effectively means no inlining. Suppose a function $f$ calls another function $m$: with depth 1, we only inline $m$'s body into $f$; with depth 2, we inline $m$'s body and also the body of every function called by $m$ into $f$; and so on, and with depth *infinite*, we inline the body of every function called by $f$, either directly or indirectly, into $f$. Similarly, in our vector inlining, we can choose to inline our characteristic vectors with various depths.

Algorithm 1 effectively inlines vectors with depth 1. Algorithm 2 extends it to allow arbitrary depths. The correctness of this algorithm can be easily proved based on induction on the depth and the correctness of Algorithm 1. The complexity of the algorithm is linear with respect to the number of vectors involved and the depth of inlining.

#### 3.2.6 Indices for Efficiency

The most time-consuming operations in the above algorithms are related to the repeated lookups in the callgraph for callees in a code fragment (especially when the callgraph is large), and the repeated lookups for vectors corresponding to callee signatures, which become particularly expensive when the set of vectors to be inlined is large. However, these operations can be implemented in an efficient way by having various indices to speed up the lookups. The idea is to construct indices among source code locations (file names, method names, and line numbers), methods, and their corresponding vectors. Algorithm 3 optimizes Algorithm 1 and is much more efficient when using the additional indices. If we used multi-sets, instead of sets, to store methods (Lines 9 and 12 in Algorithm 3), we can then inline the same method more than once as discussed in Section 3.2.3.

**Algorithm 2** Vector Inlining with Arbitrary Depths

**Input**: $T$: a set of target vectors that may require inlining
**Input**: $V$: a set of candidate vectors that may be inlined into vectors in $T$; $V$ may or may not be the same as $T$
**Input**: $G$: a call graph of all code involved
**Input**: $d$: a desired depth of inlining ($d < 0$ means an infinite depth, i.e., to inline as deep as possible)
**Output**: $I_1, \ldots, I_d$: sets of inlined vectors

Let $i := 0$
Let $I_0 := T$
Let $V_0 := V$
**while** $|V_i| > 0$ and ($d > i$ or $d < 0$) **do**
  $i := i + 1$
  $I_i := \emptyset$
  $V_i := \emptyset$
  **for all** $t \in T$ **do**
    $I_i := I_i \bigcup \{\text{Call } Algorithm\ 1(t, V_{i-1}, G)\}$
  **end for**
  **for all** $v \in V_{i-1}$ **do**
    $V_i := V_i \bigcup \{\text{Call } Algorithm\ 1(v, V, G)\}$
  **end for**
**end while**
$d := i$
**return** $I_1, I_2, ..., I_d$

**Algorithm 3** Vector Inlining With Depth 1: With Indices

1: **Input**: $v$: a vector that requires inlining
2: **Input**: $V$: a set of candidate vectors that may be inlined into $v$
3: **Input**: $G$: a call graph of all code involved
4: **Output**: $v_{in}$: inlined vector for $v$
5:
6: Let $c$ be the corresponding code fragment of $v$
7: Let $L$ be the set of $\langle filename, lineNumber \rangle$ in $c$
8: Let $MV$ of type: $String \rightarrow SetOfVectors$ the index for vectors
9: Let $LM$ of type: $String \rightarrow lineNumber \rightarrow SetOfStrings$ the index for call relations
10: $M_{called} := \emptyset$
11: **for all** $\langle filename, lineNumber \rangle \in L$ **do**
12:   $M_{called} := M_{called} \bigcup LM[filename][lineNumber]$
13: **end for**
14: $V_{M_{called}} := \emptyset$
15: **for all** $m \in M_{called}$ **do**
16:   $V_m := MV[m]$
17:   $V_{M_{called}} := V_{M_{called}} \cup V_m$
18: **end for**
19: $v_{in} :=$ `inlineVector`($v$, $V_{M_{called}}$)

## 3.3 Vector Query And Filtering

With vector inlining that emulates the effect of method extraction and inlining, the problem of scalable detection of cross-function refactoring can be reduced to finding similarity among base-level and inlined vectors by means of *vector query* and *filtering*. The intuition is: *If two pieces of code become similar, syntactically or semantically, only after the methods called in them are inlined, then they are likely to indicate a cross-function refactoring opportunity, especially if the two code pieces are not similar to each other before inlining.*

The purpose of *vector query* is to find vectors, from a given set of candidate vectors, that are similar to a vector used as an *query*. Our vector querying engine takes as input a pair of vector sets: the first is the *query set* containing all query vectors, and the second is the *target set* containing all candidate vectors. The query engine then returns a set of pairs; each pair represents a match between a query vector and a target vector. As an example, consider a query set only containing the base-level vector for the code fragment I(a) in Figure 1 and a target set only containing the inlined vector for the cross-function code fragment II(a) and (b) in Figure 1. Running the query engine will return the pair formed by the base-level vector for I(a) and the inlined vector for II(a) and (b), and the corresponding source code would be presented as a potential refactoring opportunity.

