Enabling Near Real-Time NLU-Driven Natural Language Programming through Dynamic Grammar Graph-Based Translation

Zifan Nan
Department of Computer Science
North CarolinaState University
Raleigh, North Carolina, USA
znan@ncsu.edu

Xipeng Shen
Department of Computer Science
North CarolinaState University
Raleigh, North Carolina, USA
xshen5@ncsu.edu

Hui Guan
College of Information and Computer Sciences
University of Massachusetts Amherst
Amherst, Massachusetts, USA
huiguan@cs.umass.edu

***Abstract*—Recently, natural language (NL)-based program synthesis has drawn increasing interest. Conventional methods that depend on some predefined domain-specific rules suffer from the lack of robustness and generality. Recent efforts on adopting deep learning to map queries to code requires a large number of labeled examples, making them not applicable on domains with scarce labeled examples. Although a third alternative, natural language understanding (NLU)-driven approach addresses the problems, the long response time hinders its adoption in practice, especially in an interactive scenario. This paper presents a solution to enable near real-time NLU-driven NL programming. The solution features a new algorithm, dynamic grammar graph-based translation (DGGT), for identifying the best grammar tree for a query via dynamic programming. It also introduces two new optimizations, grammar-based pruning and orphan node relocation, to further reduce the search space and address the special complexities from queries. Evaluations on two domains, text editing and program source code analysis, show that the DGGT algorithm and the optimizations shortens the response time of a state-of-the-art NLU-driven synthesizer by up to 1887× (25-133× on average) while improving the accuracy by 2-12%.**

***Index Terms*—Natural language programming, program synthesis, dynamic programming**

I. Introduction

Recently, natural language (NL)-based program synthesis (or called NL Programming) has drawn lots of research interest. Given NL descriptions (also called *queries*) from users, NL-based synthesizers automatically generate desired codelets. A codelet is often an expression in either a Domain-Specific Language (DSL) or some domain-specific APIs. Figure 1 shows a simple example.

Allowing the use of intuitive NL-based inputs offers conveniences to general users (e.g. in the IoT domain) who do not need to learn programming in the DSL. It can also serve as part of an Integrated Development Environment (IDE) to offer quick hints to programmers especially when the domain of interest contains a large number of API functions that are difficult to memorize (e.g., Android API [10], ASTMatcher in Compilers [7]).

Conventional methods mostly fall into two categories: *rule-driven* and *sample-driven* methods. *Rule-driven* methods depend on some predefined domain-specific rules. *Sample-driven* methods feature the reliance on many labeled query-codelet pairs as training data to build up some statistical models. The *rule-based* approach had shown some success in the early stage of the field development (e.g., Smartsynth [26]), but have gradually lost attractions due to the lack of robustness and the difficulties in generalizing across domains. The *data-driven* approach has dominated recent efforts, represented by the adoption of deep learning to map NL queries to code via various neural networks (e.g., [2], [15], [30], [38], [39]). Although this approach has shown more promise than the previous *rule-driven* approach, its requirement of large numbers of labeled examples hinders its adoptions, especially for domains where labeled examples are scarce. Although recent proposals show the possibility of generating examples for a certain domain [3], it is yet unclear how well these methods can generate truly representative examples in complex domains.

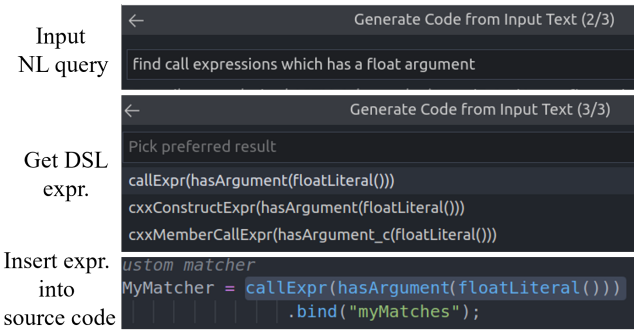

Figure 1. An example of NL programming using VSCode.

Recently, a third alternative, *NLU-driven approach*, has been proposed [34], [35]. This approach is driven by *natural language understanding (NLU)*. It is inspired by how human codes. Rather than going through tens of thousands of examples as *data-driven approach* does, it centers around natural language understanding of the domain-specific language or APIs of interest—just like how humans learn programming, and hence

978-1-6654-0584-3/22 © 2022 IEEE

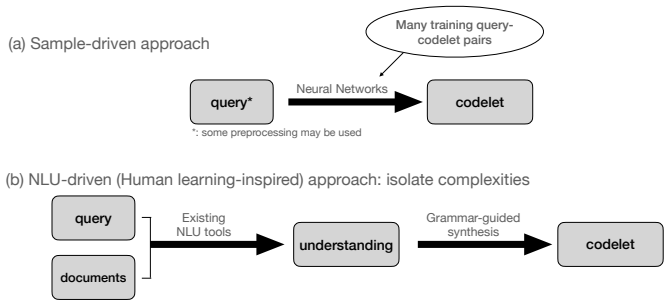

Figure 2. By isolating complexities and leveraging existing NLU tools, NLU-driven method foregoes the needs for many training samples.

foregoes the need for many labeled training data. It achieves it by isolating complexities and leveraging existing NLU tools, as Figure 2 shows.

The NLU-driven approach is appealing: It avoids the difficulties and the cost in collecting a large number of training samples while achieving similar accuracies as the sample-driven approach achieves [34], [35]. Moreover, it makes an NL programming solution much easier to extend: When the APIs in the target domain change (e.g., new functionalities are added into a smartphone), an NLU-driven solution needs no retraining on new training samples; it needs only the incorporation of the updated document of the changed APIs.

But the current NLU-driven approach is slow. When synthesizing 200 cases in TextEditing [9] domain, 38% of cases needs more than one second to produce the result; around 18% cases need more than 20 seconds. Among these cases, five of them are not finished even after 20 minutes.

The speed limitation of the NLU-driven approach is a major roadblock for its adoption in practice. NL programming is often used in an interactive scenario, such as in a smart home, vehicle, or IDE, where the slow response of the current NLU-driven approach is intolerable.

The problem is challenging, fundamentally due to the nature of the code generation process in NLU-driven NL programming. Unlike the use of statistical models to directly map queries to code, the NLU-driven approach generates code by finding the best grammar tree that connects all the key elements in the query in a way both syntactically and semantically sound. As the possible grammar trees in a domain are exponential to the size of the grammar, the space is huge for a non-trivial domain.

This paper presents a solution to the problem, which speeds up the state of the art by up to 133$times$ and for the first time makes NLU-driven NL programming possible for interactive uses. As a lossless algorithm-level optimization, it uses no more computing units and loses no accuracy.

At the core of the solution is *dynamic grammar graph-based translation* (DGGT), a new algorithm for identifying the best grammar tree for a query. DGGT is inspired by Dynamic Programming, an approach that solves an optimization problem by breaking it down to simpler subproblems and utilizing the optimal solution to its subproblems to identify the optimal solution to the overall problem. It memorizes the optimal solutions to subproblems to avoid redundant computations and achieve high efficiency. A key step in DGGT is to generate the optimal code generation tree (CGT) based on the query dependency graph. DGGT decomposes the problem into the subproblems of finding the optimal partial CGT and then generates the optimal CGT by joining the optimal partial CGT recorded in a *dynamic grammar graph* structure.

Besides DGGT, the solution introduces two new optimizations, *grammar-based pruning* and *orphan node relocation*, which further reduces the search space and addresses some special complexities from queries respectively. These optimizations, coupled with an existing optimization (size-based early pruning), effectively save more than 90% of computations for complex queries.

We evaluate the techniques on two domains, text editing, and program source code analysis. The results show that the optimizations shorten the response time of a state-of-the-art NLU-driven synthesizer by up to 1887$\times$ (25-133$\times$ on average), while improving the accuracy by 2-12% for the fewer timeout cases. It, for the first time, removes the barriers for NLU-driven NL programming to serve for interactive tasks.

Overall, this work makes the following major contributions:

- It offers the first systematic study on the efficiency issues of NLU-driven NL programming, and provides a solution that for the first time enables near real-time NLU-driven NL programming.
- It proposes a novel algorithm *dynamic grammar graph-based translation*for NLU-driven NL programming that significantly reduces the computational complexity.
- It introduces two new optimizations, *grammar-based pruning* and *orphan node relocation*, for efficient NLU-driven NL programming.
- It demonstrates the effectiveness of the proposed solution through a set of experiments on two domains, showing that the techniques can shorten the average synthesis time by up to 133X.

