On Synthesizing Code from Free-Form Queries

Tihomir Gvero and Viktor Kuncak
École Polytechnique Fédérale de Lausanne (EPFL), Switzerland
Email: firstname.lastname@epfl.ch

***Abstract*—We present a new code assistance tool for integrated development environments. Our system accepts free-form queries allowing a mixture of English and Java as an input, and produces Java code fragments that take the query into account and respect syntax, types, and scoping rules of Java as well as statistical usage patterns. The returned results need not have the structure of any previously seen code fragment. As part of our system we have constructed a probabilistic context free grammar for Java constructs and library invocations, as well as an algorithm that uses a customized natural language processing tool chain to extract information from free-form text queries. We present the results on a number of examples showing that our technique can tolerate much of the flexibility present in natural language, and can also be used to repair incorrect Java expressions that contain useful information about the developer's intent.**

I. Introduction

Application programming interfaces (APIs) are becoming more and more complex, presenting a bottleneck when solving simple tasks, especially for new developers. APIs contain many types and declarations, so it is difficult to know how to combine them to achieve a task of interest. Instead of focusing on more creative aspects of the development, a developer ends up spending a lot of time trying to understand informal documentation or adapt the API documentation examples. Integrated development environments (IDEs) help in this task by listing declarations that belong to a given type, but leave it to the developer to decide how to combine the declarations.

On the other hand, on-line repository host services such as GitHub [1], BitBucket [2], SourceForge [3] and Google Developers [4] are becoming more and more popular, hosting a large number of freely accessible projects. Such repositories are an excellent sources of code examples that the developers can use to learn API usage. Moreover, their large size and variety suggests that they can be used by machine learning techniques to create more sophisticated IDE support. A natural first step is to perform code search [5], though this still leaves the user with the task of understanding the context and adapting it to their needs. Several researchers have pursued the problem of generalizing from such examples in repositories, combining non-trivial program analysis and machine learning techniques [6].

In this paper, we present a new approach that synthesizes code appropriate for a given program point, guided by hints given in *free-form* text. We have implemented our approach in a system *anyCode*, and have found it to be very useful in our experience (see Figure 2). Our approach builds a model of the Java language based on corpus of code in repositories, and adapts the model to a given text input. In that sense, our approach combines some of the advantages of statistical programming language models [6] but also of natural language processing of the input containing English phrases, previously done for restricted APIs [7].

Our approach builds on our past experience with the InSynth tool [8], in which a user only indicates the desired API type; the tool InSynth then generates ranked expressions of a given type. With our new tool, *anyCode*, the input can be interpreted as a result type, but, more generally, it can be any text, referring to any part of an expected expression. We find this interface more convenient and expressive than in InSynth. Furthermore, InSynth uses only the unigram model [9, Chapter 4], which assigns a probability to a declaration based on its call frequency in a corpus. InSynth uses the model to synthesize and rank expressions. In *anyCode*, besides unigram, we use the more sophisticated probabilistic context free grammar (PCFG) model [9, Chapter 14], to synthesize and rank the expressions. We perform synthesis in three phases: (1) we use natural language processing (NLP) tools [10]–[12] to structure the input text and split the text input into chunks of words based on their relationships in the sentence parse tree; (2) we use the structured text and unigram model to select a set of most likely API declarations, using a set of scoring metrics and the Hungarian method [13] to solve the resulting assignment problem [14]; (3) we finally use the selected declarations, PCFG and unigram model to unfold the declaration arguments. The result is a list of ranked (partial) expressions, which *anyCode* offers to the developer using the familiar code completion interface of Eclipse.

By introducing a textual input interface, we aim to automatically reduce the gap between a natural and a programming language. *anyCode* allows the developer to formulate a query using a mixture of English and code fragments. *anyCode* takes into account English grammar when processing input text. To improve the input flexibility and expressiveness we also consider word synonyms and other related words (hypernyms and hyponyms). We build a related word map based on WordNet [15], a large lexical database of English. We present a technique to make WordNet usable in our context by automatically projecting it onto the API jargon. We use these techniques along with the NLP tools to support the natural language aspect in *anyCode*. The techniques we implement in *anyCode* are inspired by stochastic machine translation. However, we had to overcome the lack of a parallel corpus relating English and Java, as well as the gap between an informal medium such as English and the rigorous syntax and type rules of a programming language such as Java.

We aim to relieve the user of the strict structure of a programming language when describing their intention. From our perspective, IDE tools should allow a user to gloss over aspects such as the number and the order of arguments in method calls, or parenthesis usage. Instead, the developers should focus more on solving important higher-level software architecture and decomposition problems. Finally, we also hope to lower the entry for those who are learning to program, for whom syntax is often one of the first obstacles. To achieve this, we find that a short text input that approximately describe the structure of the desired expression is the most convenient. To make the input useful for programming, we also allow a user to explicitly write literals and local variable names in input. Using such input, *anyCode* manages to synthesize valid Java code fragments. It can do that because it does not impose any strict requirement on the input: it has the ability to generate likely expressions according to the Java language model, and uses as much of the information from the input as it can extract to steer the generation towards developer's intention.

In summary, this paper makes the following contributions:

- A new technique that maps text to a list of ranked (partial) expressions. It combines NLP tools, a text-declaration matching, PCFG and unigram models to process input, synthesize and rank expressions.
- A new text and declaration models that encode input and a declaration as a set of words. They prioritize words based on their importance and position, both in text and a declaration.
- Efficiently performing text-declaration matching thanks in part to the creation of appropriate indices and the use of the algorithms for the assignment problem, in particular the Hungarian method.
- A fast corpus analysis and extraction algorithm. The algorithm extracts method compositions and frequencies and build PCFG and unigram models. The implementation combines Eclipse JDT parser, our symbol table and type-checker.
- A customized related word-map that maps a word to its related words. We use relations in WordNet and a novel scoring technique that for a given word ranks and finds the closest related words. We use a set of API words to build the score and filter out irrelevant words.
- A benchmark set of 45 text-expression (input-output) benchmarks and experimental result that shows the effectiveness of our techniques and can also be used to calibrate future open ended tool that map free-form text into Java API invocations.

II. Examples

In this section we use five examples to illustrate main functionality of *anyCode*. The first example demonstrates *anyCode*'s interactive deployment, the text interface, and the use of program context to guide the synthesis.