Similar to previous studies on clone detection [12, 21], we adapt Locality-Sensitive Hashing (LSH) [14], which is designed to efficiently handle nearest-neighbor queries of high-dimensional data, to implement our query engine. Our query engine first stores the target set into LSH's internal hash tables, then uses every query vector from the query set to get matching target vectors for each query vector via LSH backend, and presents all query results as a set of pairs of matching vectors. The LSH backend from Alex Andoni (`http://www.mit.edu/~andoni/LSH/`) is capable of handling a couple of millions of vectors at a time.

Besides querying for matching code, we also need to identify matching code that may manifest cross-function refactoring. Thus, our query engine also defines a set of *filters* for matching vectors, based on heuristics, to identify more likely cross-function refactoring opportunities.

The following defines the query and filters used in ReDex.

Definition 3.1 (Split Query). *Given two vector sets $Ba$ and $In$, where $Ba$ contains only base-level vectors and $In$ contains only inlined vectors, a* Split Query *returns a set of pairs of similar vectors; every pair in the set contains one vector from $Ba$ and another vector from $In$.*

A Split query uses base-level vectors in the query set and inlined vectors in the target set. It allows us to ask whether code contained in one function is similar to code that spans more than one function. A positive answer may provide an opportunity to create a more modular version of the code used as the query, by means of method extraction.

Results from the above query can then be refined by filters. A *filter* defines a set of constraints over a pair of vectors, and *removes the pairs that satisfy the constraints.* Some of the filters we have defined look into the origin of the inlined vectors to make filtering decisions. To facilitate discussion, let us define several notations. Given a method or code fragment $m$, $I_m$ denotes the set of methods invoked by $m$, $v_m^0$ denotes the base-level vector (no inlining) for $m$, and $v_m$ denotes the inlined vector when the vectors for all methods in $I_m$ are inlined into $v_m^0$. Now, we define the following filters for refining query results.

Definition 3.2 (Filter Equal). *Given a pair of vectors $(v_q^0, v_r)$, corresponding to methods $q$ and $r$ respectively,* Filter Equal *first determines the base-level vectors for $q$ and $r$, $(v_q^0, v_r^0)$, and then removes the pair if $v_q^0$ and $v_r^0$ are clones.*

*Filter Equal* aims to eliminate those pairs where the vectors before inlining are equal: As the code fragments for the two base-level vectors are equal, they are unlikely to indicate a refactoring opportunity.

Definition 3.3 (Filter Simple). *Given a pair of vectors $(v_q^0, v_r)$, corresponding to methods $q$ and $r$ respectively, and $r$ invokes a method $i$, this* Filter Simple *removes the pair if $|I_r| = 1$ and $v_q^0$ is equal to $v_i^0$.*

It is obvious that when $|I_r| = 1$, $i$ is the only method invoked by $r$. In addition, when $v_q^0 = v_i^0$, together with the query premise $v_q^0 = v_r$, we have $v^r = v_i^0$ and can infer that the method $r$ does nothing except invoke $i$. Thus, *Filter Simple* eliminates those pairs where the possible refactoring opportunity is to simply fold or unfold a method wrapper.

Definition 3.4 (Filter Size). *Given a pair of vectors $(v_q, v_r)$ or $(v_q^0, v_r)$, this* Filter Size *removes the pair if $v_q$ or $v_q^0$ contains less than 20 nodes.*

*Filter Size* filters out query results whose query vectors are too small in terms of numbers of nodes contained so that we only report refactoring opportunities for code of non-trivial sizes to help reduce possible false positives. An example of such a pair of small vectors is shown in Figure 7.

