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

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

Xipeng Shen
Department of Computer Science
North Carolina State 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 NL programming 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 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.

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
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× 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× (25-133× 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 ASTMatcher1; (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 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*<sub>*l*</sub> dependency edges and each edge corresponds to *p*<sub>*l*</sub> number of path candidates in a grammar graph. The current NLU-driven synthesis algorithms enumerate all paths, with ∏<sub>*l*</sub> *p*<sup>*e*<sub>*l*</sub> path combinations in the worst case and the overall computational complexity is O(∏<sub>*l*</sub> *p*<sup>*e*<sub>*l*</sub>). For a non-trivial query with 4 dependency levels with *e*<sub>1</sub> = 4 dependency edges per level and *p*<sub>1</sub> = 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*<sub>1</sub>, *e*<sub>2</sub>, *e*<sub>3</sub>} with each dependency edge having two grammar paths. We denote the six paths as {*p*<sub>1a</sub>, *p*<sub>1b</sub>}, {*p*<sub>2a</sub>, *p*<sub>2b</sub>}, {*p*<sub>3a</sub>, *p*<sub>3b</sub>}. There are hence eight combinations: {*p*<sub>1a</sub>, *p*<sub>2a</sub>, *p*<sub>3a</sub>}, {*p*<sub>1a</sub>, *p*<sub>2b</sub>, *p*<sub>3b</sub>}, ... These combinations have lots of overlaps: {*p*<sub>1a</sub>, *p*<sub>2a</sub>}, 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*. 
A grammar graph is a graph representation of a context-free grammar (CFG) $\langle T, N^T, N, S, P \rangle$ where $T$: terminals, $N^T$: non-terminals, $S$: start symbol, $P$: 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.

\[
\begin{align*}
\text{insert} & ::= \text{string pos iter} \\
\text{string} & ::= \text{STRING} \\
\text{pos} & ::= \text{POSITION} \mid \text{START} \\
\end{align*}
\]

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.
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 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. \( N_{\text{API}} \) is the set of nodes that represent APIs; \( N_{\text{PCGT}} \) is the set of nodes that represent
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.

### Algorithm 1: DGGT algorithm

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

```
Function DGGT(dg, gg)
    // Bottom-up dynamic grammar graph generation
    new DynamicGraph dyng
    add start_node and leaf_API_node into dg
    foreach level in dg from bottom to top do
        Let $e : n_1 \rightarrow n_2$ be an edge at the current level
        if $n_1 \in dg$ has a single child $n_2$ then
            create $N_{n_1}$ in dyng for each candidate API $a_i$ of $n_1$
            foreach grammar path $\in EdgeToPath(e)$ do
                create a path edge ($N_{b_i} \rightarrow N_{a_i}$) in dyng where $b_i$ is a candidate API of $n_2$
            foreach $N_{n_2}$ do
                update its min_cgt and min_size with the smallest partial CGT
            else if $n_1$ in dg has multiple children then
                comb = the set of combinations of the grammar paths of $n_1$’s children
                comb = grammar_based_pruning(comb)
                comb = size_based_pruning(comb)
                foreach $e \in$ comb do
                    merge the paths in $e$ into a tree $pt$, and create node $N_{pt}$ in dyng;
                    update min_cgt and min_size of $N_{pt}$
                    create node $N_{root}$ in dyng for the root of pt;
                    if it does not exist;
                    connect $N_{pt}$ with the nodes in dyng that correspond to the leaves of $pt$;
                    connect $N_{pt}$ with $N_{root}$
                update min_cgt and min_size of $N_{root}$
            Backtrack dyng to create the optimal CGT and generate code
```

**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 $a_1, a_2, \ldots, a_n$, the algorithm generates API nodes $\{N_{a_i}\}$ and edges $\{E_{default} = (N_{start} \rightarrow N_{a_i})\}$ (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 Word2API map, $n_1$ has candidate APIs $a_1, a_2, \ldots, a_n$ and $n_2$ has candidate APIs $b_1, b_2, \ldots, b_m$. Because the visit is bottom up, by this time, the dynamic grammar graph shall already have nodes corresponding to $b_j$ ($j=1, 2, \ldots, m$). The algorithm creates a node for each $a_i$ and adds the nodes into the dynamic grammar graph, denoted as $N_{a_i}$ ($i=1, 2, \ldots, n$). It adds edges $\{E_{pk} = (N_{aj} \rightarrow N_{a_i})\}$ 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. 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 \( \text{min}\_\text{size} \) 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 \([\text{STRING} \rightarrow \cdot \cdot \cdot \cdot \cdot \text{;}\] \). When traversing the dependency edge 2 in level-3, we create a node \( N_{\text{STRING}} \) and a path edge \( E_{\text{path}2} = (N_i \rightarrow N_{\text{STRING}}) \) in the dynamic grammar graph. \( N_{\text{STRING}} \) 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 \( N_{\text{root}} \) for the root node and two edges in the dynamic grammar graph for the two grammar paths that connect the root node to the \( \text{INSERT} \) in the grammar graph. The \( \text{min}\_\text{size} \) of \( N_{\text{root}} \) 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 \( \text{INSERT} \) and \( \text{insert}_\text{arg} \). 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 \( \text{EdgeToPath} \) 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 \( \text{min}\_\text{size} \) 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 “\( \text{INSERT} \)” 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 dependency 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 \( N_{\text{PCGT}_1} \) and \( N_{\text{PCGT}_2} \) are created for each prefix tree respectively. Six path edges such as \( E_{\text{path}2} = (N_{\text{STRING}} \rightarrow N_{\text{PCGT}_1}) \) are created to link the APIs nodes and the partial CGT nodes. Since the two prefix trees have common root API \( \text{INSERT} \), we create the API node \( N_{\text{INSERT}} \) and two auxiliary edges to link the two partial CGT nodes and their root node \( N_{\text{INSERT}} \). For node \( N_{\text{INSERT}} \), the optimal partial CGT is partial CGT #1 with minimum size of 5 and the corresponding node is \( N_{\text{PCGT}_1} \).