II. Prerequisites and Background

This section reviews the background of NLU-driven NL programming, based on HISyn [34], the state-of-the-art NLU-driven code synthesizer. The input to an NLU-driven code synthesizer consists of three items: (i) NL query, such as "insert a string at the start of each line" in a text editing domain; (ii) a document that contains all the APIs and their descriptions, such as the reference to ASTMatcher[1]; (iii) the context-free grammar of the target domain, written in Backus-Naur form (BNF) and converted to a directed graph called *grammar graph*. Figure 4(a) shows part of the grammar graph of a text editing domain. The output of the synthesizer is a codelet that implements the intent expressed in the input query with the APIs in the target domain.

The workflow consists of six steps, as shown in Figure 3.

[1]https://clang.llvm.org/docs/LibASTMatchersReference.html

- *Step-1: Dependency parsing.* It is an NLP technique for identifying the dependency relations among words. The outcome is an *query dependency graph* for the input query. A dependency relation is composed of a subordinate word (called *dependent*), a word on which it depends (called *governor*), and an asymmetrical grammatical relation between the two words (called *dependency type*). A dependency relation is marked as an arrow pointing from a governor to a dependent and is labeled with the dependency type. In the example in the top left in Figure 3, the arrow from "insert" to "string" with the label "obj" indicates that "string" is an object of "insert". Each node in the *query dependency graph* contains a word in the query.
- *Step-2: Query graph pruning.* This step prunes the non-essential words from the query dependency graph based on the Part-Of-Speech (POS) of words and their relations, producing a *pruned dependency graph* (top middle in Figure 3).
- *Step-3: WordToAPI.* This step tries to find all the APIs in the target domain that may semantically match each query word in the *pruned dependency graph*. It does it by matching the query words with the descriptions of each API via NLU techniques, producing a *WordToAPI* map (top right in Figure 3).
- *Step-4: EdgeToPath.* This step tries to find the set of paths on the grammar graph (called *grammar paths*) that corresponds to each edge in the *pruned dependency graph*. It does it through a *reversed all-path search*. For edge (2) "insert"→"string" in the pruned dependency graph in Figure 3, for instance, the search starts from the grammar graph node that contains one of the candidate APIs of "string", and follows the grammar graph backward until reaching a node that contains one of the candidate APIs of "insert"; the grammar path is then recorded in the EdgeToPath map. It does this for every candidate API of each node in the *pruned dependency graph*.
- *Step-5: PathMerging.* This step enumerates every combination of the grammar paths of all the edges in the *pruned dependency graph*. For each combination, it tries to merge the grammar paths to form a tree, called *code generation tree (CGT)*. The merging process fuses the same nodes or edges into one.
- *Step-6: TreeToExpression.* This step finds the smallest CGT, traverses it in a depth-first order, and puts the API contained in the nodes together to form the final expression. The children of a node are regarded as parameters of the API in their parent node.

III. Overview of Optimizations

This section presents some observations on the performance issues of the current NLU-driven synthesis, and the key insights in our solutions for resolving the performance bottleneck.

A. Bottleneck

Among the six steps of the current NLU-driven analysis, step five is the most time-consuming step. It enumerates all the combinations of the candidate grammar paths corresponding to all the dependency edges in the *pruned dependency graph*. In an query dependency graph, assume that the l-th level has e_l dependency edges and each edge corresponds to p_l number of path candidates in a grammar graph. The current NLU-driven synthesis algorithms enumerate all paths, with $\prod_l {p_l}^{e_l}$ path combinations in the worst case and the overall computational complexity is $O(\prod_l {p_l}^{e_l})$. For a non-trivial query with 4 dependency levels with $e_l = 4$ dependency edges per level and $p_l = 4$ candidate paths per dependency edge, there are four billion combinations. The number grows exponentially as the complexity of the query increases. Our empirical measurements confirm that for the queries that take the current NLU-driven synthesizer HISyn more than two seconds to process, this step dominates the execution time, weighing 90.24% of the total time.

B. Key Insights

(i) The bottleneck analysis shows that to make NLU-driven synthesis usable in interactive scenarios, the key is in speeding up step 5. A key insight underlying our solution is that there is lots of overlap among the combinations of the grammar paths. For instance, assume that this step merges the grammar paths of three dependency edge $\{e_1, e_2, e_3\}$ with each dependency edge having two grammar paths. We denote the six paths as $\{p_{1a}, p_{1b}\}$, $\{p_{2a}, p_{2b}\}$, $\{p_{3a}, p_{3b}\}$. There are hence eight combinations: $\{p_{1a}, p_{2a}, p_{3a}\}$, $\{p_{1a}, p_{2a}, p_{3b}\}$, ... These combinations have lots of overlaps: $\{p_{1a}, p_{2a}\}$, for instance, appears in two of them. Merging each of the eight combinations hence incurs repeated merging operations on those overlapped paths.

Therefore, in the previous algorithm, lots of repeated work has been spent on processing the same branches and nodes in the combinations. So the first and most important insight of our solution is in the design of a data structure (*dynamic grammar graph*) and algorithm optimization (*DGGT*) to effectively avoid the redundant work in the combinations. Inspired by dynamic programming, DGGT decomposes the problem into the subproblems of finding the optimal partial CGT and then generates the optimal CGT by joining the optimal partial CGT recorded in a *dynamic grammar graph* structure.

(ii) A second observation is that due to the "or" relations in grammar rules, many combinations produced in step 5 are syntactically incorrect and ruled out in later steps. For instance, the non-terminal node `pos` has two derivations `POSITION` and `START`, which are exclusive from each other. If one combination has two path containing both derivations, that combination must be syntactically incorrect. The second insight underlying our optimization is that intelligent use of the grammar may prevent many of the incorrect combinations from being produced at the first place. It is the basis of our optimization *grammar-based pruning*.

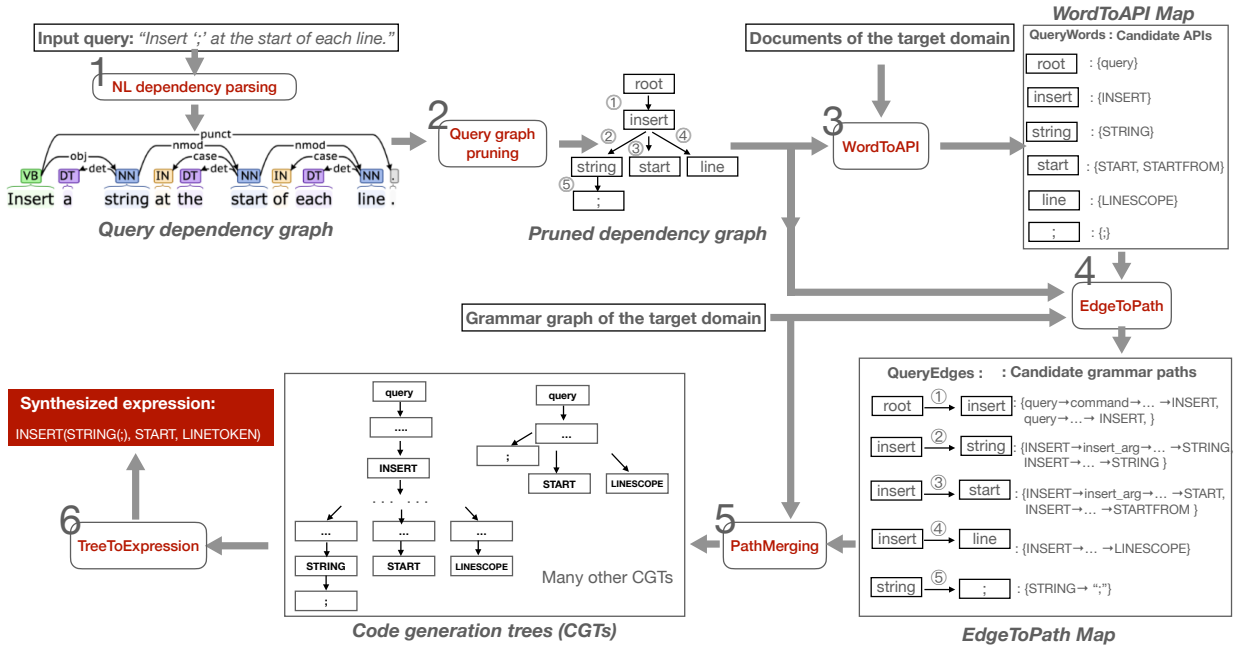

Figure 3. Workflow of existing NLU-driven NL programming (HISyn). (The full version of the EdgeToPath map is shown in Figure 5(b).)