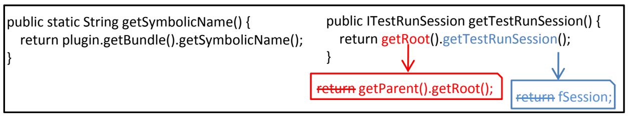



**Figure 7:** Small Vector Inlined

# 4. EMPIRICAL EVALUATION

This section evaluates the effectiveness of our vector-based approach in detecting cross-function refactoring opportunities. We show that our approach scales to a large code base and detects many refactoring opportunities with a high accuracy. Section 4.1 presents the experimental setup, Section 4.2 discusses the performance, and Section 4.3 presents the detected refactoring opportunities and accuracies.

## 4.1 Experimental Setup

While the general idea of using vector inlining to detect refactoring is independent of programming languages, the tools for constructing the data structures needed by our approach are not. In this study, we have implemented a prototype named ReDex and present results for Java programs. The experiments were performed on a PC running Ubuntu 10.04 with Intel Xeon at 2.67GHz and 24GB of RAM.

Our evaluation comprises of 200 bundle projects in the Eclipse 4.2.2 ecosystem, including Eclipse Core, Eclipse JDT, Eclipse PDE, Eclipse Equinox, Apache Commons, Apache Lucene, Hamcrest, etc. The projects encompass more than 20,000 Java files, 40,000 classes, 7,000 interfaces, and contain about 4.5 million lines of code and a long evolution history.

Our implementation uses a modified version of Deckard, in which vectors can be generated to represent either whole methods, slices of methods obtained from PDGs, or any fragment of code in a method. However, in this study, we focus on detecting refactoring opportunities at method level and only generate vectors that represent whole methods.[3] The vectors we experimented with throughout this study have dimensionality equal to 98. The first 84 features of the vectors are the types of ASTNodes generated by Eclipse JDT [7]. Separated from the usual *method_invocation* feature, our vectors also contain the *api_invocation* feature that refers to invocations of methods not defined in the subject programs. Specifically, the last 12 features of the vectors are *method_invocation_paramno* and *api_invocation_paramno*

[3] Code in a whole method includes all executable code in the method, but excludes function headers, variable declaration, simple elements unlikely responsible for the main function- ality of the code (e.g., simple null-check and return, throw exceptions), and non-executable lines (e.g., comments, blank lines, lines with only curly braces).

that denote invocations with the number of actual arguments denoted by *paramno* where $paramno \in \{0, 6\}$.

Also, we focused on detecting refactoring within and across projects in our code base and ignored potential refactoring that may span across methods defined in external libraries. Thus, our inlining algorithm was configured to only inline a method if the method is defined in a project in the code base (checked by `isAPI` and `isInlinable` in Algorithm 1). In addition, inlining was carried out in an *all* mode, i.e., all inlinable and non-API methods called in a code fragment are inlined; if any one of the methods in a code fragment cannot be inlined due to any reason (e.g., missing vectors due to parsing errors, unresolved call targets, etc.), the inlining for the code fragment would be cancelled. Further, we focus on evaluation of depth-1 inlining using ReDex and the type of queries and filters described in Section 3.3.

## 4.2 Scalability

The most expensive parts in terms of both time and memory consumption are the construction of the callgraph (CG), ASTs, PDGs, and the generation of indices (cf. Section 3.2.6) for these data structures. PDG and CG construction built on WALA [19] took about 44minutes; vector inlining (including building indices) took 3 minutes, while indexing took most of the time, and the actual vector inlining (c.f. Algorithm 3) took less than 1 minute for inlining depth 1. Fortunately, such constructions are one-time cost, and more optimizations can be performed in future for the constructions.

ReDex generated about 186K base-level characteristic vectors, each of which represents the body[3] of a defined method in the Eclipse ecosystem (excluding abstract, native and interface methods or external methods defined outside of Eclipse). Thus, about 186K queries were performed and filtered; they accumulatively took less than 2 minutes to report potential refactoring opportunities. Figure 8 shows the distribution of the refactoring oportunities in the projects. One can see that many projects are covered by the vectors.

## 4.3 Cross-Function Refactoring Opportunities

We have detected many missed refactoring opportunities in the bundle projects of Eclipse. Specifically, ReDex generated 277 reports for the evaluation code base. Each report is a pair of two pieces of code that may span multiple functions: one corresponds to the query generating the report, and the other corresponds to the target matching the query. Each of the two pieces of code may reveal a refactoring opportunity and could be refactored according to its counter-part.

