Interactive Code Snippet Synthesis Through Repository Mining

Zvonimir Pavlinovic
Domagoj Babic

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2013-23
http://www.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-23.html

March 29, 2013

Interactive Code Snippet Synthesis Through Repository Mining

Zvonimir Pavlinović
UC Berkeley
zvonimir@eecs.berkeley.edu

Domagoj Babić
Facebook, Inc.
babic.domagoj@gmail.com

Abstract

Programmers repeatedly reuse code snippets. Retyping boilerplate code, and rediscovering how to correctly sequence API calls, programmers waste time. In this paper, we develop techniques that automatically synthesize code snippets upon a programmer's request. Our approach is based on discovering snippets located in repositories; we mine repositories offline and suggest discovered snippets to programmers. Upon request, our synthesis procedure uses programmer's current code to find the best fitting snippets, which are then presented to the programmer. The programmer can then either learn the proper API usage or integrate the synthesized snippets directly into her code. We call this approach interactive code snippet synthesis through repository mining. We show that this approach reduces the time spent developing code for 32% in our experiments.

Categories and Subject Descriptors I.2.2 [*Artificial Intelligence*]: Automatic Programming–Program Synthesis; D.2.2 [*Software Engineering*]: Design Tools and Techniques–Computer-aided software engineering

General Terms Languages, Algorithms

Keywords program synthesis, code mining, code reuse

1. Introduction

Software development is expensive. In the USA, there were approximately one million software developers in 2010 [1]. Considering developers' salaries alone, cost of software development amounts to over hundred billion dollars per year, not including outsourcing. Therefore, even modest improvements in programmers' efficiency promise significant savings. Our work focuses on improving programmers' efficiency by eliminating repetitive coding tasks, like retyping boilerplate code snippets, and speeding up the learning of unfamiliar APIs.

Indeed, a substantial portion of coding is a repetitive activity [2]. In our user study, programmers repeatedly used code snippets available locally or online to solve their programming tasks. More precisely, we observed two main types of snippets. The first type are boilerplate snippets, which represent generic pieces of code that reappear frequently. For instance, such snippets are common in testing code. Instead of writing test cases from scratch, programmers often reused code from previously written test cases. The second type are snippets containing frequent sequences of API calls, representing common usage patterns. In order to learn the proper usage of unfamiliar APIs, programmers searched for snippets available online. Both types of snippets have one thing in common: they waste programmers' time.

Learning the correct usage of an unfamiliar API is time consuming [3], as APIs are often complex and poorly documented. Programmers hence invest significant effort in developing even small snippets that use such APIs, which is why we wish to generate such snippets automatically on demand. For instance, consider the following piece of code that uses Eclipse's `ASTParser` API [4]:

```
ASTParser parser =
                ASTParser.newParser(AST.JLS3);
char[] source = readInputCode();
parser.setSource(source);
CompilationUnit unit = (CompilationUnit)
                       parser.createAST(null);
```

Although a quite short one, this snippet took our test subjects almost 25 minutes to write. Moreover, previous research [5] also found that writing this example is time consuming. Development of this piece of code is delicate because of `ASTParser`'s inadequate documentation and lack of available code samples. In contrast, using our tool that synthesizes code snippets upon request, our test subjects developed the above snippet in just few minutes.

Reusing boilerplate snippets wastes time as well. Programmers waste time on manually locating and integrating appropriate snippets, instead of focusing on the core program logic. To illustrate this, we use the following test case:

```
FileTester f =
      TestFactory.createFileTester(filePath);
f.assertExists();
f.assertContentType(FileContent.XML);
f.assertSize(1024, PredicateOperator.LESS);
```

This code sample is a concrete instance of a generic code template used for writing test cases that check basic file properties. Discovering this code sample manually took our

test subjects 13 minutes on average, while only 5 minutes when using our synthesis approach.

A considerable amount of research has been done on automatic generation of code snippets [5–11]. Most of the techniques are autocompleting only a single line of code [5, 6, 10, 11]. Existing multi-line synthesis approaches are either inaccurate [8] or have slow response time [7, 9], in the order of tens of seconds. Some of the approaches, like [5, 7, 9], require programmers to manually provide input of a special format, thus hindering adoption. Ideally, programmers' interaction with a synthesis procedure should be simple, and the synthesis procedure fast and accurate. In this paper, we introduce a new approach that reasonably accurately synthesizes multi-line code snippets, requires minimal interaction, and is effective in practice. Our work is based on mining code repositories; we mine frequent repository snippets and then use them to synthesize suggestion snippets upon a programmer's request. We call this approach interactive code snippet synthesis through repository mining, or shortly INSYREM.

The INSYREM technique works in two phases. The first phase, executed offline, mines frequent code snippets from a given repository. Our mining algorithm searches for snippets that occur more often than a given occurrence threshold and are syntactically correct. After the mining process has finished, discovered snippets are compressed into a succinct representation that will be used to synthesize suggestion snippets. The second phase is triggered by programmers' requests. Programmers request suggestions by simply positioning the cursor appropriately and pressing a shortcut. The code before the cursor is then compared to the discovered snippets, yielding the best fitting code suggestions that the programmers can either learn from or directly integrate. We show by experiments that this technique increases programmers' productivity. In particular, we make the following contributions:

- We identify and describe two types of snippets programmers frequently use when developing code.
- We describe a practical code mining algorithm that finds frequent code snippets.
- We introduce the INSYREM approach for automatic generation of snippets and report results that show a significant increase in programmers' productivity.

2. Overview

In this section, we give an end-to-end description of our approach using the first example from Section 1. As mentioned earlier, INSYREM operates in two phases. The first phase mines frequent code snippets and builds their compressed representation — a *snippet index*. This phase is done offline. The second phase is initiated by programmers, whose code before the cursor is then used to find the best fitting snippets in the snippet index. Our synthesis algorithm rewrites the found snippets so as to replace variables in snippets with variables live at the point where the cursor is in the program, while matching types. Resulting snippets are then presented to programmers. We now describe each phase in more detail using the given example.

2.1 Snippet Mining

The first snippet from Section 1 begins with a statement that constructs a new `ASTParser`. Our approach mines the available repositories for such constructors. To locate such repositories, one can use tools like S6 [12]. In our approach, administrators or programmers themselves provide repositories that contain code using APIs of interest, as well as an occurrence threshold. For simplicity, we assume that provided repositories are joined into a single repository. Prior to mining, the repository code is transformed into an abstract syntax tree-like data structure that simplifies snippet comparison, needed for computing a number of occurrences with which various snippets appear in the repository.

The mining algorithm first finds statements that occur more often than a given occurrence threshold, and then proceeds by finding such statement sequences, i.e., snippets. We will refer to such statements and snippets as *frequent*. During the mining, the algorithm also comes across snippets that are very similar and differ only in, for instance, parameters to API calls or subexpressions. The mining algorithm enumerates all syntactically correct snippets abstracted by substituting parameters and expressions with a placeholder symbol. Those snippet abstractions are then treated as any other snippets. This way, we discover similar snippets that would individually get classified as infrequent.

Naïvely enumerating and memoizing all snippet abstractions would quickly exhaust all available memory. This problem is exacerbated when occurrence thresholds are low. The mining algorithm saves memory by disregarding frequent snippets that are abstractions of other frequent snippets. Such abstractions are redundant since information they contain is already present in their concretizations, which occur often enough alone to be identified as frequent. At the end, the mining procedure builds a compressed representation of all frequent snippets by exploiting shared statements and expressions.

2.2 Suggestion Synthesis

To describe the synthesis procedure, we use a programmer's request observed while investigating the INSYREM approach. Consider the partial code given in Figure 1a. The "|" symbol denotes the current cursor position. Programmers initiate a synthesis request by positioning the cursor at a desired position and pressing a shortcut. The synthesis procedure then collects information needed for suggestion synthesis: code located between the cursor and the beginning of a current method, and all variables live at the point where the cursor is.

```
    ...
a)  ASTParser parser = |
```

```
    ASTParser parser = ASTParser.newParser(AST.JLS3);
b)  char[] source = <?>;
    parser.setSource(source);
    CompilationUnit u = (CompilationUnit) parser.createAST(null);
```

Figure 1. An example of a) programmer's input and b) the code synthesized by INSYREM.