A. Making a Backup of a File

Suppose that a user wishes to create a method that backs up the content of a file. The method should take a file name as a parameter and copy the content of the file to a new file with an appropriately modified name. To implement such a backup method, the user needs to identify the appropriate API, select the set of its declarations (typically method calls) and combine them into an expression. In practice, to perform this, a user might follow these steps manually: (1) search the Internet, or API documentation (if it exists) to find the examples of API use, (2) select the most suitable example, (3) copy-paste it into the working editor with code, and (4) edit the example such that it fits into the context, using appropriate values in the program scope. *anyCode* offers an automatic approach that combines all the mentioned operations, and more. Suppose that a user writes an incomplete piece of code of the method that takes the parameter fname that stores the file name, as shown in Figure 1. She also creates a variable bname that stores the backup file name. Here, the backup name is obtained by adding ".bak" extension to bname. In the next line the user invokes *anyCode*. A pop-up text field appears where she can insert the text, such as copy file fname to bname that specifies her desire to copy the file content. *anyCode* automatically extracts the program context from Eclipse and identifies words fname and bname in the input as values referring to a parameter and a local variable. *anyCode* then uses this information to generate and present several ranked expressions to the user. When the user makes her choice, the tool inserts the chosen expression at the invocation point. In this example, *anyCode* works for less than 50 milliseconds and then presents five solutions of which the first one copies the file fname content to a file with name bname:

```
FileUtils.copyFile(new File(fname), new File(bname))
```

This is a valid solution; it uses the method FileUtils.copyFile from the popular "Commons IO" library.

B. Invoking the Class Loader

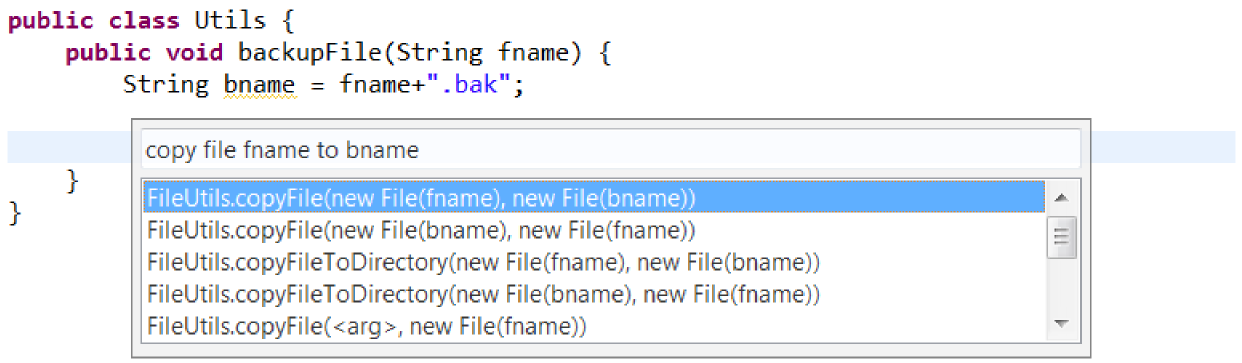
Suppose that a user intends to load a class with a name "MyClas.class". She invokes the tool with the free-form input

```
load class "MyClas.class"
```

anyCode automatically synthesizes and suggests the following (partial) expressions:

```
1  Thread.currentThread().getContextClassLoader()
                       .loadClass("MyClas.class").getClass()
2  Thread.currentThread().getContextClassLoader()
                       .loadClass("MyClas.class")
3  Thread.currentThread().getContextClassLoader()
                       .loadClass(⟨arg⟩).getClass()
4  "MyClas.class".getClass()
5  Thread.currentThread().getContextClassLoader()
                       .loadClass(⟨arg⟩)
```

In this example *anyCode* generates suggestions in less than 40 milliseconds. The second suggestion turns out to be the desired one.



```
public class Utils {
    public void backupFile(String fname) {
        String bname = fname+".bak";
```

copy file fname to bname

```
    }
}
```

FileUtils.copyFile(new File(fname), new File(bname))
FileUtils.copyFile(new File(bname), new File(fname))
FileUtils.copyFileToDirectory(new File(fname), new File(bname))
FileUtils.copyFileToDirectory(new File(bname), new File(fname))
FileUtils.copyFile(<arg>, new File(fname))

Fig. 1. After the user inserts text input, *anyCode* suggests five highest-ranked well-typed expressions that it synthesized for this input.

The suggestions 1, 2, and 4 represent complete expressions. On the other hand, suggestions 3 and 5 represent templates that include the symbol ⟨arg⟩ that marks the places where local variables are often used. The main reason why we present templates is that a user often inserts incomplete input and for an incomplete input the best solution is an incomplete output, i.e., a template. If we have insisted only on completed expressions, we would miss many interesting solutions that are more convenient for such an incomplete input. Thus, *anyCode* treats a textual input both as complete and incomplete, and tries to find both complete and incomplete solutions.

Note that the complete expressions 1 and 2 include declarations whose selection and integration does not directly depend on the textual input. For instance, method loadClass contains both input words load and class, whereas currentThread does not. To reach the currentThread from loadClass we use probabilistic language model for Java and its API calls, derived from a corpus of code. Without such a model we would not be able to construct complex expressions such as the above one.

C. Creating a Temporary File

In the third example we demonstrate the use of semantically related words. For instance, if a user wants to discover templates that make a new file, she may insert 'make file'. In a less than 80 milliseconds, *anyCode* generates the following output:

1 **new** File(⟨arg⟩).createNewFile()
2 **new** File(⟨arg⟩).isFile()
3 **new** File(⟨arg⟩)
4 **new** FileInputStream(⟨arg⟩)
5 **new** FileOutputStream(⟨arg⟩)

Note that word make does not appear among the solutions, because API designers used the word create. *anyCode* succeeds in finding the solution because it considers, in addition to the words such as make appearing in the input, its related words, which includes create. *anyCode* uses a custom related-word map to compute the relevant words. We built this map by automatically processing and adapting WordNet, a large lexical semantic network of English words.

D. Writing to a File

Consider next the following input of a developer:

write "hello" to file "text.txt"

For this input, in less than 50 milliseconds *anyCode* outputs:

1 FileUtils.writeStringToFile(**new** File("text.txt"), "hello")
2 FileUtils.writeStringToFile(**new** File("hello"), "text.txt")
3 FileUtils.writeStringToFile(**new** File("hello"), "hello")
4 FileUtils.writeStringToFile(**new** File("text.txt"), "text.txt")
5 FileUtils.writeStringToFile(**new** File(⟨arg⟩), "hello")

The expressions under 1 and 2 are ranked higher than, e.g., the solution 3 because they have a higher usage of the input text elements. Indeed, solution 3 does not refer to one of the string literals in the input. The synthesis algorithm and our scoring techniques favor solutions with the greater input coverage. In this example, the first expression performs the desired task.