The validation of the results was done through a user study with 5 graduate students with good knowledge of Java and refactoring. These report inspectors were required to classify each of the reports into one of the following four options: (1) `refactoring`, (2) `not refactoring but clone`, (3) `not refactoring or clone`, or (4) `I don't know`. In the case when the `refactoring` option is selected, they were required to classify, additionally, the refactoring opportunity that might be applied to one of the code fragements in the report.

Overall, the results after inspection showed a high accuracy at 80% for the cross-function refactoring opportunities detected by ReDex: 80% of all reports were classified as option (1), 16% as option (2), and 4% as options (3) or (4). The validation exercise discovered 223 out of the 277 analysed cases to have true refactoring opportunities. These true refactoring opportunities are matched to many categories and variants of Fowler's catalog. Table 1 shows these categories

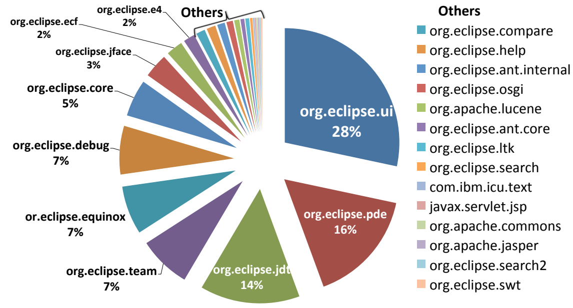



**Figure 8: Distribution of Covered Code**

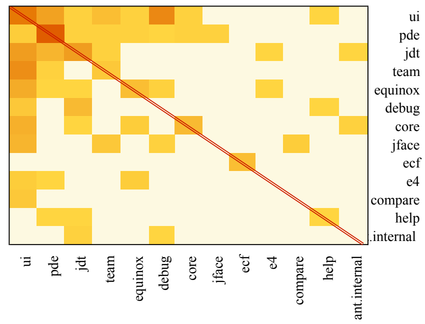



**Figure 9: Heat Map of Refactoring Opportunities**

and the number of validated refactoring opportunities. These provide strong evidence to support the ability of ReDex in detecting missed refactoring opportunities.[4] Some examples of refactorings have been shown in Sections 1 and 2.

Figure 9 shows a heatmap of the number of reports between the projects in the evaluation. Values on the diagonal indicate refactoring opportunities within the same project. However, Figure 9 also shows many cross-project refactoring opportunities. The results also indicate that many similar code and refactoring opportunities across different functions and projects diverge, which increases the difficulty for their identification, and techniques that can detect cross-function refactoring opportunities are indeed needed.

Furthermore, our evaluation showed that a large number of vectors in the result set, 97%, are the result of inlining one method. This is consistent with the fact that most refactorings that involve cross-function changes in Fowler's catalog commonly only involve extracting/inlining one method.

During the investigation, the report inspectors checked if one of the code fragments in a report can be refactored in accordance with Fowler's categorization [9], by comparing its shape with its counter-part. At the same time, we allow their best judgements and small variants to Fowler's categories as shown in Table 1. Some of the reports were validated by multiple inspectors, which resulted in interesting observations. For simple refactorings, such as Self Encapsulate Field, the type of refactoring was mostly correctly identified by all. For more complex refactorings, such as Preserve Whole Object or Separate Query from Modifier, there were variations between the types of refactoring classified by the inspectors. The example in Figure 10 was classified as both "I don't know" and `Preserve Whole Object`. The report consists of two methods `convertSeverity` and `convertLevel` that return an integer. Although they have similar functionality,

[3]The total number of occurences in Table 1 may be bigger than the number of reports returned by ReDex. This is due to the fact that code may be refactored in multiple ways.

Table 1: **Categories of Refactoring Opportunities**

| Refactoring Categories | Ocurrences |
|---|---|
| Self Encapsulate Field | 76 |
| Encapsulate collection access and downcast | 19 |
| Downcast encapsulate | 2 |
| Decompose Conditional Expression | 2 |
| Substitute Algorithm | 23 |
| Extract/Inline method | 25 |
| Separate Query from Modifier | 2 |
| Introduce Query method | 18 |
| Replace duplicated functionality by existing method | 10 |
| Hide Delegate | 23 |
| Preserve Whole Object | 3 |
| Introduce Parameter Object | 7 |
| Reverse conditional | 3 |
| Replace temp with chain | 1 |
| Make method static | 11 |

