Statistical Translation of English Texts to API Code Templates

Anh Tuan Nguyen,[†] Peter C. Rigby,[‡] Thanh Nguyen,[◇] Dharani Palani[‡], Mark Karanfil[‡], Tien N. Nguyen[△]
Axon US Corp.[†]
Concordia University, Montreal, Canada[‡]
Iowa State University[◇]
University of Texas, Dallas[△]
Email: ntanhbk44@gmail.com, peter.rigby@concordia.ca, thanhng@iastate.edu,
dharani.kumar@gmail.com, m.karanf@gmail.com, tien.n.nguyen@utdallas.edu

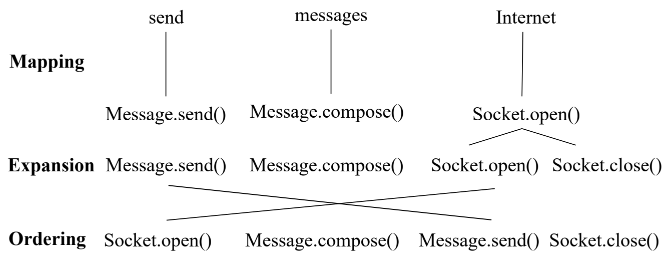
***Abstract*—We develop T2Api, a context-sensitive, graph-based statistical translation approach that takes as input an English description of a programming task and synthesizes the corresponding API code template for the task. We train T2Api to statistically learn the alignments between English and API elements and determine the relevant API elements. The training is done on StackOverflow, a bilingual corpus on which developers discuss programming problems in two types of language: English and programming language. T2Api considers both the context of the words in the input query and the context of API elements that often go together in the corpus. The derived API elements with their relevance scores are assembled into an API usage by GraSyn, a novel graph-based API synthesis algorithm that generates a graph representing an API usage from a large code corpus. Importantly, it is capable of generating new API usages from previously seen sub-usages. We curate a test benchmark of 250 real-world StackOverflow posts. Across the benchmark, T2Api's synthesized snippets have the correct API elements with a median top-1 precision and recall of 67% and 100%, respectively. Four professional developers and five graduate students judged that 77% of our top synthesized API code templates are useful to solve the problem presented in the StackOverflow posts.**

I. Introduction

Developers use the functionality of libraries via Application Programming Interfaces (APIs). Software libraries can be used in different ways, but not all of them are well documented in the official documentation and programming guides [1]. Researchers have focused on supporting the discovery of knowledge of API usages [2], including suggesting existing code snippets based on English queries [3], [4], [5], [6], [7], or synthesizing common API code usages [8], [9], [10].

To suggest API code from a query, earlier works use *information retrieval* (IR) [4], [5], [6], [7] with their goal being centered on *API code search* [2]. Recently, going beyond searching for existing code, researchers have aimed to *generate new API code*, by exploring *statistical approaches* including machine translation [9], probabilistic CFG [10], domain specific translation [11], and deep neural network [3]. Compared to these previous IR methods, statistical approaches can synthesize new code.

We continue this work by viewing the problem of generating API usages as a translation problem. We develop a graph-based statistical machine translation (SMT) model, named T2Api, to translate from English input text into an output code graph and template. To illustrate the key concepts of our approach, we start with the simplified example below and expand it to include code graphs in Fig. 1. The architecture of T2Api involves three stages: mapping, expansion, and ordering. We illustrate the stages by translating the English input *"Send messages over the Internet"* into code. After removing stopwords, one possible translation is as follows:



Statistical machine translation relies on finding the most probable *mappings* between the input and output languages, such as the term *"messages"* mapping to the code element *Message.compose()*. Some probable terms may be missing, therefore, the model requires term *expansion*, *e.g.,*Socket.close() is added. The words' order in the languages may be quite different, thus, it requires an *ordering* stage, such as composing a message before sending it.

In the *mapping stage*, we need a mapping model which is trained on a bilingual corpus or a set of aligned texts. An example corpus for natural languages are the Canadian parliamentary proceedings which are in both English and French. In the context of code, we adapt StackOverflow posts, which then serve as a *bilingual corpus* that discusses programming tasks in both English and in code. To perform the translation, we also need a language model for code, which we could extract from a large corpus of source code in the open-source projects in GitHub. We can combine the mapping model and code language model to perform the translation using a noisy-channel model and Bayes rule. We find the most probable output code, c according to the following:

$$\arg\max_{c \in C} P\left(C|E\right) = \arg\max_{c \in C} p\left(C\right) \times p\left(E|C\right)$$

where $p\left(E|C\right)$ is calculated from the bilingual StackOverflow corpus as the number of posts that an English word, E, and a code element, C, co-occur in. $p\left(C\right)$ is calculated by the code language model based on the occurrence frequencies of the code elements in the GitHub corpus. This simple model is at the heart of statistical machine translation and in our example suggests a useful but incorrectly ordered set of code elements.

Natural languages tend to have similar word *ordering* (*e.g.,* left to right). Even in a translation from English to German one may only need to switch the order of the noun and verb. Mistakes in word order can lead to rough translations that require the reader to perform reordering manually. In stark contrast, code must compile and the unlike English and German does not always work sequentially from left to right. For these reasons, unlike previous works, we do not use a sequence of tokens to represent code [9], [11], [10], [3]. Instead, we develop two new techniques to address that challenge. First, we develop an algorithm, called **context expansion**, for the mapping stage, that prioritizes the order of translation for each English word in the query by considering the contexts of the surrounding words and the already-translated code elements in the result. Second, we design GRASYN, a **graph synthesis** algorithm that synthesizes the graphs representing API usages. We use GraLan graphs [12] as the underlying representation, which are more suitable to source code than the n-grams, *i.e.,* sequences of tokens used by other translation approaches. We extract a large number of graphs from the GitHub corpus and learn from these graphs to order the translated code elements into the most likely API usage graph with control units, and data and control dependencies among API elements. Our code ordering stage requires the stepwise synthesis and combination of likely subgraphs based on the API elements we obtained from the mapping stage. We use a beam search on graph synthesis, which is more challenging than on n-grams, to limit the set of likely graphs in each synthesis iteration, making T2API efficient. The output of T2API is an API graph and an API template. In brief, we make the following contributions:

1) We treat StackOverflow as a bilingual corpus describing programming tasks in English and code. We perform substantial automated cleaning on over 236k StackOverflow posts. The alignments between code and English on StackOverflow describe a wide range of tasks allowing us to make high-quality translations.
2) We adapt the simple IBM 1 lexical translation approach to a software engineering context by introducing a novel code element expansion stage. This stage is based on the co-occurrence frequency of code elements in StackOverflow posts. This expansion stage is optional and increases the precision and recall of our approach by as many as 20 percentage points.
3) Our underlying representation is a graph, which is more suitable for API usages than the sequence of API elements used in previous work [9], [3]. 85% of the synthesized graphs do not exist as a whole in the training data, thus, they cannot be found by code search.
4) Our first evaluation involves a manually curated benchmark consisting of 250 unseen, randomly sampled StackOverflow posts which we make publicly available. These posts contain substantially more and longer inputs and complex translated outputs than those from the state-of-the-art approaches. Our best approach attains a median precision of 67% and recall of 100%.

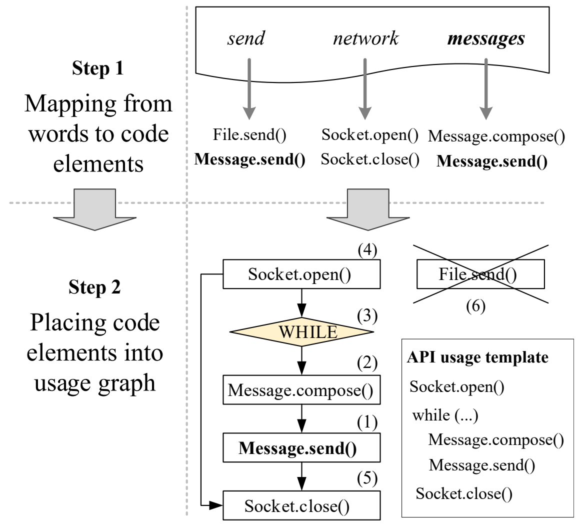


Fig. 1: Example of API Usage Graph Generation in T2API

5) In our second evaluation of T2API, we had four professional developers and five graduate students judge the usefulness of our synthesized code snippets with respect to the purpose of the StackOverflow posts. 77% of the translated code templates were deemed to be useful.
6) T2API is available as an online web-based tool, allowing users to type in an English query and receive a usable code template and view the code graph.

