Mining Fine-Grained Code Changes to Detect Unknown Change Patterns

Stas Negara
University of Illinois
Urbana, IL, USA
snegara2@illinois.edu

Mihai Codoban
Oregon State University
Corvallis, OR, USA
codobanm@eecs.oregonstate.edu

Danny Dig
Oregon State University
Corvallis, OR, USA
digd@eecs.oregonstate.edu

Ralph E. Johnson
University of Illinois
Urbana, IL, USA
rjohnson@illinois.edu

ABSTRACT

Identifying repetitive code changes benefits developers, tool builders, and researchers. Tool builders can automate the popular code changes, thus improving the productivity of developers. Researchers can better understand the practice of code evolution, advancing existing code assistance tools and benefiting developers even further. Unfortunately, existing research either predominantly uses coarse-grained Version Control System (VCS) snapshots as the primary source of code evolution data or considers only a small subset of program transformations of a single kind — refactorings.

We present the first approach that identifies *previously unknown* frequent code change patterns from a *fine-grained* sequence of code changes. Our novel algorithm effectively handles challenges that distinguish *continuous* code change pattern mining from the existing data mining techniques. We evaluated our algorithm on 1,520 hours of code development collected from 23 developers, and showed that it is effective, useful, and scales to large amounts of data. We analyzed some of the mined code change patterns and discovered ten popular kinds of high-level program transformations. More than half of our 420 survey participants acknowledged that eight out of ten transformations are relevant to their programming activities.

Categories and Subject Descriptors: D.2.7 [Software Engineering]: Distribution, Maintenance, and Enhancement

General Terms: Algorithms, Experimentation

Keywords: Program Transformation, Code Changes, Data Mining

1. INTRODUCTION

Many code changes are repetitive by nature [14, 17, 25], thus forming code change *patterns*. Frequent pattern mining [12] is successfully applied in a broad range of domains. For example, Amazon.com recommends related products based on "customers who bought this also bought that". Netflix recommends new movies based on "customers who watched this also watched that". Similar frequent pattern mining has revolutionized other services such as iTunes, GoodReads, social platforms, etc. More recently, data mining techniques became popular in the domain of genetics [27, 29, 31]. In particular, these techniques are employed to identify similar sequences of genes, which is a common task in DNA studies. We conjecture that mining frequent code changes can be similarly transformative for software development.

Identifying frequent code change patterns benefits Integrated Development Environment (IDE) designers, code evolution researchers, and developers. IDE designers can build tools that automate execution of frequent code changes, recommend code changes, or offer intelligent code completion [3, 24, 26], thus improving the productivity of developers. Researchers would better understand the practice of code evolution and also would be able to focus their attention on the most popular development scenarios. Library developers can notice and fix the common mistakes in the library API usage.

Existing research [3–5, 15, 19, 21, 30, 32, 35, 36, 39] predominantly detects frequent code change patterns either analyzing the static source code of a single version of an application or comparing the application's Version Control System (VCS) snapshots. In our previous study [23], we showed that data stored in VCS is *imprecise*, *incomplete*, and makes it *impossible* to perform analysis that involves the time dimension inside a single VCS snapshot. Recent research [9, 11] considered more precise data captured directly from IDE, but their code change identification techniques were limited in two ways: (i) they were looking for a single kind of code change patterns — refactorings, (ii) they considered only a small subset of previously known kinds of refactorings.

In this paper, we employed data mining techniques to detect *previously unknown* frequent code change patterns from a *fine-grained* sequence of code changes. Our mining algorithm does not use any predefined templates to look for patterns, and thus, all patterns it detects are previously unknown. We recorded the code changes as soon as they were produced by developers. Consequently, our algorithm's input sequence is the most fine-grained and precise representation of code evolution.

ICSE'14, May 31 – June 7, 2014, Hyderabad, India
Copyright 2014 ACM 978-1-4503-2756-5/14/05...$15.00
http://dx.doi.org/10.1145/2568225.2568317

There are unique challenges posed by our problem domain of program transformations, which render previous off-the-shelf data mining techniques [12] inapplicable. First, for program transformations, we need to mine a *continuous* sequence of code changes that are ordered by their timestamps, without any previous knowledge of where the boundaries between patterns of transformations are. In contrast, standard data mining techniques operate on a database of transactions with well known boundaries. Second, unlike the standard frequent itemset mining [1,13,37,38], when mining frequent code change patterns, a high-level program transformation corresponding to a given pattern may contain several instances of the same kind of code change. For example, Rename Local Variable refactoring involves the same change for all references of the renamed variable. Consequently, in our mining problem, a transaction may contain multiple instances of the same item kind, thus forming itembags rather than itemsets.

In this paper, we present our novel frequent code change patterns mining algorithm that effectively handles both challenges specific to our mining problem. We applied our algorithm on a large corpus of real world data that we collected during our previous user study [23], in which we accumulated 1,520 hours of code development from 23 developers working in their natural settings. Our evaluation shows that our algorithm is effective, useful, and scales well for large amounts of data. In particular, our algorithm mined more than half a million of item instances in less than six hours. We analyzed some of the frequent code change patterns detected by our algorithm and identified ten kinds of popular high-level program transformations. On average, 32% of the pattern occurrences reported by the algorithm led to high-level program transformation discoveries.

To assess the popularity of the identified high-level program transformations, we conducted a survey study with 420 participants. More than half of our survey participants found eight out of ten kinds of program transformations relevant to their programming activities. Moreover, most participants regularly perform six transformation kinds and would like to get automated support for five transformation kinds. These results confirm that our mining algorithm helps identify useful and popular program transformations.

This paper makes the following contributions:

- **Algorithm:** We designed a novel algorithm that effectively addresses the challenges of frequent code change patterns mining.
- **Implementation:** We implemented our algorithm as part of CodingTracker. CodingTracker is open source and is available at http://codingtracker.web.engr.illinois.edu.
- **Evaluation:** We evaluated our algorithm on 1,520 hours of real world code development data and showed that it is effective, useful, and scalable.
- **Results:** We analyzed some of the mined code change patterns and identified ten kinds of popular high-level program transformations.

2. MOTIVATING EXAMPLE

Figure 1 shows a code editing example in which a developer repeatedly applies a high-level program transformation. In this and all subsequent examples of program transformations, we represent the changed parts of code as underlined text. Figure 1 shows that the developer edits two methods, `getDistance` and `computeDirection`. Method `getDistance` computes distance between two points, `p1` and `p2`. Method `computeDirection` computes a direction angle from some origin point `o` to a given point `p`. In this code editing example, the developer first renames `getDistance` to `computeDistance` to better reflect its meaning. Then, the developer decides to increase accuracy of direction angle computation and changes accordingly the return type of `computeDirection` method from `float` to `double`. Finally, the developer improves the performance of both methods — if the method's arguments are equal, the method returns `0` without performing any additional computations. We call this high-level program transformation Return If Arguments Are Equal.

Before

```
double getDistance(Point p1, Point p2) {
  ...
}

float computeDirection(Point o, Point p) {
  ...
}
```

After

```
double computeDistance(Point p1, Point p2) {
  if (p1.equals(p2)) return 0;
  ...
}

double computeDirection(Point o, Point p) {
  if (o.equals(p)) return 0;
  ...
}
```

Figure 1: An example of a repeated high-level program transformation. Changed code is underlined.

Table 1 shows code changes of the code editing scenario in Figure 1. The first column presents the relative order of the changes. Note, however, that the ordering of changes is partial. For example, according to the code editing scenario in Figure 1, change 2 happened after change 1. At the same time, the relative order of changes 3 – 9 is undefined. The second and the third columns correspondingly show the kind of the Abstract Syntax Tree (AST) node operation and the affected AST node's type. The next column presents the content of the affected AST node (or the content's change for *change* operations). We explain the content of the last column in the following section.