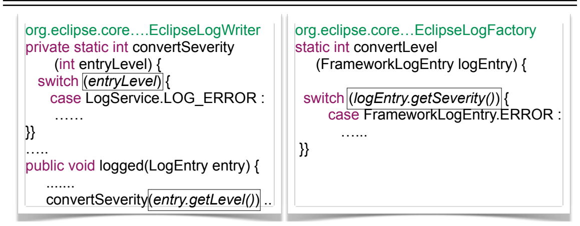



**Figure 10: Preserve Whole Object**

the two methods differ in the parameters. `convertLevel` receives an object as parameter and calls a member function of that object to access the data needed inside `convertLevel`. `convertSeverity` on the other hand, receives an `int` value obtained by a call to a member method of the object `entry` of type `LogEntry` before calling `convertSeverity`. Sending the whole object into a method makes it more robust to certain functionality changes and avoids problems when the method needs new data values from the object later. Thus, to make `convertSeverity` more robust against changes, we may refactor `convertSeverity` to use an object of type `LogEntry` as parameter without affecting its performance, through the operation `Preserve Whole Object`. For such cases where the refactoring types varied, we applied our best judgment and chose from the types selected by the reporters.

An interesting class of reports from the results is represented by code fragments where the `Make Method Static` refactoring available in IntelliJ IDEA [20] can be applied. 40 such reports were classified as `not refactoring but clone` by an inspector with the comment that "one side of the code is not directly refactorable into the other yet they are similar (possible diverged from one source) and can be made more reasonable with refactoring techniques." These reports are however not counted positively toward the accuracy of our approach. Counting these reports as refactoring opportunities would have increased the accuracy to 94%.

## 4.4 Discussion & Threats to Validity

Our approach depends on the setting of some parameters. Some of them are related to code similarity metrics and common to most clone detection tools. For example, the minimal number of tokens or nodes that a code fragment needs to contain, and the difference (or similarity) allowed between two code fragments for them to be detected. ReDex is only evaluated with code sizes larger than 20 (cf. Filter Size in Definition 3.4) and similarity 1.0.

ReDex is only evaluated with Split Query (cf. Definition 3.1) that uses all generated base-level vectors for Eclipse as the query set and all inlined vectors as the target set. Our approach doesn't need users to choose queries though it is possible to provide tailored query and target sets to find additional cross-function refactoring opportunities. The filters we used are relatively simplistic; more comprehensive filtering constraints may be developed based on common refactoring operations (e.g., Fowler's and other collections [9, 20]) to look for refactoring opportunities more accurately.

A number of other parameters control what can be inlined in our algorithms. For example, the *depth* of inlining in Algorithm 2 and whether to inline a function more than once (c.f. Section 3.2.3) affect the number of functions inlined together. Also, since vectors can be generated for arbitrary code fragments, not just whole methods, inlining can be carried out for vectors corresponding to arbitrary code, which may be expected to produce more refactoring opportunities. We currently use an all mode to inline all vectors for all methods invoked by a code fragment; we will expect to detect more refactoring opportunities if we allow *partial inlining*.

Our approach doesn't consider flow sensitivity since most refactorings we can detect don't affect flow sensitivity, but may be more accurate for some cases if we make the vectors "flow sensitive". We leave it as future work to explore the large configuration and parameter space to balance the number detected refactoring opportunities with their accuracies.

In our empirical evaluation, we measured the accuracy of the results via manual investigation by students. This introduces experimental bias. The students' Java programming skills and knowledge about refactoring may also affect how they label the reports. We also limit our evaluation to Java and thus our results may not be applicable to other programming languages. In the near future, we plan to port to other languages, extend our evaluation to more programs, and conduct both automated evaluation against historical refactoring operations and more systematic user studies to alleviate the above threats to the validity of our approach.

# 5. RELATED WORK

This paper searches for refactoring opportunities, a goal related to many studies in refactoring and code clone detection, which are also broadly related to software maintenance and evolution. The discussion here is by no means complete.

Many studies on refactoring focus on the specification and implementation of refactoring operations. A classical work by Opdyke [33], describes a set of refactoring operations for C++ in terms of the preconditions needed to preserve behaviour. Griswold specifies refactoring from the perspective of their effects on program dependence graphs [16]. Lämmel [28] and Garrido [13] use rewriting rules to represent refactoring. Recent studies also aim to allow programmers to script their own refactoring operations. To this end, Verbaere et al. [46] propose a domain specific language for expressing dataflow properties on a graph representation of the program. Scäfer et al. [38] improve on this and provides high-level specifications for many refactoring operations implemented in Eclipse. Our work complements those studies in that it searches for new refactoring opportunities. As future work, we plan to investigate the development of a query language and of abstractions that would allow us to more comprehensively and precisely specify the refactoring opportunities to search for.