II. ILLUSTRATING EXAMPLE

T2API is designed with the principle of statistical machine translation [13], [14]. Since programs are viewed as graphs, we develop GRASYN, a novel graph synthesis approach within T2API, to place the API elements generated via a context expansion process into the most *natural* API usage graph that is relevant to the query. Fig. 1 illustrates T2API via an example. Assume that a user want to have an API template for sending message over Internet. (S)he types the query *"Send network messages"*. T2API processes it in two steps:

(1) **Mapping from words to API elements and Contextual Expansion**: We use maximum likelihood IBM Model [15] to derive the m-to-n mappings for the pairs of words and API elements with their likelihoods. Contextual expansion is proceeded as follows:

- Initially, T2API selects a pair of central words and API element as a starting point for expansion. The central API element needs to be highly relevant to the input. Thus, we choose the API element having highest total mapping score with all the words (measured by the IBM Model). For example, Message.send is the central API since it has the highest mapping scores to *"send"* and *"messages"*. Then, the word *"messages"* becomes our central word.
- T2API then starts expanding by considering the words in the input in a prioritized order, *e.g.,"messages", "send", "network"*, with respect to the co-occurrences with the central word. It maps each word to k elements. The result is a collection of mapped API elements considering

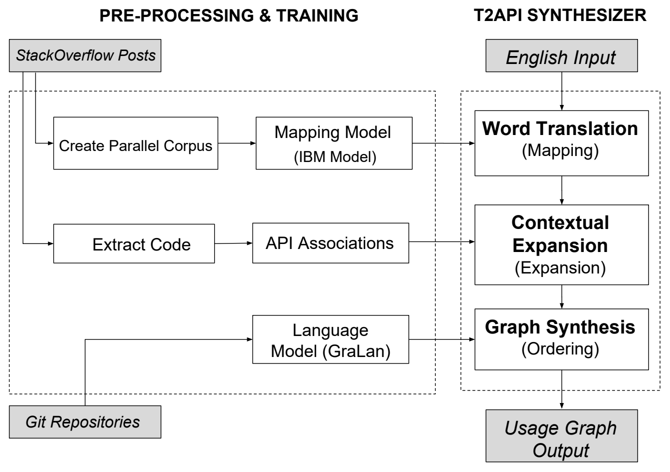


Fig. 2: T2API Architecture

Question 9292954
Title: How to make a copy of a file in Android
In my app I want to save a copy of a certain file with a different name (which I get from user) Do I really need to open the contents of the file and write it to another? What is the best way to do so?
Answer: (Rating 132)
To copy a file and save it to your destination path you can use the method below.

```
public void copy(File src, File dst) throws IOException {
    FileInputStream in = new FileInputStream(src);
    FileOutputStream out = new FileOutputStream(dst);

    // Transfer bytes from in to out
    byte[] buf = new byte[1024];
    while (in.read(buf) > 0) {
        out.write(buf,0);
    }
    in.close();
    out.close();
}
```

Fig. 3: StackOverflow Question and Answer 9292954

the contexts. In Fig. 1, after this step, we have Message.compose, Message.send, File.send, Socket.open, and Socket.close.

(2) **Placing API elements into a usage graph via GRASYN**: This step corresponds to the addition, removal and re-ordering of terms in the target language in a SMT [14]. Since we use graphs rather than sequences, we perform graph synthesis:

- T2API starts with the central API element Message.send as the initial node of the graph synthesis process. It then gradually adds other nodes (and inducing edges) according to *the occurrence likelihood of the new graph*. It also considers the addition of control units (e.g., for, if) to make the graph complete. During this, there may be disconnected graphs that could be later joined by adding likely nodes. The final graph is connected. In Fig. 1, the numbers show the order that the nodes are added.
- T2API stops after all the APIs produced via context expansion are covered. Nodes considered redundant due to low relevancy with other nodes (*e.g.,*File.send) are removed. The output template is shown in Fig. 1.

III. PRE-PROCESSING AND TRAINING OF MAPPING AND LANGUAGE MODELS

We develop T2API, a graph-based machine translation model with the general architecture shown in Fig. 2. This section presents T2API's pre-processing and training with two key building blocks: the mapping and language models.

Create Parallel Corpus and Mapping Model. The goal of the mapping model is to learn the mappings between individual words and API elements. We make use of the IBM Model [15] for this word-to-API mapping task. The principle of IBM Model is to learn the mapping pairs via maximizing the likelihoods of observing them over a large number of pairs of English texts and corresponding sequences of APIs. For example, in such a parallel corpus, a description *"write to a socket"* corresponds to the code with the following API elements Socket.new, Socket.write, and Socket.close.

ACE tool. To build such training pairs for IBM Model, we processed a large number of StackOverflow (SO) posts using the ACE tool [16]. Fig. 3 shows the question and an answer for the post #9292954. ACE can identify type and package information from API elements in freeform texts and incomplete code snippets. It extracts APIs embedded within texts. It removes stopwords (a, the, etc.) from the posts and extracts keywords/keyphrases (copy, file, save, etc.).

The training data for the IBM Model produced by Create Parallel Corpus is the collection of pairs in which each pair consists of a *textual description* (excluding the API elements in the text and the code snippet) and the set of *extracted API elements* from both places. Such exclusion is needed since we aim to map the words and the code elements. Otherwise, the embedded API elements will affect the mappings of the English words in the query. For the SO post above, the extracted texts include *"make copy file Android", "app save copy file different name", "open contents file", "write to file", "save destination path", etc*. The corresponding set of API elements includes File, FileInputStream.new, FileOutputStream.new, etc. We also keep the control units, e.g., while, for, if, etc.

After training, the result includes m-to-n *mappings from individual words to individual API elements*: ['file'$\rightarrow$File, 'file' $\rightarrow$ FileInputStream.new, 'save' $\rightarrow$ FileOutputStream.write, 'write' $\rightarrow$ FileOutputStream.write, 'contents' $\rightarrow$ byte[].new, 'close' $\rightarrow$ FileInputStream.close, 'close' $\rightarrow$ FileOutputStream.close, etc.]. The mappings and scores are used to infer the API elements during expansion.

Language Model. A language model estimates how likely a sentence occurs in the target language. Since we perform graph synthesis, we need a graph-based language model. We use GrouMiner [17] to build *API usage graph* in which the nodes represent API object instantiations, variables, API calls, field accesses, and control units (*e.g.,*while, for, if). The edges represent the control and data dependencies between the nodes. Fig. 4 shows the usage graph for the code in Fig. 3. We then use GraLan [12], a graph-based language model that can compute the naturalness via the occurrence likelihood of any API usage graph after we train it with a large corpus of GitHub projects (from which usage graphs are built).