Given a sequence of code changes like the one in Table 1, we would like to identify repetitive code change patterns that correspond to some high-level program transformations. For example, we would like to detect that code changes 3 – 9 and 10 – 16 represent the same code change pattern — the pattern of high-level program transformation Return If Arguments Are Equal. To achieve this goal, we use data mining techniques. In the following section, we discuss how our approach differs from the canonical data mining problem.

3. BACKGROUND

Definition 1 (Code Change): A code change is a pair `(<operation kind>, <AST node type>)`.

We ignore the contents of the affected AST nodes to avoid making our code changes too specific. Too specific representation hinders detecting similar changes, e.g., changes 9 and 16 in Table 1 would be different, if we considered the affected nodes' contents.

Definition 2 (Code Change Pattern): A code change pattern is an unordered bag of code changes.

Table 1: Code changes of the code editing scenario in Figure 1.

Order index	Operation	AST node type	AST node content change	Item
1	*change*	SimpleName	`getDistance` → `computeDistance`	a
2	*change*	PrimitiveType	`float` → `double`	b
3	*add*	SimpleName	`p1`	c
4	*add*	SimpleName	`equals`	c
5	*add*	SimpleName	`p2`	c
6	*add*	MethodInvocation	`p1.equals(p2)`	d
7	*add*	NumberLiteral	`0`	e
8	*add*	ReturnStatement	`return 0`	f
9	*add*	IfStatement	`if (p1.equals(p2)) return 0`	g
10	*add*	SimpleName	`o`	c
11	*add*	SimpleName	`equals`	c
12	*add*	SimpleName	`p`	c
13	*add*	MethodInvocation	`o.equals(p)`	d
14	*add*	NumberLiteral	`0`	e
15	*add*	ReturnStatement	`return 0`	f
16	*add*	IfStatement	`if (o.equals(p)) return 0`	g

Note that a code change pattern is a bag rather than a set, since the same code change might occur several times in a pattern, e.g., `(add, SimpleName)` occurs three times in the pattern of transformation Return If Arguments Are Equal.

In data mining terminology, an *item* corresponds to our code change and an *itembag* corresponds to our code change pattern. For brevity, items are usually represented as characters. The last column of Table 1 shows the items corresponding to every distinct code change. For example, item *c* corresponds to code change `(add, SimpleName)`, while item *d* — to code change `(add, MethodInvocation)`.

The number of repetitions of a pattern (itembag) is called *frequency*. For example, the frequency of a pattern containing only item *c* is 6. Note, however, that our goal is to detect full code change patterns rather than their parts (i.e., we would like to detect a pattern with code changes 3 – 9 rather than 3 – 7 or 5 – 8, or any other subset of code changes). Therefore, we mine *closed* itembags, i.e., itembags that are not part of a bigger itembag with the same frequency. In other words, a closed itembag represents the maximal size code change pattern for a given frequency.

To discuss how our problem of mining frequent code change patterns differs from the canonical one, we first present several definitions from the data mining domain.

Definition 3 (Transaction): A transaction is a tuple $< tid, X >$*, where tid is a unique transaction identifier and* X *is a set of items.*

Definition 4 (Transaction Database): A transaction database D *is a set of transactions.*

The canonical problem of mining frequent itemsets from a given transaction database D consists in finding all itemsets, whose frequency is not lower than a user-specified minimum frequency threshold. Thus, the off-the-shelf data mining algorithms are designed to mine itemsets rather than itembags. Also, they assume that transactions are disjoint.

For code change mining, a transaction is a window in which an algorithm looks for a pattern. The size of the window determines the maximum size of a pattern that a mining algorithm can detect. To mine the actual code changes, we use a time window, while for presentation purposes, we define the size of a window as the number of code changes. For example, let's consider that our window is of size eight. Then, according to Table 1, we have two disjoint windows (transactions), the first spans code changes 1 – 8, and the second — changes 9 – 16. As a result, the pattern with code changes 3 – 9 crosses the boundary between windows. Consequently, an off-the-shelf mining algorithm would not be able to correctly detect the whole pattern. To avoid this problem, we designed a mining algorithm that uses *overlapping* windows (transactions).

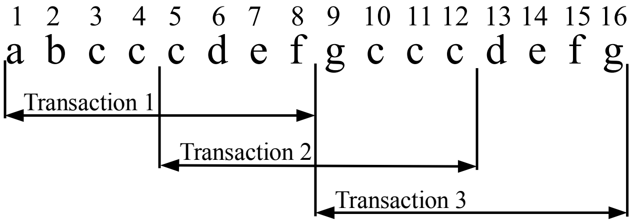


Figure 2: The overlapping transactions of size eight for the sequence of code changes from Table 1.

Figure 2 shows overlapping transactions of size eight for the sequence of code changes from Table 1. The number above each item reflects the order index of the corresponding code change. We use these numbers as IDs that help distinguish separate *occurrences* of the same item in the same transaction. For example, three occurrences of item *c* in the first transaction have IDs 3, 4, and 5.

Note, however, that just overlapping transactions of size eight is insufficient to detect code change patterns of size eight. In particular, Figure 2 shows that the pattern with code changes 3 – 9 still crosses the boundary between transactions. Consequently, the size of a transaction should be larger than the maximum-size pattern we are looking for. In the following section, we present the high-level overview of our mining algorithm and discuss handling of overlapping transactions and itembags in more detail.

4. OVERVIEW OF OUR ALGORITHM

Definition 5 (Vertical Data Format): The vertical data format represents a transaction database as a set of tuples $< item, tidset >$*, where tidset is a set of identifiers of transactions that contain the corresponding item.*

Table 2 shows the transaction database in the vertical data format for the items from Table 1 according to transactions in Figure 2. Note that to present the basic idea of the verti-

Table 2: The items from Table 1 in the vertical data format according to transactions in Figure 2.

Item	Tidset
a	$\{1\}$
b	$\{1\}$
c	$\{1, 2, 3\}$
d	$\{1, 2, 3\}$
e	$\{1, 2, 3\}$
f	$\{1, 2, 3\}$
g	$\{2, 3\}$

cal data format mining algorithm, we first disregard the fact that items in Table 1 form itembags rather than itemsets and are shared between overlapping transactions.

To accommodate such properties of code change pattern mining as overlapping transactions and itembags, it is crucial to access the transaction identifiers directly while computing new itemsets. The vertical data format grants such access to a mining algorithm. Therefore, our approach is inspired by several ideas from CHARM [38], an advanced algorithm for mining data in the vertical data format, which introduces the notion of itemset-tidset tree (IT-tree), employs several optimizations, and searches for closed itemsets, thus considerably reducing the size of the mining result. In particular, our algorithm extends the notion of IT-tree and adapts several optimization insights of CHARM. We first introduce the basic idea of mining data in the vertical data format and present the CHARM's definition of IT-tree. Then, we discuss how our approach builds upon CHARM to handle overlapping transactions and itembags.

Definition 6 (n-itemset): An n-itemset is an itemset that contains n items.

Definition 7 (Support of an Itemset): In a given transaction database D, the support of an itemset X, which we denote as $sup(X)$, is the number of transactions in D that contain X. That is, $sup(X) = |t(X)|$.