(iii) The third observation is that for some queries, the dependency parsing makes mistakes, giving wrong dependence relations between some words. Consequently, some nodes in the dependency graph become "orphans", that is, no valid edges connect them with the rest of the parsing result. To avoid missing the valid grammar paths involving these orphans, the current NLU-driven synthesis algorithm finds all paths in the grammar graph from the orphan to the root of the grammar. That treatment gives out many candidate grammar paths and increases the computational complexity. The third insight in this work is that a suitable use of the grammar can actually rule out many invalid paths that involve orphans, and hence significantly reduce the complexity. It is the basis of our optimization *orphan node relocation*.

These three optimizations, *DGGT*, *grammar-based pruning* and *orphan node relocation*, form a synergy. They work together seamlessly, drastically reducing the computation complexity and execution time of NLU-driven synthesis. We next explain them each in detail.

IV. Dynamic Grammar Graph-Based Translation (DGGT)

This section presents DGGT, a new algorithm to avoid redundant work in the bottleneck (step 5) of NLU-driven synthesis. We start by introducing a set of terms used in the algorithm, and then elaborate the algorithm. We postpone the discussion of the computational complexity in Section VI after the other optimizations are also introduced.

A. Path-Voted Grammar Graph

To prepare for understanding DGGT, we first give some more detailed explanation of *grammar graph*, and introduce a new term *path-voted grammar graph*.

A grammar graph is a graph representation of a context-free grammar (CFG) ($\mathcal{T}$: terminals, $\mathcal{NT}$: non-terminals, $\mathcal{S}$: start symbol, $\mathcal{P}$: production rules). Figure 4(a) shows part of the grammar graph of the Text Editing DSL [9]. We use the following production rules in the DSL to explain grammar graph.

insert_arg ::= string pos iter
string ::= STRING
pos ::= POSITION | START

There are three types of nodes in a grammar graph. Node "insert_arg" in Figure 4(a), for instance, is a *non-terminal node* representing a non-terminal symbol in the grammar; node "string pos iter" is a *derivation node* representing the entire right-hand side of a production rule (i.e., the "insert_arg" rule); node "STRING" (in red) is an *API node* representing the name of an API function in the DSL. There are two types of edges. Solid-headed edges (e.g., those from "insert_arg" to "string", "pos", and "iter") are *concatenation edges* representing concatenations of the sibling nodes; hollowed-headed edges (e.g., those from "pos" to "POSITION" and "START") are *"Or" edges* representing the alternative relations among the siblings.

A key operation in NLU-driven synthesis is to find a grammar path on the CFG that matches (i.e., nodes semantically match) with an edge on the query dependency graph. Edge (2) (insert→ string) in Figure 4(a), for instance, matches with two *candidate paths* in the CFG in Figure 4(c). They are shown as paths `2.1` and `2.2` in Figure 4(b). If we label an edge in a CFG with the candidate grammar paths that cover it, we get a **path-voted grammar graph**, as shown in Figure 4(c). We say that an edge has more votes if it is covered by more grammar paths. We will see how the votes can help DGGT

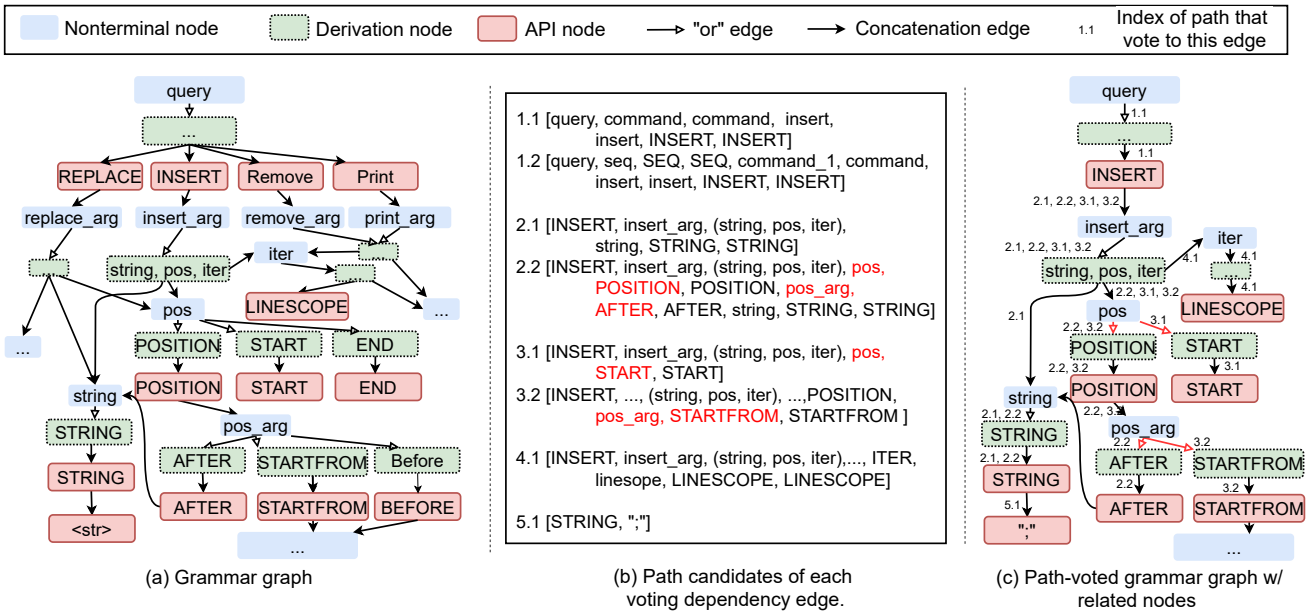

Figure 4. The grammar graph (a) and the path voted grammar graph (c) of the editing domain annotated by the grammar paths (b) of the example query in Figure 3.

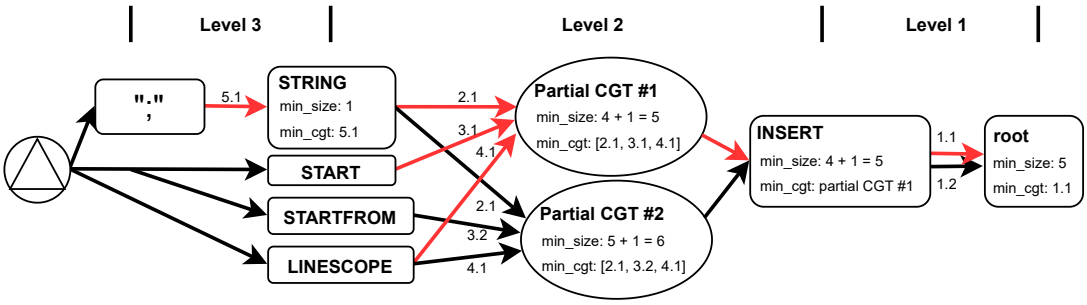

Figure 5. The *dynamic grammar graph* of the example in Figure 4. (Fields min_size and min_cgt are omitted for nodes whose min_size is 0.)

take effects.

If after the candidate paths of all dependency edges are fused (by merging common nodes and edges), they form a tree, we call the tree a **code generation tree** (CGT). By definition, a CGT is a subgraph of the CFG. A CGT can be hence reformatted into a grammar valid codelet in the DSL.

For quick references, we summarize the three important acronyms as follows:

CFG	Context-free grammar
CGT	Code generation tree
DGGT	Dynamic grammar graph-based translation

B. Dynamic Grammar Graph-Based Translation

Recall that the goal of the NLU-driven synthesis is to identify the smallest CGT (for the shortest code to be produced) for a given query dependency graph by examining the CFG. Such a CGT (i) covers all the semantic inside the query, and at the same time (ii) carries the minimum unmentioned semantic. With these two conditions, the generated CGT is more likely to represent the semantic meaning of input NL queries faithfully. The *reversed all-path search* in step four in Figure 3 searches for the paths that correspond to each dependency edge, which ensures that the created CGTs fulfill the first condition. The selection of the smallest CGT in steps 5 and 6 satisfies the second condition.

