2016 IEEE/ACM 38th IEEE International Conference on Software Engineering

Program Synthesis using Natural Language

Aditya Desai
IIT Kanpur
adityapd@cse.iitk.ac.in

Sumit Gulwani
MSR Redmond
sumitg@microsoft.com

Vineet Hingorani Nidhi Jain
IIT Kanpur
viner,nidhij@cse.iitk.ac.in

Amey Karkare
IIT Kanpur
karkare@cse.iitk.ac.in

Mark Marron
MSR Redmond
marron@microsoft.com

Sailesh R Subhajit Roy
IIT Kanpur
sairaj,subhajit@cse.iitk.ac.in

ABSTRACT

Interacting with computers is a ubiquitous activity for millions of people. Repetitive or specialized tasks often require creation of small, often one-off, programs. End-users struggle with learning and using the myriad of domain-specific languages (DSLs) to effectively accomplish these tasks.

We present a general framework for constructing program synthesizers that take natural language (NL) inputs and produce expressions in a target DSL. The framework takes as input a DSL definition and training data consisting of NL/DSL pairs. From these it constructs a synthesizer by learning optimal weights and classifiers (using NLP features) that rank the outputs of a keyword-programming based translation. We applied our framework to three domains: repetitive text editing, an intelligent tutoring system, and flight information queries. On 1200+ English descriptions, the respective synthesizers rank the desired program as the top-1 and top-3 for 80% and 90% descriptions respectively.

1. INTRODUCTION

Program synthesis is the task of automatically synthesizing a program in some underlying *domain-specific language (DSL)* from a given specification [12]. Traditional program synthesis, synthesizing programs from *complete specifications* [18, 35, 51, 52], has not yet seen wide adoption due to the difficultly of writing such specifications and verifying the synthesized program satisfies this specification.

Recent work has experimented with another class of (*possibly incomplete*) specifications, namely *examples* [7, 13, 14, 17, 31]. Programming by Example (PBE) systems have seen much wider adoption, thanks to the ease of providing such a specification. However, they can suffer in cases where many examples are required to accurately specify intent or where examples are difficult to construct. The classic L* algorithm [3], a PBE system for describing a regular language, has the well-known drawback of requiring too many examples. A domain such as ATIS queries (Air Travel Information System [8]) is a case where constructing an example input/output pair is a non-trivial task for an end-user. However, tasks in these domains can easily be specified natural language as, as we show in this work, can be used to reliably synthesize the desired program.

In this paper, we address the problem of synthesizing programs in an underlying domain specific language (DSL) from natural language (NL). NL is inherently imprecise; hence, it may not be possible to guarantee the correctness of the synthesized program. Instead, we aim to generate a ranked set of programs and let the user possibly select one of those programs by either inspecting the source code of the program, or the result of executing them on some test inputs [37]. The synthesis algorithm in this paper is able to consistently produce and rank the desired result program in the top spot, over 80% of the time, or in the top 3 spots, over 90% of the time in our benchmarks. To give users confidence in the program they choose, we show both the translation of the code into disambiguated English and/or run it to show the result as a preview.

Unlike most existing synthesis techniques that specialize to a specific DSL, such as [13, 15, 50, 51], our approach can be applied to a variety of DSLs. Our approach requires two inputs from the synthesis designer: (i) The DSL definition, which we assume provides a set of operations that are similar to the concepts an end-user might express in NL, (ii) Training data consisting of example pairs of English sentences and corresponding intended programs in the DSL. A training phase infers a *dictionary* relation over pairs of English words and DSL terminals (in a semi-automated interactive manner), and optimal weights/classifiers (in a completely automated manner) for use by the generic synthesis algorithm. Our approach can be seen as a meta-synthesis framework for constructing NL-to-DSL synthesizers.

The generic synthesis algorithm (Alg. 1) takes as input an English sentence and generates a ranked set of likely programs. First, it uses a *bag algorithm* (Alg. 2) to efficiently compute the set of all well-typed DSL programs whose terminals are related to the words that occur in the sentence. For this, it uses a *dictionary* (learned during the training phase) that is a relation over English words and DSL terminals. Then, it ranks these programs based on a set of scoring functions (§4.2) inspired by our view of the Abstract Syntax Tree (AST) of a program as involving two constituents: the set of terminals in the program, and the tree structure between those terminals. A weighted linear combination of 3 scores determines the rank of each program: (i) a *coverage score* that captures the intuition that results that ignore many words in the user input are unlikely to be correct (ii) a *mapping score* that captures the intuition that English words can have multiple meanings wrt. the DSL but we prefer the more probable interpretations (iii) a *structure score* that uses the insight that natural and programming languages have common idiomatic structures [22], and prefer more *natural* results.

ICSE '16, May 14-22, 2016, Austin, TX, USA

DOI: http://dx.doi.org/10.1145/2884781.2884786

(a) Grammar

```
S:= Command | SEQ(Command, Command)
Command:= ReplaceCmd | RemoveCmd | InsertCmd | PrintCmd
ReplaceCmd:= Replace(SelectStr, NewString, IterScope)
RemoveCmd:= Remove(SelectStr, IterScope)
InsertCmd:= Insert(PString, Position, IterScope)
PrintCmd:= Print(SelectStr, IterScope)
SelectStr:= (Token, BCond, Occurrence)
IterScope:= (Scope, BCond, Occurrence) | DOCUMENT
Token:= PString | LINE | WORD | NUMBER | ...
BCond:= AtomicCond |Not(AtomicCond)|And(AtomicCond,AtomicCond)|...
AtomicCond:= StartsWith(Token) | Contains(Token) | CommonCond
CommonCond:= Between(Token, Token)| After(Token)| ...
Occurrence:= ALL | AtomicOccurrence | ...
AtomicOccurrence:= IntSetByAnd | FirstFew(Integer) | ...
IntSetByAnd:= Integer | INTSET(IntegerSet)
Scope:= LINESCOPE | WORDSCOPE
Position:= START | END | ...
NewString:= BY(PString)
```

(b) Sample Benchmarks

1. Remove the first word of lines which start with number.
2. Replace "&" with "&&" unless it is inside "[" and "]".
3. Add "$" at the beginning of those lines that do not already start with "$".
4. Add "..???" at the last of every 2nd statement.
5. In every line, delete the text after "//".
6. Remove 1st "&" from every line.
7. Add the suffix "_IDM" to the word right after "idiom:".
8. Delete all but the 1st occurrence of "Cook".
9. Delete the word "the" wherever it comes after "all".

(c) Variations in NL for description of the same task.