E. Reading from a File

In the final example we show that our input interface may also accept an approximate expression. For instance, if a user attempts to write an expression that reads the file, in the first iteration she may write the following expression

readFile("text.txt","UTF−8")

Unfortunately, this expression is not well-typed according to common Java APIs. Nevertheless, if *anyCode* takes such a broken expression, it puts it apart and recomposes it into a correct one, suggesting (again in less than 40 milliseconds) the following solutions:

1 FileUtils.readFileToString(**new** File("text.txt"))
2 FileUtils.readFileToString(**new** File("UTF−8"))
3 FileUtils.readFileToString(⟨arg⟩)
4 FileUtils.readFileToString(**new** File(⟨arg⟩))
5 FileUtils.readFileToString(**new** File("text.txt"), "UTF−8")

anyCode first transforms the input by ignoring the language specific symbols (e.g., parenthesis and commas). It then slices complex identifiers, so called *k-words*, into single words. Here, readFile is a 2-word that gets sliced into read and file. Despite the loss of some structure in treating the input, our language

	Input	Output	Rank			Time
			NoPU	NoP	All	[ms]
1	copy file fname to bname	FileUtils.copyFile(new File(fname), new File(bname))	>10	>10	1	47
2	does x begin with y	x.startsWith(y)	>10	>10	1	62
3	load class ”MyClass.class”	Thread.currentThread().getContextClassLoader() .loadClass(”MyClass.class”)	>10	>10	2	31
4	make file	new File(<arg>).createNewFile()	>10	>10	1	78
5	write ”hello” to file ”text.txt”	FileUtils.writeStringToFile(new File(”text.txt”), ”hello”)	>10	>10	1	47
6	readFile(”text.txt”,”UTF-8”)	FileUtils.readFileToString(new File(”text.txt”), ”UTF-8”)	>10	>10	5	31
7	parse ”2015”	Integer.parseInt(”2015”)	>10	>10	1	16
8	substring ”ICSE2015” 4	”ICSE2015”.substring(4)	>10	>10	1	31
9	new buffered stream ”text.txt”	new BufferedReader(new InputStreamReader(new BufferedInputStream(new FileInputStream(”text.txt”))))	>10	1	1	47
10	get the current year	new Date().getYear()	>10	>10	6	62
11	current time	System.currentTimeMillis()	1	1	1	16
12	open connection ”http://www.oracle.com/”	new URL(”http://www.oracle.com/”).openConnection()	>10	>10	1	31
13	create socket ”http://www.oracle.com/” 8080	new Socket(”http://www.oracle.com/”, 8080)	>10	>10	6	47
14	put a pair (”Mike”,”+007-345-89-23”) into a map	new HashMap().put(”Mike”, ”+007-345-89-23”)	>10	9	1	109
15	set thread max priority	Thread.currentThread().setPriority(Thread.MAX_PRIORITY)	>10	>10	1	109
16	set property ”gate.home” to value ”http://gate.ac.uk/”	new Properties().setProperty(”gate.home”, ”http://gate.ac.uk/”)	>10	>10	3	94
17	does the file ”text.txt” exist	new File(”text.txt”).exists()	>10	4	1	47
18	min 1 3	Math.min(1, 3)	>10	7	1	31
19	get thread id	Thread.currentThread().getId()	>10	1	1	31
20	join threads	Thread.currentThread().join()	>10	1	2	16
21	delete file ”text.txt”	new File(”text.txt”).delete()	>10	1	1	31
22	print exception ex stack trace	ex.printStackTrace()	>10	>10	7	47
23	is file ”text.txt” directory	new File(”text.txt”).isDirectory()	>10	>10	2	46
24	get thread stack trace	Thread.currentThread().getStackTrace()	>10	1	1	47
25	read line by line file ”text.txt”	FileUtils.readLines(new File(”text.txt”))	>10	>10	2	94
26	set time zone to ”GMT”	Calendar.getInstance().setTimeZone(TimeZone.getTimeZone(”GMT”))	>10	>10	1	47
27	pi	Math.PI	2	1	1	15
28	split ”ICSE-2015” with ”-”	”ICSE-2015”.split(”-”)	>10	>10	1	16
29	memory	Runtime.getRuntime().freeMemory()	2	2	1	15
30	free memory	Runtime.getRuntime().freeMemory()	3	4	1	16
31	total memory	Runtime.getRuntime().totalMemory()	2	2	1	31
32	exec ”javac.exe MyClass.java”	Runtime.getRuntime().exec(”javac.exe MyClass.java”)	>10	1	1	16
33	new data stream ”text.txt”	new DataInputStream(new FileInputStream(”text.txt”))	>10	>10	4	47
34	rename file fname1 to fname2	new File(fname1).renameTo(new File(fname2))	>10	>10	1	31
35	move file fname1 to fname2	FileUtils.moveFile(new File(fname1), new File(fname2))	>10	>10	1	62
36	concat ”ICSE” ”2015”	”ICSE”.concat(”2015”)	>10	3	1	16
37	read utf from the file ”text.txt”	new DataInputStream(new FileInputStream(”text.txt”)).readUTF()	>10	>10	10	47
38	java home	SystemUtils.getJavaHome()	2	1	1	15
39	upper(text)	text.toUpperCase()	>10	2	1	31
40	compare x y	x.compareTo(y)	>10	>10	1	32
41	BufferedInput ”text.txt”	new BufferedInputStream(new FileInputStream(”text.txt”))	>10	>10	1	15
42	set thread min priority	Thread.currentThread().setPriority(Thread.MIN_PRIORITY)	>10	1	1	78
43	create panel and set layout to border	new Panel().setLayout(new BorderLayout())	>10	1	1	172
44	sort array	Arrays.sort(array)	>10	>10	1	15
45	add label ”names” to panel	new Panel().add(new Label(”names”))	>10	>10	1	78

Fig. 2. The table that shows the results of the comparison of the different *anyCode* configurations with and without unigram and PCFG models.

model gives us the power to recover meaningful expressions from such an input. This shows that *anyCode* can be used as a simple expression repair system. The desired solution is ranked fourth because it uses a version of readFileToString method with two arguments, which appears less frequently in the corpus than the simpler versions of the method.

We have evaluated our system on a number of examples; Figure 2 shows 45 text queries and the code that we expected to obtain in return. The “All” column indicates the rank on which the expression was found with all features of our system turned on, as discussed in Section IV.