Eclat [37] is the first algorithm for mining data in the vertical data format without candidate generation. The basic idea of the algorithm is to compute $(n + 1)$-itemsets from n-itemsets by intersecting their tidsets. The algorithm starts with frequent 1-itemsets and finishes when no more frequent itemsets can be found. For example, let's consider two 1-itemsets, $\{a\}$ and $\{b\}$, from Table 2 (note that for any item x there is a corresponding 1-itemset $\{x\}$). The algorithm computes the tidset of a 2-itemset $\{a, b\}$ by intersecting the tidsets of $\{a\}$ and $\{b\}$: $t(\{a, b\}) = t(\{a\}) \cap t(\{b\}) = \{1\}$. The support of the itemset $\{a, b\}$ is 1: $sup(\{a, b\}) = |t(\{a, b\})| = 1$. If the minimum frequency threshold is greater than 1, then itemset $\{a, b\}$ is discarded. Otherwise, it is added to the results and considered for computing 3-itemsets.

The nodes in an IT-tree are pairs *itemset* : *tidset*. The root of the tree represents an empty itemset, and thus, its tidset is T, the set of all *tids*. The immediate children of the root node are 1-itemsets that are computed by scanning the transaction database. The immediate children of a non-root node are computed by intersecting this node's tidset with the tidsets of the *not yet considered* 1-itemsets, traversing them from left to right. If the resulting tidset's size falls below the minimum frequency threshold, the new node is not added to the IT-tree. The IT-tree is completed when no more nodes can be added to it. Figure 3 shows the partially completed first three levels of the IT-tree for the transaction database from Table 2. The first level of the tree consists of the 1-itemsets from the database that are paired with their *tids*, e.g., $\{a\} : \{1\}$, $\{b\} : \{1\}$, $\{c\} : \{1, 2, 3\}$, etc. The following levels are computed according to the procedure described above. For example, node $\{a, b\} : \{1\}$ is the result of combination of nodes $\{a\} : \{1\}$ and $\{b\} : \{1\}$.

Handling overlapping transactions and itembags. We mine frequent code change patterns from an ordered sequence of code changes (note that code changes are naturally ordered according to when they happened, i.e., according to their timestamps). To populate our transaction database, we divide this continuous sequence into individual transactions. Our code editing example in Section 2 shows that making transactions disjoint and sizing them according to the maximum length of a pattern, *max_length*, does not account for patterns that cross the boundary of two transactions. Therefore, we use *overlapping* transactions whose size is $2 * max\_length$. The size of the overlap between two neighboring transactions is *max_length*. As a result, our mining algorithm finds all patterns whose length does not exceed *max_length* and some patterns whose length lies in between *max_length* and $2 * max\_length$. Note that as an alternative to overlapping transactions of size $2 * max\_length$, we could employ a sliding window of transactions of size *max_length*. However, this approach would not simplify the algorithm and would lead to a significant increase in the number of transactions, thus undermining the algorithm's scalability.

Figure 2 shows overlapping transactions of size eight for the sequence of code changes from Table 1. Such choice of transactions ensures that the algorithm detects all patterns whose length does not exceed four. To detect the pattern with code changes 3 – 9, which is of size seven, the size of the transactions should be at least 14. We specify the size of a transaction as the number of code changes (i.e., items) for presentation purposes only, while for the actual mining, we set *max_length* to five minutes, and thus, transactions contained various numbers of items.

An important observation is that although code changes form an ordered sequence, a code change pattern is unordered because the corresponding high-level program transformation may be performed in different orders. For example, a developer who performs a Rename Local Variable refactoring might first change the variable's declaration and then its references or vice versa, or even intersperse changing the declaration and the references. Thus, the order of a transaction's items does not matter.

Another observation is that a high-level program transformation may contain several instances of the same kind of code change. For example, Rename Local Variable refactoring involves the same change for all references to the renamed variable. Section 2 presents an example of another high-level program transformation, Return If Arguments Are Equal, that also contains multiple instances of the same kind of code change, `(add, SimpleName)`. These examples show that for mining code change patterns, a transaction's items form a bag rather than a set. In particular, the first transaction in Figure 2 contains three items *c*.

The major difference between our frequent code change pattern mining algorithm and the existing approaches to mining frequent itemsets is that our algorithm handles overlapping transactions and itembags rather than itemsets. To distinguish different occurrences of an item in the same transaction as well as the overlapped parts of two transactions, our algorithm assigns a unique ID to each item's occur-

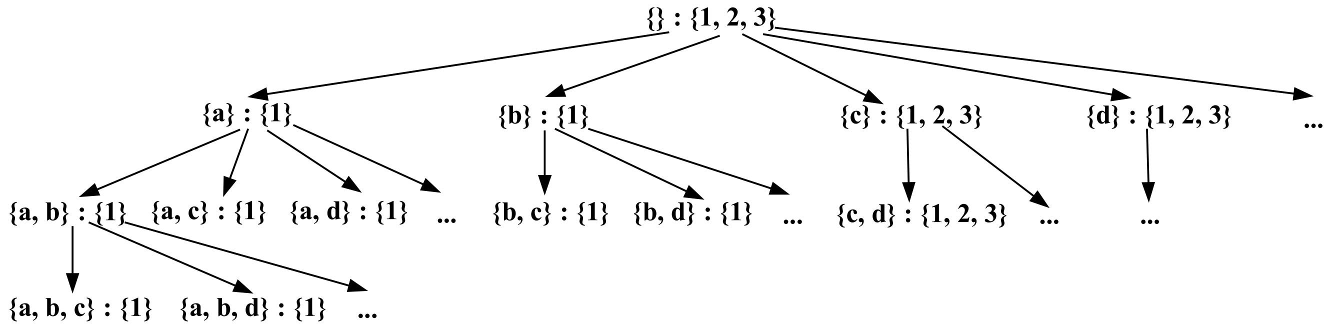


Figure 3: The partially completed first three levels of the IT-tree for the transaction database from Table 2.

rence. The first line in Figure 2 shows the IDs assigned to the underlying items' occurrences. For example, the first transaction contains occurrences of item c with IDs $\{3, 4, 5\}$, the second transaction — $\{5, 10, 11, 12\}$, and the third — $\{10, 11, 12\}$. Although our algorithm handles itembags, we use the notion of itemsets throughout our presentation, since the fact that our itemsets are actually itembags is accounted for by explicitly tracking each item's occurrences.

Tracking an item's occurrences. In order to track items' occurrences, a node in our itemset-tidset tree (IT-tree) is defined as follows:

$$\begin{aligned}
&[item_1, item_2, ..., item_n] : \\
&\quad [tid_1 : [[occurrences_1], [occurrences_2], ..., [occurrences_n]], \\
&\quad\; tid_2 : [[occurrences_1], [occurrences_2], ..., [occurrences_n]], \\
&\quad\; ..., \\
&\quad\; tid_m : [[occurrences_1], [occurrences_2], ..., [occurrences_n]]]
\end{aligned}$$

We use square brackets to denote ordered sets. The order of items in an itemset does not matter for a pattern, but it helps our algorithm to track occurrences of every item in each transaction that contains this itemset. Thus, we represent an n-itemset as an ordered set of items $[item_1, item_2, ..., item_n]$. For a given itemset, a node in an IT-tree contains an ordered set of tids of transactions that contain this itemset. Ordering of transactions enables our algorithm to effectively handle overlapping parts of the neighboring transactions. For each transaction, the IT-tree node also tracks all occurrences for every item in the given itemset (in the above representation, $[occurrences_i]$ are all occurrences of $item_i$ in a particular transaction). Our algorithm also orders an item's occurrences to ensure the optimal result of our itemset frequency computation technique that we discuss below.