1. Prepend the line containing "P.O. BOX" with "*"
2. Add a "*" at the beginning of the line in which the string "P.O. BOX" occurs
3. Put a "*" before each line that has "P.O. BOX" in it
4. Put "*" in front of lines with "P.O. Box" as a substring
5. Insert "*" at the start of every line which has "P.O. BOX" word

Table 1: Grammar and sample benchmarks for the Text Editing domain.

The classifiers to compute these scores, as well as the weights for combining the scores are learned during the training phase using off-the-shelf machine learning algorithms. The novelty of of our approach lies in the generation of training data for classifier learning from the top-level training data (Alg. 3 and 4), and in smoothing a discrete scoring metric into a continuous and differentiable loss function for effective learning of weights (§5.4).

This paper makes the following contributions:

- We describe a meta-synthesis framework for constructing NL-to-DSL synthesizers, consisting of a synthesis algorithm (§4) for translating English sentences into corresponding programs in the underlying DSL, and a training phase for learning a dictionary and weights that are used by the synthesis algorithm (§5). Our method can be applied to new DSLs, and requires only the DSL definition along with translation pair training data.
- We apply our generic framework to three different domains, namely automating end-user data manipulation (§2.1), generating problem descriptions in intelligent tutoring systems (§2.2), and database querying (§2.3). In cases where comparisons can be made with state-of-the-art NLP based approaches, the results of the approach presented in this paper are competitive.
- We gather an extensive corpus of data consisting of 1272 pairs of English descriptions and corresponding programs. We use this data for evaluation in this paper, and provide it as a resource [9] for researchers in the community. Of these, 535 English descriptions come from the *Air Travel Information System (ATIS)* benchmark suite [8], 227 come from another corpus [36], while 510 English descriptions were collected by us from various online sources (including help forums and course materials), textbooks, and user studies.
- We evaluate the effectiveness of our approach on three different DSLs (§6). The synthesizers produced by our framework run in $1-2$ seconds on average per benchmark and produce a ranked set of candidate programs with the correct result in the top-1/top-3 choices for over 80%/90% benchmarks respectively.

2. MOTIVATING SCENARIOS

We describe 3 different domains where a NL-to-DSL synthesizer is useful: text editing (§2.1), automata construction problems for intelligent tutoring (§2.2), and answering queries for an air travel information systems (§2.3).

2.1 Text Editing (End-User Programming)

Through a study of help forums for Office suite applications like Microsoft Excel and Word, we observed that users frequently request help with repetitive *text editing* operations such as insertion, deletion, replacement, or extraction in text files. These operations (Table 1(b)) are more complicated than simple search-and-replace of a constant string by another in two ways. First, the string being searched for is often not constant and instead requires regular expression matching. Second, the editing is often conditional on the surrounding context. Programming of even such relatively simple tasks requires the user to understand syntax and semantics of regular expressions, conditionals, and loops, which are beyond the ability of most end-users.

This inspired us to design a command language for text-editing (a subset of the grammar is shown in Table 1(a)) that includes key commands `Insert`, `Remove`, `Print` and `Replace`. Each of these commands relies on an `IterScope` expression that specifies the region (a set of lines, a set of words, or the entire Word document) that the text editing operation is on. The `SelectStr` production includes a `Token`, which allows for limited wild-card matching (e.g., an entire `WORD`, `NUMBER`, or a pattern specified by `PString`), a Boolean condition `BCond` that acts as an additional (local) filter on the matched value, and an `Occurrence` value that performs an index based selection from the resultant matches. Use of the occurrence values like `FirstFew`(N) (from `AtomicOccurrence`) when performing a `Remove` results in the removal of *only* the first N items (here N is a positive integer) that match the condition, while use of `ALL` will instead result in all matches of the condition being removed. The Boolean conditions `BCond` cover the standard range of string matching predicates (`Contains`, `StartsWith` etc.) and allow conjunction of conditions (`And`, `Not` etc.). The `CommonCond` production specifies the position relative to the string token(s) that occurs after it (`After`), before it (`Before`), or around it (`Between`) acts as another (global) filter.

EXAMPLE 1. *For text editing task 1 in Table 1(b), our system produces the following translation:*

```
Remove(SelectStr(WORD, ALWAYS, 1),
       IterScope(LINESCOPE, StartsWith(NUMBER), ALL))
```

EXAMPLE 2. *For text editing task 2 in Table 1(b), our system produces the following translation:*

```
Replace(SelectStr("&", Not(Between("[", "]")), ALL),
        By("&&"),
        DOCUMENT)
```

1. Consider the set of all binary strings where the difference between the number of "0" and the number of "1" is even.
2. The set of strings of "0" and "1" such that at least one of the last 10 positions is a "1".
3. the set of strings w such that the symbol at every odd position in w is "a".
4. Let L1 be the set of words w that contain an even number of "a", let L2 be the set of words w that end with "b", let L3 = L1 intersect L2.
5. The set of strings over alphabet 0 to 9 such that the final digit has not appeared before.

Table 2: Sample benchmarks for the Automata domain.

1. I would like the time of your earliest flight in the morning from Philadelphia to Washington on American Airlines.
2. I need information on a flight from San Francisco to Atlanta that would stop in Fort Worth.
3. What is the earliest flight from Washington to Atlanta leaving on Wednesday September fourth.
4. Okay we're going from Washington to Denver first class ticket I would like to know the cost of a first class ticket.
5. What ground transportation is there from the airport in Atlanta to downtown.

Table 3: Sample benchmarks for the ATIS domain.

Table 1(c) describes a sample of the variations that our system can handle for description of a task that is expressible in our DSL. Our belief is that once users are able to accomplish these types of smaller conditional and repetitive tasks, they can accomplish complex tasks by reducing them to a sequence of smaller tasks using end-user programming environments [20, 37, 55].

2.2 Automata Theory (Intelligent Tutoring)

Results from formal methods research have been used in many parts of intelligent tutoring systems [16] including problem generation, solution generation, and especially feedback generation for a variety of subject domains including geometry [19] and automata theory [1]. Each of these domains involves a specialized DSL that is used by a problem generator tool to create new problems, a solution generation tool to produce solutions, and more significantly, a feedback generation tool to provide feedback on student solutions.

Consider the domain of automata constructions, where students are asked to construct an automaton that accepts a language whose description is provided in English (For some examples, see Table 2). We designed a DSL based on the description provided by Alur et.al. [1] on constructs required to formally specify such languages. This DSL contains predicates over strings, Boolean connectives, functions that return positions of substrings, and universal/existential quantification over string positions. As stated in [1], such a language is used to generate feedback for students' incorrect attempts in two ways: (i) it is used by a solution generation tool to generate correct solutions against which a student's attempts are graded, (ii) it is also used to provide feedback and generate problem variations consistent with a student's attempt. This feedback generation tool has been deployed in the classroom and has been able to assign grades and generate feedback in a meaningful way while being both faster and more consistent than a human.