In identifying the smallest CGT, instead of enumerating all the candidate CGTs as the current NLU-driven synthesis does [34], DGGT breaks the synthesis problem into sub-problems, by generating the optimal partial CGTs for each level of the query dependency graph via *bottom-up dynamic grammar graph generation*, and then combining these partial CGTs into the final CGT via *the optimal CGT backtrack*.

We next present a central data structure *dynamic grammar graph*. It is used in DGGT to record optimal partial CGTs and to backtrack. We then present the main algorithm.

1) Dynamic Grammar Graph: There are three kinds of nodes in a dynamic grammar graph. N_{API} is the set of nodes that represent APIs; N_{PCGT} is the set of nodes that represent

Algorithm 1: DGGT algorithm

Input : query dependency graph dg, grammar graph gg
Result : the optimal code generation tree and synthesized code

```
1  Function DGGT(dg, gg)
       // Bottom-up dynamic grammar graph
          generation
2      new DynamicGraph dyng
3      add start_node and leaf_API_node into dg
4      foreach level in dg from bottom to top do
5          Let e : n1 → n2 be an edge at the current level
6          if n1 ∈ dg has a single child n2 then
7              create N_ai in dyng for each candidate API a_i of
                 n1
8              foreach grammar path ∈ EdgeToPath(e) do
9                  create a path edge (N_bj → N_ai) in dyng
                     where b_j is a candidate API of n2
10             foreach N_ai do
11                 update its min_cgt and min_size with the
                     smallest partial CGT
12         else if n1 in dg has multiple children then
13             comb = the set of combinations of the grammar
                 paths of n1's children
14             comb = grammar_based_pruning(comb)
15             comb = size_based_pruning(comb)
16             foreach c in comb do
17                 merge the paths in c into a tree pt, and
                     create node N_pt in dyng;
18                 update min_cgt and min_size of N_pt;
19                 create node N_root in dyng for the root of pt
                     if it does not exist;
20                 connect N_pt with the nodes in dyng that
                     correspond to the leaves of pt;
21                 connect N_pt with N_root
22             update min_cgt and min_size of N_root
23     Backtrack dyng to create the optimal CGT and generate
         code
```

partial CGTs; N_{start} is the start node of the dynamic grammar graph. Figure 5 is the dynamic grammar graph of the example in Figure 4; the three types of nodes are represented by round-cornered rectangles, circles, and triangles respectively.

There are two kinds of edges in a dynamic grammar graph. $E_{path} = \{(N_{API_1} \rightarrow N_{API_2})\} \cup \{(N_{API} \rightarrow N_{PCGT})\}$ is the set of *path edges*, with each representing one grammar path between two APIs or between an API and a partial CGT. Its length is the number of APIs on the path. $E_{default} = \{(N_{start} \rightarrow N_{API})\} \cup (N_{PCGT} \rightarrow N_{API})\}$ is a set of *auxiliary edges* whose lengths are regarded as zero. A path edge carries the ID of the corresponding grammar path, while an auxiliary edge carries no IDs, as shown in Figure 5.

2) Main Algorithm: At the core of the DGGT algorithm is the construction of a *dynamic grammar graph* through a bottom-up traversal of the pruned dependency graph. The nodes in the dynamic grammar graph record the optimal partial CGT from the start node to this node. So after the construction, it takes just a simple backtracking to generate the desired code expression. Algorithm 1 outlines the DGGT algorithm. We explain it in detail while drawing on the pruned dependency graph in Figure 3 (top middle) as an example.

Step 1: dynamic grammar graph construction through a bottom-up traversal. This step traverses the pruned dependency graphs in a bottom-up order to generate a dynamic grammar graph (lines 2-22 in Algorithm 1). For the example in Figure 3, the algorithm starts from edge (5) in level-3, then moves to the 3 sibling edges ((2), (3), and (4)) in level-2, and finally reaches (1) in level-1.

Initially, the dynamic grammar graph has a single node N_{start}. For each leaf node in the pruned dependency graph whose candidate APIs are $a_1, a_2, \cdots, a_n$, the algorithm generates API nodes $\{N_{a_i}\}$ and edges $\{E_{default_i} = (N_{start} \rightarrow N_{a_i})\}$ (line 3) and put them into the dynamic grammar graph. In Figure 4(c), the leaf node "`;`" for instance leads to the creation of the ";" node in the dynamic grammar graph in Figure 5 and the auxiliary edge from the start node to it. The leaf node "start" in the dependency graph leads to the creations of the "START" and "STARTFROM" nodes (for "start" has two APIs in the `Word2API` map in Figure 3 (top right)) and the edges to them from the start node in Figure 5, and the "LINETOKEN" node is created because of the leaf node "line" in the dependency graph.

Then from the leaf nodes in the pruned dependency graph, the algorithm visits one level up each time. In the process, it treats edges with siblings and without siblings differently. (We say that an edge $e1$ has a sibling edge $e2$ if they share the same source node but have different sink nodes.)

Case I: a dependency edge e from n_1 to n_2 has no siblings. This case is simpler. We just need to create nodes to denote the candidate APIs of n_1 and add them into the dynamic grammar graph. Specifically, suppose in the `WordToAPI` map, n_1 has candidate APIs $a_1, a_2, \cdots, a_n$ and n_2 has candidate APIs $b_1, b_2, \cdots, b_m$. Because the visit is bottom up, by this time, the dynamic grammar graph shall already have nodes corresponding to b_j (j=1,2,$\cdots$,m). The algorithm creates a node for each a_i and adds the nodes into the dynamic grammar graph, denoted as N_{a_i} (i=1,2,$\cdots$,n). It adds edges $\{E_{p_k} = (N_{b_j} \rightarrow N_{a_i})\}$ into the dynamic grammar graph to represent the grammar paths p_k between a_i and b_j, where k is the ID of the grammar path (lines 5-9). An example is the node "STRING" in Figure 5 and the edge from ";" to "STRING". Note, it is possible that $EdgeToPath(e)$ contains multiple grammar paths. They are handled in the same way. After the algorithm finishes processing all the grammar paths of e, an important step is to record the optimal partial CGT from the start to the newly added nodes[2]. This step is made efficient in a way similar to dynamic programming. Every node in the dynamic grammar graph has two fields, `min_cgt` and `min_size`. The field `min_cgt` records the optimal partial CGT from the start node to this node. The field `min_size` records its size. An optimal partial CGT is the smallest CGT from the start to this node, that is, the CGT covers the smallest number of APIs (illustrated later). So, when we try to find the

[2]It is a graph, but turns into a tree rooted at the newly added node after the start node is omitted.

optimal partial CGT for the newly added a_i nodes, the existing b_j nodes should already have both fields set. The algorithm can then compute the size of the partial CGT to a_i via b_j by just adding the size of the grammar path from b_j to a_i into the `min_size` of b_j. It can hence quickly find out the optimal partial CGT of a_i and its size, and record them in its own fields (lines 10-11).

For instance, in Figure 4(c), the query dependency edge (5) has only one grammar path [STRING→";"]. When traversing the dependency edge (2) in level-3, we create a node N_{STRING} and a path edge $E_{path_{5.1}} = (N_{;} \rightarrow N_{STRING})$ in the dynamic grammar graph. N_{STRING} records that the minimum size of the partial CGT is 1 as the represented grammar path 5.1 has only one API. Similarly, when traversing the dependency edge (1) in level-1, we create a node N_{root} for the root node and two edges in the dynamic grammar graph for the two grammar paths that connect the `root` node to the `INSERT` in the grammar graph. The min_size of N_{root} is set to 5, and the path ID is 1.1.

Case II: a dependency edge e from n_1 to n_2 has siblings. It means that the grammar paths of these sibling dependency edges must have common predecessors. For example, the grammar paths 2.1 and 3.1 in Figure 4(b) share the same predecessors, nodes `INSERT` and `insert_arg`. It usually indicates that some of the APIs in the sibling nodes are arguments of a common API.