Similarly to CHARM [38], we compute our IT-tree by traversing the 1-itemsets from left to right and intersecting the tidset of a particular itemset with the tidsets of the not yet considered 1-itemsets to generate new IT-tree nodes. The major difference from the CHARM's approach is that our algorithm tracks items' occurrences, and thus, whenever a new item is added to an itemset, the item's occurrences are appended to the set of occurrences of every transaction in the corresponding IT-tree node. Table 3 shows several examples of itemsets and their corresponding IT-tree nodes for the sequence of code changes from Figure 2. The third row presents the IT-tree node for itemset $\{c\}$. The node consists of the itemset itself, $[c]$, followed by tids of transactions that this itemset appears in — $1, 2, 3$. For each transaction, the node tracks all occurrences of item c: transaction 1 contains occurrences $3, 4, 5$, transaction 2 — $5, 10, 11, 12$, and transaction 3 — $10, 11, 12$. Note that storing an item's occurrences in every IT-tree node that contains this item is not only redundant, but also prohibitively expensive. Instead, our algorithm stores occurrences of individual items and then just refers these occurrences from the containing IT-tree nodes. We inline the referred occurrences for presentation purposes only.

Table 3: Examples of itemsets and their corresponding IT-tree nodes for code changes from Figure 2.

Itemset	IT-tree node
$\{a\}$	$[a] : [1 : [[1]]]$
$\{b\}$	$[b] : [1 : [[2]]]$
$\{c\}$	$[c] : [1 : [[3, 4, 5]], 2 : [[5, 10, 11, 12]], 3 : [[10, 11, 12]]]$
$\{a, c\}$	$[a, c] : [1 : [[1], [3, 4, 5]]]$

Computing the frequency of an itemset. Due to overlapping transactions and multiple occurrences of an item in the same transaction, our algorithm cannot compute the frequency of an itemset as the number of transactions that contain this itemset (as it is done in the existing frequent itemset mining techniques). Instead, we devised our own frequency computation technique that accounts for the particularities of our mining problem. In a given transaction k:

$tid_k : [[occurrences_1], [occurrences_2], ..., [occurrences_n]]$

the frequency of the corresponding itemset is:

$$f_k = \min_{i=1..n} |[occurrences_i]| \tag{1}$$

That is, f_k is the number of occurrences of an itemset's item that appears the least number of times. The *overall* frequency of an itemset contained in m transactions is:

$$F = \sum_{k=1}^{m} f_k \tag{2}$$

If an item occurrence is shared between two neighboring transactions, k and l, the algorithm should count this occurrence only once, either as part of f_k or as part of f_l. In an ordered set of transactions, two transactions, k and l, are neighboring if and only if $|k - l| = 1$ and $|tid_k - tid_l| = 1$. That is, the neighboring transactions follow each other both in the ordered set and in the original sequence of code changes. For example, in Figure 2 transactions of the ordered set $[1, 2]$ are neighboring, while transactions of the ordered set $[1, 3]$ are not neighboring.

Let's denote $[occurrences_i^k]$ the ordered set of occurrences of $item_i$ in a transaction k. Let's denote o_j an occurrence o with the index j in the ordered set of occurrences. To compute the frequency of an n-itemset that is contained in m transactions, our algorithm visits each pair of transactions k and $k + 1$, where $1 \leq k < m$. First, our algorithm computes f_k using formula (1). Then, if transactions k and $k + 1$

are neighboring, our algorithm visits every occurrence $o_j \in [occurrences_i^k]$, where $1 \leq i \leq n$. If $o \in [occurrences_i^{k+1}]$, then our algorithm checks whether the shared occurrence o should be removed from the transaction k or $k+1$. If $j \leq f_k$, then o is removed from the transaction $k + 1$. Otherwise, it is removed from the transaction k. Note that removing shared occurrences never affects the initially computed f_k). Finally, our algorithm computes the overall frequency using formula (2).

For the best performance of our algorithm, we order an item's occurrences such that those that happened earlier in time appear earlier in the ordered set. Since occurrences' IDs are generated incrementally (see Figure 2), such ordering is easily achieved by sorting occurrences in ascending order of their IDs. Consequently, the occurrences that are shared between transactions k and $k+1$ are placed at the end of the ordered set of all occurrences for the transaction k. Hence, our algorithm computes the maximal possible frequency for the transaction k employing the shared occurrences only when needed, while the unused part of them is attributed to the subsequent transaction $k+1$, thus maximizing its frequency too. Going through each pair of transactions k and $k + 1$, $1 \leq k < m$, our algorithm propagates this maximization, thus computing the optimal overall frequency F.

More details about our mining algorithm, including optimizations to handle large amounts of data, computation of closed itemsets, and establishing the frequency thresholds can be found in our technical report [22].

5. EVALUATION

In our evaluation, we would like to answer these questions:

- **Q1**(scalability): Is our algorithm scalable to handle large amounts of data?
- **Q2**(effectiveness): Does our algorithm mine code change patterns that simplify identification of high-level program transformations?
- **Q3**(usefulness): Are there useful high-level program transformations among the mined code change patterns?

To answer these questions, we applied our frequent code change pattern mining algorithm on a large corpus of real world data. In the following, we first describe how we collected the data and performed the evaluation of the algorithm. Then, we present our results.

5.1 Experimental Setup

To evaluate our mining algorithm, we first applied it on the previously collected code development data. We analyzed the mining results and identified ten kinds of high-level program transformations. Then, we performed a survey study to assess the popularity of the identified transformation kinds.

5.1.1 *Mining Code Changes*

We applied our algorithm on the data collected during our previous user study [23], which involved 23 participants: ten professional programmers who worked on different projects in domains such as marketing, banking, business process management, and database management; and 13 Computer Science graduate students and senior undergraduate summer interns who worked on a variety of research projects from six research labs at the University of Illinois at U-C.

According to the responses of our participants, 1, 4, 11, and 6 of them had 1 – 2, 2 – 5, 5 – 10, and more than 10 years of programming experience, respectively. In the course of our study, we collected code evolution data for 1,520 hours of code development with a mean distribution of 66 hours per developer and a standard deviation of 52.

The participants of our study installed the CodingTracker plug-in in their Eclipse IDEs. Throughout the study, CodingTracker recorded the detailed code evolution data ranging from individual code edits up to the high-level events like automated refactoring invocations. CodingTracker uploaded the collected data to our centralized repository using the existing infrastructure [33].

We first applied our AST node operations inference algorithm [23] on the collected raw data to represent code changes as *add*, *delete*, and *update* operations on the underlying AST. Next, we represented distinct kinds of code changes as combinations of the operation and the type of the affected AST node. For example, `(add, IfStatement)`, `(delete, IfStatement)`, and `(add, InfixExpression)` are three different kinds of code changes. The instances of code change kinds serve as input to our frequent code change pattern mining algorithm. That is, in our mining algorithm, a code change kind is an *item* and an instance of a code change kind is an item's *occurrence*.

For each mined code change pattern, our algorithm reports all occurrences of the pattern in the input sequence of code changes. We use CodingTracker's replayer to manually investigate these occurrences. We replay the code changes of a particular occurrence to detect the corresponding high-level program transformation.

Since the mining result is huge, we order the mined patterns along three dimensions: by frequency of the pattern (F), by size of the pattern (S), and by $F * S$. Then, we output the top 1,000 patterns for each dimension and investigate them starting from the top of the list.

We noticed that some items (i.e., atomic AST node operations like `(add, IfStatement)`) are much more frequent than the others. Thus, using fixed threshold values to analyze our data is not practical. If these values are too low, our algorithm's scalability would degrade, while the output would become disproportionately big. On the other hand, too high values would hinder the mining of patterns that involve less frequent items. Therefore, we divided the input items into three groups, applying different threshold values to each. Table 4 shows all three groups as well as the corresponding thresholds and the mining time. The *absolute frequency threshold* ensures that the frequency of a code change pattern does not fall below a particular absolute value. The *dynamic threshold* represents a pattern's frequency multiplied by its size, thus ensuring that the longer a pattern is, the less frequent it can be in order to pass the threshold. More details can be found in our technical report [22]. We performed all mining on a quad-core i7 2GHz machine with 8GB of RAM.