EXAMPLE 3. *Specification 1 in Table 2 is translated as:*

```
IsEven(Diff(OccurCount(0), OccurCount(1)))
```

EXAMPLE 4. *Specification 2 in Table 2 is translated as:*

```
Exists(LastK(10), StrEquals(SymbolAtPos(), 1))
```

2.3 Air Travel Information Systems (ATIS)

ATIS [8] is a standard benchmark for querying air travel information, consisting of English queries and an associated database containing flight information. It has long been used as a standard benchmark in both natural language processing and speech processing communities. Table 3 shows some sample queries from the ATIS suite. For ATIS, we designed a DSL that is based around SQL style row/column operations and provided support for predicates/-expressions that correspond to important concepts in air-travel queries, arrival/departure locations, times, dates, prices, etc.

EXAMPLE 5. *The first query in the ATIS examples, Table 3, is translated into our DSL as:*

```
ColSelect(DEP_TIME, RowMin(DEP_TIME,
   RowPred(EqDepart(PHILADELPHIA, Time(MORNING)),
           EqArrive(WASHINGTON, Time(ANY)),
           EqAirline(AMERICAN))))
```

3. PROBLEM DEFINITION

We study the problem of synthesizing NL-to-DSL synthesizers, given a DSL definition and training data. A DSL $L = (G, TC)$ consists of a context free grammar G (with terminal symbols denoted by G_T and production rules by G_R), and a type/semantic checker TC that can check if a given program is well-typed. The *training data* consists of a set of pairs (S, P), where S is an English sentence and P is the corresponding intended program from the DSL L. A sentence is simply a sequence of words $[w_1, w_2, \ldots, w_n]$. The goal of the generated NL-to-DSL synthesizer is to translate an English sentence to a ranked set of programs, $[P_1, P_2, \ldots, P_k]$, in L.

4. NL TO DSL SYNTHESIS ALGORITHM

Our synthesis algorithm (Alg. 1) takes a natural language command from the user and creates a ranked list of candidate DSL programs. The first step (loop on line 2) is to convert each of the words in the user input into one or more terminals (function names or values) using the NL to program terminal Dictionary *NLDict*. This loop ranges over the length of the input sentence and for each index looks up the set of terminals in the DSL that are associated with the word at that index in *NLDict*. Fundamentally, *NLDict* encodes, for each terminal, which English language words are likely to indicate the presence of that terminal in the desired result program. This map can be constructed in a semi-automated manner (§5.3). Once this association has been made for a terminal t we store a tuple of the terminal and a singleton map, relating the index of the word to a terminal that was produced, into the set R_0 (line 4).

For each natural language word, the dictionary *NLDict* associates a set of terminals with it. The terminals may be constant values or function applications with *holes* (□) as arguments. Thus, Alg. 1, when applying NLDict on line 3, can create incomplete programs, where some arguments to functions are missing. For example, Consider the sentence "Print all lines that do not contain 834". Since the grammar contains `PrintCmd:= Print(SelectStr, IterScope)` as a production and the dictionary relates the word "print" to the function `Print`, the partial program `Print(□, □)` will be generated. These holes are later replaced by other programs that match the argument types SelectStr and IterScope.

Once the base set of terminals has been constructed, the algorithm uses the *Bag* algorithm (Alg. 2) to generate the set of all *consistent* programs, Res_T, that can be constructed from it (line 5). The final step is to rank (§4.2) this set of programs, using a combination of scores and weights, in the loop on line 8.

4.1 Synthesizing Consistent Programs

A program P in the DSL is either an atomic value (i.e., a terminal in G), or a function/operator applied to a list of arguments. By convention we represent function application as *s-expressions* where a function F applied to k arguments is written $(F, P_1, \ldots, P_k)$.

Algorithm 1: Top-Level NL to DSL Synthesis Algorithm

Input: NL sentence S, Word-to-Terminal dictionary *NLDict*
Output: Ranked set of programs

$R_0 \leftarrow \emptyset$;
for $i \in [0, S.Length - 1]$ **do**
  $T \leftarrow NLDict(S[i])$;
  foreach $t \in T$ **do** $R_0 \leftarrow R_0 \cup (t, SingletonMap(i,t))$;
$Res_T \leftarrow Bag(S, R_0)$;
$Res_P \leftarrow \{P | \exists M \text{ s.t. } (P,M) \in Res_T\}$;
foreach *program* $P \in Res_P$ **do** $score(P) \leftarrow -\infty$;
foreach *program* $(P,M) \in Res_T$ **do**
  $s_{cov} \leftarrow CoverageScore(P,S,M) \times \omega_{cov}$;
  $s_{map} \leftarrow MappingScore(P,S,M) \times \omega_{map}$;
  $s_{str} \leftarrow StructureScore(P,S,M) \times \omega_{str}$;
  $score(P) \leftarrow \max(score(P), s_{cov} + s_{map} + s_{str})$;
return *set of programs in* Res_P *ordered by score*

Consistent Programs and Witness Maps.

Given a DSL $L = (G, TC)$, we say a program P in language L is consistent with a sentence S if there exists a map M that maps (some) word occurrences in S to terminals in G_T such that the range of M equals the set of terminals in the program P.[1] We refer to such a map M as a *witness* map, and use the notation $\texttt{WitnessMaps}(P,S)$ to denote the set of all such maps.

Usable and Used Words.

Let S be an English sentence, P be a program consistent with S, and M be any witness map. $\mathit{UsableWords}(S)$ are those word occurrences in S that are mapped to some grammar terminal and hence might be useful in translation. $\mathit{UsedWords}(S,M)$ is the set of usable word occurrences in S that are used as part of the map M.

$$\mathit{UsableWords}(S) = \{i \mid S[i] \in Domain(NLDict)\}$$
$$\mathit{UsedWords}(S,M) = \mathit{UsableWords}(S) \cap Domain(M)$$

Partial Programs.

A partial program extends the notion of a program to also allow for a *hole* ($\square$) as an argument. A hole is a symbolic placeholder where another complete program (program without any hole) can be placed to form a larger program. To avoid verbosity, we often refer to a *partial program* as simply a *program*.