The synthesis procedure proceeds by searching for statements in the snippet index that resemble the statement located immediately before the cursor (see Figure 1a). We refer to such statements in the snippet index as *fixed statements*. The rest of programmers' code, above the cursor line, is compared to statements in the snippet index that precede fixed statements; the number of matched statements and expressions is later used to rank autocompletion suggestions. Suggestion snippets are generated by taking code from the snippet index that comes after fixed statements and rewriting it so as to include programmers' live variables. The suggestions are then sorted heuristically and presented to programmers. The best ranked snippet for the running example is shown in Figure 1b. The `<?>` symbol will be described later. The reader might notice the resemblance to the troublesome first example from Section 1. Once synthesized, programmers can integrate the snippet directly into their code, obtaining the desired functionality in the matter of seconds.

3. Definitions and Notation

In this section, we present basic definitions, notation, and terminology used throughout the paper. To formalize abstraction of snippets, we will use unranked trees [13] and unranked top-down tree transducers [14], defined in the rest of this section. Tree transducers perform transformations on tree-structured data, like unranked trees, which makes them a powerful transformation engine successfully used in term rewriting and unification [15, 16].

Definition 1. *Let Σ be a finite unranked alphabet. An unranked tree over the alphabet Σ is a string over the symbols from $\Sigma \cup \{`(', `)', `,'\}$, where $\Sigma \cap \{`(', `)', `,'\} = \emptyset$, of the form $f(t_1, ..., t_n)$, such that:*

- *$f \in \Sigma$ and*
- *$t_1, ..., t_n$ are unranked trees over Σ*

The f node is the root of the tree, while $t_1, ..., t_n$ are root's child subtrees. For leaves, we omit the parentheses for clarity. The set of all trees over Σ is denoted by $\mathcal{T}_\Sigma$.

Definition 2. *A single state propagation top-down unranked tree transducer is a tuple (Q, Σ_1, Σ_2, q, $\mathcal{R}$), where Q is a finite set of states, Σ_1 (resp. Σ_2) is a finite unranked input (resp. output) alphabet, q is the initial state, and $\mathcal{R}$ is a finite set of transformation rules. Given $c \in \Sigma_1$, $d \in \Sigma_2$, and $q, p \in Q$, transformation rules $T \in \mathcal{R}$ can have two possible forms; either $(q, c) \to d$ or $(q, c) \to d(p)$. The first form substitutes the subtree rooted at c with d, and the second form renames node c to d and propagates a single state[1] p to all of c's (after substitution, d's) child subtrees.*

Transformation of an unranked tree t is defined inductively on t's subtrees. In a state q, transformation $T^q(t)$ transforms a tree t as follows. If the tree is empty (ϵ), the transformation will return an empty tree. If the tree is of the form $f(t_1, ..., t_n)$, there are two cases that need to be considered. In the first case, there is no applicable rule for the node f in state q and the transformation returns an empty tree. In the case there exists an applicable rule, the tree is transformed as follows. If a matching rule is of the first form, the tree will be replaced by a single node. If the rule is of the second form, f will be renamed and the transformation rules will be applied to its children with the propagated state. Figure 2 illustrates two possible transformation rules.

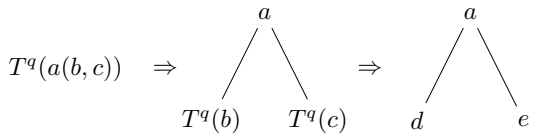

Figure 2. Transforming the $a(b, c)$ tree using the $(q, a) \to a(q)$, $(q, b) \to d$, and $(q, c) \to e$ transformation rules.

The first transformation propagates the initial state q to a's children. Then, node b is transformed into d using the second rule, and node c is transformed into e using the third rule.

4. Snippet Mining

The goal of the snippet mining is to find all frequent code segments (i.e., snippets) in a given repository. Moreover, in our analysis, we observed that infrequent snippets often have similar counterparts, which differ in only few subexpressions. We thus mine such snippets as well by replacing differing subexpressions with a placeholder, effectively abstracting snippets. The mining algorithm enumerates increasingly abstract snippet abstractions. As snippets become more abstract, they tend to match an increasing number of code segments in the repository.

The main challenge during the enumeration is to avoid excessive memory usage. To circumvent this problem, the mining algorithm prunes the search to save memory. More precisely, as soon as an abstraction of a snippet appears often enough, we stop the further search for abstractions. The

[1] Our single state propagation top-down tree transducer is a simplified version of the uniform top-down tree transducer, introduced by Martens and Neven [14], which can propagate multiple states.

algorithm then builds an index that compresses all discovered snippet abstractions. The synthesis procedure uses this index later to synthesize suggestions on demand.

4.1 Code Representation

Before the mining starts, we convert repository code into an abstract syntax tree-like data structure that enables fast snippet comparison needed for incrementing snippets' occurrence counters. The main motivation for choosing trees as the code representation is that checking the tree isomorphism, which is how we compare snippets, can be done in polynomial time [17], while for graphs only exponential time algorithms are known [17].

Plain abstract syntax trees [18, p. 69] are not a suitable code representation for accurate snippet comparison. Abstract syntax trees encode variables by their names. Often, semantically equivalent snippets can have different variable names. We therefore represent variables by their types, obtaining a necessary generalization. Our tree data structure is similar to the one developed by Raghavan et al. [19]. An example of the INSYREM code representation is shown in Figure 3.

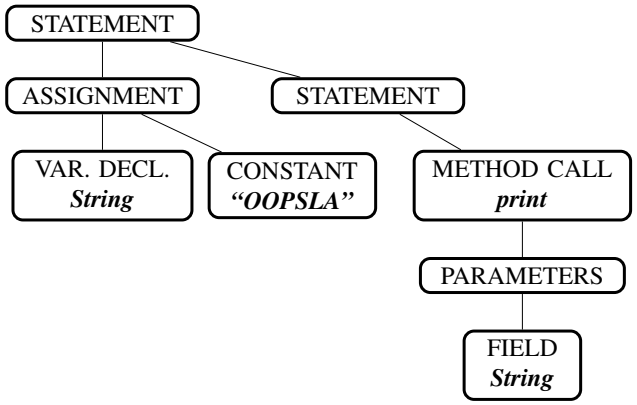

Figure 3. The INSYREM representation of the *String s = "OOPSLA"; print(s);* piece of code.

Each node in a tree has a type and a value, where applicable. For example, the `STATEMENTS` node does not have an associated value, while the `METHOD CALL` node has a value — the name of the corresponding method.

4.2 Abstract Snippets

We found that there are often many variants of essentially the same snippet, differing only in few statements or subexpressions. INSYREM mines such snippets as well. We compute such snippets by enumerating abstractions of snippets from repositories, i.e., replacing snippets' expressions with a *hole*, denoted `<?>`.