We observed that some AST node operations are too frequent to be considered at all. For example, adding and deleting `SimpleName` accompanies any code change that declares or references a program entity. Consequently, mining items that represent such AST node operations would only add noise to the detected code change patterns. Therefore, before applying our algorithm, we filtered out the noisy item kinds, thus reducing the number of item kinds from 162 to

Table 4: Grouping of item kinds by their frequency. Column NK shows the number of item kinds in each group. Columns AFT and DT show the values of the absolute frequency and dynamic thresholds.

Frequency, F	NK	AFT	DT	Mining time
$10,000 \leq F$	23	30	10,000	15 minutes
$300 \leq F < 10,000$	81	30	300	5.2 hours
$5 \leq F < 300$	32	5	5	7.7 seconds

138. According to Table 9, the total number of the considered item kinds is 136, which means that two item kinds were too infrequent to be part of any group.

5.1.2 *Survey Study*

To assess the popularity of the identified transformations, we conducted a survey study with 420 participants. To recruit our participants, we promoted the survey on Twitter and Google+ feeds that are mainly read by developers. We also released our survey on checkbox.io, a platform for software engineering empirical research.

For each transformation, our survey contains three questions — the first and the third are multiple-choice, while the second is Likert scale:

- Do you find this kind of change interesting, relevant, or applicable to your programming activities?
- How often have you manually performed this kind of change?
- Would you like your IDE to provide automated support for this change?

Tables 5, 6, 7, and 8 show that the majority of the developers who participated in our survey study are from industry, have more than five years of programming experience, and often employ refactoring and code completion features of their IDEs. The column **IDK** abbreviates the "I do not know" answer of our participants.

Table 5: Programming experience in years (%).

0 - 2	2 - 5	5 - 10	10 - 15	15 - 20	more than 20
2.87	21.05	31.34	25.60	7.18	11.96

Table 6: Software project type (%).

Open Source	Personal/Class	Proprietary	Research
7.18	10.77	76.32	5.74

Table 7: Usage of IDE refactoring features (%).

IDK	Never	Very rarely	Sometimes	Often
0.24	4.55	8.37	31.10	55.74

Table 8: Usage of IDE code completion features (%).

IDK	Never	Very rarely	Sometimes	Often
0.48	2.63	3.35	10.29	83.25

5.2 Results

Table 9 summarizes performance statistics of our experiment. Our algorithm mined more than half a million item occurrences in less than six hours, and thus, **the answer to the first question is that our mining algorithm is sufficiently scalable to handle large amounts of data with the appropriate threshold values**.

Table 9: Performance statistics of our experiment.

Item kinds	Transactions	Item occurrences	Total mining time
136	7,927	549,184	5.5 hours

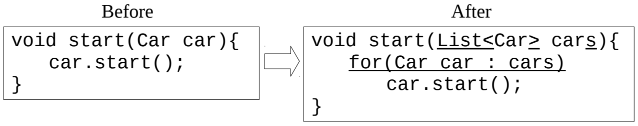
The frequent patterns mined by our algorithm helped us identify ten new kinds of program transformations. Note that among the mined patterns we encountered those that pointed to different kinds of refactorings (mostly Field Rename, Method Rename, and Change Method Signature), but we disregarded them in this evaluation, since our goal was to focus on new transformation kinds. The second author randomly picked 34 pattern occurrences among the top mined patterns ordered by frequency, size, and frequency multiplied by size. Out of 34 investigated occurrences, 11 were *fruitful*, i.e., the second author could describe them as high-level program transformations. Thus, 32% of investigated pattern occurrences were fruitful. Therefore, **our answer to the second question is that our algorithm is effective — it mines patterns that often lead to discovery of new high-level program transformations**. Note that the number of fruitful pattern occurrences is a lower bound. Although the second author could not give meaning to 68% of investigated occurrences, some of them might be considered high-level program transformations by a developer more familiar with the underlying code.

Table 10 shows the identified kinds of program transformations grouped according to their scope. The last two columns of the table show the number of pattern occurrences that we investigated and the number of pattern occurrences that led to the discovery of the corresponding program transformations. In the following, we present the discovered transformation kinds in more detail.

Table 10: Identified kinds of program transformations. Column I shows the number of the investigated pattern occurrences. Column F shows the number of pattern occurrences that were fruitful.

Scope	Identified program transformation	I	F
Statement	Convert Element to Collection	5	2
Loop	Add a Loop Collector	3	1
Method	Add Null Check for a Parameter	5	1
	Wrap Code with Timer	2	1
Class	Add a New Enum Element	2	1
	Change and Propagate Field Type	3	1
	Change Field to ThreadLocal	2	1
	Copy Field Initializer	2	1
	Create and Initialize a New Field	4	1
	Move Interface Implementation to Inner Class	6	1

Convert Element to Collection. This is a statement-level transformation in which a developer converts a field, parameter, or a local variable of a certain type into a collection (e.g., list, set, array, etc.) of that type. All references to the element need to be updated accordingly.

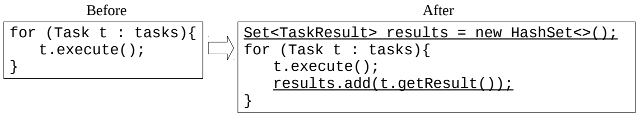

Before

```
void start(Car car){
   car.start();
}
```

After

```
void start(List<Car> cars){
   for(Car car : cars)
      car.start();
}
```

Add a Loop Collector. This is a transformation in which a developer introduces a new variable that collects or aggregates the data processed in a loop.



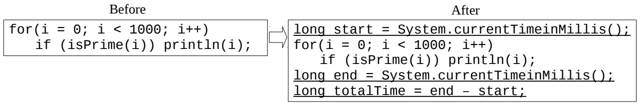
Before

```
for (Task t : tasks){
    t.execute();
}
```

After

```
Set<TaskResult> results = new HashSet<>();
for (Task t : tasks){
    t.execute();
    results.add(t.getResult());
}
```

Wrap Code with Timer. A developer applies this transformation to compute the execution time of a code fragment, e.g., a loop. The developer surrounds the code with variables that hold the time before and after the code's execution and outputs the time difference.



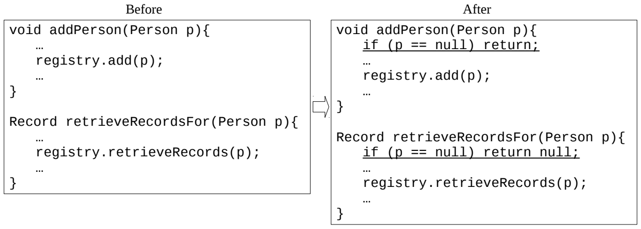
Before

```
for(i = 0; i < 1000; i++)
    if (isPrime(i)) println(i);
```

After

```
long start = System.currentTimeinMillis();
for(i = 0; i < 1000; i++)
    if (isPrime(i)) println(i);
long end = System.currentTimeinMillis();
long totalTime = end – start;
```

Add Null Check for a Parameter. This is a transformation in which a developer adds `null` precondition checks to all methods of a class that receive a particular parameter such that the methods' bodies are not executed if the parameter is `null`.