This case is trickier to handle. It is because the partial CGT would need to accommodate the APIs of all those siblings. If we still use one node to represent only one API as in Case I, it would not allow us to record the information of the partial CGT. Moreover, because in the `EdgeToPath` map, one edge in the pruned dependency graph may have multiple grammar paths, the sibling edges can form multiple partial CGTs.

Our solution is to introduce a special type of node (partial CGT node) into the dynamic grammar graph, with each node recording one combination of the grammar paths of the sibling edges. For instance, the two eclipses in Figure 5 represent the combinations of the paths {2.1, 3.1, 4.1} and {2.1,3.2,4.1} respectively. Edges are added to represent the grammar paths. For each combination, the algorithm merges the common predecessors of the grammar paths to form a *prefix tree*, and then uses the size of the tree (i.e., the number of APIs it contains) and the `min_size` of the predecessor nodes to figure out the size of the partial CGT and record the minimum partial CGT and the size in its fields.

After the constructions of a partial CGT node, the algorithm further creates an API node to represent the root of the prefix tree if it does not yet exist, such as the "INSERT" node in Figure 5, and uses auxiliary edges to connect the partial CGT node with this API node. It is needed in the processing of the dependence edges on the next level which expects edges between API nodes.

Merging grammar paths to form a prefix tree has three steps (lines 14-21 in Algorithm 1). For the combinations of the paths from each sibling dependency edge, the first two steps prune the grammarly incorrect combinations and the oversized combinations; the third step generates the prefix tree for the remaining combinations. We postpone the details to later sections.

We use the example in Figure 4 to give a full illustration of the operations on Case II. Dependence edges (2), (3), (4) are sibling edges. After pruning, there are two path combinations left, $c_1 = \{2.1, 3.1, 4.1\}$ and $c_2 = \{2.1, 3.2, 4.1\}$. For each combination, we merge the paths and create a prefix tree. Nodes N_{PCGT_1} and N_{PCGT_2} are created for each prefix tree respectively. Six path edges such as $E_{path_{2.1}} = (N_{\texttt{STRING}} \rightarrow N_{PCGT_1})$ are created to link the APIs nodes and the partial CGT nodes. Since the two prefix trees have common root API `INSERT`, we create the API node $N_{\texttt{INSERT}}$ and two auxiliary edges to link the two partial CGT nodes and their root node $N_{\texttt{INSERT}}$. For node $N_{\texttt{INSERT}}$, the optimal partial CGT is partial CGT #1 with minimum size of 5 and the corresponding node is N_{PCGT_1}.

Step 2: Optimal CGT backtrack. Optimal CGT backtrack aims to generate the optimal CGT by backtracking the dynamic grammar graph and joining the optimal partial CGTs in each level. Since each API node N_{API} records the optimal partial CGT, we can backtrack the dynamic grammar graph and join all the recorded partial CGTs to create an optimal CGT. (In Figure 5, the edges involved in the backtrack is colored in red.)

From the description, we can see that the dynamic grammar graph produced by DGGT concisely subsumes the CGTs from all possible combinations of the candidate paths. Its dynamic programming style allows it to evade repeated subtree merging and examinations while allowing the fast generation of the minimum CGT—one of the key reasons for it to dramatically outperform previous NLU-driven synthsis algorithms [34] which enumerates and examines all possible CGTs.

V. Optimizations

Besides the DGGT algorithm, we introduce two new optimizations to further reduce the computation cost.

A. Grammar-Based Pruning

The first optimization is *grammar-based pruning*, which prunes grammarly incorrect path combinations during the creation of prefix trees (line 14 in Algorithm 1) for sibling dependency edges. The pruning leverages the alternation relations among "or" edges in a grammar graph.

Recall that in a grammar graph, hollowed-headed edges (e.g., those from "pos" to "POSITION" and "START" in Figure 4(a)) are *"or" edges* representing alternative relations among the siblings. Given a set of "or" edges that share the same non-terminal node as sinks, only one of the 'or' edges should be selected at a time to produce the CGT. If in a CGT, one non-terminal node has more than one "or" edge appearing in the CGT, the CGT must be grammarly incorrect. These "or" edges are called *conflict "or" edges*. Two candidate paths are called

a *conflict paths pair* if merging the two paths leads to conflict "or" edges.

Our algorithm identifies conflict paths pairs as follows. It merges common prefix nodes on all the candidate grammar paths of sibling edges. During the merging, it records the path ID on each edge in the prefix tree. It then checks the grammar of each non-terminal node. If there are two or more "or" edges under that node, the algorithm generates conflict path pairs based on the path ID of each of the "or" edges. After getting all the conflict paths pairs, it prunes the combinations that have any of the conflict path pairs.

For example, Figure 4(c) shows the prefix tree for the sibling dependency edges (2), (3), (4) in 4(a), with path ID recorded. In this all-path prefix tree, there are two pairs of conflict "or" edges [$e_1 = (\texttt{pos} \rightarrow \texttt{POSITION})$, $e_2 = (\texttt{pos} \rightarrow \texttt{START})$] and [$e_3 = (\texttt{pos\_arg} \rightarrow \texttt{AFTER})$, $e_4 = (\texttt{pos\_arg} \rightarrow \texttt{STARTFROM})$]. For the pair [$e_1$, e_2], paths $p_{2.2}$ and $p_{3.2}$ use e_1, and path $p_{3.1}$ uses e_2. The conflict paths pair [$p_{2.2}$, $p_{3.1}$]. [$p_{3.1}$, $p_{3.2}$] are from the same dependency edge so they cannot exist in one combination. Similarly, for the pair [e_3, e_4], there are conflict paths pair [$p_{2.2}$, $p_{3.2}$]. By pruning the combinations that have conflict paths pairs, this optimization avoids many unnecessary computations.

B. Orphan Node Relocation

The other optimization we introduce in this work is *orphan node relocation*. Due to the complexity in queries or NL parsing errors, in some cases, an edge $e = (n_i, n_j)$ in a pruned dependency graph has no candidate path in the grammar graph that connects their candidate APIs; it implies that n_i is not the "real" governor of n_j. We call n_j an *orphan node*.

The previous NLU-driven synthesis algorithm [34] simply regards an orphan node as the child of the root in the pruned dependency graph. As a result, the synthesis algorithm would find all the paths on the grammar graph from the node's candidate APIs to the grammar root node. It often leads to a larger number of candidate paths and causes inefficiency. Moreover, it could lead to cross-level prefixes. The DGGT algorithm assumes that the candidate paths of dependency edges in one level (e.g. edge (2), (3), (4) in Fig. 4(a)) have no common nodes with those in other levels (i.e. (1) and (5)). This is because if cross-level prefixes exist, the optimal partial CGTs will have overlaps and need to be merged. It would affect the size of the joined CGT; the CGT generated via joining optimal partial CGT from each level could become suboptimal.

We designed an algorithm, *orphan node relocation*, to address the issue. The algorithm relocates orphan nodes in an pruned dependency graph before DGGT is applied. The main idea is to use the knowledge from the grammar graph to determine the proper locations of orphan nodes. In a pruned dependency graph, suppose that that there is a dependency edge $e = (n_1 \rightarrow n_2)$ and the two dependency nodes n_1 and n_2 are mapped to two API nodes n_{A1} and n_{A2} respectively. Then in the grammar graph, all the paths from n_{A1} to n_{A2} are

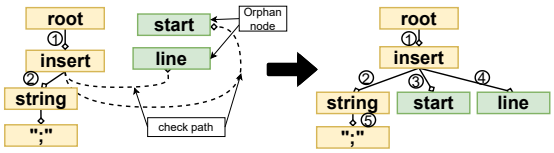

Figure 6. An illustration of orphan relocation.

the candidate paths of e. So if n_{A2} is the descendent of n_{A1} in the grammar graph but n_2 is an orphan node in the pruned dependency graph, it suggests that an edge $e = (n_1 \rightarrow n_2)$ should exist in the pruned dependency graph. Our algorithm adds such an edge and uses the augmented pruned dependency graph as the input to DGGT. Since an orphan node could have several candidate APIs, there could be many valid locations for the node inside the pruned dependency graph. In this case, the algorithm creates different pruned dependency graphs and synthesizes them separately. The smallest CGTs is chosen from all these pruned dependency graphs.