III. System Overview

In this section we give a high-level picture of the main components of our system. Input to our system consists of i) a textual description, explicitly typed by the developer and ii) a partial Java program with a position of the cursor, which *anyCode* extracts automatically from the Eclipse IDE. *anyCode* uses the input to generate, rank and present (possibly partial) expressions to the user. As Figure 3 shows, the key components of *anyCode* are the input text parser, the declaration search engine, and the expression synthesizer. The method getExpressions is the main method that performs these steps, as outlined in Figure 4.

The first goal of parsing is to identify structure of the input text using a set of natural language processing tools. *anyCode* uses the structure to group input words into WordGroups. The intuition is that the input text corresponds to several declarations, and grouping according to the rules of English helps to identify these declarations from multiple input words. Moreover, the system uses a map of related words to complete the words given in the input with some of the related meanings computed from a modified version of WordNet [15]. To complement natural language input, the system uses program context from the IDE to mark local variables and literals in the input text. Section V describes our parsing and related word completion techniques in more depth.

In the declaration search engine, the system uses the groups, WordGroups, to find a subset of API declarations that are most likely to form the final expressions. The system tries to

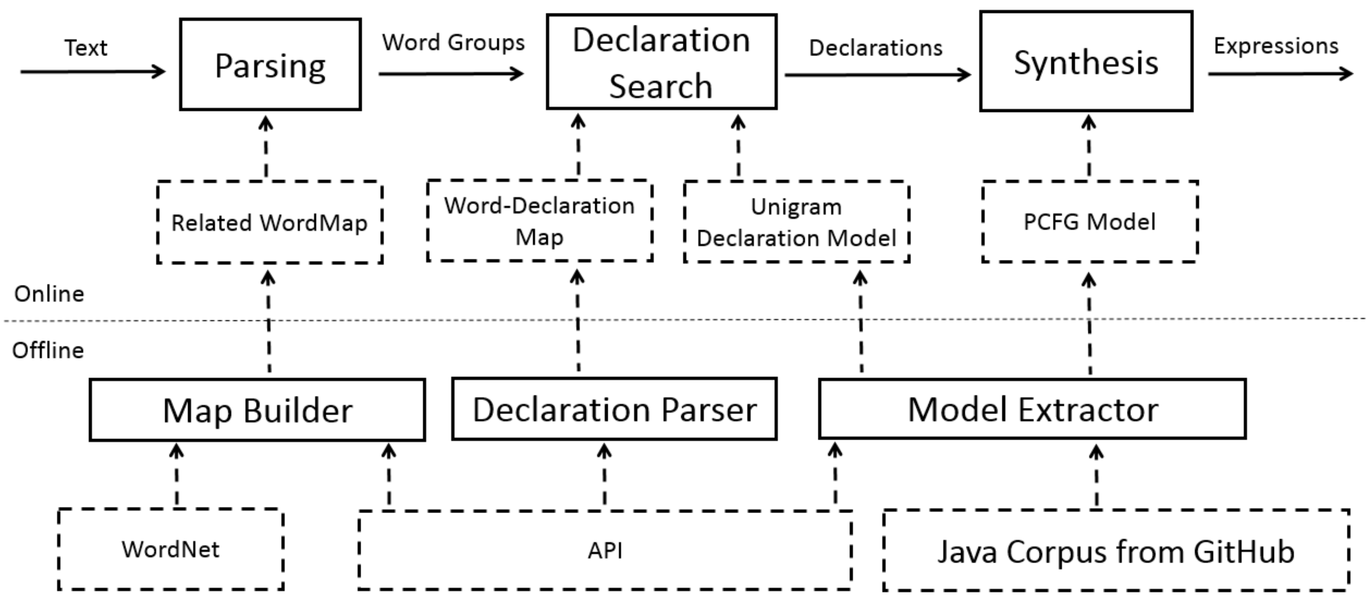


Fig. 3. *anyCode* system overview. The offline components run only once and for all. The online components run as part of the Eclipse plugin.

```
getExpressions(text, context, N−bestExprs):
    // Text Parsing
    (WordGroups, Literals, Locals) ← parse(text, context)
    // Declaration Search
    DeclGroups ← declSearch(WordGroups, API, Unigram)
    // Synthesis
    ExPCFG ← extend(PCFG, Literals, Locals)
    Exprs ← synth(DeclGroups, ExPCFG , N−steps)
    return keepBest(Exprs, N−bestExprs)
```

Fig. 4. The high level description of online portion of *anyCode*.

match word groups against declarations in our API collection. To perform matching, the system extracts a list of words from declarations and matches them against the words in the groups. Based on the number of words that match, declaration Unigram [9, Chapter 4] score, and other parameters the system estimates the matching score. We explain this in more details in Section VIII. Finally, for each word group the system selects the top N−best declarations with the highest scores and puts them in a declaration group. In summary, the method declSearch transforms each word group into a declaration group. Lastly, we would like to mention that API contains a set of declarations we collect from different APIs and packages. This is done in advance, before a user invokes *anyCode* for the first time.

In the last step the system uses declaration groups and a probabilistic context free grammar (PCFG) model [9, Chapter 14] to synthesize expression. ExPCFG consists of the initial PCFG model, pre-collected from a large Java source code corpus and the extension, a set of production rules for literals and local variables. The method extend extends PCFG with the extension and returns ExPCFG. The method synth tries to unfold declaration arguments following ExPCFG model, in N−steps. Given a declaration, ExPCFG suggests declarations that should fill in the argument places. The method synth also assigns scores to the expressions, based on the ExPCFG and declarations scores. Finally, the system uses the method keepBest in order to filter the top N−bestExprs expressions with the highest scores.

IV. Evaluation

This section discusses a set of benchmarks we use to evaluate *anyCode* and presents the experimental results.

A. Benchmarks

We wrote 45 benchmarks shown in Figure 2. Each benchmark consists of a textual description and local variables as input, and a desired expression as output. We say that the system passes the benchmark if for the given input, the desired expression appears among the synthesized expressions. We use the benchmarks to estimate the effectiveness and the importance of different aspects of our system.

We ran all experiments on a machine with a quad-core processor with 2.7Ghz clock speed and 16MB of cache. We imposed a 8GB limit for allowed memory usage. Software configuration consisted of Windows 7 (64-bit) and Java(TM) Virtual Machine 1.7.0.55. The declaration search and the expression synthesis algorithm make use of multiple CPU cores.

We parametrized *anyCode* with N−bestExprs=10 and N−steps=5. By using a N−step=5 limit, our goal was to evaluate the usability of *anyCode* in an interactive environment (which IDEs usually are).