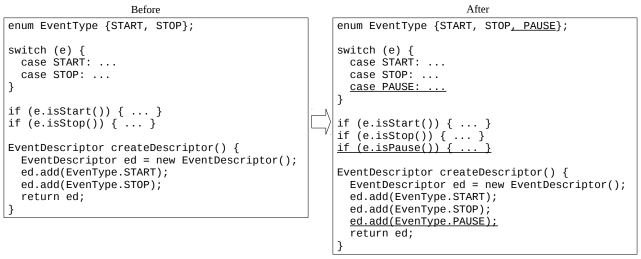
Before

```
void addPerson(Person p){
    …
    registry.add(p);
    …
}

Record retrieveRecordsFor(Person p){
    …
    registry.retrieveRecords(p);
    …
}
```

After

```
void addPerson(Person p){
    if (p == null) return;
    …
    registry.add(p);
    …
}

Record retrieveRecordsFor(Person p){
    if (p == null) return null;
    …
    registry.retrieveRecords(p);
    …
}
```

Add a New Enum Element. Adding a new element to `enum` triggers a ripple of changes such as adding new `switch` cases, `if-then-else` chains, and dealing with any duplicated code that uses the updated `enum`.



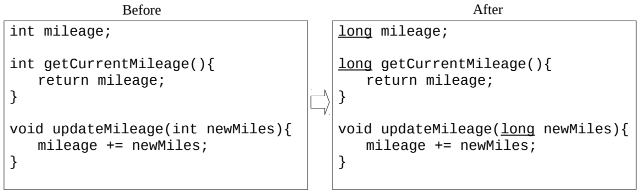
Before

```
enum EventType {START, STOP};

switch (e) {
  case START: ...
  case STOP: ...
}

if (e.isStart()) { ... }
if (e.isStop()) { ... }

EventDescriptor createDescriptor() {
  EventDescriptor ed = new EventDescriptor();
  ed.add(EvenType.START);
  ed.add(EvenType.STOP);
  return ed;
}
```

After

```
enum EventType {START, STOP, PAUSE};

switch (e) {
  case START: ...
  case STOP: ...
  case PAUSE: ...
}

if (e.isStart()) { ... }
if (e.isStop()) { ... }
if (e.isPause()) { ... }

EventDescriptor createDescriptor() {
  EventDescriptor ed = new EventDescriptor();
  ed.add(EvenType.START);
  ed.add(EvenType.STOP);
  ed.add(EvenType.PAUSE);
  return ed;
}
```

Change and Propagate Field Type. This is a transformation in which a developer changes the type of a field. As a result, the developer also has to update the type of some local variables as well as the return type of some methods.



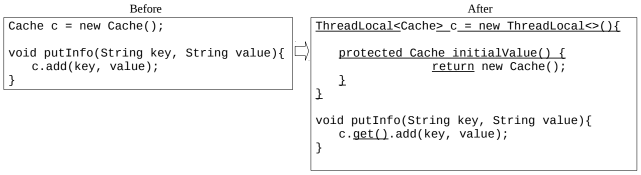
Before

```
int mileage;

int getCurrentMileage(){
    return mileage;
}

void updateMileage(int newMiles){
    mileage += newMiles;
}
```

After

```
long mileage;

long getCurrentMileage(){
    return mileage;
}

void updateMileage(long newMiles){
    mileage += newMiles;
}
```

Change Field to ThreadLocal. To improve thread safety of an application, a developer may decide to convert some fields to `ThreadLocal`. Besides changing the type and the initialization of the converted field, the developer also has to modify all field's accesses such that they use `get()` and `set()` of `ThreadLocal`.



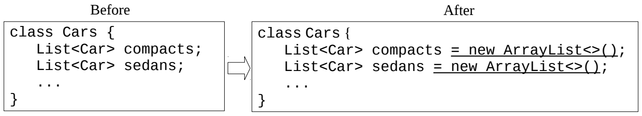
Before

```
Cache c = new Cache();

void putInfo(String key, String value){
   c.add(key, value);
}
```

After

```
ThreadLocal<Cache> c = new ThreadLocal<>(){

   protected Cache initialValue() {
             return new Cache();
   }
}

void putInfo(String key, String value){
   c.get().add(key, value);
}
```

Copy Field Initializer. This is a transformation in which a developer copies the same initializer to several fields.



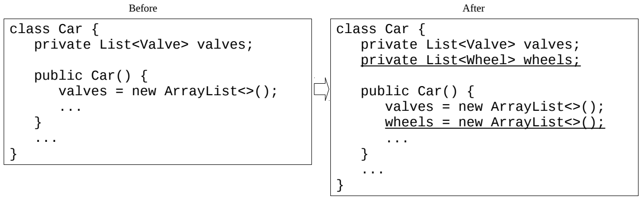
Before

```
class Cars {
   List<Car> compacts;
   List<Car> sedans;
   ...
}
```

After

```
class Cars {
   List<Car> compacts = new ArrayList<>();
   List<Car> sedans = new ArrayList<>();
   ...
}
```

Create and Initialize a New Field. When a developer adds a new field, it has to be properly initialized alongside the already present fields in constructors and other initialization places (e.g., static initialization blocks).



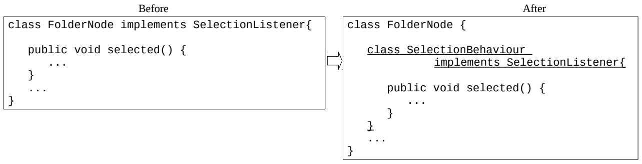
Before

```
class Car {
   private List<Valve> valves;

   public Car() {
      valves = new ArrayList<>();
      ...
   }
   ...
}
```

After

```
class Car {
   private List<Valve> valves;
   private List<Wheel> wheels;

   public Car() {
      valves = new ArrayList<>();
      wheels = new ArrayList<>();
      ...
   }
   ...
}
```

Move Interface Implementation to Inner Class. In this transformation, a developer moves the implementation of an interface from a class to its newly created inner class.



Before

```
class FolderNode implements SelectionListener{

   public void selected() {
      ...
   }
   ...
}
```

After

```
class FolderNode {

   class SelectionBehaviour
            implements SelectionListener{

      public void selected() {
         ...
      }
   }
   ...
}
```

The mined code change patterns helped us identify ten kinds of interesting program transformations whose scopes range from individual statements to whole classes. **Thus, our answer to the third question is that our algorithm is useful**.

5.2.1 *Survey Study Results*

Table 11 ranks ten transformation kinds identified by our mining algorithm according to the percentage of our survey participants who reported them as relevant. More than half of our participants recognized eight out of ten transformations as relevant to their programming activities.

Table 12 shows that more than 50% of developers who completed our survey regularly (columns **Sometimes** and **Often**) applied six out of ten transformation kinds.

Table 13 shows that more than 50% of our survey study participants would like to see five out of ten transformation kinds automated in their IDEs. Similarly to the existing automated refactorings, our automated transformations would be interactive: the developer will guide the automated execution of a transformation and provide the required input values, e.g., specifying functionality of a new case statement introduced by Add a New Enum Element transformation.

Table 11: Ranking of transformation kinds according to the percentage of our survey participants who reported them as interesting, relevant, or applicable to their programming activities.

Change Field Type	93.15
Create and Initialize a New Field	86.68
Add Precondition Checks for a Parameter	76.64
Add a New Enum Element	75.91
Wrap Code with Timer	60.77
Add a Loop Collector	60.49
Copy Field Initializer	56.97
Convert Element to Collection	55.23
Move Interface Implementation to Inner Class	38.40
Change Field to ThreadLocal	28.15

Table 12: The fraction of our survey study participants who applied the identified transformation kinds with different frequency (%).