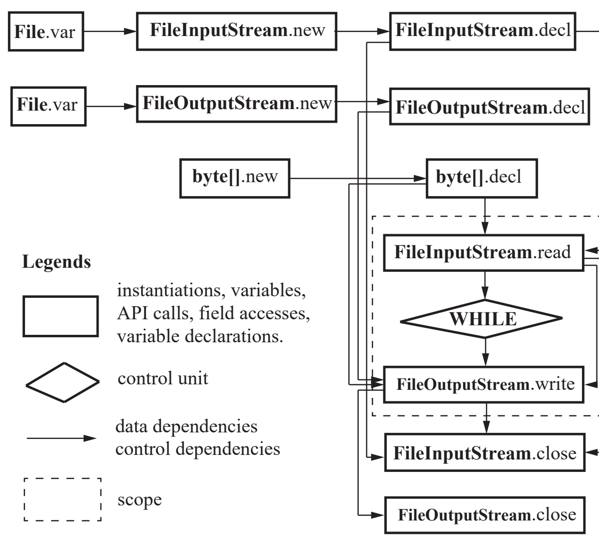


Fig. 4: API Usage Graph Representation

```
1  function Expansion ($T$, $C$, IBM_Model $M$)
2    $I$ = {$c_i \in C$ that has highest total mapping score with $t_i \in T$}
3    $J$ = {$t \in T$ that has highest total mapping score with $c_i \in I$}
4    return ExpandCodeElements($T$, $I$, $J$)
5
6  function ExpandCodeElements($T$, $I$, $J$)
7    while ($T \setminus J \neq \emptyset$)
8      Select $t \in (T \setminus J)$ with highest TextCooccurrenceScore($t$,$J$)
9      $Q$ = {$M(t)$}
10     Order m$s \in Q$ descending by CodeCooccurrenceScore($m$,$I$)
11     $I = I \cup$ {top $k$ elements m$s \in Q$ with relev_scores from $M$}
12     $J = J \cup \{t\}$
13   return $I$
14
15 function TextCooccurrenceScore($t$, $J$)
16   return $\prod_{j \in J} \frac{CountPosts(j,t)}{CountPosts(j)}$
17
18 function CodeCooccurrenceScore($m$, $I$)
19   return $\prod_{i \in I} \frac{CountPosts(i,m)}{CountPosts(i)}$
```

Fig. 5: API Element Expansion with code and word contexts

T2Api Synthesizer. We develop a graph synthesizer that uses the trained mapping model and language model to synthesizes new API usage graphs from a query in 3 steps: word translation, context expansion, and graph synthesis. Let us explain the first two steps in Section IV and graph synthesis in Section V.

IV. Contextual Expansion Algorithm

After the IBM Model is trained, it gives us the *word-to-API mappings*. We use the trained IBM Model to perform word translation on the given query, *i.e.,* for each word, it gives a ranked list of mapped API elements. We use that result for context expansion. The goal of the **expansion** step is to collect the set of API elements that are most relevant for the query. A naive solution is to collect the top-ranked API elements for each word into the resulting set. That solution faces two key challenges. First, a word can be used in different contexts with other words describing different programming tasks, and it can be mapped to multiple APIs. Thus, we cannot take advantage of the already-translated words that might provide context for expanding API elements from the current word. Second, the relevant API elements are inter-dependent on one another. A choice of mapping for an element could affect the mapping for the next API element. If we simply pick the most frequent element for a word, it may not fit with the current context of both text and code. Thus, we develop a contextual expansion algorithm to collect the set of relevant API elements S.

To account for the contexts, we rely on two principles. First, to account for code context (dependency among selected API elements), the next element must have the highest relative co-occurrence frequency with all API elements that were already selected. The co-occurrence frequencies of the API elements in the posts are computed by the API association module via ACE (Fig. 2). The idea is that if c and c' (*e.g.,*FileInputStream.new and FileInputStream.close) often go together, it indicates a usage relation. The chosen element must be among the ones that are likely mapped to the current word.

Second, to consider the word context in the given query, the next word t to be translated must have the highest relative co-occurrence frequency with all the already-translated words. Our intuition is that the co-occurrence of t and t' could be part of a description of a certain task (e.g., *"open"* and *"file"*).

Algorithm. Fig. 5 shows the pseudo-code of our algorithm. It takes the query T, the set of top-ranked API elements C produced by running the trained IBM Model (mapping model) on the words in T *without* considering contexts, the trained IBM Model with word-to-API mappings, and returns the set of API elements relevant to T and their scores.

First, we select the central element having highest total mapping score with the words in the query T (line 2) since that element is most likely relevant to the query. For example, in the query *"open file contents"*, considering the elements mapped with the word *"file"* and the elements for *"open"* and *"contents"*, the central API element could be FileInputStream.open or FileOutputStream.open. We then map the central API element back to the query (using IBM Model) to identify the central word t (line 3). Next, the API elements that are mapped to multiple words but do not fit the current context will not be considered as relevant (function ExpandCodeElements). Specifically, during expansion, that function proceeds in a stepwise manner expanding the set of API elements. We consider the *word context* in T. At a step, we select a word t for translation such that t must have highest relative co-occurrence frequency with all of the already-translated words in all SO posts (line 8, TextCooccurrenceScore). Relative co-occurrence frequency is scored by the formula at line 16.

For *code context*, the next API element to be chosen must have the highest relative co-occurrence frequency with all API elements that were already collected (line 10 and function CodeCooccurrenceScore). Such relative co-occurrence frequency is scored by the formula at line 19. That next element must be among the most likely mapped elements (*i.e.,* highest mapping scores) for t according to IBM Model (line 11).

After the expansion, for each word in the query, we take the top-k API elements (line 11). We are not interested in the order of the elements as the order of the English words does not map to a compilable order for code. The elements and *their relevance scores (to the query) computed by IBM Model* are used by the graph synthesizer to synthesize the usage graph.

V. GraSyn: Usage Graph Synthesizer

We develop a graph synthesizer that takes the API elements and their relevance scores from the previous step, and puts them together in an API usage graph relevant to the query.

Fig. 7 shows the pseudo-code for the GraSyn algorithm. It takes as input the query T, the set of API elements J with their relevance scores, and the trained graph-based language model GL. It produces a ranked list of candidate usage graphs.

At lines 2–5, we initialize the synthesis. First, we use the central API element detected in the previous step as the starting node in a single-node graph g_0. The score of this single-node graph is calculated as the product of the relevance score and the occurrence probability of the node central in the graph corpus. At lines 9–16, we extend each graph in the list of unexplored graphs CG (line 9), which is initialized with the single-node graph g_0. Using a beam search, we pick the graph g with the highest score first. We consider each node n in the API nodes in J that have not been yet explored as a potential expansion ordered by their relevance scores (lines 11–13). The rationale is that the nodes with higher scores are more likely related to the query. We attempt to extend the current graph g with the current node n via the function ExtendGraph (line 14). If the expansion yields new candidate graphs, we append them to CG. We repeat the process until there is no more candidate left. The graphs covering all nodes in J are added to the candidate list FG (line 16).

ExtendGraph: to extend the current graph (lines 19–29), we use the language model to find all possible extended graphs from the current graph g (line 21). If after removing the node n and its connecting edges from the extended graph eg, we get the exact match to the current graph g (line 24), we ask the graph-based language model for the occurrence likelihood of the new graph formed by g and the new node n (GetProb(g,n) at line 25). The score for the newly extended graph eg is computed (line 25). Relevance score of n (relev_score) allows us to give higher scores to the nodes relevant to query T. With beam searching, we need to prune the extended graphs with low scores, thus, we keep eg only if its score is in the top list among other extended graphs at this step (line 26).

An alternative solution for ExtendGraph would be to add to g a new node and keep only those with high occurrence probabilities. However, that is less efficient since there are many more possibilities to add a new node to g than the number of the feasible extended graphs from g (we need to maintain only the extended graphs for g that were observed in the corpus).

At lines 27–28, if n is not a feasible extended node according to the graph language model, we still add it to the current graph g, hoping that it will be connected in a later expansion. In this case, our new graph is disconnected. The reason for such disconnected components is as follows. Since a new API usage graph might not occur in its entirety in the training corpus, during the expansion of the current usage graph, we might face the situation where smaller, yet-unrelated usages (*i.e.,* subgraphs) might be formed first and later connected together via new edges to form a larger API graph.

To support this broader synthesis, we allow the *intermediate synthesized graph to be disconnected* (*i.e.,* containing disconnected components). Specifically, we allow an expansion in which we add a node without any inducing edges. The score for a disconnected graph is the average score of the scores of its connected components. We assign the score for a connected graph (including a single-node graph) with its occurrence probability in the corpus. We continue to process the newly extended graphs (line 14) and remaining API elements in R until all elements are covered or the scores of extended graphs are not in the top list (line 26), *i.e.,* they are not candidates.

Finally, in the final candidate graphs, we remove the single, disconnected nodes (line 17), since those isolated nodes are likely the ones that were incorrectly included by the expansion algorithm. Then, the candidate API graphs are presented.

Example. Fig. 6 illustrates the result of each expansion step for the API usage shown in Fig. 4 (we show only the top-ranked candidate graph).

1) At step 1, assume that FileInputStream.new (labeled as (1)) is chosen as the central node (among FileInputStream.new and FileOutputStream.new). At step 2, since in the training data, File.var is likely used as a parameter for the instantiation FileInputStream.new, it is newly added and marked with (2).