Given a partial program $P = (F, \ldots, \square, \ldots)$ with a hole $\square$, we can substitute a complete program P' to fill the hole:

$$P[\square \leftarrow P'] = \begin{cases} (F, \ldots, P', \ldots) \text{ if } TC((F, \ldots, P', \ldots)) \\ \bot \qquad \text{otherwise} \end{cases}$$

The validity check, TC, ensures that all synthesized programs are well defined in terms of the DSL grammar and type system (otherwise we return the *invalid* program $\bot$).

Combination.

The combination operator *SubAll* generates the set of all valid programs that can be obtained by substituting a complete program P' in some hole of a partial program P. This is done by going over all the arguments of P and producing substitutions for argument positions with holes. Given partial program $P = (F, P_1, \ldots, P_k)$ and complete program P', we have:

$$SubAll(P,P') = \{P[\square_i \leftarrow P'] | P_i = \square_i\}$$

[1] Since the same English word can occur at different positions in S, having different meanings, any map M must take the position information also as an argument. We ignore this in the paper.

Algorithm 2: Bag Synthesis Sub-Algorithm

Input: NL sentence S, Initial Tuple Set B_0

1 $result \leftarrow B_0$;
2 **repeat**
3 $oldResult \leftarrow result$;
4 **foreach** $(P_1,M_1),(P_2,M_2) \in result$ **do**
5 $okpc \leftarrow P_1$ is partial $\wedge P_2$ is complete ;
6 $disjoint \leftarrow \mathit{UsedWords}(S,M_1) \cap \mathit{UsedWords}(S,M_2) = \emptyset$;
7 **if** $okpc \wedge disjoint$ **then**
8 $combs \leftarrow SubAll(P_1,P_2) - \{\bot\}$;
9 $new \leftarrow \{(P_r, M_1 \cup M_2) | P_r \in combs\}$;
10 $result \leftarrow result \cup new$;
11 **until** $oldResult = result$;
12 return $result$;

Bag Algorithm.

The *Bag* algorithm (Alg. 2) is based on computing the closure of a set of programs by enumerating all possible well-typed combinations of the programs in the set. The main loop (line 2) is a fixpoint iteration on the *result* set of programs that have been constructed.

The requirement that P_2 is a complete program (line 5) when applying the *SubAll* function ensures that the only holes in the result programs are holes that were originally in P_1. We restrict the initialization of B_0 to include only complete programs and partial programs with holes at the top level only. Using this restriction we can inductively show that at each step all partial programs only have holes at the top-level. Thus, we can efficiently compute the fixpoint of all possible programs in a bottom-up manner. The condition $\mathit{UsedWords}(S,M_1) \cap \mathit{UsedWords}(S,M_2) = \emptyset$ (line 6) ensures that the two programs do not use overlapping sets of words from the user input. This ensures that the final program cannot create multiple sub-programs with different meanings from the same part of the user input. This also ensures that the set of possible combinations has a finite bound based on the number of words in the input. Line 8 constructs the set of all possible substitutions of P_2 into holes in P_1 (ignoring any invalid results). For each of the possible substitutions we add the result (and the union of the M maps) to the *new* program set (line 9). Since the domains of the maps were disjoint the union operation is well-defined.

The *Bag* algorithm has a high recall, but, in practice, it may generate spurious programs that arise as a result of arbitrary rearrangement of the words in the English sentence. To fix this, the correct translation is reported by selecting the top-most rank program based on features of the program and the parse tree of the sentence.

4.2 Ranking Consistent Programs

We view the abstract syntax tree of the synthesized program as consisting of two important constituents: the set of terminals in the program, and the tree structure between those terminals. We use these constituents to compute the following three scores to determine the rank of a consistent program: (i) a *coverage score* that reflects how many words in the English sentence were mapped to some operation or value in the program, (ii) a *mapping score* that reflects the likelihood that a word-to-terminal mapping is capturing the user intent (iii) a *structure score* that captures the naturalness of the tree program structure and the connections between parts of the program and the parts of the sentence that generated them.

4.2.1 Coverage Score

For a given sentence S, a candidate translation P, and a witness map M, the coverage score is defined as,

$$CoverageScore(P,S,M) = \frac{|\mathit{UsedWords}(S,M)|}{|\mathit{UsableWords}(S)|}$$

The *CoverageScore*(P,S,M) denotes the fraction of available information in S that is actually used to generate P. Intuitively we want to prefer programs that make more use of the input information.

EXAMPLE 6. *Consider possible translations for an input S:*

S: `find the cheapest flight from Washington to Atlanta`

P_1:
```
RowMin(FARE, RowPred(EqDepart(WASHINGTON),
                     EqArrive(ATLANTA)))
```

P_2: `RowPred(EqDepart(WASHINGTON), EqArrive(ATLANTA)))`

The first program P_1 makes use of all parts of the user input, including the desired cheapest fare, while the second program P_2 ignores this information. The Coverage score ranks P_1 higher than P_2.

4.2.2 Mapping Score

For any word w there may be multiple terminals (functions or values) in the set *NLDict*(w) each of which corresponds to a different interpretation of w. We use machine learning techniques to obtain a classifier C_{map} based on the part-of-speech (POS) tag provided for the word by the Stanford NLP engine [26]. C_{map}.Predict function of the classifier predicts the probability of each word-to-terminal mapping being correct. We use predictions from C_{map} to compute the *MappingScore*, the likelihood that terminals in P are correct interpretation of corresponding words in S.

$$\mathit{MappingScore}(P,S,M) = \prod_{w \in \mathit{UsableWords}(S)} C_{map}.\text{Predict}(w, \text{POS}(w,S), M(w))$$

A limitation of the *MappingScore* score is that it looks only at the mapping of a word but not its relation to other words and how they are are mapped by the translation. Thus, interchanging a pair of terminals in a correct translation gives us an incorrect translation which has the same score.

EXAMPLE 7. *Consider an input S and 2 possible translations:*

S:
```
If "XYZ" is at the beginning of the line,
replace "XYZ" with "ABC"
```

P_1:
```
Replace(SelectStr("XYZ", ALWAYS, ALL),
    By("ABC"),
    IterScope(LINESCOPE, StartsWith("XYZ"), ALL))
```

P_2:
```
Replace(SelectStr("XYZ", ALWAYS, ALL)
    By("ABC"),
    IterScope(LINESCOPE, Before("XYZ"), ALL))
```

Both the programs use same sets words, so they have the same coverage score. The only difference is that the word "beginning" is mapped to StartsWith (POS: Verb Phrase) in P_1, and to Before (POS: Prepositional Phrase) in P_2. Mapping score helps in identifying P_1 as the correct choice.

4.2.3 Structure Score

Structure score captures the notion of naturalness in the placement of sub-programs. We use connection features obtained from the sentence S, the natural language parse tree for the sentence, NLParse(S), and the corresponding program P to define the overall structure score. These features are used to produce the classifier C_{str} which computes the probability that each of the combinations in P is correct.