	IDK	Never	Very rarely	Sometimes	Often
Create and Initialize...	0.49	2.93	10.76	30.07	55.75
Change Field Type	0.00	1.23	12.07	37.19	49.51
Add Precondition ...	0.49	4.90	22.30	25.98	46.32
Add a New Enum ...	1.23	6.63	31.94	34.89	25.31
Copy Field Initializer	1.00	12.75	35.25	30.75	20.25
Add a Loop Collector	1.49	10.95	33.83	34.83	18.91
Wrap Code with ...	0.49	10.95	41.36	28.95	18.25
Convert Element ...	0.98	9.80	52.94	27.45	8.82
Move Interface Impl...	2.01	34.92	37.19	19.35	6.53
Change Field to Thr...	1.75	40.15	44.39	11.22	2.49

6. THREATS TO VALIDITY

In our experiment, we used the output of the AST node operations inference algorithm [23] to prepare the input to our frequent code change pattern mining algorithm. Consequently, imprecisions in the inferred AST node operations could negatively affect our mining results. Note, however, that our approach to mining frequent code change patterns is independent of the way its input is produced.

We investigated the mined patterns manually, and thus, might have missed some of their corresponding high-level program transformations. Also, we investigated only a fraction of the mining results. However, our experiment did not aim at discovering all program transformations performed by our participants. Instead, our goal was to show that our algorithm mines patterns that effectively point to high-level program transformations, and we believe that discovering several such transformations in a reasonable amount of time (identifying and documenting these transformations took just a couple of days) supports this claim.

In our study, we collected code evolution data from developers who use Eclipse for Java programming. Consequently, the identified high-level program transformations might not be generalizable to other programming environments and languages. Nevertheless, our approach to identifying such transformations is orthogonal to the way developers make their code changes.

Our dataset is not publicly available due the nondisclosure agreement with our participants.

7. RELATED WORK

7.1 Mining Frequent Itemsets

The major challenge in mining frequent itemsets is to develop scalable algorithms that can effectively handle large transaction databases. One of the fundamental distinctions between different approaches to mining frequent itemsets is whether mining is performed with or without candidate generation. Agrawal et al. [1] observed that an n-itemset is frequent only if all its subsets are also frequent. Their mining algorithm, Apriori, leverages this property by using frequent n-itemsets to generate $(n + 1)$-itemset candidates. Apriori checks the newly generated candidates against the transaction database to establish those of them that are frequent. The algorithm starts with detecting frequent 1-itemsets directly from the transaction database and proceeds iteratively until no more frequent itemsets can be found.

Table 13: Survey study participants' preference for automated IDE support (%).

Change Field Type	86.42
Create and Initialize a New Field	74.33
Add a New Enum Element	60.20
Add Precondition Checks for a Parameter	60.10
Wrap Code with Timer	57.32
Copy Field Initializer	44.22
Convert Element to Collection	43.52
Add a Loop Collector	42.12
Move Interface Implementation to Inner Class	33.42
Change Field to ThreadLocal	24.69

Mining with candidate generation has two major drawbacks: a) it generates redundant itemsets that are found to be infrequent; b) it repeatedly scans the transaction database while progressing through the iterations. Mining without candidate generation addresses both these limitations. Such mining can be broadly divided into mining using *horizontal data format* and mining using *vertical data format*. The horizontal data format represents a transaction database as a set of tuples $< TransactionID, itemset >$. Han et al. [13] suggested to mine frequent itemsets without candidate generation using horizontal data format. Their frequent-pattern growth (FP-growth) algorithm first scans the database to detect frequent 1-itemsets. The algorithm uses these 1-itemsets to construct FP-tree, an extended prefix-tree structure. Then, the algorithm expands the initial FP-tree by growing pattern fragments in a recursive fashion.

Zaki [37] proposed a different approach to mining frequent itemsets without candidate generation. His algorithm, Eclat, explores the vertical data format, which explicitly stores transactions' identifiers (tidsets) for every itemset (like in Table 2). Eclat computes $(n + 1)$-itemsets from n-itemsets by intersecting their tidsets. The algorithm collects the initial set of frequent 1-itemset by scanning the transaction database. Our technical report [22] compares the horizontal vs. vertical data formats and the corresponding algorithms in more detail.

Subsequently, Zaki [38] developed CHARM, a more advanced algorithm for mining data in the vertical data format. The algorithm is based on the same idea of intersecting itemsets' tidsets to produce new itemsets, but it specifies the search problem using the notion of itemset-tidset tree (IT-tree). Also, CHARM introduces several optimizations, including the search for closed itemsets.

All the approaches above operate on a database with disjoint transactions, each containing a set of items. On the contrary, our frequent code change pattern mining algorithm handles *overlapping* transactions and *itembags* rather than itemsets, which are the two major challenges specific to frequent code change pattern mining from *continuous* sequence of code changes.

7.2 Mining Source Code

Source code mining research has a long history. Here, we present several representative examples.

Michail [21] applied data mining techniques to detect how a library is reused in different applications. The mined library reuse patterns, represented as association rules, facilitate the reuse of the library components by developers.

Li et al. [19] employed frequent itemset mining to extract programming rules from the source code of an application. They also showed that source code fragments that violate the extracted rules are likely to be buggy.

Holmes et al. [15] matched the structural context of the edited source code against a code repository to present a developer with the examples demonstrating the relevant API usage. Similarly, Bruch et al. [3] proposed to improve the IDE's code completion systems by making them learn from code repositories.

Andersen et al. [2] inferred code changes from several examples of code before and after editing. The inferred transformations are context-sensitive and thus, could be applied to matching code contexts.

Hovemeyer et al. [16] developed FindBugs, a tool that detects a variety of bug patterns in an application by statically matching bug pattern descriptions against the underlying source code. Lin et al. [20] proposed an approach to search for a specific kind of bug patterns — violations of check-then-act idioms.

All these approaches mine the application's source code, while our algorithm mines *fine-grained* code changes.

Another direction of research is mining source code change patterns from the Version Control System (VCS) history of an application. Ying et al. [35] and Zimmermann et al. [39] apply data mining techniques on the application's revision history to detect software artifacts (e.g., methods, classes, etc.) that are usually changed together. The mined association rules predict what other source code locations a developer needs to consider while performing a particular change. Uddin et al. [32] proposed to mine VCS histories of client applications to study how their use of APIs evolves over time, which is helpful both to developers and users of the libraries' APIs. Canfora et al. [4, 5] and Thummalapenta et al. [30] used VCS snapshots to study and track the evolution of different software entities such as source lines, bugs, and clones. More recently, Nguyen et al. [25] extracted method-level code changes from revision histories of Java projects and studied their within-project and cross-project repetitiveness. In the domain of software testing, Zaidman et al. [36] mined software repositories to explore how production and test code co-evolve.

Mining VCS snapshots of an application is exposed to the limited nature of VCS data. In our previous study [23], we showed that data stored in VCS is *imprecise*, *incomplete*, and makes it *impossible* to perform analysis that involves the time dimension inside a single VCS snapshot. Also, similarly to other source code mining techniques, these approaches mine *static* source code, while our algorithm mines *dynamic* code changes.

7.3 Automated Inference of Refactorings

Early work by Demeyer et al. [7] inferred refactorings by comparing two different versions of source code using heuristics based only on low-level software metrics — method size, class size, and inheritance levels. Kim et al. [18] used a function similarity algorithm to detect methods that have been renamed. More recent refactoring inference approaches detect refactorings depending on how well they match a set of *characteristic properties* that are constructed from the differences between two consecutive versions of an application. Dig et al. [8] employed references of program entities like instantiation, method calls, and type imports as its set of characteristic properties. Weißgerber and Diehl [34] used characteristic properties based on names, signature analysis, and clone detection. Prete et al. [28] developed Ref-Finder, a tool that can infer the widest variety of refactorings to date — up to 63 of the 72 refactorings cataloged by Fowler [10]. Their set of characteristic properties involved accesses, calls, inherited fields, etc.