Results are shown in Figure 2. The Input column represents the textual descriptions, and the Output column represents the expected expressions. The column Rank represent the ranks of the expected expressions after we run *anyCode*. With >10 we mark the case when the expected expression is not among the top ten synthesized expressions. The Rank column is split in three sub-columns. Each column relates to a different *anyCode* configuration. The first, NoPU, denotes *anyCode* that does not use unigram and PCFG models. In this setting all declarations have the same unigram score and all PCFG productions have the same probability. The second, NoP, denotes *anyCode* that uses unigram, but does not use PCFG model. In this setting the tool uses unigram model to select declarations, but all PCFG productions still have the same probability. The last, All, denotes *anyCode* that uses both unigram and PCFG to guide the synthesis algorithm and to rank the expressions. The results show that the system without the both models generates only 6 expected expressions among the top ten solutions. The system with unigram model generates 19 and the system with both models generates all 45 expressions. Finally, the column Time shows the times needed to synthesize the top ten expressions for the *anyCode* with both models turned on. All times are between 15 and 172 milliseconds, with average of 45.4 milliseconds.

In summary, the results illustrate that our system greatly benefits from the unigram and the PCFG models. They also show that *anyCode* can efficiently synthesize the expressions in a small period of time (in less that 200 milliseconds).

B. Threats to Validity

The primary threat is to external validity: our set of examples, while fairly large by the standards of previous literature, may not be representative of general results. This limitation comes from the two facts: (1) there is no standardized set of benchmarks for the problem that we examine, and (2) we used the same set of examples to configure and evaluate our system. The primary purpose of the examples is to show that our tool is able to produce a set of real-worlds examples when configured. The parameters that we configure include declaration selection and reward-penalty parameters. The declaration selection parameters are important for our word-declaration matching. A parallel corpus with text as input and declarations (expressions) as output would be ideal for configuring the parameters, however no such corpus exists. Our attempt to create the corpus from the code and its descriptive comments led to irrelevant examples and the low quality corpus. The reward-penalty parameters are used to effect the size of the expressions, the aspect over which PCFG and unigram models have no control. In the rest of the paper we take a deeper look at the techniques we employed in *anyCode*.

V. Parsing

We perform parsing both on an input text and API declarations. We use parsing to prepare the text and the declarations for the search and the matching.

In the sequel, a k-word denotes a chain of k English words connected without a whitespace between them, as often used in Java identifiers. The words are separated at places where a small letter meets a capital letter. Usually, declaration names are k-words (e.g. "readFile" is a 2-word that contains words "read" and "file"). A 1-word is a single English word. A token is either a k-word, a literal or a local variable name. A literal is a number, a string, or a boolean value. The input text consists of tokens, whereas declarations contain only k-words.

A. Input Text Parsing

The main idea behind input parsing is to group words in the text such that the groups can be matched to desired declarations. To describe the parsing we use a running example that shows different phases of the parse method in Figure 4. Let us assume a user inserts 'copy file fname to "C:/user/text2.txt"', as shown in the first row in Table 5. Each row represent the input to the next phase, but also the output of the previous phase.

In the input we identify the following tokens: three single words, one local variable and a string literal. We mark local variables and literals as shown in the second row. In the next phase we substitute literals and local variables with their types, and produce the output in the third row. The intuition is that literals and local variables will appear as declaration arguments, accordingly, we use their types to match potential declaration argument types. To perform our next parsing step we use the Stanford CoreNLP [10]–[12]. The CoreNLP is a pipeline of a natural language processing tools that takes an English raw text as input and returns tagged text with deep semantic connections among the words. First, we use the tools to lemmatize and tag the words. A lemmatizer analyzes a word context to obtain the word's canonical form, called lemma (e.g., "good" is the lemma of "better"). The Part-of-Speech (POS) tagger assigns a part of speech tags (e.g., noun, verb, adjective and etc.) to each word, based on the context. The tagging is important for two reasons. The first reason is that tagging is a necessary preprocessing step towards deeper text analysis, which we also perform. The second reason is that different tags impose different word places in a declaration. For instance, we observe that verbs appear in method-names, whereas non-verbs can appear almost anywhere. Therefore, POS-tags can help in assigning different roles and priorities to words. We also tag the words inside each k-word (if it contains more than one word).

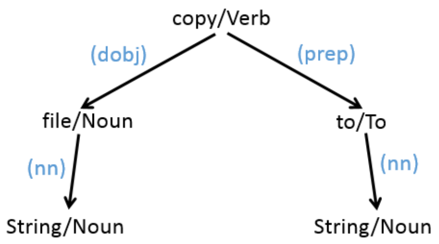


Fig. 6. Natural language semantic graph for the input from Figure 5.

To properly group the words, such that each group corresponds to a single declaration, we need more information

	Parsing		Example				
1	Text Input		copy	file	fname	to	”C:/user/text2.txt”
2	Decomposition		copy	file	Loc(fname)	to	Lit(”C:/user/text2.txt”)
3	Literals & Locals to Types		copy	file	String	to	String
4	Lemmatiz. & POS Tagging		copy/Verb	file/Noun	String/Noun	to/To	String/Noun
			1	2	3	4	5
5	Grouping	Primary Words	copy/Verb	file/Noun	String/Noun	to/To	String/Noun
		Secondary Words	file/Noun, to/To	String/Noun		String/Noun	
		Related Words	duplicate/Verb		chain/Noun		chain/Noun

Fig. 5. Phases of parsing an example input sentence.

than tags can provide. Thus, we use the parser to identify different semantical relations among the words. The result is a semantic graph, as show in Figure 6. The graph includes relations like ”nsubj” (a nominal subject) or ”dobj” (direct object) that identify a predicate, a subject and an object in a sentence. The relations are important because they separate the words that are more likely to appear in declaration names from the words that may appear as arguments. Because we expect that a user will type more often declaration name words to refer to the declaration, we call them *primary* words. The other words, which are more likely to appear as arguments, we call *secondary* words.

In the last phase, we form word groups, such that each group has primary, secondary and related words (see row five in the table). In our example we form five different groups marked with numbers 1-5. The group contains all the words below the corresponding number. The primary words are obtained directly from a k-word from the phase 4. The secondary words are connected to the primary words via the semantic graph. Namely, the secondary words are the neighbors (children) of the primary words in the graph. The related words are API words that are related to the primary words. Those words include synonyms, hypernyms and hyponyms. We build our own map of related words using set of all API words and WordNet [15], see Section V-C. An important constraint that we try to fulfill is that each group contains at least one non-verb word, because we observe that declarations usually contain at least one non-verb word.

B. Declaration Parsing

We also use parsing in pre-processing stage to create the representation of API declarations. We next define what we mean by a declaration, then sketch an algorithm that extracts words from declarations, forming a model suitable for matching.