Snippets are defined by the simplified grammar shown below. We omitted several productions for brevity. The grammar defines snippets as sequences of statements. Snippets' expressions are defined using the *expression* production. Expressions range from simple constructs, like variables and constants, to chains of method calls and field accesses.

$$
\begin{aligned}
\langle \textit{statements} \rangle &::= \langle \textit{statement} \rangle \mid \langle \textit{statement} \rangle\ \langle \textit{statements} \rangle \\
\langle \textit{statement} \rangle &::= \langle \textit{for} \rangle \mid \langle \textit{while} \rangle \mid \langle \textit{if} \rangle \mid ... \\
&\ \ \ldots \\
\langle \textit{expression} \rangle &::= \langle \textit{field} \rangle \mid \langle \textit{operation} \rangle \\
\langle \textit{operation} \rangle &::= \langle \textit{binoperation} \rangle \mid \langle \textit{unoperation} \rangle \\
\langle \textit{binoperation} \rangle &::= \langle \textit{expression} \rangle\ \textit{binop}\ \langle \textit{expression} \rangle \\
\langle \textit{unoperation} \rangle &::= \textit{unop variable} \\
\langle \textit{field} \rangle &::= \textit{constant} \mid \textit{variable} \mid \\
&\quad \mid \langle \textit{methodcall} \rangle \mid \langle \textit{chain} \rangle \\
\langle \textit{chain} \rangle &::= \textit{variable}.\langle \textit{chain} \rangle \mid \langle \textit{methodcall} \rangle.\langle \textit{chain} \rangle \\
\langle \textit{methodcall} \rangle &::= \textit{method}(\langle \textit{params} \rangle) \\
\langle \textit{params} \rangle &::= \langle \textit{expression} \rangle \mid \langle \textit{expression} \rangle, \langle \textit{params} \rangle
\end{aligned}
$$

We generate abstract snippets by applying transformations on a given snippet using a transducer from Definition 2. The transducer transforms trees over an unranked alphabet Σ consisting of all node type labels from the INSYREM code representation. The alphabet is unranked since our nodes in general don't have a fixed number of children. The transducer has only one (initial) state q and the following two sets of transformation rules:

$$\{(q, x) \to x(q) \mid x \in \Sigma\}$$

$$\{(q, x) \to <?> \mid x \in (expression \cup statement)\}$$

The rules from the first set recurse without modifying the nodes they are applied to. The rules from the second set abstract either expressions or a whole statement. More precisely, the rules replace nodes that root an expression or a statement with holes.

The rules are non-deterministic, since two different rules can be applicable to the same construct. Every possible run of the transducer outputs a unique abstract snippet. This way, snippet's subexpressions are abstracted in all possible ways. Note that the transducer can produce duplicates as transformations can be applied in an arbitrary order. We avoid producing duplicates by fixing the order in which we apply transformations.

Examples of abstract snippets are shown in Figure 4. The transducer generates the topmost snippet by applying only the rules from the first set. Such snippets are identical to snippets passed as input to the transducer and we call them *concrete*. The transducer generates the bottommost snippet by applying a rule from the second set to the root, which is of the statement type. The arrows in the figure express a

partially ordered and transitive *more specific than* relation; a snippet at the start of an arrow is more specific than the snippet at the end of the same arrow.

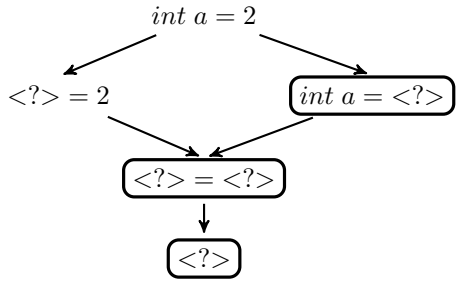

Figure 4. Abstractions of the $int\ a = 2$ snippet.

We now motivate mining of abstract snippets with an example. Consider the following scenario. Suppose a given repository consists of statements $int\ a = 2$ and $int\ b = 3$, each appearing only once. The mining procedure, with the occurrence threshold set to two, will discover the snippets shown in rectangles. The mining algorithm creates abstract snippets for both statements and computes their occurrence counts. The snippets in rectangles appear twice, as abstractions of both statements, thus satisfying the occurrence threshold condition, while the concrete statements appear only once each.

4.3 Mining Algorithm

The main goal of the mining algorithm is to find all snippets that satisfy the occurrence threshold condition. The mining problem can be stated more precisely:

> *Given a repository and an occurrence threshold* **t**, *find all abstractions of snippets that appear more often than* **t** *times in the repository*.

The mining algorithm starts by finding frequent statements and continues by finding frequent statement sequences, i.e., snippets. At the end, the mining algorithm builds a compressed representation of all discovered snippets by exploiting shared statements and expressions.

4.3.1 Statement Mining

For each statement in the repository, the statement mining algorithm finds abstractions that occur more often than a given occurrence threshold. However, the algorithm does not save all abstractions of a statement. For each statement, we save only its most specific frequent abstractions. Note that a statement can have several such abstractions since more specific than relation is partially ordered, as illustrated by Figure 4. Discovered abstractions will be used later to find frequent statement sequences, i.e., snippets.

A straightforward approach to the statement mining would be to enumerate all abstractions of statements in the repository. After finishing the enumeration, the algorithm could filter out infrequent abstractions and then save the most specific ones for each statement. However, such an approach proved to be impractical. Since the number of statement's abstractions is exponential in the number of its subexpressions, memoizing all abstractions often exhausts all available memory before filtering takes place.

INSYREM iteratively discovers the most specific frequent abstractions of each statement, enumerating increasingly more specific abstractions in each iteration. For example, suppose that the repository consists of the single statement from Figure 4. The algorithm starts with the bottommost abstraction in the first iteration, enumerates the abstraction located above the bottommost one in the second iteration, and so on. In other words, the algorithm enumerates abstractions at increasingly higher levels of specificity. Given some abstraction, we will refer to all of its abstractions located at the specificity level below as *immediate abstractions*.

In each iteration, our algorithm computes increasingly more specific abstractions while updating their occurrence counts. As abstractions get more specific, they match fewer statements in the repository, eventually becoming infrequent. Once an abstraction is identified as infrequent, we use its immediate abstractions to compute the most specific frequent abstractions for the corresponding statement. For instance, suppose that the $int\ a = <?>$ abstraction from Figure 4 was identified as infrequent. Its immediate abstraction, $<?> = <?>$, is a candidate for being the most specific frequent abstraction for that statement. In the rest of this section, we will refer to such immediate abstractions simply as candidates.

The algorithm removes candidates that are either infrequent or not the most specific for the corresponding statement. A candidate is not most specific if it is also an immediate abstraction of some other frequent abstraction. In our running example, suppose that the $<?> = 2$ abstraction, enumerated in the same iteration as $int\ a = <?>$, was found to be frequent. In that case, the $<?> = <?>$ candidate is frequent but not the most specific one, as it is more abstract than the mentioned frequent abstraction. Thus, we discard that candidate. A candidate can also be infrequent. Note that the algorithm also encounters infrequent abstractions that are more specific than some other infrequent abstractions. Thus, some of the immediate abstractions (i.e., candidates) of such encountered infrequent abstractions are also infrequent. We remove such candidates as well. The remaining candidates are saved as the most specific frequent abstractions of the corresponding statement. Additionally, for each statement, we remember its abstractions enumerated in the current iteration so that the next iteration can enumerate abstractions at the next specificity level, upon which we remove the old abstractions. Note that we use abstractions enumerated in the previous iteration also to check is a candidate infrequent or not.

Algorithm 1 finds frequent statement abstractions using the above described approach. In each iteration, the al-

Algorithm 1 Algorithm for finding frequent statement abstractions.

input: S (set of statements)
output: M (mapping from statements to set of abstract statements)
init: $O = \emptyset$ (mapping from statements and specificity levels to set of abstract statements), $level = 1$

```
1:  while |S| ≠ 0 do
2:      for each st ∈ S do
3:          A := enumerateMoreSpecific(st, O, level)
4:          if |A| = 0 then
5:              S := S/{st}
6:          else
7:              O[st, level] := A
8:          end if
9:      end for
10:     for each st ∈ S do
11:         I := getInfreqAbstractions(st, O, level)
12:         C := getImmdtAbstractions(I)
13:         P := getInfreqAbstractions(st, O, level − 1)
14:         C := C/P
15:         F := getFreqAbstractions(st, O, level)
16:         T := getImmdtAbstractions(F)
17:         C := C/T
18:         M[st] := M[st] ∪ C
19:         O[st, level − 1] := ∅
20:     end for
21:     level + +
22: end while
23: return M
```

gorithm enumerates abstractions of an increasing level of specificity. We enumerate abstractions of a statement using the transducer described in Section 4.2 (line 3). The transducer applies transformations in the breadth-first manner on statements' trees. In the first iteration, we abstract the root node producing the most abstract abstraction of a statement. In the second iteration, we start new transformations that abstract only root's children. More precisely, the second iteration abstracts all root's children. In other words, we replace all statement's immediate subexpressions with holes. For instance, applying such transformations, the transducer produces the second bottommost abstraction from Figure 4. The subsequent iterations will abstract fewer root's children (expressions), eventually continuing to the next tree level.