DEFINITION 1 (CONNECTION). *For a production $R \in G_R$ of the form $N \to N_1 \ldots N_i \ldots N_j \ldots N_k$, the tuple (R,i,j) where $1 \le i,j \le k$, and $i \ne j$ is called a connection.*

DEFINITION 2 (COMBINATION). *Consider the program $P = (P_1, P_2, \ldots, P_k)$ generated using the production $R : N \to N_1 N_2 \ldots N_k$, such that N_i generates P_i for $1 \le i \le k$. We say the pair of sub-programs (P_i, P_j) is combined via connection (R,i,j) and this combination is denoted as Conn(P_i,P_j).*

The overall *StructureScore* is obtained by taking the geometric mean of the various connection probabilities of the scores for the program P—this normalizes the score to account for programs with differing numbers of connections.

$$\mathit{StructureScore}(P,S,M) = \text{GeometricMean}(\mathit{ConnProbs}(P,S,M))$$

$$\mathit{ConnProbs}(P,S,M) = \bigcup_{\mathit{Conn}(P_i,P_j) \text{ in } P} \{C_{str}[\mathit{Conn}].\text{Predict}(\vec{f},1)\}$$

where $\vec{f} \equiv \langle f_{pos1}, f_{pos2}, f_{lca1}, f_{lca2}, f_{order}, f_{over}, f_{dist} \rangle$ computed for P_i and P_j using P, S, and M.

We obtain separate classifier, C_{str}[Conn], for each connection Conn. The function C_{str}[Conn].Predict asks the classifier to predict the probability that f-vec belongs to class 1 (i.e., present in correct translation). The other class is 0.

Given a program P and input sentence S that are related by a witness map M and the parse tree NLParse(S), the following functions define several useful relationships:

$$\begin{aligned}
&\text{TreeCover}(P,S,M) = \text{minimal sub-tree } T_{\text{sub}} \text{ of NLParse}(S)\\
&\quad \text{s.t. } \mathit{UsedWords}(S,M) \subseteq \mathit{UsableWords}(T_{\text{sub}})\\
&\text{Root}(P,S,M) = \text{root node of TreeCover}(P,S,M)\\
&\text{Span}(P,M) = [\text{Min}(\mathit{Domain}(M)), \text{Max}(\mathit{Domain}(M))]
\end{aligned}$$

In the rest of the section, we assume that P_1 and P_2 denote two sub-programs of P. The following features determine the naturalness of the connections between P, P_1, P_2, and S :

DEFINITION 3 (ROOT POS TAGS). *Part-of-speech features are the POS tags assigned by the NL Parser to the root nodes of the sub-trees associated with P_1 and P_2 respectively:*

$$f_{\text{pos1}} \equiv \mathit{POS}(\mathit{Root}(P_1,S,M)) \qquad f_{\text{pos2}} \equiv \mathit{POS}(\mathit{Root}(P_2,S,M))$$

The features f_{pos1} and f_{pos2} help to learn the phrases that are commonly combined using a particular connection.

DEFINITION 4 (LCA DISTANCES). *Let LCA be the least common ancestor of Root(P_1,S,M) and Root(P_2,S,M). The LCA distance features are the tree-distances from LCA to the root nodes of the sub-trees associated with P_1 and P_2 respectively:*

$$\begin{aligned}
f_{\text{lca1}} &\equiv \mathit{TreeDistance}(\mathit{LCA}, \mathit{Root}(P_1,S,M))\\
f_{\text{lca2}} &\equiv \mathit{TreeDistance}(\mathit{LCA}, \mathit{Root}(P_2,S,M))
\end{aligned}$$

DEFINITION 5 (ORDER). *The order feature is determined by the positions of the roots of the sub-tree roots associated with P_1 and P_2 in the* in-order *traversal of NLParse(S).*

$$f_{\text{order}} = \begin{cases} 1 & \textit{if Root}(P_1,S,M) \textit{ occurs before Root}(P_2,S,M) \\ & \quad \textit{in in-order traversal of NLParse(S)} \\ -1 & \textit{otherwise} \end{cases}$$

Features f_{lca1}, f_{lca2} and f_{order} are used to learn the correspondence between the parse tree structure and the program structure. We use these to maintain the structure of translation close to the structure of the parse tree.

DEFINITION 6 (OVERLAP). *The overlap feature captures the possibility that two programs are constructed from mixtures of two subtrees in the NL Parse tree:*

$$f_{\text{over}} = \begin{cases} 1 & \text{if } Span(P_1,M_1).end < Span(P_2,M_2).start \\ -1 & \text{if } Span(P_1,M_1).start > Span(P_2,M_2).end \\ 0 & \text{otherwise} \end{cases}$$

DEFINITION 7 (DISTANCE). *Given two programs P_1 and P_2 we define the distance feature for programs by looking at the distance between the word spans used in the programs:*

$$f_{\text{dist}} = \begin{cases} Span(P_2,M_2).start - Span(P_1,M_1).end & \text{if } f_{\text{over}} = 1 \\ Span(P_1,M_1).start - Span(P_2,M_2).end & \text{if } f_{\text{over}} = -1 \\ 0 & \text{otherwise} \end{cases}$$

The features f_{over} and f_{dist} capture the proximity information of words and are useful because related words often occur together in the input sentence.

EXAMPLE 8. *Consider possible translations for an input S:*

S: `Print all lines that do not contain "834"`

P_1: `Print(SelectStr(LINE, Not(Contains("834")), ALL), DOCUMENT)`

P_2: `Print(SelectStr("834", Not(Contains(LINE)), ALL), DOCUMENT)`

In the parse tree NLParse(S), "print" will have two arguments, what to print ("lines") and when to print("not contain 834*"). We observe the following for the candidate programs: (a) The word "lines" is closer to "print", while the word "834" is farther in NLParse(S). The same structure is observed for P_1, but not for P_2. This is captured by LCA Distances. (b) The order of the words in NLParse(S) matches the order in P_1 better than in P_2. This is captured by the Order feature. (c) The phrase "do not contain 834" is kept intact in P_1, but is split apart in P_2. Overlap and Distance features will capture this splitting and reordering.*

Both the programs use the same set of words, and the same word to terminal mappings, resulting in the same coverage score and the same mapping scores. However, the program P_1 is correct and our choice of features rank it higher.