1) Declaration Representation: A declaration can be a class method, a constructor, or a field. A declaration has a name and type, as well as an optional owner. Declaration name is a k-word. Declaration type can be: a Java primitive type, an API class type or a function type with multiple argument types, built from class and simple types. A declaration can be static or instance. A static declaration contains the owner class name. An instance declaration possesses also a receiver type, which we treat as any argument type.

2) Declaration Word Model: To match a declaration with a word group, we extract k-words from a declaration. Then we decompose complex k-words into single ones. Next, we apply lemmatizer and POS tagger to the words. In our final step we put the words into the primary or the secondary group based on the position in the declaration. The words that appear in the declaration name become the primary words and the word that appear elsewhere become the secondary words. Our goal is to assign higher priority to the primary words and lower to the secondary words. Section VIII shows the framework and the mechanisms to assign such priorities.

The declaration parsing allows us to transform API declaration set into a word-declarations hash map. This allows us to use an input word to quickly find all declarations that contain it. This optimization is crucial when searching for declarations.

C. Related WordMap: Modifying WordNet

To support inputs that does not strictly follow the actual words of API declarations, we use WordNet [15], a large lexical database of English words. WordNet groups words into synsets, which are sets of synonyms. Each synset represents a different meaning of a word. WordNet is a graph with synsets as the vertices and the relations as edges. The relations between synsets include antonyms, hypernyms and hyponyms.

API declarations contain only a subset of English words. We refer to this subset the *API words*. We build a map from all English words in WordNet to a ranked list of related API word meanings. A key of the map is an English word and the value is a ranked list of related API word meanings. The resulting related words are organized into meanings (synsets) to whom we assign scores (a value between zero and one). This score subsequently becomes the related weight $weight_r$ (see Section VIII-A) of a particular related word.

We build meanings using WordNet relations. For each English word we first find synsets (synonyms). Then, we find the synsets' closest hypernyms and hyponyms. We filter out all non-API words. Finally, we assign scores using textual descriptions assign to each synset in WordNet. We use Stanford POS-tagger to tag each word in a description. Then, we calculate a percentage of API words in the description. We assign a description score to each synonym synset. However, to a hypernym (hyponym) synset we assign a score that is a product of its description score and the score of the synonym synset that input word uses to reach the hypernym (hyponym). This way we give an advantage to synonyms over *their closest* hypernyms and hyponyms.

VI. Declaration Search

We expect that the developer will often insert a short text (two to ten words), omitting many details of a desired expression. The advantages of such an approach for developer are faster coding, with more time to focus on other development aspects. Such use, however, brings further challenges for generating expressions. A typical short and ambiguous input does not make it clear which declarations the expression should use, nor how to compose them. To solve this difficulty, our first step is to use the words as a starting point in identifying the desired declarations.

Recall from Figure 4 that the parsing phase returns word groups, making use of relations in the parse tree. Recall also that these word groups are enriched with related words based on WordNet. We use such word groups to find candidate declarations. The declarative description of this algorithm is simple: we match a word group with all declarations, calculate the matching score for each declaration, and find N−bestDecl declarations with the top score. We apply the same procedure to each group. To make the solution practical, we split the algorithm into two steps. *First*, we use a word group and the word-declaration API map to select a set of declarations that match with *at least one* word in the word group. In practice, the selected set is far smaller that the entire API collection. As a consequence, we will have far fewer calls to the expensive scoring procedure in the second step. The *second* step calculates the score using for each selected declaration using the matching procedure from Section VIII-A and the unigram model. From the unigram model we obtain the declaration probability. We apply the two steps to each group in a list of word groups.

VII. Synthesis

In this section we describe the algorithm that synthesizes the expressions using PCFG model. Because the algorithm relays on PCFG we first explain the model and later the expression synthesis.

A. Probabilistic Context Free Grammar Model

The idea is to use the information on declaration compositions to synthesize expressions. In general, a declaration has multiple arguments, meaning that it simultaneously composes with multiple declarations. We decide to treat the simultaneous compositions as one inseparable *multi-composition*. We collect from the corpus API *multi-composition*s and form the PCFG model. Intuitively, each production represents one *multi-composition*. The nonterminal symbols on the left-hand side of the production we call holes. There are three kinds of holes: a declaration, a local variable and a literal hole. The declaration hole explains how a declaration is simultaneously composed with other holes. Literal and local variables holes we treat a bit differently. During PCFG extraction, we abstract away literal and local variable names and leave only type information. Moreover, we do not build production rules for literal and local variable holes from the corpus, but from the user context. This means that our PCFG model collected from a corpus is incomplete. We complete it once we have the textual input and the context of the partial program where a user invoked *anyCode*. From the context we build the missing production rules and expand PCFG model. This allow us to synthesize expressions with the local variables from the context and literals from the textual input.

The reason why we call the non-terminals, holes, is that, during synthesis, they appear as holes in partial expressions. We use holes to access production rules that unfold them. We also keep probability for each rule. Note that a single hole can have a several different productions, e.g., denoting a several different multi-compositions for a single declaration.

B. Partial Expression Synthesis

A *partial expression* denotes an intermediate result arising in the construction of expressions of interest. A *complete expression* would be made up of variables, literals, and declaration applications. A partial expression may additionally contain different kinds of *holes*, as well as *connectors*. Holes denote places where a partial expression may expand using the rules of PCFG. A connector denotes a place in a partial expression that can expand by substituting another synthesized partial expression.

Our algorithm proceeds in two following phases:

First Phase turns every declaration group into a partial expression group. For each declaration in a declaration group:

1) We wrap a declaration name in a hole, creating the initial partial expression.
2) We use production rules of PCFG to unfold holes in partial expressions. This creates new partial expression(s). Gradually, partial expressions grow, and we continue unfolding them until we reach some maximum number of steps. When we can, we insert a connector at the place where partial expressions may merge. This way, we leave the merging until the next phase.
3) We calculate a partial expression score, based on the PCFG model and the declaration scores.
4) The expressions without holes form a partial expression group. (The expressions may have connectors.)

Second Phase tries to connect as many as possible expressions that belong to the different partial expression groups. For each partial expression in a partial expression group:

1) We find connectors in the partial expression and substitute them with the appropriate expressions from another group. Again, this creates more partial expressions. They gradually grow and we keep substituting connectors until some maximum number of steps is reached.
2) The score of a new partial expression is calculated using PCFG model, declaration scores, the text input coverage and expression repetition parameters as explained in Section IX.

3) We keep and present to a user some maximal number of expressions.

VIII. Declaration Score

WordGroup-Declaration matching score and declaration unigram score influence total declaration score. More precisely the declaration score is equal to the product of the two scores.