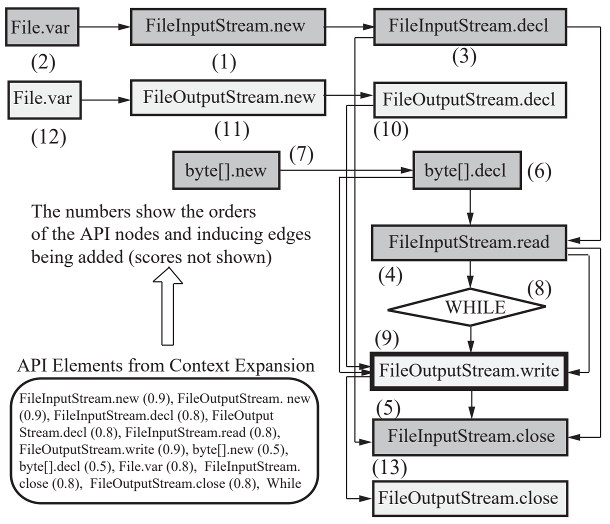
2) At steps 3-5, a variable declaration, read and close are likely to be used on a FileInputStream, thus FileInputStream.decl, FileInputStream.read, and FileInputStream.close are added. Relevance scores decide the order of adding the nodes.

3) At steps 6-8, since in the corpus, FileInputStream.read is often used to read data from a file into an array among which an array of byte matches with the element byte[].decl. Thus, it is added at step 6, leading to the addition of its instantiation byte[].new at step 7. At step 8, the WHILE node is added since in training, the model observes that FileInputStream.read with an array of bytes byte[].decl often goes with a while loop.

4) Step 9 is an interesting step because after step 8, we have a small usage for reading into a file corresponding to a subgraph (1)–(8), including the nodes highlighted in a darker color. At step 9, the node FileOutputStream.write (with a darker border) is added from WHILE, byte[].decl, and FileInputStream.read since the smaller sub-graph involving those nodes (6),(4),(8) and FileOutputStream.write (9) occurs frequently. It represents a smaller usage in which a while loop is used to read from a FileInputStream to a buffer and write its contents to a FileOutputStream.

5) That allows us to expand to the nodes (10)–(13), which correspond to another usage of writing to a file via FileOutputStream. Specifically, at the step 10, FileOutputStream.decl is added because it occurs often before FileOutputStream.write. At step 11, an instantiation with FileOutputStream.new occurs often for a declaration of that type. Then, at step 12, File.var is connected because it is used as an argument for such instantiation. After that, FileOutputStream.close is inserted because it often occurs after FileOutputSream.write. Finally, we have a larger API usage for both file reading and writing.

If the occurrence probability of FileOutputStream.new is higher in the training data, it will be the central API and the order of nodes being added will be different. The usage of writing to a file via FileOutputStream will be formed first. There could be cases where smaller, independent usages are expanded. We keep a disconnected graph with its connected components, which will be connected later for the same result.


File.var
(2)
FileInputStream.new
(1)
FileInputStream.decl
(3)
File.var
(12)
FileOutputStream.new
(11)
FileOutputStream.decl
(10)
byte[].new
(7)
byte[].decl
(6)
The numbers show the orders
of the API nodes and inducing edges
being added (scores not shown)
FileInputStream.read
(4)
WHILE
(8)
(9)
FileOutputStream.write
(5)
FileInputStream.close
(13)
FileOutputStream.close
API Elements from Context Expansion
FileInputStream.new (0.9), FileOutputStream. new (0.9), FileInputStream.decl (0.8), FileOutput Stream.decl (0.8), FileInputStream.read (0.8), FileOutputStream.write (0.9), byte[].new (0.5), byte[].decl (0.5), File.var (0.8), FileInputStream. close (0.8), FileOutputStream.close (0.8), While

Step	New nodes	New edges
1	FileInputStream.new	[]
2	File.var	File.var->FileInputStream.new
3	FileInputStream.dec	FileInputStream.new -> FileInputStream.decl
4	FileInputStream.read	FileInputStream.decl-> FileInputStream.read
5	FileInputStream.close	FileInputStream.read-> FileInputStream.close FileInputStream.decl -> FIS.close
6	byte[].decl	byte[].decl->FileInputStream.read
7	byte[].new	byte[].new -> byte[].decl
8	WHILE	FileInputStream.read->WHILE
9	FileOutputStream.write	WHILE->FileOutputStream.write {byte[].decl,FIS.read}->FOS.write
10	FileOutputStream.decl	FileOutputStream.decl-> FileOutputStream.write
11	FileOutputStream.new	FileOutputStream.new-> FileOutputStream.decl
12	File.var	File.var->FileOutputStream.new
13	FileOutputStream.close	FileOutputStream.write->...close FileOutputStream.decl->...close

Fig. 6: Graph Synthesis Example for a Top-ranked Candidate Usage Graph (Relevance scores not shown)

```
function GraphSynthesis(Query T, APINodes J, LangModel GL)
  Graph $g_0$ = new Graph();
  APINode central = GetCentralElement(J);
  $g_0$.add(central);
  $g_0$.score = GetScore($g_0$);
  Queue CG = $\emptyset$; // unexplored graphs
  Append($g_0$,CG);
  Queue FG = $\emptyset$; // candidate graphs
  while (CG $\neq \emptyset$)
    Take the graph g with highest score off CG
    R = J \ g.Nodes // R: remaining API nodes in J
    Sort(R); // Sort nodes in R by relevance scores
    foreach Node n in R
      GraphList G$^+$ = ExtendGraph(g,n,GL); //Try to extend
      if (G$^+ \neq \emptyset$) Append(G$^+$,CG)
      if (G$^+$.nodes \ J = $\emptyset$) Append(G$^+$,FG)
  return Filter(FG);

// Extend g with node n and inducing edges to get new graphs
function ExtendGraph(Graph g, Node n, LangModel GL)
  GraphList EG = GL.FindExtendingGraphs(g);
  GraphList RG = $\emptyset$;
  foreach Graph eg in EG
    if ($g = eg \ominus n$)
      eg.score = g.score $\times$ GL.GetProb(g, n) $\times$ n.relev_score(T)
      if (eg.score on the top list ) RG.add(eg);
  if (EG is $\emptyset$ or ($g \neq (eg \ominus n)$ with all $eg$s))
    eg = $g \oplus n$ with eg.score = GetScore($g \oplus n$)
return RG
```

Fig. 7: GRASYN: Usage Graph Synthesis Algorithm

VI. Empirical Evaluation

To empirically evaluate T2API, we had conducted experiments with the following questions:

RQ1. How accurate is T2API in generating API templates and usage graphs with respect to a reference benchmark?

RQ2. How useful is the context expansion algorithm?

RQ3. How useful and relevant to the queries are the synthesized API templates and usage graphs from T2API?

A. Data Collection

Large StackOverflow Corpus. For the mapping phase, we trained the IBM model (part of Berkeley Aligner [18]) with the StackOverflow dataset (Table I) after pre-processing the posts.

TABLE I
LARGE STACKOVERFLOW CORPUS FOR TRAINING IBM MODEL

Number of posts	236,919
Size of English dictionary	701,781
The number of distinct keywords	103,165
Size of code element dictionary	11,834
Number of mappings to code per keyword	17

TABLE II
LARGE CODE AND USAGE GRAPH CORPUS TO TRAIN GRALAN

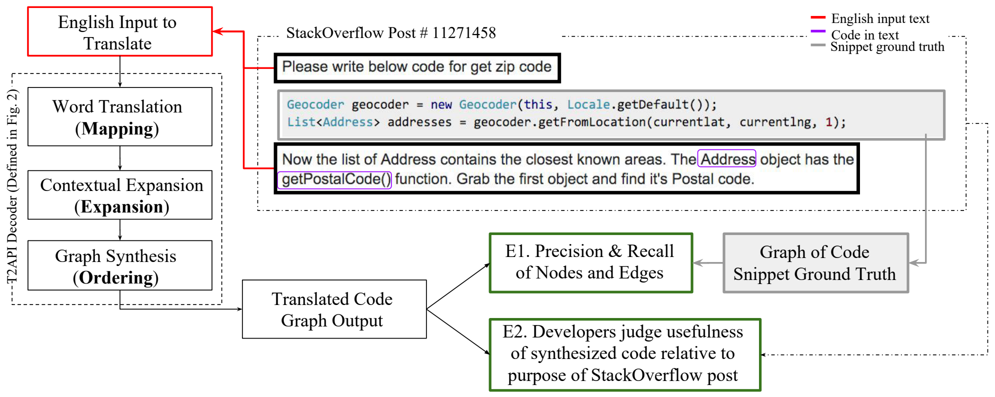
Number of projects	543
Number of source files	29,524
Number of methods	317,792
Number of extracted graphs	284,418,778
Number of unique graphs	82,312,248
Number of unique API elements	113,415

This dataset of **236,919 SO posts** was processed via ACE [16]. We randomly removed 250 posts from this dataset and used as a test set (will be explained in Translation Benchmark).