4.3 Combined Score Example

To provide some intuition into the complementary strengths and weaknesses of the various scores, we examine how they behave on a subset of the programs generated by the *Bag* algorithm for the following text editing task: `Add a "*" at the beginning of the line in which the string "P.O. BOX" occurs.` Table 4 shows some of the consistent programs generated by the *Bag* algorithm. The first program (P_1) is the intended translation. Let us look at the performance of each of the component scores:

Coverage Score: Both P_1 and P_4 use the maximum number of words from the sentence, and are tied on top score. P_4 is wrong as it adds "P. O. BOX" at the beginning of the line containing "*".

Mapping Score: The classifier learnt by our system maps the word "beginning" to the terminal StartsWith with a high probability but to the terminal START with a lower probability. Further, it maps "occurs" to the terminal Contains with a still lower probability. P_2 does not use the word "occur", otherwise it has same mappings as P_1. As a result it has higher mapping score than P_1, but suffers on coverage. P_3 maps "beginning" to StartsWith, and does not use the word "occurs". As a result it has a mapping score lower than P_2 but higher than P_1. If we had used the mapping score alone, we would not have been able to rank the desired program P_1 above the incorrect programs P_3 and P_4.

Algorithm 3: Learning Mapping Score Classifier C_{map}

Input: Training Data $\mathcal{T}$
foreach *training pair* $(S,P) \in \mathcal{T}$ **do**
  $\tilde{M} \leftarrow$ `WitnessMaps`(P,S);
  $M \leftarrow argmax_{M' \in \tilde{M}}(\mathit{Likeability}(P,S,M'))$;
  foreach $(w,t) \in M$ **do**
    C_{map}.Train(w,POS(w,S),t)
return C_{map};

Structure Score: Coverage score and mapping score look only at the mapping of a word but not its relation to other mappings and their placement with respect to the original sentence. Structure score fixes this by considering structural information (parse tree, ordering of words and distance among words) from the sentence. P_4 has poor structure score because it swaps the sentence ordering for strings "*" and "P.O. BOX". P_3 also suffers as it moves "beginning" (mapped to StartsWith) away from "Add" (mapped to Insert). P_1 gets a high structure score as it maintains the parse tree structure of the input text. Note that, P_2 and P_5 have high structure score as well. This is because structure score does not take into account the fraction of used words or word-to-terminal mappings. So, an incomplete translation that uses very few words but maps them to correct terminals and places them correctly, is likely to have a high value.

The desired program, P_1, is only top ranked by one of the scores and even in that case, the score is tied with another incorrect result. However, a combination of the scores with appropriate weights (§5) ranks P_1 as the clear winner!

5. TRAINING PHASE

This section describes the learning of classifiers, weights, and the word-to-terminal mapping used by the synthesis algorithm described in §4. The key aspects in this process are (i) deciding which machine learning algorithm to use, and (ii) generation of (lower level) training data for that machine learning algorithm from the top level training data provided by the DSL designer.

5.1 Mapping Score Classifier (C_{map})

The goal of the C_{map} classifier is to predict the likelihood of a word w mapping to a terminal $t \in G_T$ using the POS tag of the word w. The learning of this classifier is performed using an off-the-shelf implementation of a Naive Bayesian Classifier [6]. The training data for this classifier is generated as shown in Alg. 3.

The key idea is to first construct the set $\tilde{M}$ of all witness maps that can yield program P from natural language input S. We then select the most likely map M out of these witness maps based on the partial lexicographic order given by the *likeability score* tuples.

$$\mathit{Likeability}(P,S,M) = (\mathit{UsedWords}(S,M), \mathit{Disjointness}(P,S,M))$$
$$\mathit{Disjointness}(P,S,M) = \sum_{P' \in \mathit{SubProgs}(P)} \sigma(P')$$

where $\sigma((P_1,\ldots,P_n))$=1 if $\forall P_i,P_j, P_i \cap P_j = \emptyset$, 0 otherwise

The likeability tuples serve two purposes: First, via the *UsedWords*, they guide the system to prefer mappings that use all parts of the input sentence. Second, via the *Disjointness*, they guide the system to prefer mappings that penalize the use of a single part of a sentence to construct multiple different subprograms.

5.2 Structure Score Classifiers (C_{str})

We now describe how the classifiers used in structure score, $C_{str}[Conn]$ for each connection *Conn*, are learned. The goal of each

	Program Generated	Coverage Score	Mapping Score	Structure Score	Final Score
P_1	Insert("*", START, IterScope(LINESCOPE, Contains("P.O. BOX"), ALL))	8.33	5.73	4.45	322.17
P_2	Insert("*", START, IterScope(LINESCOPE, ALWAYS, ALL))	5.00	8.40	4.45	248.17
P_3	Insert("*", START, IterScope(LINESCOPE, StartsWith("P.O. BOX"), ALL))	6.67	6.35	1.09	232.57
P_4	Insert("P.O. BOX", START, IterScope(LINESCOPE, Contains("*"), ALL))	8.33	5.73	1.00	272.43
P_5	Insert("*", START, DOCUMENT)	3.33	4.74	6.84	216.33

Table 4: Ranking the set of consistent programs generated by the *Bag* algorithm.

Algorithm 4: Learning Structure Score Classifiers C_{str}

Input: Training Data $\mathcal{T}$
foreach *training pair* $(S,P) \in \mathcal{T}$ **do**
  $AllOpts = \text{SynthNoScore}(S)$;
  foreach *program* $P' \in AllOpts$ **do**
    foreach *combination c that occurs in* P' **do**
      if *c occurs in P* **then** class $\leftarrow 1$;
      else class $\leftarrow 0$;
      $Conn \leftarrow$ connection used by c;
      $\vec{f} \leftarrow \langle f_{pos1}, f_{pos2}, f_{lca1}, f_{lca2}, f_{order}, f_{over}, f_{dist} \rangle$;
      $C_{str}[Conn]$.Train($\vec{f}$, class);
return C_{str}

classifier $C_{str}[Conn]$ is to predict the likelihood that a combination c is an instance of connection *Conn* using the 7 features of c from §4.2. We use an off-the-shelf implementation of a Naive Bayesian Classifier and generate the training data for it as shown in Alg. 4.

The key idea is to run the synthesis algorithm without the scoring step, *SynthNoScore*, to construct the set of all possible programs, *AllOpts*, from the English sentence S. Any combination present in a program in P' in *AllOpts* but not present in P is used as a negative example, while that present in P is used as a positive example.

5.3 Dictionary Construction

We construct the dictionary *NLDict* in a semi-automated manner using the names of the terminals (functions and arguments) in the DSL. If the name of an operation is a proper English word, such as `Insert`, we use the *WordNet* [38] synonym list to gather commonly used words which are associated with the action. Cases where the name is not a simple word but instead concatenations of (or abbreviations of) several words, such as `StartsWith`, are handled by splitting the name and resolving the synonyms of each sub-component word.