All these approaches infer refactorings from VCS snapshots, and thus, suffer from the limitations of VCS data. Also, they mine *static* source code and consider refactorings only.

Recently, Ge et al. [11] and Foster et al. [9] proposed tools that continuously monitor code changes to detect and complete manual refactorings in *real-time*. Although this direction of research is very promising, the proposed tools are limited to a single kind of program transformations — refactorings, and detect a small subset of already known refactorings. On the contrary, our algorithm is not restricted to any specific kind of program transformations and is designed to detect *previously unknown* code change patterns.

Wit et al. [6] performed live monitoring of the clipboard to detect clones. Their tool tracked the detected clones, offering several resolution strategies whenever the clones were edited inconsistently. Our approach of detecting similar changes to different parts of the code is complementary to detecting different changes to the similar parts of the code.

8. CONCLUSIONS

Although mining frequent code change patterns has a long research history, we are the first to present an algorithm that mines *previously unknown* patterns from a *fine-grained* sequence of code changes. Our algorithm effectively handles *overlapping* transactions that contain multiple instances of the same item kind — the major challenge that distinguishes our approach from the existing frequent itemset mining techniques.

To evaluate our algorithm, we used 1,520 hours of real world code changes that we collected from 23 developers. Our experiment showed that our mining algorithm is scalable, effective, and useful. Analyzing some of the mining results, we identified ten popular kinds of high-level program transformations. To assess the popularity of the identified transformations, we conducted a survey study with 420 participants. More than half of the survey participants recognized the relevance of eight out of ten transformation kinds in their daily development activities.

Acknowledgments. We would like to thank Jiawei Han for his guidance in the field of data mining. We also thank Darko Marinov and students in the Software Engineering seminar at UIUC and the Software Evolution group at OSU for insightful comments on earlier drafts of this paper. This work was partially supported by the National Science Foundation grants number CCF-1117960 and CCF-1213091.

9. REFERENCES

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *VLDB*, 1994.

[2] J. Andersen, A. C. Nguyen, D. Lo, J. L. Lawall, and S.-C. Khoo. Semantic patch inference. In *ASE*, 2012.

[3] M. Bruch, M. Monperrus, and M. Mezini. Learning from examples to improve code completion systems. In *FSE*, 2009.

[4] G. Canfora, M. Ceccarelli, L. Cerulo, and M. D. Penta. How long does a bug survive? an empirical study. In *WCRE*, 2011.

[5] G. Canfora, L. Cerulo, and M. D. Penta. Tracking your changes: A language-independent approach. In *IEEE Software*, 2009.

[6] M. de Wit, A. Zaidman, and A. van Deursen. Managing code clones using dynamic change tracking and resolution. In *ICSM*, 2009.

[7] S. Demeyer, S. Ducasse, and O. Nierstrasz. Finding refactorings via change metrics. In *OOPSLA*, 2000.

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

[9] S. Foster, W. G. Griswold, and S. Lerner. WitchDoctor: IDE Support for Real-Time Auto-Completion of Refactorings. In *ICSE*, 2012.

[10] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[11] X. Ge, Q. L. DuBose, and E. Murphy-Hill. Reconciling manual and automatic refactoring. In *ICSE*, 2012.

[12] J. Han, H. Cheng, D. Xin, and X. Yan. Frequent pattern mining: current status and future directions. *Data Min. Knowl. Discov.*, 15, 2007.

[13] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *SIGMOD*, 2000.

[14] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu. On the naturalness of software. In *ICSE*, 2012.

[15] R. Holmes, R. J. Walker, and G. C. Murphy. Approximate structural context matching: An approach to recommend relevant examples. *IEEE Trans. Softw. Eng.*, 32, 2006.

[16] D. Hovemeyer and W. Pugh. Finding bugs is easy. In *OOPSLA*, 2004.

[17] L. Jiang and Z. Su. Automatic mining of functionally equivalent code fragments via random testing. In *ISSTA*, 2009.

[18] S. Kim, K. Pan, and J. W. Jr. When functions change their names: Automatic detection of origin relationships. In *WCRE*, 2005.

[19] Z. Li and Y. Zhou. Pr-miner: automatically extracting implicit programming rules and detecting violations in large software code. In *FSE*, 2005.

[20] Y. Lin and D. Dig. Check-then-act misuse of java concurrent collections. In *ICST*, 2013.

[21] A. Michail. Data mining library reuse patterns in user-selected applications. In *ASE*, 1999.

[22] S. Negara, M. Codoban, D. Dig, and R. E. Johnson. Mining continuous code changes to detect frequent program transformations. Technical Report http://hdl.handle.net/2142/43889, University of Illinois at Urbana-Champaign, 2013.

[23] S. Negara, M. Vakilian, N. Chen, R. E. Johnson, and D. Dig. Is it dangerous to use version control histories to study source code evolution? In *ECOOP*, 2012.

[24] A. T. Nguyen, T. T. Nguyen, H. A. Nguyen, A. Tamrawi, H. V. Nguyen, J. Al-Kofahi, and T. N. Nguyen. Graph-based pattern-oriented, context-sensitive source code completion. In *ICSE*, 2012.

[25] H. A. Nguyen, A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, and H. Rajan. A study of repetitiveness of code changes in software evolution. In *To appear in ASE*, 2013.

[26] C. Omar, Y. Yoon, T. D. LaToza, and B. A. Myers. Active code completion. In *ICSE*, 2012.

[27] H. T. T. Petteri Sevon and P. Onkamo. Gene mapping by pattern discovery. In *Data Mining in Bioinformatics*, 2005.

[28] K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim. Template-based reconstruction of complex refactorings. In *ICSM*, 2010.

[29] P. Sevon, H. Toivonen, and V. Ollikainen. TreeDT: Tree pattern mining for gene mapping. *IEEE/ACM Trans. Comput. Biol. Bioinformatics*, 3, 2006.

[30] S. Thummalapenta, L. Cerulo, L. Aversano, and M. D. Penta. An empirical study on the maintenance of source code clones. In *Empirical Software Engineering*, 2010.

[31] H. Toivonen, P. Onkamo, P. Hintsanen, E. Terzi, and P. Sevon. Data mining for gene mapping. In *Next Generation of Data Mining Applications*, 2005.

[32] G. Uddin, B. Dagenais, and M. P. Robillard. Analyzing temporal api usage patterns. In *ASE*, 2011.

[33] M. Vakilian, N. Chen, S. Negara, B. A. Rajkumar, B. P. Bailey, and R. E. Johnson. Use, disuse, and misuse of automated refactorings. In *ICSE*, 2012.

[34] P. Weißgerber and S. Diehl. Identifying refactorings from source-code changes. In *ASE*, 2006.

[35] A. T. Ying, G. C. Murphy, R. Ng, and M. C. Chu-Carroll. Predicting source code changes by mining change history. *IEEE Trans. Softw. Eng.*, 30, 2004.

[36] A. Zaidman, B. V. Rompaey, S. Demeyer, and A. van Deursen. Mining software repositories to study co-evolution of production & test code. In *ICST*, 2008.

[37] M. J. Zaki. Scalable algorithms for association mining. *IEEE Transactions on Knowledge and Data Engineering*, 2000.

[38] M. J. Zaki and C.-J. Hsiao. CHARM: An efficient algorithm for closed itemset mining. In *SDM*, 2002.

[39] T. Zimmermann, P. Weissgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 31, 2005.