We associate enumerated abstractions to the corresponding statements (line 7). However, if the $enumerateMoreSpecific$ function is called on a concrete statement, the function returns an empty set. In that case, we remove the statement from the statement list (line 5). After we enumerated abstractions for all statements, the algorithm can identify infrequent abstractions. Thus, we proceed to computing the most specific frequent abstractions of each statement. First, we collect all infrequent abstractions of a statement (line 11). Then, we collect immediate abstractions, i.e., candidates, of those infrequent abstractions (line 12). After that, we remove candidates that were identified as infrequent in the previous iteration (line 13, 14). The algorithm next collects statement's frequent abstractions enumerated in the same iteration (line 15) and removes candidates that are more abstract than those frequent abstractions (line 16, 17). We save remaining candidates as the most specific frequent abstractions for the corresponding statement (line 18). We then remove abstractions of a statement enumerated in the previous iteration (line 19). The algorithm stops when there are no more statements in the statement list.

We now show an example of the statement mining. Figure 5b shows the discovered abstractions for the repository shown in Figure 5a, where the occurrence threshold is set to two. The first discovered abstraction in Figure 5b is saved as the most specific abstraction for all statements in the repository that create a new map. The second discovered abstraction is saved as the most specific for the last two statements of the first two snippets in the repository, as both statements have "a" as the first parameter, but differ in the second parameter. The third discovered abstraction in Figure 5b is saved as the most specific only for the last statement of the third snippet in the repository.

Note that an algorithm that would enumerate statement abstractions in the other direction, from the most specific to the most abstract abstractions, is impractical. Such an algorithm would enumerate abstractions at increasingly higher levels of abstraction. Two different statements can share the same abstraction at different abstraction levels for each of the statements. Thus, that abstraction will be enumerated at different iterations. For example, the abstraction in the topmost rectangle in Figure 4 would be enumerated in the second iteration and the same abstraction for the statement $int\ a = 2 + 3$ would be enumerated in the third iteration. Consequently, the algorithm could not discard infrequent abstractions early on, as they could become frequent in the subsequent iterations. In our approach, the same abstraction is always located at the same specificity level and, consequently, enumerated in the same iteration. For instance, the $<?> = <?>$ abstraction is always one level above the most abstract abstraction, no matter what the left/right hand sides of the assignment are.

4.3.2 Statement Sequence Mining

The goal of the statement sequence mining is to find snippet abstractions that occur more often than a given occurrence threshold. The algorithm, however, does not save all abstractions of a snippet in the repository. For each snippet, we save only its most specific frequent abstractions. The algorithm builds upon results of the statement mining. More precisely, the algorithm finds frequent sequences of discovered statement abstractions.

Our algorithm iteratively discovers increasingly longer sequences of statement abstractions. For each statement in

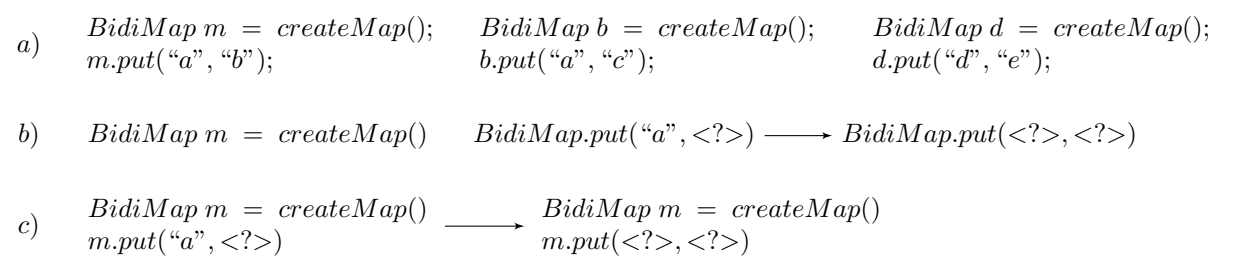

Figure 5. Given a repository composed of snippets in a) and an occurrence threshold **t=**2, the algorithm discovers abstractions of statements in b) and abstractions of snippets in c). The algorithm names variables used in discovered abstractions using names of variables from the repository. In this case, the algorithm used the name from the first snippet in a).

the repository, we enumerate increasingly longer abstract sequences starting with abstractions of that statement. When doing so, we utilize discovered shorter sequences. For each statement, we take its abstractions and prepend each of them separately to discovered sequences starting at the next statement.

Algorithm 2 discovers abstract sequences iteratively using the above described approach. In each iteration, we mine statement sequences of an increasingly greater length, as follows. If the following statement of a current statement does not exist, we remove the current statement from the statement list (line 6). If there exists a following statement, we first collect that statement (line 9) and then all sequences starting with that statement that were discovered by the previous iteration (line 10). The algorithm then computes new sequences by prepending each abstraction of the current statement (line 14) to the collected shorter sequences, thus producing longer sequences (line 15). We associate generated sequences to the statement (line 16), later removing infrequent sequences. The algorithm updates occurrence count for each sequence it generates (line 17). As statement sequences get longer, they match fewer code samples in the repository. Eventually, sequences become infrequent and we cannot generate sequences of increasing lengths. Thus, if the algorithm did not discover any sequence starting with the following statement in the previous iteration, we remove the current statement from the statement list (line 12), as we cannot generate sequences of increasing lengths starting with the current statement.

Note that the most specific abstraction of one statement can be more specific than the most specific abstraction of some other statement, as depicted by the arrow in Figure 5b. Consequently, one sequence of statement abstractions can be more specific than some other sequence of statement abstractions, as depicted by the arrow in Figure 5c. Algorithm 2 directly generates only the most specific abstractions of some sequence, without enumerating all of its abstractions. Thus, the algorithm will not correctly compute occurrence counts for all abstract sequences. To solve this problem, we do the following. First, we generate all

Algorithm 2 Algorithm for finding frequent snippet abstractions.

input: S (set of statements), M (mapping from statements to their set of most specific frequent abstractions, produced by Algorithm 1), F (mapping from statements to their following statements), L (mapping from statements and lengths to multiset of abstract sequences; for length one, statements are mapped to the multiset of their corresponding abstract statements)
output: P (set of frequent most specific abstract snippets)
init: $length = 2$

```
1:  while |S| ≠ 0 do
2:      // Multiset of abstract statement sequences
3:      T := ∅
4:      for each st ∈ S do
5:          if F[st] is not defined then
6:              S := S/{st}
7:              continue
8:          end if
9:          ns := F[st]
10:         Q := L[ns, length − 1]
11:         if |Q| = 0 then
12:             S := S/{st}
13:         else
14:             A := M[st]
15:             N := generateNewSequences(A, Q)
16:             L[st, length] := N
17:             T := T ∪ N
18:         end if
19:     end for
20:     T := updateCountsForAbstractions(T)
21:     T := removeInfrequentSequences(T)
22:     for each st ∈ S do
23:         L[st, length] := L[st, length] ∩ T
24:         L[st, length − 1] := ∅
25:     end for
26:     P := P ∪ T
27:     length + +
28: end while
29: return P
```

abstractions of a sequence using discovered abstractions of its statements. Then, we increase occurrence counts of generated abstract sequences by the occurrence count of the sequence. For instance, the algorithm will increase the occurrence count of the second sequence from Figure 5c by the occurrence count of the first sequence, as denoted by the arrow. In Algorithm 2, this is implemented in the $updateCountsForAbstractions$ function (line 20). Then, the algorithm removes infrequent sequences for all statements (lines 21–23). We stop the algorithm where there are no statements in the statement list.

An example of abstract snippets is shown in Figure 5c. The algorithm generates the first snippet abstraction from the first two repository snippets, thus identifying that sequence as frequent. The second snippet abstraction is generated only from the last repository snippet. However, that abstract snippet is an abstraction of the first abstract snippet (denoted by the arrow). Thus, the algorithm increased the occurrence count of the second snippet by the occurrence count of the first snippet; the second snippet then became frequent.