It is possible that the general purpose synonym sets provided by WordNet contain English words that are not useful for the particular domain we are constructing the translator for. However, the mapping score learning in §5.1 will simply assign these words low scores. Once the learning algorithm for the mappings has finished assigning weights to each word/terminal we discard all mappings below a certain threshold. Conversely, it is also possible that an important domain specific synonym will not be provided by the WordNet sets or that the names in the DSL are not well matched with proper English words. Our system automatically detects these cases as a result of being unable to find witness maps for programs (in the training data) involving certain DSL terminals. In these cases, it prompts the user to identify a word in an input sentence that corresponds to an unmapped terminal in a program. These new seed words are then further used to build a more extensive synonym set using WordNet.

5.4 Learning Combination Weights

In the previous section, we defined 3 component scores for a translation. A standard mechanism for combining multiple scores into a single final score is to use a weighted sum of the component scores. In this section we describe a novel method for learning the required weights to use to maximize the following function.

Optimization Function: *Number of benchmarks in the training set, for which the correct translation is assigned rank 1.*

In numerical optimization, maximization of an optimization function is a standard problem which can be solved using *stochastic gradient descent* [5]. In order to use gradient descent to find the weight values that maximize our optimization function we need to define a continuous and differentiable *loss function*, F_{loss}. This loss function is used to guide the iterative search for a set of weights that maximizes the value of the optimization function as follows:

$$\vec{w_{n+1}} = \vec{w_n} - \gamma \vec{\nabla} F_{loss}(\vec{w_n}) \quad n = 0, 1, 2, ..$$

where $\vec{\nabla}$ denotes the gradient and γ is a positive constant. At each step, $\vec{w}$ moves in the direction in which the value of F_{loss} decreases and the process is stopped when the change in the function value in successive steps drops below a specified threshold ε.

A common form for loss functions is a sigmoid. We can convert our ill-behaved optimization function into a loss function that is closer to what is needed to perform gradient descent by basing the sigmoid on the ratio score given to the best incorrect result and the score given to the desired rate via the following construction:

$$F_{loss}(\vec{w}) = \sum_{\forall \text{ training } S} f(\vec{w}, S)$$

$$f(\vec{w}, S) = \frac{1}{1 + e^{-c(\lambda - 1)}} \text{ where } \lambda = \frac{v_{wrong}}{\text{Score}(P_{desired})} \wedge c > 0$$

$$P_{desired} = \textit{correct translation of } S$$

$$v_{wrong} = max(\{\text{Score}(P) | P \in Bag(S) \ \wedge \ P \neq P_{desired}\})$$

Although the above transformation results in a loss function which is mostly well behaved, it saturates appropriately and is piecewise continuous and differentiable, there are still points were the function is not continuous. In particular the presence of the *max* function in the definition of v_{wrong} creates discontinuous points in F_{loss}. However, the following insight enables us to replace the discontinuous *max* operation with a continuous approximation:

$$\max(a, b) \approx \log(e^{ca} + e^{cb})/c \text{ where } c \geq 1 \text{ if } a \ll b \vee b \ll a$$

Thus, we can replace the *max* operator with this function, extended in the natural way to k arguments, in the computation of v_{wrong} to produce a globally continuous and differentiable loss function. The cases where there are several incorrect results which are given very similar scores are minimized by the selection of a large value for c, which amplifies small differences. Additionally, in the worst case where two scores are extremely close, the impact of the approximation is to drive the gradient descent to increase the ratio between v_{wrong} and $\text{Score}(P_{desired})$. Thus, the correctness of the gradient descent algorithm is not impacted.

In addition to satisfying the basic requirements for performing gradient descent, our loss function, F_{loss}, saturates for large values of λ. This implies that if an input S, has $\frac{v_{wrong}}{\text{Score}(P_{desired})} \gg 1$ it will

not dominate the gradient descent causing it to improve the ranking results for a single benchmark at the expense of rank quality on a large number of other benchmarks. The saturation also implies that the descent will not become stuck trying to find weights for an input where there is no assignment to the weights that will improve the ranking, i.e., there is an incorrect result program P_i where every component score has a higher value than the desired program P_d.

6. EXPERIMENTAL EVALUATION

The (online) synthesis algorithm, consisting of the *Bag* algorithm and feature extraction (for ranking), was implemented in C# and used the Stanford NLP Engine (Version 2.0.2) [53] for POS tagging and extracting other NL features. The offline gradient decent was implemented in C# while the classifiers used for training the component features were built using MATLAB.

A major goal of this research is the production of a generic framework for synthesizing programs in a given DSL from English sentences. Thus, we selected 3 different categories of tasks, question answering (ATIS), constraint based model construction (Automata Theory Tutoring), and command execution on unstructured data (Repetitive Text Editing). These domains, described in detail in §2, present a variety of structure in the underlying DSL, the language idioms that are used, and the complexity of the English sentences that are seen. For benchmarks, automata descriptions are taken verbatim from textbooks and online assignments. Text editing descriptions are taken verbatim from help forums and user studies. ATIS descriptions are part of a standard suite. Tables 1(b), 1(c), 2, and 3 describe a sample of the benchmarks. The details of the benchmarks and their sources can be obtained from companion website: https://sites.google.com/site/nl2pgm/ [9].

Air Travel Information System (ATIS).

We selected 535 queries at random from the full ATIS suite (which consists of few thousand queries) and, by hand, constructed the corresponding program in our DSL to realize the query. Each task in ATIS domain is a query over flight related information.

Automata Theory Tutoring.

We collected 245 natural language specifications (accepting conditions) of finite state automata from books and online courses on automata theory.

Repetitive Text Editing.

We collected a description of 21 text editing tasks from Excel books and help forums. We collected 265 English descriptions for these 21 tasks via a user study, which involved 25 participants (who were first and second year undergraduate students). The large number of participants ensured variety in the English descriptions (e.g., see Table 1(c)). In order to remove any description bias, each of these tasks was described not using English but using representative pairs of input and output examples. Additionally, we obtained 227 English descriptions for 227 text editing tasks (one for each task) from an independent corpus [36].

6.1 Precision, Recall, and Computational Cost

In this study we used standard *10-fold cross-validation* to evaluate the precision and recall of the translators on each of the domains. Thus, we select 90% of the data at random to use for learning the classifiers/weights and then evaluate the system on the remaining 10% of data which was held back (and not seen during training). In the ranking we handle ties in the scores assigned to an element using a *1334 ranking* scheme [46]. In 1334 ranking, in the case of tied scores, each element in the tied group is assigned a rank corresponding to the lowest position in the ordered result list (as opposed to the highest). This ensures that the reported results represent the *worst case* number of items that may appear in a ranked list before the desired program is found.