Training. We used the remaining posts to build the training corpus, which is the collection of pairs in which each pair consists of 1) a textual description (excluding the API elements in the text and the code snippets), and 2) the set of extracted API elements by ACE in text and code. See Section III, *Create Parallel Corpus and Mapping Model* on building those pairs. After training, T2API performed context expansion. For diversity, SO posts with the same snippets are excluded from test set. Each keyword is mapped to an average of 17 APIs.

Large Code and Usage Graph Corpus. To train the graph-based language model GraLan [12], we collected a set of 543 Java and Android projects from GitHub (Table II). We selected the projects with well-established histories so that their code is compilable and can be semantically analyzed to build usage graphs. We collected a large number of usage graphs (284M) with 82M unique graphs and 113,415 unique API elements.

Translation Benchmark. The results from statistical machine


English Input to Translate
T2API Decoder (Defined in Fig. 2)
Word Translation (Mapping)
Contextual Expansion (Expansion)
Graph Synthesis (Ordering)
Translated Code Graph Output
StackOverflow Post # 11271458
English input text
Code in text
Snippet ground truth
E1. Precision & Recall of Nodes and Edges
Graph of Code Snippet Ground Truth
E2. Developers judge usefulness of synthesized code relative to purpose of StackOverflow post

Please write below code for get zip code

```
Geocoder geocoder = new Geocoder(this, Locale.getDefault());
List<Address> addresses = geocoder.getFromLocation(currentlat, currentlng, 1);
```

Now the list of Address contains the closest known areas. The Address object has the getPostalCode() function. Grab the first object and find it's Postal code.

Fig. 8: First evaluation (E1): precision and recall of synthesized code is calculated relative to the StackOverflow code snippets. Second evaluation (E2): developer judges compare the purpose of the post to determine if the synthesized code is useful.

translations are typically evaluated against a reference translation benchmark that is manually created by humans [13, Ch.8]. Unfortunately, no such corpus exists for English and API usages. However, SO is in effect a bilingual corpus that describes programming tasks in both English and code. If we use the SO posts with English text that describes the code snippets in the same post, we would have a benchmark. For example, consider the post in Fig. 8, the texts in the boxes in bold, describes how to get the zip code from an address using the Android APIs. We can use the text as an input to T2API. The code snippet serves as a ground truth. We curate a benchmark of 250 posts randomly taken from the Large SO Corpus; each test post has high rating with only one code snippet since we want automatic comparison with our result. We'll explain how to extract texts to be used as input next.

B. Evaluation Settings and Procedure

E1. Evaluation against Code Snippets in StackOverflow in Benchmark. We compared the synthesized code from T2API against SO code snippets. We conducted our experiments with two distinct use cases in mind. The persona in the first use case does not know any of the API elements and writes their input entirely in English. We refer to this as the *pure input* case. This persona may be a developer who is learning the API. The second persona is a developer who knows some of the API, so he or she uses a mix of API elements and freeform text. This case is called *mixed input*, *e.g.,"I want to use getPostCode() for a GPS location"*. Mixed input is common in SO.

To build the input, we extract the text surrounding a code snippet within a SO post (thick-border boxes in Fig. 8), get the keywords and use them as a query. For **mixed input**, *all keywords and the API elements within the texts are used as input*. We do not include code snippet in the mixed input to T2API (code snippet is used as ground truth). For **pure input**, *those API elements as well as the code snippet are removed.*

E2. Developers' Judgement. Developer judges compare the purpose of the post to see if the synthesized code is useful (Section VI-E).

Procedure. Our evaluation includes the following steps:

1. Use T2API to translate a mixed or pure input into an output usage graph (T2API's pipeline in Fig. 8),

2 Use the code snippet in the post as the ground truth (the shaded box in Fig. 8),

3. In Evaluation 1, calculate precision and recall for the nodes (API elements, control units) and the edges (data/control dependencies) among the API elements. The edges also indicate the order,

4. In Evaluation 2, developers judge how useful our code snippet is with respect to the purpose of a post (Section VI-E).

C. Evaluation Metrics

We take a conservative and stringent approach with the calculation of accuracy. We evaluated only the top-1 results from T2API instead of, for example, the top-5 results. We evaluated node accuracy (API elements and control units) and edge accuracy (orders and dependencies) against SO code snippets. If in the ground truth, the developer creates object instances in a different order, we conservatively marked each of our edges as incorrect even if the ordering would not affect the behavior of the code. For each synthesized graph g_{syn}, we compared it against the usage graph g extracted from the StackOverflow code snippet. We use PPA [19], a partial program analysis tool to parse the code and build g. We measured Precision and Recall for the sets of nodes and edges. Recall on nodes is defined as the ratio between the number of shared nodes in g and g_{syn} and the number of nodes in g. Precision on nodes is the ratio between the number of shared nodes in g and g_{syn} over the number of nodes in g_{syn}. Similarly, we define Recall and Precision on edges. BLEU score [20] was not used since it is not defined for graphs.

D. Precision and Recall Results

Fig. 9 show the distribution of precision and recall for nodes and edges for each post through violin plots. A violin plot combines a boxplot and a kernel density plot (shown vertically). The boxplot is represented as the box in the middle of the plot. The bottom of the box is the 25th percentile and the top is the 75th. The horizontal line is the median. The left of each violin plot is precision and the right is recall. Table III shows the median precision and recall for each input type (pure English or mixed code and English), and each mapping model: the Word Mapping rows correspond to the mapping model

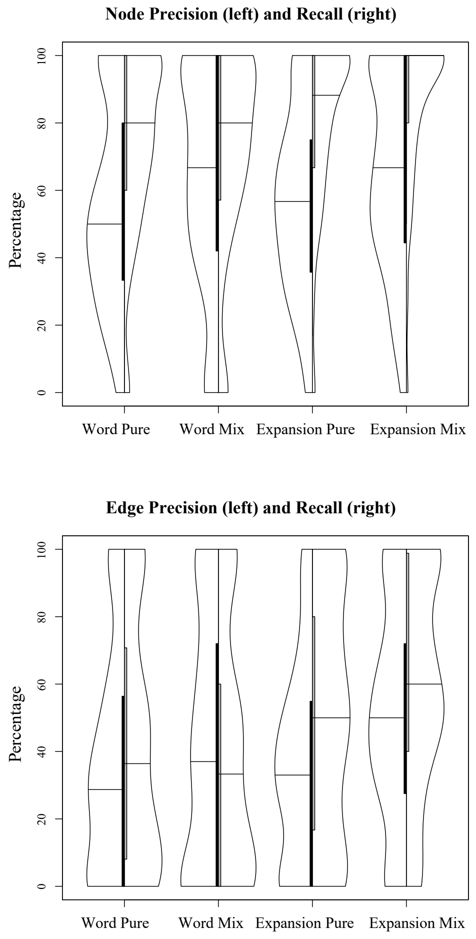


Fig. 9: Violin plots of the precision and recall distributions for the 250 posts in our benchmark. Our Contextual Expansion approach outperforms Word Mapping (IBM Model) approach.

without context expansion (IBM Model), while the Expansion rows are for the ones with context expansion.