There are also many studies aiming for automatic detection of refactoring (besides the studies on clone detection). Many of these studies rely on changes recorded in version control systems; their focus is to reconstruct refactoring operations that have happened. Demeyer et al. [4] define heuristic metrics to search for refactoring between successive versions. Hayashi et al. [17] model refactoring detection as a graph search representing structural differences between two versions of a program. Weißgerber and Diehl [47] define various signatures based on code clone detection results to look for refactoring. Prete and Kim [25, 35] use template logic queries to represent refactoring operations and a logic programming engine to reconstruct the refactoring between versions of a program; their tool Ref-Finder can detect 63 kinds of refactoring in Fowler's catalog. Taneja and Dig et al. [5, 43] present tools (`RefactoringCrawler` and `RefacLib`) to detect refactoring between different versions of libraries. Soetens et al. [41] detect refactoring operations as actual changes are happening in an integrated development environment, and thus achieve higher accuracy than previous work. Origin analysis has also been used to detect refactoring [15] by capturing certain kinds of cross-function changes and how call relations change between two versions of a program. Our vector-based inlining and query technique is not limited for changes between versions; it can search whole code bases and detect refactoring opportunities within the same version.

The approach of Meng et al. [32] can also be applied to find refactoring opportunities. Their idea is to create context-aware edit scripts from two or more examples and use the scripts to identify edit locations and transform the code. However, edit-scripts are limited within a single method, as from their experience combining inter-procedural analysis and the expressiveness of general-purpose edits is a very hard problem. They can thus not detect changes that require moving code from one method to another or coordinating changes to multiple methods in the way our approach does.

Cider [40] is another work that can detect refactoring without code change histories. Their algorithm relies on graph matching and requires initial seeds that are similar code fragments first, and is limited within individual methods too. Our technique does not need seeds and relies on vector matching, making it more scalable to large code bases where code divergences across functions occur more often.

Code clone detection has also been touted as an important way to detect refactoring opportunities (Extract or Pull-Up Method in particular). Fontana et al. [8] manually refactor code clones detected by three different clone detection tools and find that certain code quality metrics are improved after the refactoring. Higo et al. [18] define several metrics for code clones and demonstrate a tool that can suggest refactoring operations for code clones. Tairas [42] visualize clones so that it may become easy to select candidates for refactoring. Van Rysselberghe and Demeyer [45] investigate three different kinds of clone detection techniques (simple line matching, parameterized matching, and metric fingerprints) and find that clones detected by different techniques may be suitable for different kinds of refactoring.

Generally speaking, clone detection techniques can be string-based [1], token-based [3, 29, 31, 36], tree-based [2, 21], graph-based [12, 26, 27, 30], functionality-based [22, 24, 34], and some detect clones in bytecode or binary code [23, 37]. Although they can in theory detect cross-function clones, especially the ones using program slicing [12, 27], their scalability is still limited to intra-procedural for large code bases. Our approach builds on, and extends a previous study on clone detection with the capability to check cross-function similar code. This paper focuses on detecting cross-function refactoring opportunities, while it may be possible to adapt our technique to help improve clone detection.

Some studies and tools can also automatically perform identified refactoring. For example, modern development environments, such as Eclipse and NetBeans, have refactoring capabilities. Concurrencer [6] can identify and convert sequential code that may be benefited from the `java.util.concurrent` supports. LambdaFicator [10], automatically refactors certain anonymous inner Java classes and `for` loops to use lambda expressions and functional operations available in Java 8. Our tool currently focuses on scalable detection only. We plan in future work to make our tool perform identified refactoring automatically.

# 6. CONCLUSION AND FUTURE WORK

This paper presents a novel, automated and scalable technique for identifying cross-function refactoring opportunities that span more than one function (e.g., Extract/Inline Method). The key of our technique is the design of efficient *vector inlining* operations that emulate the effect of method inlining among code fragments, so that the problem of identifying cross-function refactoring can be reduced to finding similar vectors before and after vector inlining.