4.3.3 Snippet Index

The mining algorithm finishes by building a compressed representation of all discovered snippets — a snippet index — a map that maps statements to the snippets in which those statements appear. To save memory, snippets in the snippet index share common statements and expressions.

The snippet index facilitates fast retrieval of snippets by maintaining a mapping between statement types and snippets that have statements of the corresponding type. The synthesis procedure uses the snippet index to quickly retrieve snippets that match programmers' code. During our experiments, the programmers often requested for the following types of statements: assignment, variable declaration, and field access.

We proceed by presenting two optimizations: (1) statement and subsequence sharing and (2) eliminating subsumed snippets. The first optimization saves memory by exploiting shared abstract statements and subsequences. Snippets in the snippet index are represented as sequences of pointers to discovered abstract statements. Additionally, two snippets sharing a subsequence of abstract statements have a pointer to that subsequence. The second optimization avoids indexing of snippets that are a part of some larger snippets. The synthesis procedure will suggest only those larger snippets, without withholding any potentially valuable code from programmers. This way, we save memory and don't overwhelm programmers with too many similar suggestions.

5. Interactive Code Snippet Synthesis

We suggest snippets from the snippet index upon request. Programmer initiates a request by simply positioning her cursor at a desired position in the program and pressing a shortcut. Then, we find snippets in the snippet index that best match her statement located immediately before the cursor. In the rest of this section, we will refer to such snippets as *the best fitting snippets*. Once the best fitting snippets are found, we rewrite them so as to include variables live at the point where the cursor is. We use the rest of programmer's statements, located above the cursor line statement, to sort the snippets heuristically. At the end, we suggest the synthesized snippets to the programmer.

5.1 Suggestion Synthesis

Upon a programmer's request, the synthesis procedure collects data needed for the synthesis. First, it collects code between the cursor and the beginning of the current method. Then, we collect all variables live at the cursor position. The synthesis procedures uses the input code to find and rank the best fitting discovered snippets, which we then rewrite so as to include the live variables.

We use only programmer's statement located immediately before the cursor to find the best fitting discovered snippets. We call that last programmer's statement a *cursor statement*. We find all snippets in the snippet index that have statements that resemble the cursor statement, using the type of the cursor statement. As mentioned earlier, we refer to those resembling statements in the snippet index as fixed statements. Synthesized snippets are merely statements in the snippet index that come after fixed statements. In the case that the cursor statement is partial, synthesized snippets also include fixed statements. We then rewrite synthesized snippets so as to include programmers' live variables.

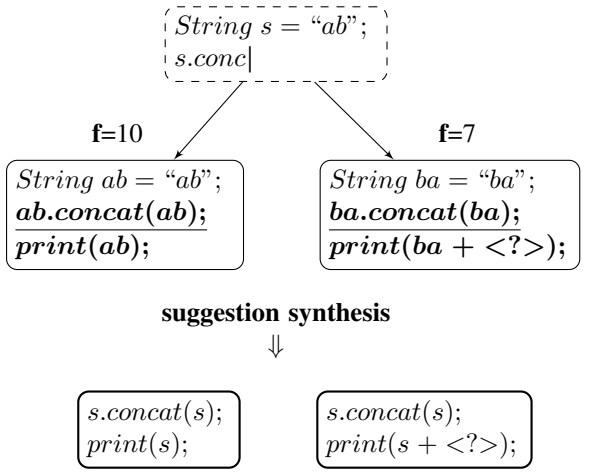

Figure 6. An example of suggestion synthesis. Programmer's code is shown in the dashed rectangle. The cursor is represented with the "|" symbol. The occurrence counts of discovered snippets in the snippet index are denoted by **f**.

We now describe the example of the suggestion synthesis shown in Figure 6. The synthesis procedure first collects the programmer's code, where the last partial statement is the cursor statement. Then, we collect all live variables, which in this case is the single `String` variable s. Using the type

of the cursor statement (field access), the synthesis procedure retrieves two snippets, pointed to by the arrows, from the snippet index. Then, we identify the fixed statements, shown underlined, by traversing the retrieved snippets' statements and performing a resemblance comparison to the cursor statement. Then, we generate suggestions, shown in bold, by taking the statements that come after the fixed statements. Additionally, we include the fixed statements since the cursor statement is partial. The synthesis procedure then rewrites the suggestions so as to include the s variable. The synthesized suggestions are shown in the bottommost rectangles.

In the case that a synthesized snippet uses a type for which there is no corresponding programmers' live variable, the synthesis procedure declares a variable of that type. Programmers can then position their cursors immediately after those declarations and initiate requests to get variable initialization suggestions. We name those variables using names of variables from the repository, instead of creating meaningless names.

5.2 Suggestion Ranking

The synthesis procedure sorts suggestions using a ranking function and presents them to programmers. We sort suggestions snippets based on their score, heuristically computed from four parameters: snippet frequency, contextual match, specificity, and size.

Frequency We consider snippet frequency (f) as an important parameter of our ranking function. If some snippet appeared more frequently in the repository used for mining, then there are higher chances that it will be useful to programmers.

Contextual match For each synthesized suggestion, we compare it to programmer's code above the cursor line statement. We perform the comparison using the INSYREM tree representation of code. We compute the percentage (m) of nodes of programmer's code above the cursor line that match nodes of statements preceding the fixed statement of a suggestion. We call this value a *contextual match*. We plan to research more sophisticated code comparison procedures in the future, as well as integrate ones from XSnippet [6] and Strathcona [7].

Abstraction In our initial experiments, we observed that programmers prefer snippets that have fewer holes. Therefore, our ranking function uses snippet abstraction parameter (a), computed as the ratio between the number of holes in a snippet and the number of its subexpressions.

Size We also observed that programmers prefer longer snippets. Thus, we score longer snippets higher. Our ranking function uses the size parameter (n), computed as the number of statements in a suggestion snippet.

Snippet score We use the following formula to heuristically compute the score of a snippet. Formula uses all of the above described parameters and the occurrence threshold **t** used to discover snippets in the repository.

$$\ln(1 + \frac{f}{\mathbf{t}}) \times m \times n \times a$$

We now explain the intuition behind our heuristic scoring. Snippets with higher scores will show up higher in the suggestion list presented to programmers. The first factor gives a high score to snippets that are highly frequent. However, we scale the frequency by the occurrence threshold. More precisely, we consider snippets with frequencies much higher than the occurrence threshold almost equal; we want to rank them relatively to each other using the other factors. Using the contextual match factor, we put highly frequent snippets at the top of the suggestion list only if they match programmers' code very well. We use the size factor to favor longer snippets. However, longer snippets might not be useful if they have a lot of holes, which is why we use the abstraction factor. The above formula sorts the synthesized suggestions in Figure 6 so that the first suggestion is listed before the second suggestion, since the first suggestion has higher frequency, fewer holes, and better contextual match.

6. Evaluation

To evaluate our approach, we implemented a tool based on INSYREM and conducted a user study to measure development time savings. Our tool is an Eclipse [20] plugin that synthesizes Java suggestions upon a programmer's request. The user study consisted of 5 programming tasks. For each task, we randomly arranged 11 test subjects into two equally sized groups where one group solved the task with the help of the tool, and the other without. We now describe implementation details of our tool and present the results of our evaluation.

6.1 Implementation

We implemented the INSYREM approach as an Eclipse plugin targeted for Java programs. The plugin provides functionality for mining repositories and synthesizing suggestions on demand using snippets discovered in those repositories. In the rest of this section, we will assume that repositories have been already mined. Programmers interact with the tool by positioning their cursors at a desired position and pressing the `Ctrl + Space` shortcut. This is the standard Eclipse shortcut used for built-in autocompletion. The suggestion synthesis is then carried on as described in Section 5. Figure 7 shows an example of suggested snippets.