Figure 6 illustrates the orphan node relocation step. Both $n_{\texttt{start}}$ and $n_{\texttt{line}}$ are orphan nodes. In the grammar graph shown in Figure 4(a), the API `INSERT` (mapped to $n_{\texttt{insert}}$) is the ancestor of API `START` (mapped to $n_{\texttt{start}}$) and API `LINESCOPE` (mapped to $n_{\texttt{line}}$). The algorithm can hence relocate the orphan nodes $n_{\texttt{start}}$ and $n_{\texttt{line}}$ as the dependent of the node $n_{\texttt{insert}}$.

C. Other Optimizations

Another more detailed optimization in DGGT happens after enumerating all the path combinations, before merging the combinations. It prunes paths based on the size of a path combination. For a path p, the size of p is the number of APIs in the path, denoted as $size(p)$. Then for a collection of candidate paths, it is easy to see that the size of the merged combination $c = \{p_1, p_2, ..., p_n\}$ must meet the following: $len(\bigcup_{i=1}^{n}\{a|APIs\ in\ p_i\}) \leq size(c) \leq \sum_{i=1}^{n} size(p_i) - (n-1)$. The maximum size of the combination is reached when only the first node of each path can be merged. The minimum size is reached when all the common APIs in each path can be merged. Then for all the possible combinations, $\mathcal{C} = \{c_1, c_2, \cdots, c_m\}$, we calculate the minimum max_size among all the combinations as an upper bound for the size of each possible path combination: $\mathcal{C}.min\_size = \min_{i=1}^{m}(c_i.max\_size)$. So for a path combination c, if $c.min\_size > \mathcal{C}.min\_size$, then the prefix tree of this combination is not worth creating as it is definitely not the minimum tree. For example, in Figure 5, after grammar-based pruning (described next), the remaining combinations are c_1 = [2.1, 3.1, 4.1], c_2 = [2.1, 3.2, 4.1]. $c_1.min\_size = 5$, $c_1.max\_size = 5$, while $c_2.min\_size = 6$, $c_2.max\_size = 6$. Since $c_2.min\_size > c_1.max\_size$, we can hence directly prune c_2.

VI. Computational Complexity

In an query dependency graph, assume that the l-th level has e_l dependency edges and each edge corresponds to p_l number of path candidates in a grammar graph. Previous NLU-driven

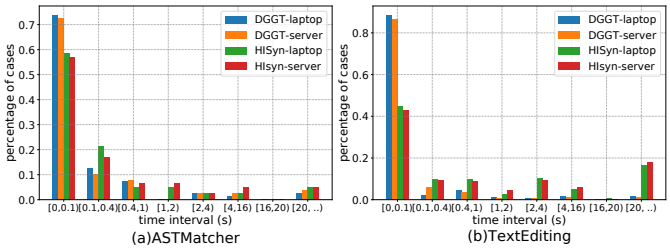

Figure 7. Execution time comparison.

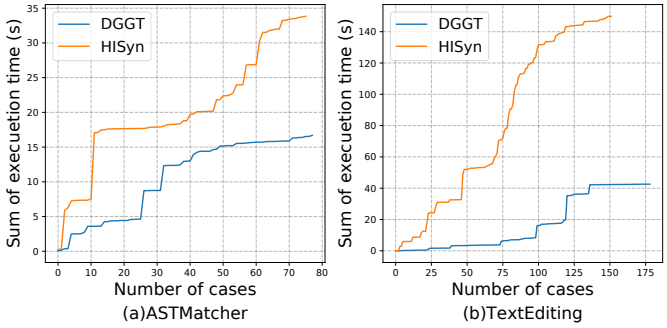

Figure 8. Accumulated execution time (on Laptop).

synthesis algorithms enumerate all paths, with $\prod_l {p_l}^{e_l}$ path combinations in the worst case and the overall computational complexity is $O(\prod_l {p_l}^{e_l})$. In the DGGT algorithm, the time complexity for generating partial optimal CGT for each level of an query dependency graph is $O(p_l^{e_l})$ in the worst case. Then by joining the partial optimal CGTs, the time complexity of the DGGT algorithm becomes only $O(\sum_l {p_l}^{e_l})$. The optimizations further reduce the cost.

VII. Evaluations

To evaluate the efficacy of DGGT, we conduct a set of experiments on two domains. We focus on answering the following research questions: (Q1) How much efficiency benefit can the techniques bring to NLU-driven code synthesis? (Q2) Does the DGGT algorithm affect the synthesis accuracy? (Q3) How much does each of the optimizations contribute to the speedups?

A. Methodology

Domains, Dataset, and Baselines. We experiment with two domains, ASTMatcher [7] in Code Analysis and Text Editing [9]. Table I lists the basic information of both domains and several example queries and code expressions from each domain. We compare to the most recent NLU-driven solution HISyn [34]. For comparison, the used query sets are the same as those used in the prior work [34].

Evaluation Metrics. We use **synthesis time** (t) to evaluate the performance of the DGGT and HISyn algorithms and use **speedup** $t(HISyn)$ $/t(DGGT)$ to measure the improvement brought by DGGT. Besides, we report DSL codes **synthesis accuracy**, which is the ratio between the number of *correctly* synthesized DSL code expressions and the number of total test cases. A synthesized DSL code is *correct* if it is identical to the ground truth code in terms of both the set of APIs, arguments, and their relative order.

Hardware. We conduct our experiments on two different machines: a laptop with 2.6GHz 6-Core Intel® Core(TM) i7 CPU and 16GB RAM; a server machine with 2.20GHz Intel® Xeon® Silver 4114 CPU and 128GB RAM.

B. Experimental Results

1) Synthesis Efficiency: In this experiment, we set 20 seconds as the timeout limit for processing one query. If the synthesizer fails to finish in time, we stop synthesizing, regard it an error case and record 20sec as the execution time. We choose 20sec as time limit because previous studies have shown that "10 seconds is about the limit for keeping the user's attention focused on the dialogue" [37]. Therefore, a response longer than 10 seconds would lose its usability significantly in an interactive scenario. We hence choose 20 seconds as the cut-off time to well cover the critical range of tolerable delays.

Table II reports the overall performance and accuracy of HISyn and DGGT. (The accuracy of HISyn differs from the accuracy reported in the previous paper [34] because the previous work uses hours as the timeout limit which is unreasonable for interactive usage.) Figure 7 provides the distributions of the response times.

Overall, in both domains, DGGT runs much faster than HISyn. In ASTMatcher domain, on laptop and server, DGGT completes 73.8% and 72.7% of the cases in less than 0.1s; only 6.3% and 9.1% of the cases take more than 1s. For HISyn, only 58.8% and 57.1% of cases take less than 0.1s; 15.0% and 19.5% of cases needs more than 1s to synthesis. In TextEditing domain, on laptop and server, DGGT completes 88.5% and 86.7% of the cases in less than 0.1s; only 4.9% and 3.2% of the cases take more than 1s. For HISyn, 45.1% and 43.1% of the cases take less than 0.1s; 35.1% and 38.3% of cases needs more than 1s to synthesis. We further show the accumulated execution time in Figure 8. The accumulated execution time $time(x)$ shows the total time needed to synthesis the codes from case 0 to x. The curves of DGGT raise much slower than those of HISyn.

To answer the research question Q1, we measure speedup brought by the DGGT algorithm over HISyn for each test case. The result is shown in Table II. In the TextEditing domain, the maximum speedup is 1887x, in which case, it costs HISyn 16.4s to synthesis the correct code, but costs DGGT only 0.00869s. The average speedup in the TextEditing domain is 133.2x, and the median is 12.86x . In the ASTMatcher domain, the maximum speedup is 537.7x, the average speedup is 25.02x, and the median is 3.463x.

The performance comparison between HISyn and DGGT on server is similar to that on the laptop, demonstrating the portability of the benefits from DGGT.