We have implemented our technique in a prototype tool named ReDex which encodes Java programs to particular vectors. We have evaluated our technique on a large code base (4.5 MLOC) and the results show that ReDex can find 277 cross-function refactoring opportunities in 2 minutes, and 223 cases were labelled as true opportunities by users, and cover many categories of cross-function refactoring operations in classical refactoring books, such as Decompose Conditional Expression, Hide Delegate, Preserve Whole Object, etc.

In the future, we plan to improve ReDex so that it can automatically categorize the detected refactoring opportunities, which may require us to incorporate more diverse filtering criteria into our approach. The filtering criteria might consider the particular structures and features of inlined methods, the call relations between code fragments, or some other characteristics of the composition of code fragments. We have also experimented with a number of abstractions of the vectors, such as considering literals and simple names as the same program elements, to allow us to encode more refactoring operations in vectors. With appropriate abstractions and filtering criteria we aim to detect a broad range of refactorings. We also plan to develop a query language that allows us to specify the refactoring opportunities to search for. The query language will also allow us to specify the composition of multiple refactoring types. We also plan to incorporate code change histories from version control systems to further improve the accuracy of ReDex, and evaluate whether refactoring detection results can help to track code changes better and facilitate code evolution and program understanding.

# 7. ACKNOWLEDGMENTS

We would like to express our gratitude to the following report inspectors, who have spent much time in evaluating all our experiment results: Chenhong Xie, Liu Yang, Lucia, Pavneet Kochhar Singh, and Ta Quang Trung. This research is supported by NUS Research Grant R-252-000-484-112.

# 8. REFERENCES


[1] B. S. Baker. Finding clones with Dup: Analysis of an experiment. *IEEE TSE*, 33(9):608–621, 2007.

[2] I. D. Baxter, C. Pidgeon, and M. Mehlich. DMS©: Program transformations for practical scalable software evolution. In *ICSE*, pages 625–634, 2004.

[3] Y. Dang, D. Zhang, S. Ge, C. Chu, Y. Qiu, and T. Xie. XIAO: tuning code clones at hands of engineers in practice. In *ACSAC*, pages 369–378, 2012.

[4] S. Demeyer, S. Ducasse, and O. Nierstrasz. Finding refactorings via change metrics. In *OOPSLA*, pages 166–177, 2000.

[5] D. Dig, C. Comertoglu, D. Marinov, and R. E. Johnson. Automated detection of refactorings in evolving components. In *ECOOP*, pages 404–428, 2006.

[6] D. Dig, J. Marrero, and M. D. Ernst. Refactoring sequential java code for concurrency via concurrent libraries. In *ICSE*, pages 397–407, 2009.

[7] Eclipse Foundation. Eclipse Java development tools (JDT). `http://help.eclipse.org/indigo/topic/org.eclipse.jdt.doc.isv/reference/api/org/eclipse/jdt/core/dom/ASTNode.html`.

[8] F. A. Fontana, M. Zanoni, A. Ranchetti, and D. Ranchetti. Software clone detection and refactoring. *ISRN Software Engineering*, online open access, 2013.

[9] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.

[10] L. Franklin, A. Gyori, J. Lahoda, and D. Dig. LAMBDAFICATOR: from imperative to functional programming through automated refactoring. In *ICSE*, pages 1287–1290, 2013.

[11] R. M. Fuhrer, M. Keller, and A. Kiezun. Refactoring in the eclipse jdt: Past, present, and future. In *Workshop on Refactoring Tools (WRT)*, pages 30–31, 2007.

[12] M. Gabel, L. Jiang, and Z. Su. Scalable detection of semantic clones. In *ICSE*, pages 321–330, 2008.

[13] A. Garrido and J. Meseguer. Formal specification and verification of java refactorings. In *6th IEEE International Workshop on Source Code Analysis and Manipulation (SCAM)*, pages 165–174, 2006.

[14] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *VLDB*, pages 518–529, 1999.

[15] M. W. Godfrey and L. Zou. Using origin analysis to detect merging and splitting of source code entities. *IEEE TSE*, 31(2):166–181, 2005.

[16] W. G. Griswold. Program restructuring as an aid to software maintenance, phd thesis. *University of Washington*, 1991.

[17] S. Hayashi, Y. Tsuda, and M. Saeki. Detecting occurrences of refactoring with heuristic search. In *APSEC*, pages 453–460, 2008.

[18] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue. ARIES: refactoring support tool for code clone. *ACM SIGSOFT Software Engineering Notes*, 30(4):1–4, 2005.