Node Accuracy. In Fig. 9, all the node distributions are clearly skewed towards higher precision and recall. For example, with Expansion and mixed inputs, *for over half of the posts*, **T2API achieves 100% recall and for 75% of the posts, it reaches 80% recall**. The precision is also skewed towards higher values, with **a median of 67% and over 25% of the posts having 100% precision**. To put the results in context, if SO code snippets have a median of 10 API elements, T2API produces all of them (recall is 100%). It correctly derives 6.7 API elements. Users need to remove 3.3 elements. This is important for developers since they do not need to find extra elements, which cost much more effort than removing a few incorrect ones. For the bottom 25% of posts, users need to find 1 additional API element. The accumulated average **precision and recall are 64.9% and 94.1%** (not shown).

The **mapping approach with expansion substantially outperforms the word mapping approach (mapping without expansion) on node recall, up to 20% points higher**. For precision, two approaches are more evenly matched, however in the pure-input setting, the precision for nodes is 9% higher.

TABLE III
GRAPH SYNTHESIS PRECISION AND RECALL (%). DISTRIBUTIONS ARE SKEWED AND DISPLAYED IN FIGURE 9

		Median Node		**Median Edge**	
Input	**Mapping Model**	**Prec**	**Rec**	**Prec**	**Rec**
Pure	Word Mapping	50	80	29	36
Pure	**Con. Expansion (T2API)**	57	88	33	50
Mixed	Word Mapping	67	80	37	33
Mixed	**Con. Expansion (T2API)**	67	100	50	60

Edge Accuracy. When we predict the edges (orders and dependencies), with Expansion for mixed input, the result is skewed towards larger values, with median precision and recall of 50% and 60%, respectively. At the 75th percentile, the corresponding values are 72% and 99%. For the simple Word Mapping, the edge precision and recall are skewed towards lower values with the median precision and recall of 29% and 37%, respectively. By considering contexts, **T2API with Expansion drastically outperforms Word Mapping** by including the missing but relevant nodes.

T2API produces good translation even when users do not know any of the API elements. For pure English inputs, the graph synthesizer with Expansion model has high node precision and recall of 57% and 88%, respectively. Contextual Expansion still outperforms Word Mapping in this use case. Interestingly, comparing the top performing graph synthesizer with Expansion for mixed and pure inputs, we see that people who use pure English descriptions only have a 10 point and 12 point reduction in median precision and recall compared with those who mix code and English in their queries. A similar phenomenon applies to the edges. The reason is that irrelevant code elements could be included in the mixed posts.

Note: **85% of the synthesized graphs do not exist as a whole in the training data, thus, cannot be found by code search, and all top-1 graphs are unique**.

E. Result from Evaluation by Developers

In this experiment, we asked human judges to evaluate how useful our synthesized templates/graphs are. Four professional developers and five graduate students who are not involved in this work evaluated the top synthesized results for 100 randomly sampled posts from our benchmark. We used the following evaluation procedure:

1) As described in Fig. 8, we synthesized code by translating the English text (*mixed input*) contained in each StackOverflow post. We used our most accurate approach, *i.e.,* the graph synthesizer with Expansion model.
2) We randomly assigned 100 StackOverflow posts in our benchmark to two judges each.
3) Each judge was shown the post title, textual descriptions, existing snippet, and synthesized code (*e.g.,* Fig. 10) and API usage graph.
4) Judges independently determined if the synthesized code for each post is 'useless' or 'useful' given the original purpose of the StackOverflow post.
5) When disagreement occurred, we included a discussion phase between pairs of judges. If a consensus is not reached between judges, we conservatively assign T2API's synthesized code to the 'useless' category.

Developers judge the usefulness of our synthesized code snippet with respect to the purpose of the post. A synthesized API snippet is *considered useful if a subject thinks that the code snippet satisfies the purpose of the post. It is useless if the subject would rather not use the generated snippet and would rather re-write the code.* The judges use the existing snippet as context to evaluate the purpose. If our generated code is incorrect, they could more easily recognize the problem. If our solution is a correct alternative but different, the judges may see it as incorrect, biasing against our solution. In either case, the bias is not toward a "useful" categorization.

Of 100 resulting API templates, 77 are deemed to be useful by two judges. This is consistent with the state-of-the-art approaches. SWIM's evaluation [9] involved one professional developer judging the output of 30 queries. They found that 21/30 or 70% of their top recommendations were relevant to the query. DeepAPI [3] had two developers evaluate 30 queries and found that 23/30 or 77% of the top snippets were relevant. Not only is T2API competitive with the existing work in terms of developers perceived usefulness, it is able to translate *larger queries into more complex code templates (with control units and dependencies).* All the data for this study is available on our website. Let us explain in details the result next.

F. Result Analysis

There are two recent related work to T2API: SWIM [9] (phrase-based SMT with IBM Model and n-gram) and DeepAPI [3] (with RNN Encoder-Decoder).

T2API can take a large input of English text as in SO posts, and synthesize a sophisticated API template and usage graph specifically with data and control dependencies. In our study, we used T2API on the texts of real-world SO posts with +131 words. For the output, it synthesizes a usage with up to 23 API elements and control units. Among 250 resulting templates, 39 of them (15.6%) have control units (*e.g.,*for, if, while). All of them have more than one API element.

In the evaluation for SWIM [9], the number of words in the inputs is from 1–4, while the number of generated API elements is from 1–3. T2API can synthesize conditions for conditional statements, loop bodies, method arguments, and overriding methods because our graph representation, API Usage Graph [17] contains all of those. In contrast, SWIM uses syntactic rules to generate control units, but does not use program dependencies and so cannot generate sophisticated data and control dependencies.

For DeepAPI, since it relies on RNN, the larger the size of the input, the higher complexity of the training and predicting. In fact, when DeepAPI was publicly available, as we entered the larger JDK SO posts in our benchmark into DeepAPI online web tool [21], it took several minutes (due to its high computation) and most of the resulting code were irrelevant. Regarding output, DeepAPI's sequences are a list of API elements without data/control dependencies.

Illustrating examples. To illustrate the differences, let us analyze the outputs to the same queries reported in their papers.

All three tools reported the result dealing with the query on file copying. The SWIM output is simply File.Copy (Table 1 of [9]). T2API generates an usage graph and code template

```
 1  LocationManager varLocationManager;
 2  varLocationManager.removeUpdates();
 3  Location varLocation = varLocationManager.
          getLastKnownLocation();
 4  if (varLocation.isProviderEnabled();){
 5      varLocation.getLatitude();
 6      varLocation.getLongitude();
 7      Log.i();
 8      varLocation.requestLocationUpdates();
 9  }
10  Context varContext;
11  varContext.getSystemService();
```

Fig. 10: Translated template for SO post 13761430 (how to find the previous location when GPS is not present)

that opens an Input- and an OutputStream and loops through each byte (see Fig. 6). The result from DeepAPI has only 6 out of those 13 APIs produced by T2API (Table 2 of [3]). Moreover, DeepAPI does not produce program dependencies and control units as in T2API.

As another example, the query *"play audio"* leads to the incomplete sequence from DeepAPI: SourceDataLine.open SourceDataLine.start. In contrast, our synthesized snippets to play audio contain various nodes/edges that set up a MediaPlayer and a data source, play the audio, and create a callback listener to handle what happens when the audio finishes.

SWIM's result for the query *"execute sql statement"* includes SqlCommand.ExecuteNonQuery and ExecuteReader, while that from DeepAPI is not quite relevant to SQL: Connection.prepareStatement, PreparedStatement.execute, and PreparedStatement.close. T2API is able to generate the API usage that creates an SQLLiteDatabase object, prepares a command, and executes it via SQLLiteDatabase.execSQL.

The preceding complex snippets are not an exception. One of the more sophisticated cases that T2API handles involves getting the GPS location when the GPS is off (Fig. 10).

Limitations. T2API also has the following limitations. First, there are sometimes extra code elements that GRASYN found were common in complete programs in frequent patterns, but may not be useful in an answer to a particular query. For example, the Log statement (line 7, Fig. 10) is not needed, and getSystemService (line 11) is a common next step once the location is found, but not required for this query. Advanced NLP analysis on queries could improve the result. Second, most of incorrect cases are caused by the out-of-vocabulary issue (un-seen APIs). Finally, we depend on large corpora. Less common APIs might not fit with this approach. For JDK or Android, SO is a useful but noisy corpus.