A. WordGroup-Declaration Matching Score

We next present a word group and a declaration matching algorithm. Recall that the word group contains primary, secondary and related words. We further split them based on their input origin. Namely, each word has a word in input to which it belongs. We call this word an origin; a group can have a several origins. Our goal is to maximize the matching score between group origins and declaration words. However, we do not match an origin with a declaration word. Instead, we split the group into disjoint subgroups based on origins, and match subgroups with declaration words. A word matches with a subgroup if a subgroup contains the lexically identical word with the same tag. Let us create the bipartite graph such that one set, SGS, of vertices is a set of subgroups, and the other, W, is the set of declaration words. Also, we create edges between subgroups and declaration words that match. Each edge has a weight, a value between zero and one. Now, calculating the maximum matching score turns into finding a maximum weight matching in a weighted bipartite graph. This is the well-known assignment problem [14]. To solve it, we choose the Hungarian method [13], with the following optimizations. We observed that many vertices remain without edges, so we remove all such vertices. We also observe that the bipartite graph is usually disconnected. We identify its connected components, and decompose it into smaller bipartite graphs. We calculate the score of each component and sum them up. The final score is the word group-declaration matching score. Another optimization is that, if we need to calculate matching 1 to n, we simply find the maximum among the weights. This is an important optimization because this sort of matching and its special case (1 to 1 matching) often occur. We define a total match weight between a group word w_g and a declaration word w_d as follows:

$$weight_{match}(w_g, w_d) = weight_i(w_g) * wieght_i(w_d) * \\ * weight_k(w_g, w_d) * weight_r(w_g)$$

Word Importance Weight: Primary words are more important than secondary and related words. To encode this, we introduce word importance weight, $weight_i$, which is a real value between one and zero and assigns a higher value to a primary than to the secondary and related words.

Kind Weight We reward more a matching between primary input and primary declaration words (a primary-primary match) than a primary-secondary and a secondary-secondary matching. (Related words are treated same as primary words in this context.) We use a quantitative function, called kind weight, $weight_k$, that returns a real value between one and zero. It assigns a higher value to a primary-primary than to a secondary-primary and a secondary-secondary matching.

Related Word Weight We use WordNet and collection of API words to calculate $weight_r$ (see also Section V-C). We penalize related words that are *far* from the primary words. To measure a primary-related word *distance* we introduce a quantitative function, called related word weight, $weight_r$, that returns a real value between one and zero. Synonyms are closer to a word than hyponyms and hypernyms.

To apply the matching algorithm, we take the maximum weight between a subgroup SG and a declaration word w, to be the edge (SG, w) weight:

$$weight_{match}(\mathsf{SG}, \mathsf{w}) = \max(\{weight_{match}(w_g, \mathsf{w}) \mid w_g \in \mathsf{SG}\})$$

B. Declaration Unigram Score

The unigram model [9, Chapter 4] assigns a probability to each declaration based on call frequency in a corpus. The higher the declaration frequency, the higher the probability. We smooth the model by assigning the minimal frequency value (collected in the corpus) to a declaration that does not appear in the corpus. The declaration unigram score is equal to the declaration probability.

IX. Partial Expression Score

To rank the (partial) expressions, $expr$, and to guide the synthesis algorithm we use the score $score(expr)$ that is computed by the following formula:

$$score(expr) = \log(score(expr)_{pcfg}) + \log(score(expr)_{decl}) \\ + score(expr)_{cov} - score(expr)_{rep}$$

The PCFG score, $score(expr)_{pcfg}$, is equal to the product of all composition probabilities used in the (partial) expression. The declaration score, $score(expr)_{decl}$, is equal to the product of all declaration scores whose declarations appear in $expr$. The coverage score, $score(expr)_{cov}$, estimates and favors the higher coverage of the input text words. We say that the word is covered if it selects a declaration in $expr$. Finally, the repetition score, $score(expr)_{rep}$, is proportional to the number of extra partial expressions used in $score(expr)_{rep}$. We say that a partial expression is extra if another partial expression from the same partial expression group is also used in $score(expr)_{rep}$. Preferably, we would like to use a single partial expression per each partial expression group.

X. Constructing PCFG and Unigram Models

We build both unigram and probabilistic context-free grammar models by analyzing the corpus of projects from GitHub.

Java Code Corpus. We use the GitHub Java corpus [16] that contains over 14'500 Java projects. The corpus includes only projects from GitHub that were forked at least once, to select more popular repositories. We decide to analyze each Java source file individually to reduce analysis time and avoid the need to execute essentially arbitrary build processes of various projects. Whereas this reduces the quality of the data we can extract from corpus, it is compensated by the fact that we can analyze many more projects: we were able to analyze 14'500 Java projects containing over 1.8 million files.

Model Extraction. We use Eclipse JDT parser [17] to parse each file. To improve the model we build our own symbol table and type-checker. The symbol table keeps track of all imported API declarations (of our interest). We analyze an expression using the symbol table and the type-checker. The symbol table identifies API declarations in an expression and the type-checker checks if the expression type-checks against them. Using those methods we extract a declaration multi-composition along with its occurrence frequency. We also extract the declaration occurrence frequency. Then, we use them to calculate the composition's and the declaration occurrence probabilities. Finally, we use the composition and its probability to build PCFG model, and the declaration and its probability to build Unigram model. The models are formed once we extract the information for all compositions and declarations in the corpus. In general, an expression contains user-defined declarations. We reduce their number to improve the quality of the analysis. In particular, where suitable, we inline local variables. The rest we mark with a special symbols and encode them in the PCFG model.

XI. Related Work

We mention related work that combine NLP and program synthesis techniques. as well as program synthesis tools with similar goals as our work.

SmartSynth [7] generates smartphone automation scripts from natural language descriptions. It uses NLP techniques to infer components and their partial dataflow from NL description. Then, it uses type based synthesis to constructs the scripts. Macho [18] transforms natural language descriptions into a simple programs using a natural language parser, a database of corpus and input-output examples. It maps English into database queries, then selects them, combines them and test them using examples. The queries are based on variables names. Little and Miller [19] built a system that translates a small number of keywords, provided by the user, into a valid expression. It extracts words from a declaration in the context, and tries to match them using *explanatory* vectors. The system tries to cover as many as possible words from the input, using declaration words. It also penalizes the unmatched words. NaturalJava [20] allows a user to create and manipulate Java programs using an NL input. It uses a restricted form of NL, based on Java‘s programming concepts, and translates it to Java statements. It requires the user to think and explicitly describe commands at the syntactical level of Java. Also, Metafor [21] transforms a story (in NL form) into a program template. It tries to obtain program structure by interpreting nouns as program objects, verbs as functions and adjectives as properties.