[19] IBM T.J. Watson. T.J. Watson Libraries for Analysis (WALA). `http://wala.sourceforge.net/`.

[20] JetBrains. Refactoring Source Code in IntelliJ IDEA 13. `http://www.jetbrains.com/idea/webhelp/refactoring-source-code.html`.

[21] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. DECKARD: Scalable and accurate tree-based detection of code clones. In *ICSE*, pages 96–105, 2007.

[22] L. Jiang and Z. Su. Automatic mining of functionally equivalent code fragments via random testing. In *ISSTA*, pages 81–92, 2009.

[23] I. Keivanloo, C. K. Roy, and J. Rilling. SeByte: A semantic clone detection tool for intermediate languages. In *ICPC*, pages 247–249, 2012.

[24] H. Kim, Y. Jung, S. Kim, and K. Yi. MeCC: memory comparison-based clone detector. In *ICSE*, pages 301–310, 2011.

[25] M. Kim, M. Gee, A. Loh, and N. Rachatasumrit. Ref-finder: A refactoring reconstruction tool based on logic query templates. In *FSE*, pages 371–372, 2010.

[26] R. Komondoor and S. Horwitz. Semantics-preserving procedure extraction. In *POPL*, pages 155–169, 2000.

[27] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. In *SAS*, pages 40–56, 2001.

[28] R. Lämmel. Towards generic refactoring. In *ACM SIGPLAN workshop on Rule-based programming*, pages 15–28, 2002.

[29] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: A tool for finding copy-paste and related bugs in operating system code. In *OSDI*, pages 289–302, 2004.

[30] C. Liu, C. Chen, J. Han, and P. S. Yu. GPLAG: Detection of software plagiarism by program dependence graph analysis. In *KDD*, pages 872–881, 2006.

[31] S. Livieri, Y. Higo, M. Matsushita, and K. Inoue. Very-large scale code clone analysis and visualization of open source programs using distributed CCFinder. In *ICSE*, pages 106–115, 2007.

[32] N. Meng, M. Kim, and K. S. McKinley. Lase: locating and applying systematic edits by learning from examples. In *ICSE*, pages 502–511, 2013.

[33] W. F. Opdyke. Refactoring object-oriented frameworks, phd thesis. *University of Illinois at Urbana-Champaign*, 1992.

[34] A. Podgurski and L. Pierce. Retrieving reusable software by sampling behavior. *ACM TOSEM*, 2(3):286–303, 1993.

[35] K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim. Template-based reconstruction of complex refactorings. In *ICSM*, pages 1–10, 2010.

[36] C. K. Roy and J. R. Cordy. NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *ICPC*, pages 172–181, 2008.

[37] A. Sæbjørnsen, J. Willcock, T. Panas, D. Quinlan, and Z. Su. Detecting code clones in binary executables. In *ISSTA*, pages 117–128, 2009.

[38] M. Schäfer and O. de Moor. Specifying and implementing refactorings. In *OOPSLA*, pages 286–301, 2010.

[39] M. Schäfer, J. Dolby, M. Sridharan, E. Torlak, and F. Tip. Correct refactoring of concurrent java code. In *ECOOP*, pages 225–249, 2010.

[40] M. Shomrat and Y. Feldman. Detecting refactored clones. In *ECOOP*, volume 7920 of *Lecture Notes in Computer Science*, pages 502–526, 2013.

[41] Q. D. Soetens, J. Perez, and S. Demeyer. An initial investigation into change-based reconstruction of floss-refactorings. In *ICSM*, pages 384–387, 2013.

[42] R. Tairas. Clone detection and refactoring. In *OOPSLA*, pages 780–781, 2006.

[43] K. Taneja, D. Dig, and T. Xie. Automated detection of API refactorings in libraries. In *ASE*, pages 377–380, 2007.

[44] F. Tip, R. M. Fuhrer, A. Kiezun, M. D. Ernst, I. Balaban, and B. D. Sutter. Refactoring using type constraints. *TOPLAS*, 33(3):9, 2011.

[45] F. Van Rysselberghe and S. Demeyer. Evaluating clone detection techniques from a refactoring perspective. In *19th ASE*, pages 336–339, 2004.

[46] M. Verbaere, R. Ettinger, and O. de Moor. Jungl: a scripting language for refactoring. In *ICSE*, pages 172–181, 2006.

[47] P. Weissgerber and S. Diehl. Identifying refactorings from source-code changes. In *ASE*, pages 231–240, 2006.