G. Performance: Time and Space Complexity

Table IV shows T2API's complexity (measured on a computer with AMD Phenom II X4 965 3.0GHz, 8GB RAM, and Linux). Storage cost is reasonable since we use beam search. Training time is extensive. However, one can train it off-line. With the median of 7 keywords leading to 8 synthesized API elements for a post, the suggestion time is 11 seconds. In comparison, SWIM [9] takes an average of 15 seconds to make suggestions of 1–3 API elements from an input of max 4 words. Although Desai *et al.* [11] can make suggestions

TABLE IV
TIME AND SPACE COMPLEXITY

	Mapping + Expansion	GraLan + GRASYN
Storage	2.5GBs/236K posts	3.7GBs total
Training time	8hrs/236K posts	45 hrs/543 projects
Suggestion time	0.08 seconds/post	11.2 seconds/post

in under 2 seconds, their vocabulary is for a domain-specific language and contains only 144 words compared to our 103k words. DeepAPI does not report the suggestion time [3].

H. Threats to Validity

Our evaluation is based on sampled SO posts. Unlike reference translations that were designed to evaluate statistical machine translation between natural languages, these posts were not created with the intention of evaluating English to code translation and have the following limitations: English text may lack descriptiveness compared to code snippets, code snippets may be incomplete compared to the English text, and code may be mixed in with freeform text. In our benchmark of 250 posts we tried to balance code and English. We found 12 cases where synthesized code is more complete than the SO code snippets. They were automatically counted toward incorrect ones. However, they were considered as useful by our judges. Thus, T2API's actual accuracy could be higher.

In our human study, to address the balance of English and code in our benchmark, we had developers judge the usefulness of generated code. To reduce bias, we used the standard approach having two judges for each post. To reduce human errors, we asked professional developers. Showing subjects the SO code snippet might bias against T2API as explained.

Unlike a search problem that uses short inputs, our inputs are an the entire English portion of the StackOverflow post and the comparison output is the corresponding code snippet. This is consistent with the use of use of reference translation in SMT. It is unclear how well T2API would perform with short or poorly formed input queries.

A final threat is that we only considered Android and the associated Android and Java API calls. Although the technique is not inherently tied to a particular API the current evaluation was limited to Android.

VII. RELATED WORK

Our work is related to the *statistical NLP approaches to generate code from text.* SWIM [9] first uses IBM Model with word translation to produce code elements. It then uses syntactic rules on those elements to build code sequences close to the query. In comparison, after IBM Model, we perform context expansion that expand the set of APIs in a priority order. We showed that with context expansion, the results improves over the traditional IBM model (Table III). Second, we use graph synthesis, while SWIM generates code based on syntactic rules, and does not support program dependencies. Third, while both SWIM and T2API produce conditions for conditional statements, loops, method arguments, each relies on different mechanisms. T2API uses graph synthesis where the API usage graph in GraLan [12] contains the information. In contrast, SWIM relies on *syntactic rules* to generate those statements, but does not use program dependencies and so cannot generate data/control dependencies.

DeepAPI [3] uses RNN to generate API sequences for a given text by using deep learning to relate APIs. In comparison, we use graph-based translation with graph synthesis. DeepAPI uses deep learning on sequences. Their synthesized code is a sequence of APIs without parameters, arguments, data/control dependencies, and control units as in T2API.

Desai *et al.* [11] synthesize domain-specific languages from English. A user is required to map key terms in English to the terminals in the DSL. Ye *et al.* [22] use Skip-gram model on API documentation/tutorials to produce embeddings to relate words and APIs. They improve code retrieval from texts, rather than code synthesis. T2API also improves over the T2API tool [23] in which context expansion and graph synthesis were tightly integrated and extensive empirical studies were performed for the evaluation of T2API.

Allamanis *et al.* [24] introduce a jointly probabilistic model with tree representation for short natural language utterances and code snippets. Anycode [10] uses a probabilistic CFG with trees for Java constructs and API calls to synthesize small Java expressions. Maddison and Tarlow [25] present a generative model for source code, which is based on AST structures. Other NLP approaches have been used to support code suggestion [26], [27], [28], code convention [29], name suggestion [30], API suggestions [31], code mining [32], etc.

Others use **program analysis to synthesize code**. Buse and Weimer [8] use path sensitive dataflow analysis, clustering, and pattern abstraction to synthesize API usages from code examples. They do not handle *textual queries*. Others explore structure relations [33], call graphs (FACG [34]) program dependencies (MAPO [35], Altair [36]), input/output types [37].

While we generate code, **code search approaches** mainly use IR approaches to improve accuracy of retrieving existing code [38], [39], [40], [33], [41], [42], [43]. Other IR-based search approaches consider the relations among APIs [6], [44], [45], [41] via using API graphs, call graphs, or dependencies [7], [4], [5]. Semantic code search approaches use context-aware retrieval [46], temporal specifications [47], constraint solver [48], symbolic execution [49], formal logic and theorem prover [50], test execution [51], [52], control flow [53].

VIII. CONCLUSION

We present a novel context-sensitive, graph-based statistical translation approach that takes a query and synthesizes complex API code templates and graphs with control units and dependencies among API elements. While we use texts from real-world SO with large numbers of words, the synthesized code captures more complex usages than the state-of-the-art sequence-based approaches. Human subjects including professional developers judged that 77% of our top-1 synthesized templates are useful to solving the problem in the posts. Our snippets are made “natural” by our graph synthesis algorithm.

ACKNOWLEDGMENTS

This work was supported in part by the US National Science Foundation (NSF) grants CCF-1723215, CCF-1723432, TWC-1723198, CCF-1518897, and CNS-1513263.

References

[1] E. Duala-Ekoko and M. P. Robillard, “Asking and answering questions about unfamiliar APIs: An exploratory study,” in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE’12. IEEE Press, 2012, pp. 266–276.

[2] E. Duala-Ekoko and M. P. Robillard, “Using structure-based recommendations to facilitate discoverability in APIs,” in *Proceedings of the 25th European Conference on Object-oriented Programming*, ser. ECOOP’11. Springer-Verlag, 2011, pp. 79–104.

[3] X. Gu, H. Zhang, D. Zhang, and S. Kim, “Deep API Learning,” in *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. ACM, 2016.

[4] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu, “Portfolio: Finding relevant functions and their usage,” in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE ’11. ACM, 2011, pp. 111–120.

[5] C. McMillan, D. Poshyvanyk, and M. Grechanik, “Recommending Source Code Examples via API Call Usages and Documentation,” in *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering*, ser. RSSE ’10. ACM, 2010, pp. 21–25.

[6] W. Zheng, Q. Zhang, and M. Lyu, “Cross-library API recommendation using Web search engines,” in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE ’11. ACM, 2011, pp. 480–483.

[7] W.-K. Chan, H. Cheng, and D. Lo, “Searching Connected API Subgraph via Text Phrases,” in *Proceedings of the 20th International Symposium on the Foundations of Software Engineering*, ser. FSE ’12. ACM, 2012, pp. 10:1–10:11.

[8] R. P. L. Buse and W. Weimer, “Synthesizing API Usage Examples,” in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE ’12. IEEE Press, 2012, pp. 782–792.

[9] M. Raghothaman, Y. Wei, and Y. Hamadi, “SWIM: synthesizing what I mean,” in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE 2016. ACM Press, 2016.

[10] T. Gvero and V. Kuncak, “Synthesizing Java expressions from free-form queries,” in *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA 2015. New York, NY, USA: ACM, 2015, pp. 416–432. [Online]. Available: http://doi.acm.org/10.1145/2814270.2814295

[11] A. Desai, S. Gulwani, V. Hingorani, N. Jain, A. Karkare, M. Marron, S. R, and S. Roy, “Program synthesis using natural language,” in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE ’16. ACM, 2016, pp. 345–356.

[12] A. T. Nguyen and T. N. Nguyen, “Graph-based statistical language model for code,” in *Proceedings of the 37th International Conference on Software Engineering*, ser. ICSE 2015. IEEE CS, 2015.

[13] P. Koehn, *Statistical Machine Translation*, 1st ed. New York, NY, USA: Cambridge University Press, 2010.