Programmers can use synthesized suggestions in three ways. Firstly, programmers can integrate suggestions at positions where their cursors are located. Programmers can accomplish this by simply double-clicking on suggestions. Secondly, programmers can copy a snippet and then paste it at a desired position in the program. This can be done by simply clicking on a suggestion, upon which suggestion

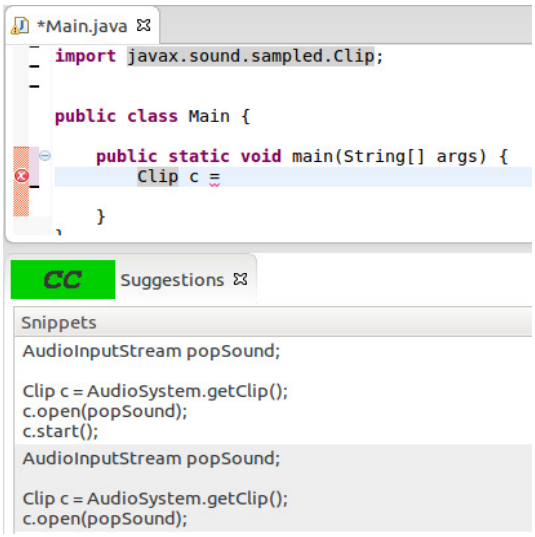

Figure 7. An example of suggestions synthesized by INSYREM in Eclipse.

code is copied to the clipboard. Lastly, programmers can learn from suggestions and write the code on their own.

The major challenge in developing the tool was to accurately parse partial cursor line statements. The Eclipse's built-in parser [4] performs recovery steps on syntactically invalid code by removing lexems. Given a partial statement, the parser thus produces a syntax tree that does not correspond to the statement. Using such incorrect trees, our synthesis procedure would inaccurately synthesize suggestions. Instead, we listen for error messages from the Eclipse's parser. Using those messages, we infer what recovery steps the compiler made and then reconstruct the syntax tree so it corresponds to the programmers' cursor statement. Effectively, we try to undo Eclipse's error recovery. This technique works well for partial code that programmers seem to use most often, although it does not work for all cases. If we cannot reconstruct the programmers' cursor statement, we indicate to programmers that the synthesis was not accurate. We indicate this by setting the plugin icon's background color to red. Otherwise, we set the background color to green, as shown in Figure 7.

6.2 Scalability

We now analyze the complexity of INSYREM algorithms. Let n be the number of statements in the repository, l the maximal specificity level of all statements (see Figure 4), a the maximal number of statement abstractions at any specificity level, s the length of the longest frequent sequence of abstract statements, and m the number of frequent abstract snippets. The time complexity of the statement mining algorithm is $\mathcal{O}(l \cdot n \cdot a)$; the algorithm iterates at most l times, traversing at most n statements where it performs constant time operations on at most a statement abstractions. The algorithm has the $\mathcal{O}(n \cdot a)$ space complexity, as it remembers all abstractions enumerated in the previous and the current iteration. The statement sequence mining algorithm iterates at most s times. For each statement, the algorithm creates at most a^s new sequences. After traversing the statements, the algorithm updates occurrence counts for all abstractions of discovered sequences, which takes at most $m \cdot a^s$ steps. Thus, the time complexity of the sequence mining algorithm is $\mathcal{O}(s \cdot a^s \cdot (m+n))$. In any iteration, the algorithm keeps track of at most a^s sequences, which gives the $\mathcal{O}(a^s)$ space complexity. The synthesis algorithm finds at most m best fitting snippets. Locating fixed statements in those snippets requires at most s constant time comparisons. The synthesis procedure thus has $\mathcal{O}(m \cdot s)$ time complexity and $\mathcal{O}(m)$ space complexity. Note that all complexities are highly conservative and that variables a and s are small in practice.

6.3 Experiments

We evaluated our tool by conducting a user study, in which we asked our test subjects to solve 5 Java programming tasks. For each task, we asked half of the subjects, chosen randomly, to solve the task with the help of our tool, and the other half without. We chose the groups so that each subject solved at least 2 tasks with the help of the tool. During the evaluation, the subjects were allowed to use all available resources, like Web, just as they would normally do in the classical working environment. In our experiments, our tool improved programmers' efficiency by saving 32% of time per task on average. Additionally, four test subjects gave up on solving a task when they were not using the tool, while only two subjects who used the tool quit a task.

6.3.1 Setup

We now present more details on how we conducted the study. Our test subject group consisted of 11 UC Berkeley undergraduate students. All subjects had experience in developing Java software using Eclipse IDE. None of the subjects had any experience in using our tool prior to the user study. Upon arrival, we gave our test subjects a 30 minute introductory session to INSYREM, which included a brief tutorial and three simple exercises. We introduced the subjects to the approach at a high level and then explained the main features of the tool. Additionally, we asked the subjects to solve three simple programming tasks consisting of constructing an object with the help of our tool.

In our preliminary experiments, we noticed that test subjects can negatively affect each other. More precisely, finishing a task earlier by one subject distracts other subjects or makes them uncomfortable for being slower. In this user study, we gave our test subjects all tasks at once and asked them to solve one task at a time. Instead of notifying others, we asked the subjects to write the start and finish times for each task. Additionally, we gave the subjects a difficult 6th task that requires at least couple of hours of development time. Using such a lengthy task, we avoided the situation where one subject finishes all the tasks before other subjects,

since in that case other subjects can clearly see she is done. When we saw that all the subjects reached the sixth task, we stopped the testing. We don't report results for that 6th task. We asked all subjects to write comments on how the tool helped them on each task, if they were allowed to use it.

We chose 5 Java programming tasks that require usage of unfamiliar APIs or boilerplate code. For each task that requires usage of unfamiliar API, we collected Java files from the Web that use that API. We used the API names as a search keyword. Using the keyword, we queried GitHub [21] for relevant code samples. We collected the first 10 Java files that were returned as a search result. Additionally, we collected all Java files needed for collected files to compile correctly. Note that some of the search result files were irrelevant since their authors declared their own classes with the name same as the name of the corresponding unfamiliar API. We discarded such irrelevant files.

Together with the files collected from GitHub, we also used files from our INSYREM implementation project. We did that for two reasons. First, our project was using some of the APIs our testing tasks required. Second, we wanted to simulate API diversity in repositories, which we expect in practice. We summarized all the files into one repository. Altogether, the repository consisted of 20,000 lines of code, measured by `cloc` [22]. We mined the repository on a 4 core Lenovo `T420si` laptop with 4GB of RAM and Ubuntu 11.04 installed. The mining took roughly around 30 seconds with an occurrence threshold set to 3, where 21 second was spent on code transformation. We did not parallelize our algorithms to utilize the multi-core environment. The resulting snippet index occupied 601 KB of RAM, while the size of the repository was 3.07 MB.

6.3.2 Results

We now present the results of our experiments. The graph in Figure 8 shows the times required for solving each task by each test subject. Additionally, the graph shows the mean time and the standard deviation for each task and the subject group. Our tool reduces the time spent on developing code for 32% on average in our experiments. Moreover, using our tool, only two test subjects were not able to solve a task, while four test subjects quit a task when they were not using the tool. We now describe the results of each task in detail, show examples of synthesized suggestions, and explain motivation the behind each task.

Task 1. The first task required subjects to write code that operates over the `BidiMap` [23] object. More concretely, the subjects were asked to initialize a bidirectional `BidiMap` map, put their last name as a value with their first name as the key, and then construct an inverse of the map. The purpose of this task was to find out how much time our tool saves on developing small pieces of code that use unfamiliar APIs. Test subjects who were not allowed to use the tool spent 7.2 minutes on average on solving this task, while the subjects who used the tool only 2.2 minutes. All test subjects who used the tool were faster than those who didn't.