2) Synthesis Accuracy: The synthesis accuracy (Q2) of DGGT are 76.5% and 79.1% in the two domains on the laptop, exceeding the accuracy of HISyn which are 74.7% and 67.5%

Table I
TESTING DOMAINS AND TEST CASES

Domain	Source	Description	#APIs	#Queries	Example queries and code expressions
TextEditing	[9]	A command language that aims to free Office suite application end-users from understanding syntax and semantics of regular expressions, conditionals, and loops	52	200	1) Append ":" in every line containing numerals. - *INSERT(STRING(:), END(), IterationScope(LINESCOPE(), BConditionOccurrence(CONTAINS(NUMBERTOKEN()), ALL()))* 2) if a sentence starts with "–", add ":" after 14 characters - *INSERT(STRING(–), Position(AFTER(CHARTOKEN()), IntegerSet(INTEGER(14))), IterationScope(LINESCOPE(), BConditionOccurrence(STARTSWITH(STRING(Exercise)), ALL())))* 3) Remove the 3rd and 4th line - *REMOVE(SelectString(LINETOKEN(), BConditionOccurrence(ALWAYS(), IntegerSet(INTEGER(3), INTEGER(4)))), IterationScope(LINESCOPE(), BConditionOccurrence(ALWAYS(), ALL()))* 4) Put '+' at the beginning of 2nd, 6th line - *INSERT(STRING(+), START(), IterationScope(LINESCOPE(), BConditionOccurrence(ALWAYS(), IntegerSet(INTEGER(2), INTEGER(6)))))*
ASTMatcher	[34]	A tool in Clang/LLVM [8] for constructing Abstract Syntax Trees (AST) Matching expressions to find code patterns of interest.	505	100	5) find cxx constructor expressions which declare a cxx method named "PI" - *cxxConstructExpr(hasDeclaration(cxxMethodDecl(hasName("PI"))* 6) serach for call expressions whose argument is a float literal - *callExpr(hasArgument(floatLiteral()))* 7) list all binary operators named "*" - *binaryOperator(hasOperatorName("*"))*

Table II
PERFORMANCE COMPARISON. (20SEC TIMEOUT)

Domain	H/W	Speedup (X)			Accuracy	
		Max	Mean	Median	HISyn	DGGT
AST-Matcher	Laptop	537.7	25.02	3.463	0.744	0.765
	Server	611.9	32.72	4.038	0.737	0.769
Text-Editing	Laptop	1887	133.2	12.86	0.675	0.791
	Server	1911	102.6	10.00	0.67	0.75

Table III
DETAILED RESULTS OF DGGT ALGORITHM ON 4 CASES.

Ex. ID	# of dep edge	HISyn		DGGT					Time Speedup (X)
		# of orig. path	# of comb.	After orph reloc		Optimize			
				# of path	# of comb.	gram.-based	size-based	remain comb.	
1	5	388	3.8e6	71	3744	3545	182	17	8186
2	7	555	1.3e10	179	673920	673500	4	416	1902
3	4	472	2.9e5	62	7600	7368	0	232	1887
4	6	880	2.8e9	103	226800	226714	0	86	>2748

respectively. The results on the server are similar. Theoretically, as DGGT only accelerates the synthesis process in HISyn, it should produce identical synthesis results in all the cases and hence has the same accuracy as HISyn has. Things change when timeout is considered, as a timeout is an error. DGGT has fewer timeout cases and hence higher accuracy.

3) Case Study: To examine the reasons for the dramatic speedups that DGGT brings (Q3), we report in Table III detailed results on 4 cases. The queries are the No.1-4 examples in Table I. Take case 1 as an example. In its query dependency graph, there are 5 dependency edges and 388 paths in total. The word *lines* and *contains* are two orphan nodes. There are as many as 3,810,240 possible combinations. In DGGT, after orphan relocation, the number of total paths reduces to 71. For levels with sibling edges, there are 3744 combinations, the grammar pruning removes 3545 (94.7%) of them. Then for the left 199 cases, size-based pruning removes 182 (91.4%) of them. For the remaining 17 combinations, DGGT merges the paths to prefix trees and chooses the smallest prefix trees as the optimal partial CGTs. It then joins them with the optimal partial CGTs of other levels. The previous algorithm takes 85.7 seconds to synthesize, while DGGT takes only 0.0105 seconds. Cases 2, 3 and 4 share the same optimization pattern with case 1. In case 4, HISyn did not finish synthesize within 20min, while DGGT gives the correct result in 0.436 sec. Therefore, the speedup is as much as 2748×.

According to Table III, the orphan node relocation can reduce the number of candidate paths in the query dependency graph by selecting proper governor for orphan nodes. The optimizations, grammar-based pruning and size-based pruning, further help avoid over 90% of combinations. DGGT needs to generate only a small number of prefix trees for the remaining combinations. These greatly reduce the execution time, bringing a huge speedup in challenging cases.

4) Error Analysis and Discussion: It is worth nothing that even though the synthesis accuracy is not perfect, as previous studies have mentioned [9], such results can be already useful in reducing the burden of general users. The technique, for instance, can be integrated into an IDE, offering a list of ranked candidate expressions for the programmer to choose when she types in her intent in natural language.

VIII. RELATED WORK

In prior work, NLP has been used in software maintenance and other software engineering tasks [1], [4], [12], [13], [16], [22], [36], [48], [49], [51]. Here we focus on work closely related with code synthesis.

Various specifications are used for code synthesis. A specification can be first order logic expressions [19], [24], a set of examples [17], [18], [43], natural language [9], [20], [26], [27], [42], [47], partial programs [14], [44] or any other form that is easier to write than the expected program. We concentrate on work on program synthesis from natural language (NL).

Besides NLU-driven NL programming [34], there are rule-based and data-driven methods developed in prior work. Example rule-based work includes smartphone automation [26], SQL queries [28], [47], and SpreadSheet data analysis [20].

Many recent studies have pursued modern machine learning for NL programming [2], [6], [9], [11], [15], [21], [23], [25], [29]–[31], [38]–[40], [42], [45], [50]. For example, Chen [6] has applied LSTM-based sequence-to-sequence model with other specifications for code synthesis. Desai and others [9] have presented a general framework for constructing program synthesizers based on a domain-specific language (DSL) definition and training data. Quirk [40] uses the semantic mapping approach which learns to map natural-language descriptions of "if-then" rules to executable code. Lin [29] leverages recurrent neural networks (RNNs) for NL to code translation. Chen [5] trained a semantic parser that translate NL to sketch, then perform synthesis from sketch with examples. Applying these approaches to program analysis would require many training examples to cover the vast space of possible code complexities and situations. They are difficult to apply to areas where labeled training data is scarce, which is especially common for program synthesis on multi-sentence queries.

Another body of work is API learning. These work tries to identify some statistical patterns of API usage. Examples include code search tools [32], [33], API usage pattern mining [41], [46], API sequence generation [15]. These studies rely on statistical machine learning techniques. They hence require a large set of examples, requiring extra efforts when applying to other domains.

IX. Conclusion

The paper presented the first known solution that enables near real-time NLU-driven natural language programming. It introduces a new algorithm dynamic grammar graph-based translation for identifying the best grammar tree for a given query and two optimizations, grammar-based pruning and orphan node relocation, to reduce the search space and address complexities from queries. It demonstrates the effectiveness of the proposed solution through a set of experiments on two domains, showing that the techniques can shorten the synthesis time of a state-of-the-art NLU-driven algorithm significantly. It brings up to 1887× (25-133× on average) speedups, while improving the accuracy by 2-12% for its effects in reducing the timeout cases.

Acknowledgement

This material is based upon work supported by the National Science Foundation (NSF) under Grants CNS-1717425, CCF-1703487, CCF-2028850, and the Department of Energy (DOE) under Grant DE-SC0013700. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF or DOE.

References

[1] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, 51(4):1–37, 2018.

[2] Rohan Bavishi, Caroline Lemieux, Roy Fox, Koushik Sen, and Ion Stoica. Autopandas: neural-backed generators for program synthesis. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–27, 2019.

[3] Giovanni Campagna, Silei Xu, Mehrad Moradshahi, Richard Socher, and Monica S Lam. Genie: A generator of natural language semantic parsers for virtual assistant commands. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 394–410, 2019.

[4] Oscar Chaparro, Jing Lu, Fiorella Zampetti, Laura Moreno, Massimiliano Di Penta, Andrian Marcus, Gabriele Bavota, and Vincent Ng. Detecting missing information in bug descriptions. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 396–407, 2017.

[5] Qiaochu Chen, Xinyu Wang, Xi Ye, Greg Durrett, and Isil Dillig. Multi-modal synthesis of regular expressions. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 487–502, 2020.