[14] P. Koehn, F. J. Och, and D. Marcu, “Statistical phrase-based translation,” in *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology - Volume 1*, ser. NAACL ’03. Stroudsburg, PA, USA: Association for Computational Linguistics, 2003, pp. 48–54. [Online]. Available: http://dx.doi.org/10.3115/1073445.1073462

[15] P. F. Brown, V. J. D. Pietra, S. A. D. Pietra, and R. L. Mercer, “The mathematics of statistical machine translation: parameter estimation,” *Comput. Linguist.*, vol. 19, no. 2, pp. 263–311, Jun. 1993.

[16] P. C. Rigby and M. P. Robillard, “Discovering essential code elements in informal documentation,” in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE ’13. IEEE Press, 2013, pp. 832–841.

[17] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen, “Graph-based mining of multiple object usage patterns,” in *Proceedings of Conference on the Foundations of Software Engineering*, ser. ESEC/FSE ’09. ACM, 2009, pp. 383–392.

[18] “The BerkeleyAligner,” https://code.google.com/p/berkeleyaligner/.

[19] B. Dagenais and L. Hendren, “Enabling static analysis for partial Java programs,” in *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*, ser. OOPSLA ’08. New York, NY, USA: ACM, 2008, pp. 313–328. [Online]. Available: http://doi.acm.org/10.1145/1449764.1449790

[20] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, “BLEU: A Method for Automatic Evaluation of Machine Translation,” in *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, ser. ACL ’02. Stroudsburg, PA, USA: Association for Computational Linguistics, 2002, pp. 311–318. [Online]. Available: http://dx.doi.org/10.3115/1073083.1073135

[21] “Deep API Learning,” http://bda-codehow.cloudapp.net:88/.

[22] X. Ye, H. Shen, X. Ma, R. Bunescu, and C. Liu, “From word embeddings to document similarities for improved information retrieval in software engineering,” in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE ’16. ACM, 2016, pp. 404–415.

[23] T. Nguyen, P. C. Rigby, A. T. Nguyen, M. Karanfil, and T. N. Nguyen, “T2api: Synthesizing api code usage templates from english texts with statistical translation,” in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: ACM, 2016, pp. 1013–1017. [Online]. Available: http://doi.acm.org/10.1145/2950290.2983931

[24] M. Allamanis, D. Tarlow, A. Gordon, and Y. Wei, “Bimodal modelling of source code and natural language,” in *Proceedings of the 32nd International Conference on Machine Learning*, ser. ICML ’15. ACM, 2015.

[25] C. J. Maddison and D. Tarlow, “Structured generative models of natural source code,” in *Proceedings of the 31st International Conference on Machine Learning (ICML)*, June 2014.

[26] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, “On the naturalness of software,” in *Proceedings of the 2012 International Conference on Software Engineering*, ser. ICSE 2012. IEEE Press, 2012, pp. 837–847.

[27] M. White, C. Vendome, M. Linares-Vasquez, and D. Poshyvanyk, “Toward deep learning software repositories,” in *Proceedings of 12th IEEE Working Conference on Mining Software Repositories (MSR’15)*. IEEE CS, May 2015.

[28] L. Mou, G. Li, Z. Jin, L. Zhang, and T. Wang, “TBCNN: A tree-based convolutional neural network for programming language processing,” *CoRR*, vol. abs/1409.5718, 2014. [Online]. Available: http://arxiv.org/abs/1409.5718

[29] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, “Learning natural coding conventions,” in *Proceedings of the International Symposium on Foundations of Software Engineering*, ser. FSE 2014. ACM, 2014, pp. 281–293.

[30] ——, “Suggesting accurate method and class names,” in *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. ESEC/FSE 2015. ACM, 2015.

[31] V. Raychev, M. Vechev, and E. Yahav, “Code completion with statistical language models,” in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’14. ACM, 2014, pp. 419–428.

[32] M. Allamanis and C. Sutton, “Mining source code repositories at massive scale using language modeling,” in *Proceedings of the 10th IEEE Working Conference on Mining Software Repositories (MSR’13)*. IEEE CS, May 2013, pp. 207–216.

[33] S. Bajracharya, T. Ngo, E. Linstead, Y. Dou, P. Rigor, P. Baldi, and C. Lopes, “Sourcerer: A search engine for open source code supporting structure-based search,” in *Proceedings of the ACM Conference on Object-oriented Programming Systems, Languages, and Applications*, ser. OOPSLA ’06. ACM, 2006, pp. 681–682.

[34] Q. Zhang, W. Zheng, and M. R. Lyu, “Flow-augmented Call Graph: A New Foundation for Taming API Complexity,” in *Proceedings of the 14th International Conference on Fundamental Approaches to Software Engineering*, ser. FASE’11/ETAPS’11. Springer-Verlag, 2011, pp. 386–400.

[35] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei, “MAPO: Mining and Recommending API Usage Patterns,” in *Proceedings of the 23rd European Conference on Object-Oriented Programming*. Springer, 2009, pp. 318–343.

[36] F. Long, X. Wang, and Y. Cai, “Api hyperlinking via structural overlap,” in *Proceedings of the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC/FSE ’09. ACM, 2009, pp. 203–212.

[37] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman, “Jungloid Mining: Helping to Navigate the API Jungle,” in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’05. ACM, 2005, pp. 48–61.

[38] “Black Duck Open Hub,” http://code.openhub.net/.

[39] “Codase,” http://www.codase.com/.

[40] K. Inoue, R. Yokomori, H. Fujiwara, T. Yamamoto, M. Matsushita, and S. Kusumoto, “Component rank: Relative significance rank for software component search,” in *Proceedings of the 25th International Conference on Software Engineering*, ser. ICSE ’03. IEEE, 2003, pp. 14–24.

[41] D. Puppin and F. Silvestri, “The Social Network of Java Classes,” in *Proceedings of the 2006 ACM Symposium on Applied Computing*, ser. SAC ’06. ACM, 2006, pp. 1409–1413.

[42] N. Sawadsky, G. C. Murphy, and R. Jiresal, "Reverb: Recommending code-related web pages," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. IEEE Press, 2013, pp. 812–821.

[43] J. Brandt, M. Dontcheva, M. Weskamp, and S. R. Klemmer, "Example-centric programming: Integrating web search into the development environment," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '10. ACM, 2010, pp. 513–522.

[44] M. Grechanik, C. Fu, Q. Xie, C. McMillan, D. Poshyvanyk, and C. Cumby, "A Search Engine for Finding Highly Relevant Applications," in *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10. ACM, 2010, pp. 475–484.

[45] Z. M. Saul, V. Filkov, P. Devanbu, and C. Bird, "Recommending random walks," in *Proceedings of the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC-FSE '07. ACM, 2007, pp. 15–24.

[46] N. Sahavechaphan and K. Claypool, "Xsnippet: Mining for sample code," in *Proceedings of the Conference on Object-oriented Programming Systems, Languages, and Applications*, ser. OOPSLA'06. ACM, 2006, pp. 413–430.

[47] A. Mishne, S. Shoham, and E. Yahav, "Typestate-based semantic code search over partial programs," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '12. ACM, 2012, pp. 997–1016.

[48] K. T. Stolee, S. Elbaum, and D. Dobos, "Solving the search for source code," *ACM Trans. Softw. Eng. Methodol.*, vol. 23, no. 3, pp. 26:1–26:45, Jun. 2014.

[49] K. T. Stolee, S. Elbaum, and M. Dwyer, "Code search with input/output queries: Generalizing, ranking, and assessment," *J. Syst. Softw.*, 2015.

[50] J. Penix and P. Alexander, "Efficient specification-based component retrieval," *Automated Software Engg.*, vol. 6, no. 2, pp. 139–170, Apr. 1999.

[51] O. A. Lazzarini Lemos, S. K. Bajracharya, and J. Ossher, "CodeGenie: A tool for test-driven source code search," in *Companion to the 22nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion*, ser. OOPSLA '07. ACM, 2007, pp. 917–918.

[52] S. P. Reiss, "Semantics-based code search," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09. IEEE Computer Society, 2009, pp. 243–253.

[53] S. Thummalapenta and T. Xie, "Parseweb: A programmer assistant for reusing open source code on the web," in *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '07. New York, NY, USA: ACM, 2007, pp. 204–213. [Online]. Available: http://doi.acm.org/10.1145/1321631.1321663