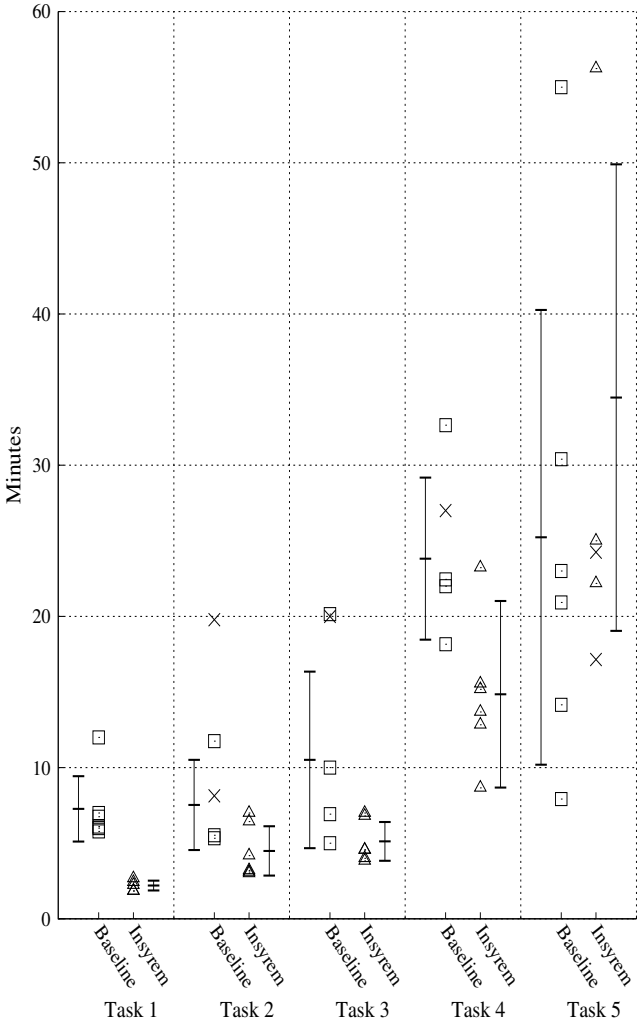

Figure 8. The times test subjects required to solve programming tasks with and without the help of the INSYREM tool. The cross symbols denote the times when a test subject decided to quit a task.

This task is difficult as it requires significant amount of time to find examples on the Web that show how to construct an object implementing the `BidiMap` interface. The subjects noted that the following synthesized suggestion helped them in solving the task:

```
BidiMap map = new DualHashBidiMap();
map.put(<?>, <?>);
```

The snippet shows how to initialize the map and how to put values into it. The parameters of the `put` method differed in the repository so INSYREM replaced them with placeholders. The test subjects integrated this snippet automatically into their code and then used the standard Eclipse's autocompletion to construct the inverse map.

Task 2. The second task had a similar purpose as the first. We asked the subjects to create a `CompressorInputStream` [24] object with the Gzip compression method and a proprietary file as an input. Test subjects who were allowed to use the tool were faster by 3 minutes on average. Moreover, one test subject who was not allowed to use the tool quit a task after 20 minutes, while all test subjects using the tool completed the task.

Initializing `CompressorInputStream` is not trivial since it is created by a factory class, as suggested by our tool:

```
InputStream is;
CompressorInputStream s =
    new CompressorStreamFactory().
     createCompressorInputStream("bzip2", is);
```

The suggested snippet shows how to create `CompressorInputStream`. The first parameter to the method that constructs a compressor hints how to specify the compression method. The second parameter, object `InputStream is`, hints how to set the input to the compressor. At the time when the corresponding requests were made, the test subjects did not have any live `InputStream` variables. Our tool hence declared the `is` variable. The subjects then requested suggestions that initialize that variable, upon which the tool suggested snippets that create `InputStream` using a `File`, a well known API.

Task 3. We used the third task to evaluate our approach on tasks that require reuse of boilerplate snippets. First, we created our own proprietary `FileTester` API that is used to perform various checks on files. Then, using this API, we implemented 20 tests that we included in the repository. We gave the subjects who were not allowed to use the tool a separate repository where they could find those tests and reuse the boilerplate code. Additionally, that repository contained other irrelevant files, just as repositories do in practice.

The task required subjects to write a test that checks that a file contains some XML content and that the file size is less than 1KB. Subjects who were not allowed to use the tool resorted to browsing the repository, which made them 5.39 minutes slower on average. Moreover, two of them were not able to complete the task. The subjects who were allowed to use the tool found the suggested snippets helpful and they were not compelled to browse the repository. We present one of the few suggested snippets:

```
FileTester t = TestFactory.
                  createFileTester(<?>);
t.assertExists();
t.assertContentType(FileContent.XML);
```

The suggestions did not provide the exact snippet the subjects needed, but it suggested snippets that contained lines that the subjects integrated into their code. The above snippet is one such example. The snippet shows how to create a `FileTester` object where the hole represents a placeholder for the file path parameter, which differed in the repository from test to test.

Task 4. We designed the fourth task so that it requires more complex solution and usage of a significant amount of code that our mining repositories did not contain. We simulated the scenario where programmers' tasks require usage of freshly developed unfamiliar APIs for which exists only a small number of examples that can be used for mining. We asked the subjects to parse a given Java file into a `CompilationUnit` [4]. The task required the subjects to read the contents of a given file and parse it using the mentioned API. The subjects allowed to use the tool were 37% faster on average.

A similar, but shorter, problem was used by Jungloid [5], XSnippet [6], and Strathcona [7], since at that time it was hard to find relevant examples on the Web. We decided to use this problem even though examples of the `ASTParser` API usage can be found quickly. The subjects reported that the tool was very helpful by providing the following snippet:

```
ASTParser parser;
CompilationUnit u = (CompilationUnit)
                  (parser.createAST(null));
```

The snippet shows how to construct a `CompilationUnit` object. Then, the test subjects had to initiate a request for an `ASTParser` [4] object. The tool responded with several suggestions, one of them being the following snippet:

```
String source;
ASTParser p = ASTParser.newParser(AST.JLS3);
p.setSource(source.toCharArray());
```

The subjects integrated this snippet and then proceeded to reading the contents of the file. They reused examples from the Web since our tool did not suggest any useful snippets; our repository did not contain enough relevant samples to be mined. The files we collected from GitHub did not contain code that reads from a file and the files from our projects did not use such code frequently. More precisely, we implemented a single method that reads contents from a file and then reused that method whenever needed.

Task 5. The fifth task had the same purpose as the fourth task, but this time the task was even more complex. The subjects were asked to parse an XML file using the `DocumentBuilderFactory` [25] and then traverse the document and count the number of XML nodes that satisfy a given predicate. We provided the description of the XML format, as well as an example file.

The tool suggested the useful snippet shown below. The snippet shows how to construct a parser and parse XML content from a file given its path. However, INSYREM had little impact on the overall solving time for this task. The tool did not provide any suggestions that were valuable for traversing an XML file and extracting its content, which was the most difficult part of the task.

```
String path;
DocumentBuilderFactory f =
        DocumentBuilderFactory.newInstance();
DocumentBuilder b = f.newDocumentBuilder();
Document doc = b.parse(path);
```

We would like to note that all suggestions for each task were synthesized in less than 400 milliseconds.

7. Future Work

We identified few directions in which one can improve the INSYREM approach. We believe that solving the following challenges can make the INSYREM approach more accurate.

INSYREM checks for the equivalence between snippets by comparing hashes of their corresponding trees. This approach classifies two semantically equivalent snippets as different even if they slightly differ syntactically. Moreover, our current implementation does not account for data-flow dependencies between statements. Two data-flow dependent statements that appear frequently can be separated by an arbitrary number of unrelated statements in each occurrence. Our mining algorithm misses such dependent statements, since it directly mines whole sequences. Additionally, a snippet might be wrapped by a single method that is reused throughout the code. In that case, our algorithm will not discover such snippets since its code does not appear often enough, although it might be used frequently. We believe that by canonicalizing code one can improve the accuracy of the mining algorithm and provide a higher synthesis quality.

For the mining purposes, we define the snippet similarity only in terms of the syntactic (structural) resemblance. More precisely, we identify snippets that differ in only few subtrees as similar. Our future work will investigate more semantic approaches to computing similarity between snippets like, for instance, using subtyping information.

We currently compare programmer's code before the cursor line to discovered snippets to compute how well does programmer's current context match suggestions. We compute the match by counting the number of matched nodes in corresponding trees. In our future work, we plan to design more advanced techniques for quantifying similarity between code. Additionally, we will consider techniques that take into account more of programmer's current code context than just statements before the cursor.