**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 \( N_{\text{API}} \) 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 synthesis 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 $[e_1 = (pos \rightarrow POSITION), e_2 = (pos \rightarrow START)]$ and $[e_3 = (pos_{arg} \rightarrow AFTER), e_4 = (pos_{arg} \rightarrow STARTFROM)]$. 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 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 $e = (n \rightarrow n_2)$ 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 $n_{start}$ and $n_{line}$ are orphan nodes. In the grammar graph shown in Figure 4(a), the API INSERT (mapped to $n_{insert}$) is the ancestor of API START (mapped to $n_{start}$) and API LINESCOPE (mapped to $n_{line}$). The algorithm can hence relocate the orphan nodes $n_{start}$ and $n_{line}$ as the dependent of the node $n_{insert}$.

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 $c = \{p_1, p_2, \ldots, p_n\}$ must meet the following: $len(\bigcup_{i=1}^{n} \{APIs in p_i\}) \leq size(c) \leq \sum_{i=1}^{n} size(p_i) - (n - 1)$. 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, $C = \{c_1, c_2, \ldots, c_m\}$, we calculate the minimum $max.size$ among all the combinations as an upper bound for the size of each possible path combination: $C.min.size = \min_{c \in C} (c.max.size)$. So for a path combination $c$, if $c.min.size > C.min.size$, 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
synthesis algorithms enumerate all paths, with $\prod_{l} p_l^{c_l}$ path combinations in the worst case and the overall computational complexity is $O(\prod_{l} p_l^{c_l})$. In the DGGT algorithm, the time complexity for generating partial optimal CGT for each level of an query dependency graph is $O(p_l^{c_l})$ in the worst case. Then by joining the partial optimal CGTs, the time complexity of the DGGT algorithm becomes only $O(\sum_{l} p_l^{c_l})$. 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 $t_{\text{HISyn}} / t_{\text{DGGT}}$ 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 $\text{time}(x)$ 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. Take case 1 as an example. In its query dependency graph, there are 5 dependency edges and 388 paths in total. DGGT needs to generate only a small number of prefix trees for the remaining levels with sibling edges, there are 3744 combinations, the grammar pruning removes 3545 (94.7%) of them. Then for the remaining 17 combinations, DGGT merges the paths to prefix trees and chooses the smallest prefix trees as much as 3,810,240 possible combinations. In DGGT, after 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].

### Table II

<table>
<thead>
<tr>
<th>Domain</th>
<th>Source</th>
<th>Description</th>
<th>#APIS</th>
<th>#Queries</th>
<th>Example queries and code expressions</th>
</tr>
</thead>
</table>
| 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 `~` or `~` 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(SELECT(String(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 [9] 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 `PP`  
- `cxxConstructExpr(hasDeclaration(cxxMethodDecl(hasName("PF")));`  
6) search for call expressions whose argument is a float literal  
- `callExpr(hasArgument(floatLiteral()))`  
7) list all binary operators named `*`  
- `binaryOperator(hasOperatorName("*"));` |

### Table III

**Performance Comparison. (20sec Timeout)**

<table>
<thead>
<tr>
<th>Domain</th>
<th>H/W</th>
<th>Speedup (X)</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>AST-Matcher</td>
<td>Laptop</td>
<td>537.7</td>
<td>25.02</td>
</tr>
<tr>
<td>Text-Editing</td>
<td>Server</td>
<td>611.9</td>
<td>32.72</td>
</tr>
</tbody>
</table>

### Table IV

**Detailed Results of DGGT Algorithm on 4 Cases.**

<table>
<thead>
<tr>
<th>Ex.</th>
<th># of dep edge</th>
<th># of orig. comb. path</th>
<th>After orph reloc</th>
<th># of path comb.</th>
<th>Optimize</th>
<th>Time Speedup (X)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>388</td>
<td>3.86</td>
<td>71</td>
<td>3744</td>
<td>3545</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>555</td>
<td>1.3e10</td>
<td>179</td>
<td>673920</td>
<td>673500</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>472</td>
<td>2.9e5</td>
<td>62</td>
<td>7600</td>
<td>7368</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>880</td>
<td>2.8e9</td>
<td>103</td>
<td>226800</td>
<td>226714</td>
</tr>
</tbody>
</table>

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 related with code synthesis.
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