Unlike all mentioned tools, *anyCode* uses code corpus, PCFG and unigram model to synthesize and rank the expressions. We also *automatically* infer the set of words that map to declaration (components). Finally, those tools usually relay on the mapping model, where verbs are mapped to actions (methods), and nouns to objects (arguments). As discussed in Section VIII-A, we introduce a more sophisticated model and its framework that maps an input text to declarations, resolves complex declaration names and takes into account related words (e.g. synonyms). We also show how to encoded the mapping into the assignment problem and solve it using Hungarian method.

SLANG [6] takes a program with holes and produces the most likely completions, sequences of method calls. It uses an N-gram language model to predict and synthesize a likely method invocation sequence, as well as method arguments. SNIFF [22] uses natural language to search for code examples. It collects a corpus, code examples, and uses API documentation to annotate the examples, and method calls, with keywords. In our previous work, InSynth [8] asks user to specify the desired type and produces a set of ranked expressions, instances of the desired type. InSynth ranks the solutions based on the declaration unigram model. In this paper, we change the input interface to the textual one, giving a user more freedom in specifying his wishes. We additionally use more sophisticated PCFG model, extracted from a far bigger corpus than the one used in InSynth. CodeHint [23] is a dynamic synthesis tool that uses a runtime information to generate and filter candidate expressions. A user provides tests and a specification, and tool generate candidates and checks them against the tests and specification. To guide a generation, the tool uses only a declaration unigram model. XSnippet [24] takes a user query to extract Java code from the sample repository. It offers a range of queries from generalized to specialized. XSnippet ranks solutions based on their length, frequency, and context-sensitive as well as context-independent heuristics. The user needs to initiate additional queries to fill in the method arguments. Strathcona [25] automatically extracts a query based on the structure of the developed code. It does not allow a user to explicitly describe their needs. PARSEWeb [5] uses the Google code search engine to get relevant code examples. The solutions are ranked by length and frequency. The advanced code completions tools [26], [27], proposes declarations and code templates. Both systems use API declaration call statistics from the existing code examples to present a solutions with appropriate statistical confidence value.

XII. Conclusions

We presented *anyCode*, to the best of our knowledge the first tool for code synthesis that combines unique flexibility in both its input and its output. On the one hand, *anyCode* performs parsing of the free-form text input that may contain a mixture of English and code fragments. On the other hand, *anyCode* automatically constructs valid Java expressions for a given program point and is able to generate combinations of methods not encountered previously in the corpus. Ensuring this flexibility required a new combination of techniques from natural language processing, code synthesis, and statistical inference. Our experience with the tool, as reported on 45 diverse examples, suggests that there is a number of scenarios where such functionality can be useful for the developer.

References

[1] GitHub repository hosting service, https://github.com/.

[2] BitBucket repository hosting service, https://bitbucket.org/.

[3] SourceForge source code repository, http://sourceforge.net/.

[4] Google Developers, https://developers.google.com/.

[5] S. Thummalapenta and T. Xie, “PARSEWeb: a programmer assistant for reusing open source code on the web,” in *ASE*, 2007.

[6] V. Raychev, M. T. Vechev, and E. Yahav, “Code completion with statistical language models,” in *PLDI*, 2014, p. 44.

[7] V. Le, S. Gulwani, and Z. Su, “Smartsynth: Synthesizing smartphone automation scripts from natural language,” in *MobiSys*, 2013, pp. 193–206.

[8] T. Gvero, V. Kuncak, I. Kuraj, and R. Piskac, “Complete completion using types and weights,” in *PLDI*, 2013, pp. 27–38.

[9] D. Jurafsky and J. H. Martin, *Speech and Language Processing*, 2nd ed. Prentice Hall, 2008.

[10] C. D. Manning, M. Surdeanu, J. Bauer, J. Finkel, S. J. Bethard, and D. McClosky, “The Stanford CoreNLP natural language processing toolkit,” in *ACL*, 2014, pp. 55–60.

[11] M.-C. de Marneffe, B. MacCartney, and C. D. Manning, “Generating typed dependency parses from phrase structure parses,” in *LREC*, 2006, pp. 449–454.

[12] K. Toutanova, D. Klein, C. D. Manning, and Y. Singer, “Feature-rich part-of-speech tagging with a cyclic dependency network,” in *HLT-NAACL*, 2003.

[13] H. W. Kuhn, “The hungarian method for the assignment problem,” *Naval Research Logistics Quarterly*, vol. 2, pp. 83–97, 1955.

[14] R. Burkard, M. Dell’Amico, and S. Martello, *Assignment Problems*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2009.

[15] C. Fellbaum, *WordNet: An Electronic Lexical Database*. Bradford Books, 1998.

[16] M. Allamanis and S. Charles, “Mining Source Code Repositories at Massive Scale using Language Modeling,” in *The 10th Working Conference on Mining Software Repositories*. IEEE, 2013, pp. 207–216.

[17] EclipseJDT, http://www.eclipse.org/jdt/.

[18] A. Cozzie and S. T. King, “Macho: Writing programs with natural language and examples,” University of Illinois at Urbana-Champaign, Tech. Rep., 2012.

[19] G. Little and R. C. Miller, “Keyword programming in java,” in *ASE*, 2007, pp. 84–93.

[20] D. Price, E. Riloff, J. L. Zachary, and B. Harvey, “Naturaljava: A natural language interface for programming in java,” in *IUI*, 2000, pp. 207–211.

[21] H. Liu and H. Lieberman, “Metafor: Visualizing stories as code,” in *IUI*, 2005, pp. 305–307.

[22] S. Chatterjee, S. Juvekar, and K. Sen, “SNIFF: A search engine for java using free-form queries,” in *FASE*, 2009, pp. 385–400.

[23] J. Galenson, P. Reames, R. Bodík, B. Hartmann, and K. Sen, “Codehint: Dynamic and interactive synthesis of code snippets,” in *ICSE*, 2014, pp. 653–663.

[24] N. Sahavechaphan and K. Claypool, “Xsnippet: mining for sample code,” in *OOPSLA*, 2006.

[25] R. Holmes and G. C. Murphy, “Using structural context to recommend source code examples,” in *ICSE*, 2005, pp. 117–125.

[26] M. Bruch, M. Monperrus, and M. Mezini, “Learning from examples to improve code completion systems,” in *ESEC/SIGSOFT FSE*, 2009, pp. 213–222.

[27] http://www.eclipse.org/recommenders/.