8. Related Work

There has been a considerable amount of research done on synthesizing code snippets using code repositories. We now shortly present that work and compare it to INSYREM.

Jungloid [5] automatically synthesizes jungloid code fragments that help programmers write code using unfamiliar APIs. A *jungloid* is a sequence of object and method call chains synthesized from API signatures and other jungloids discovered in repositories. Our work differs in two aspects. First, our approach can synthesize snippets with no restrictions on the code format, while Jungloid synthesizes snippets of chains of objects and method calls. Second, programmers initiate a Jungloid request by writing a query in a dedicated language, while in our approach programmers only position their cursors and press a key.

Strathcona [7] suggests code samples from repositories that heuristically match programmers' code. Upon a programmer's request, Strathcona searches through all available repositories for code samples that heuristically match programmer's code and then presents them to programmers. Our work differs in two ways. First, Strathcona does not synthesize snippets and, consequently, does not provide programmers with an autocompletion option. Second, the authors of Strathcona reported that suggesting relevant snippets sometimes takes dozens of seconds, which we consider to be a significant impediment to adoption. INSYREM synthesizes suggestions in less than half a second.

XSnippet [6] introduces new heuristics for synthesizing object instantiation snippets. XSnippet mines repositories to extract useful snippet information that is later used to heuristically synthesize relevant object instantiation snippets on demand. Our work differs in that INSYREM does not impose any constraints on the code format, and thus can synthesize snippets beyond object instantiation.

PARSEWeb [9] helps programmers instantiate objects of an unfamiliar type using code search engines. Programmers manually provide a query specifying the source object type and destination object type. More precisely, programmers ask for chains of method calls and field accesses, starting with an object of the source type, that return an object of the destination type. PARSEWeb then searches for such chains using code search engines. In contrast to INSYREM, PARSEWeb requires programmers to manually craft the synthesis request. Moreover, INSYREM synthesizes snippets faster, as it mines repositories offline. Finally, PARSEWeb suggests only object initialization snippets, while INSYREM does not impose such constraints.

Hill and Rideout [8] propose an approach that autocompletes whole method bodies by employing machine learning techniques on frequent snippets in repositories. This approach is not very accurate, as reported by its authors. Gvero et al. [10] propose an approach that synthesizes single-line expressions by taking into account polymorphic type constraints of programmers' values in scope and API usage patterns in repositories. Perelman et al. [11] developed a technique that completes programmers' partial expression in a type-directed fashion. Hipikat [26] recommends artifacts from project archives to programmers depending on the task they are trying to solve.

9. Conclusion

In this paper, we presented INSYREM, a new approach to improving programmers' efficiency by eliminating repeti-

tive coding tasks and speeding up the learning of unfamiliar APIs. INSYREM automatically synthesizes code snippets upon a programmer's request using available code repositories. INSYREM operates in two phases. In the first phase, executed offline, we discover frequent code snippets from given repositories and save them into a compressed data structure called a snippet index. Programmers trigger the second phase by initiating a request, upon which they are served with snippets in the snippet index that best match their code before the cursor. Programmers can then integrate the snippets directly into their code or they can learn from the snippets.

This work was motivated by observing that programmers waste time developing repetitive coding tasks. More precisely, programmers often waste time by looking for code samples that use API they are unfamiliar with. Likewise, programmers spend precious time on reusing boilerplate code. To save programmers' time, we suggest code snippets to programmers upon their requests using the INSYREM approach.

Finally, we presented the results of our evaluation of the INSYREM approach. We developed a tool based on INSYREM and performed a user study where we measured how much the tool speeds up code development. In our experiments, the INSYREM approach reduced the time spent developing code for 32% on average.

References

[1] U.S. Department of Labor-Bureau of Labor Statistics. Occupational outlook handbook, 2012-13 edition, software developers. `http://goo.gl/kFnkD`. [Online; visited Feb. 08, 2013].

[2] Joel Brandt, Philip J. Guo, Joel Lewenstein, Mira Dontcheva, and Scott R. Klemmer. An empirical investigation of opportunistic programming: Interleaving web foraging, learning, and writing code.

[3] M.P. Robillard. What makes apis hard to learn? answers from developers. *Software, IEEE*, 26(6):27 –34, Nov.-Dec. 2009.

[4] Eclipse documentation Archived Release. Astparser api. `http://goo.gl/EgEWp`. [Online; visited Feb. 08, 2013].

[5] David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. Jungloid mining: helping to navigate the api jungle. In *Proceedings of PLDI*, pages 48–61, New York, NY, USA, 2005. ACM.

[6] Naiyana Sahavechaphan. Xsnippet: mining for sample code. In *Proceedings of OOPSLA*, pages 413–430. ACM Press, 2006.

[7] Reid Holmes and Gail C. Murphy. Using structural context to recommend source code examples. pages 117–125. ACM Press, 2005.

[8] Rosco Hill and Joe Rideout. Automatic method completion. In *Proceedings of ASE*, pages 228–235, Washington, DC, USA, 2004. IEEE Computer Society.

[9] Suresh Thummalapenta and Tao Xie. Parseweb: a programmer assistant for reusing open source code on the web. In *Proceedings of ASE*, pages 204–213, New York, NY, USA, 2007. ACM.

[10] Tihomir Gvero, Viktor Kuncak, and Ruzica Piskac. Code completion using quantitative type inhabitation. In *EPFL-REPORT-170040*. EPFL, 2011.

[11] Daniel Perelman, Sumit Gulwani, Thomas Ball, and Dan Grossman. Type-directed completion of partial expressions. In *Proceedings of PLDI*, pages 275–286, New York, NY, USA, 2012. ACM.

[12] Steven P. Reiss. Semantics-based code search. In *Proceedings of ICSE*, pages 243–253, Washington, DC, USA, 2009. IEEE Computer Society.

[13] Hubert Comon, Max Dauchet, Remi Gilleron, Florent Jacquemard, Denis Lugiez, Sophie Tison, and Marc Tommasi. Tree automata techniques and applications. 2002.

[14] Wim Martens and Frank Neven. Typechecking top-down uniform unranked tree transducers. *Database Theory–ICDT 2003*, pages 64–78, 2002.

[15] Franz Baader. Unification theory. *Word Equations and Related Topics*, pages 151–170, 1992.

[16] Sophie Tison. Tree automata and term rewrite systems. In *Rewriting Techniques and Applications*, pages 27–30. Springer, 2000.

[17] Alfred V. Aho and John E. Hopcroft. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1974.

[18] Alfred V Aho, Monica S Lam, Ravi Sethi, and Jeffrey D Ullman. *Compilers: principles, techniques, and tools*, volume 1009. Pearson/Addison Wesley, 2nd edition, 2007.

[19] S. Raghavan, R. Rohana, D. Leon, A. Podgurski, and V. Augustine. Dex: a semantic-graph differencing tool for studying changes in large code bases. In *Proceedings of ICSM*, pages 188 – 197, Sept. 2004.

[20] The Eclipse Foundation. Eclipse - the eclipse foundation open source community website. `www.eclipse.org`. [Online; visited Mar. 14, 2013].

[21] Inc. GitHub. Github - build software better, together. `https://github.com/`. [Online; visited Feb. 09, 2013].

[22] sourceforge.net. Cloc — count lines of code. `http://cloc.sourceforge.net/`. [Online; visited Mar. 16, 2013].

[23] The Apache Software Foundation. Bidimap (commons collections 3.2.1 api). `http://goo.gl/pju8c`. [Online; visited Mar. 16, 2013].

[24] The Apache Software Foundation. Compressorinputstream (commons compress 1.0 api). `http://goo.gl/52PAm`. [Online; visited Mar. 16, 2013].

[25] Oracle and/or its affiliates. Documentbuilderfactory (java 2 platform se 5.0). `http://goo.gl/PKeBg`. [Online; visited Mar. 17, 2013].

[26] Davor Cubranic and Gail C Murphy. Hipikat: Recommending pertinent software development artifacts. In *Proceedings of ICSE*, pages 408–418. IEEE, 2003.