[6] Yanju Chen, Ruben Martins, and Yu Feng. Maximal multi-layer specification synthesis. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 602–612, 2019.

[7] The Clang-Team. Astmatcher reference. clang.llvm.org/docs/LibASTMatchersReference.html.

[8] The Clang-Team. Clang. clang.llvm.org.

[9] Aditya Desai, Sumit Gulwani, Vineet Hingorani, Nidhi Jain, Amey Karkare, Mark Marron, Subhajit Roy, et al. Program synthesis using natural language. In *Proceedings of the 38th International Conference on Software Engineering*, pages 345–356. ACM, 2016.

[10] Android developers. Android api reference. https://developer.android.com/reference.

[11] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. Robustfill: Neural program learning under noisy i/o. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 990–998. JMLR. org, 2017.

[12] Pradeep Dogga, Karthik Narasimhan, Anirudh Sivaraman, and Ravi Netravali. A system-wide debugging assistant powered by natural language processing. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 171–177, 2019.

[13] Michael D Ernst. Natural language is a programming language: Applying natural language processing to software development. In *2nd Summit on Advances in Programming Languages (SNAPL 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[14] Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas W Reps. Component-based synthesis for complex apis. *ACM SIGPLAN Notices*, 52(1):599–612, 2017.

[15] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. Deep api learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 631–642. ACM, 2016.

[16] Hui Guan, Xipeng Shen, and Hamid Krim. Egeria: a framework for automatic synthesis of hpc advising tools through multi-layered natural language processing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2017.

[17] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *ACM SIGPLAN Notices*, volume 46, pages 317–330. ACM, 2011.

[18] Sumit Gulwani. Synthesis from examples: Interaction models and algorithms. In *Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2012 14th International Symposium on*, pages 8–14. IEEE, 2012.

[19] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Synthesis of loop-free programs. *ACM SIGPLAN Notices*, 46(6):62–73, 2011.

[20] Sumit Gulwani and Mark Marron. Nlyze: Interactive programming by natural language for spreadsheet data analysis and manipulation. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 803–814. ACM, 2014.

[21] Tihomir Gvero and Viktor Kuncak. Synthesizing java expressions from free-form queries. In *Acm Sigplan Notices*, volume 50, pages 416–432. ACM, 2015.

[22] Sonia Haiduc, Venera Arnaoudova, Andrian Marcus, and Giuliano Antoniol. The use of text retrieval and natural language processing in software engineering. In *Proceedings of the 38th International Conference on Software Engineering Companion*, pages 898–899, 2016.

[23] Gang Hu, Linjie Zhu, and Junfeng Yang. Appflow: using machine learning to synthesize robust, reusable ui tests. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 269–282, 2018.

[24] Susmit Jha, Sumit Gulwani, Sanjit A Seshia, and Ashish Tiwari. Oracle-guided component-based program synthesis. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, volume 1, pages 215–224. IEEE, 2010.

[25] Gregory Kuhlmann, Peter Stone, Raymond Mooney, and Jude Shavlik. Guiding a reinforcement learner with natural language advice: Initial results in robocup soccer. In *The AAAI-2004 workshop on supervisory control of learning and adaptive systems*. San Jose, CA, 2004.

[26] Vu Le, Sumit Gulwani, and Zhendong Su. Smartsynth: Synthesizing smartphone automation scripts from natural language. In *Proceeding of the 11th annual international conference on Mobile systems, applications, and services*, pages 193–206. ACM, 2013.

[27] Xuan-Bach D Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. S3: syntax-and semantic-guided repair synthesis via programming by examples. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 593–604, 2017.

[28] Fei Li and HV Jagadish. Constructing an interactive natural language interface for relational databases. *Proceedings of the VLDB Endowment*, 8(1):73–84, 2014.

[29] Xi Victoria Lin, Chenglong Wang, Deric Pang, Kevin Vu, Luke Zettlemoyer, and Michael D Ernst. Program synthesis from natural language using recurrent neural networks. *University of Washington Department of Computer Science and Engineering, Seattle, WA, USA, Tech. Rep. UW-CSE-17-03-01*, 2017.

[30] Xi Victoria Lin, Chenglong Wang, Luke Zettlemoyer, and Michael D Ernst. Nl2bash: A corpus and semantic parser for natural language interface to the linux operating system. *arXiv preprint arXiv:1802.08979*, 2018.

[31] Wang Ling, Edward Grefenstette, Karl Moritz Hermann, Tomáš Kočiskỳ, Andrew Senior, Fumin Wang, and Phil Blunsom. Latent predictor networks for code generation. *arXiv preprint arXiv:1603.06744*, 2016.

[32] Fei Lv, Hongyu Zhang, Jian-guang Lou, Shaowei Wang, Dongmei Zhang, and Jianjun Zhao. Codehow: Effective code search based on api understanding and extended boolean model (e). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 260–270. IEEE, 2015.

[33] Collin McMillan, Mark Grechanik, Denys Poshyvanyk, Qing Xie, and Chen Fu. Portfolio: finding relevant functions and their usage. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 111–120. ACM, 2011.

[34] Zifan Nan, Hui Guan, and Xipeng Shen. Hisyn: human learning-inspired natural language programming. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 75–86, 2020.

[35] Zifan Nan, Hui Guan, Xipeng Shen, and Chunhua Liao. Deep nlp-based co-evolvement for synthesizing code analysis from natural language. In *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction (CC)*, pages 141–152, 2021.

[36] Pengyu Nie, Junyi Jessy Li, Sarfraz Khurshid, Raymond Mooney, and Milos Gligoric. Natural language processing and program analysis for supporting todo comments as software evolves. In *Workshops at the Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[37] Jakob Nielsen. *Usability engineering*. Morgan Kaufmann, 1994.

[38] Emilio Parisotto, Abdel-rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. Neuro-symbolic program synthesis. *arXiv preprint arXiv:1611.01855*, 2016.

[39] Illia Polosukhin and Alexander Skidanov. Neural program search: Solving programming tasks from description and examples. *arXiv preprint arXiv:1802.04335*, 2018.

[40] Chris Quirk, Raymond Mooney, and Michel Galley. Language to code: Learning semantic parsers for if-this-then-that recipes. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 878–888, 2015.

[41] Mukund Raghothaman, Yi Wei, and Youssef Hamadi. Swim: Synthesizing what i mean-code search and idiomatic snippet synthesis. In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*, pages 357–367. IEEE, 2016.

[42] Mohammad Raza, Sumit Gulwani, and Natasa Milic-Frayling. Compositional program synthesis from natural language and examples. In *IJCAI*, pages 792–800, 2015.

[43] Xujie Si, Woosuk Lee, Richard Zhang, Aws Albarghouthi, Paraschos Koutris, and Mayur Naik. Syntax-guided synthesis of datalog programs. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 515–527, 2018.

[44] Armando Solar-Lezama and Rastislav Bodik. *Program synthesis by sketching*. Citeseer, 2008.

[45] Yu Su, Ahmed Hassan Awadallah, Madian Khabsa, Patrick Pantel, Michael Gamon, and Mark Encarnacion. Building natural language interfaces to web apis. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, pages 177–186. ACM, 2017.

[46] Tao Xie and Jian Pei. Mapo: Mining api usages from open source repositories. In *Proceedings of the 2006 international workshop on Mining software repositories*, pages 54–57. ACM, 2006.

[47] Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, and Thomas Dillig. Sqlizer: query synthesis from natural language. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):63, 2017.

[48] Hao Yu, Wing Lam, Long Chen, Ge Li, Tao Xie, and Qianxiang Wang. Neural detection of semantic code clones via tree-based convolution. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 70–80. IEEE, 2019.

[49] Juan Zhai, Xiangzhe Xu, Yu Shi, Minxue Pan, Shiqing Ma, Lei Xu, Weifeng Zhang, Lin Tan, and Xiangyu Zhang. Cpc: automatically classifying and propagating natural language comments via program analysis. 2019.

[50] Victor Zhong, Caiming Xiong, and Richard Socher. Seq2sql: Generating structured queries from natural language using reinforcement learning. *arXiv preprint arXiv:1709.00103*, 2017.

[51] Yu Zhou, Ruihang Gu, Taolue Chen, Zhiqiu Huang, Sebastiano Panichella, and Harald Gall. Analyzing apis documentation and code to detect directive defects. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 27–37. IEEE, 2017.