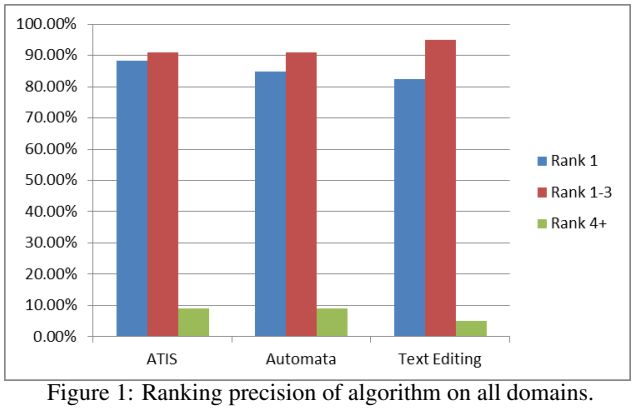

Figure 1: Ranking precision of algorithm on all domains.

Precision.

Fig. 1 shows the percentage of inputs for which the desired program in the DSL is the top ranked result and the percentage of inputs where the desired result is in the first three results. As shown in the figure, for every domain, on over 80% of the inputs the desired program is unambiguously identified as the top ranked result. Further, for the ATIS domain the desired result is the top ranked result for 88.4% of the natural language inputs. Given the size of our sample from the full ATIS suite we can infer that the desired program will be the top ranked result for $88.4 \pm 4.2\%$ of the natural language inputs at a 95% confidence interval. These results show that our novel *program synthesis* based translation approach is competitive with the state-of-the-art *natural language processing* systems: 85% in [57], 84% in [42], and 83% in [27].

Recall.

In addition to consistently producing the desired program as the top ranked result for most inputs the ranking algorithm places the desired program in the top 3 results an additional 5%-12% of the time. Thus, across all three of the domains, for over 90% of the natural language inputs the desired program is one of the three top ranked results. This leaves less than 10% of the inputs for any of the domains, and only 5% in the case of the text editing domain, where the synthesizer was unable to produce and place the desired program in the top three spots.

Computational Cost.

Fig. 2 shows the distribution of the time required to run the synthesis algorithm and perform the ranking on a 2.80 GHz Intel(R) Core(TM) i7 CPU with 8 GB RAM. On average translation takes 0.68 seconds for Text Editing, 1.72 seconds for Automata and 1.38 seconds for the ATIS inputs. Further, the distribution of times is heavily skewed with more than 85% of the inputs taking under 1 second and very few taking more than 3 seconds. The outliers tend to be inputs in which the user has specified an action in an exceptionally redundant manner.

As with any program synthesis technique which fundamentally involve search over exponential spaces, the cost of our technique is also worst case exponential in the size of the DSL. However, the key issue is doing this efficiently for practical cases. Our synthesis works efficiently (usually under 1 second) for a range of useful DSLs. The size of the dictionary has minimal impact on the run-

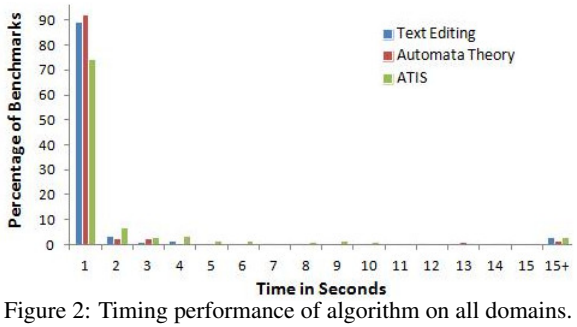

Figure 2: Timing performance of algorithm on all domains.

Domain	1334 Top Rank		
	CoverageScore	*MappingScore*	*StructureScore*
ATIS	5.6%	1.9%	20.2%
Automata	17.9%	31.0%	51.4%
Text Editing	8.1%	8.9%	34.4%

Table 5: Performance of individual component scores in ranking the desired program as top result.

time as the translation only depends on the subset of the dictionary corresponding to the words in the input sentence.

6.2 Individual Component Evaluation

In §4 we defined various components for ranking and provided intuition into their usefulness. To validate that these component scores are important to achieving good results, we evaluated our choices by using various subsets of the component scores, learning the best weights for the subset, and re-ranking the programs.

Performance of Individual Scores.

The results of using each component in isolation are presented in Table 5. This table shows that when identifying the top-ranked program the best performance for using only *CoverageScore* is 17.9%, using *MappingScore* is 31.0% and for *StructureScore* is 51.4%. This is far worse than the result obtained by using the combined ranking which placed the desired program as the top result for 84.9% of the inputs. Thus, we conclude that the components are not individually sufficient.

Score Independence.

Although these results show that independently none of the components are sufficient for the program ranking it may be the case that one of the components is, effectively, a combination of the other two. Table 6 shows the results of ranking the programs when dropping one of the components. Dropping *StructureScore* results in the largest decrease, as high as 81.86% in the worst case, and even the best case has a decrease of 47.75%. Dropping *CoverageScore* also results in substantial degradation, although not as high as for *StructureScore*. The impact of dropping *MappingScore* is much smaller, between 2.04% and 4.67%. However, the consistent positive contribution of *MappingScore* shows that it still provides useful information for the ranking. Thus, all of the components provide distinct and useful information.

Dictionary Construction.

In practice the semi-automated approach makes dictionary construction a task that, while usually requiring manual assistance, does not require expertise in natural language processing or program synthesis. On average the dictionaries for each domain contained 144 English words averaging 4.51 words/terminal and 1.48 terminals/word. The user was prompted to provide 20.7% mappings on average across the three DSLs. Although beyond the scope of this work, as it requires a larger corpus of training data, the amount of user intervention can be further reduced by using statistical alignment to automatically extract the domain specific synonyms from the training data.

Domain	% change on dropping		
	CoverageScore	*MappingScore*	*StructureScore*
ATIS	-30.6%	-4.7%	-81.9%
Automata	-22.4%	-2.0%	-47.7%
Text Editing	-19.7%	-4.7%	-65.8%

Table 6: Impact of dropping individual component scores on top rank percentage.

Domain	Total Count	Equal Wt.		Rank Boost		Gradient	
		Top	Top-3	Top	Top-3	Top	Top-3
ATIS	535	73.2%	90.8 %	79.8%	89.9 %	88.4%	93.3 %
Automata	245	74.2%	91.4 %	73.1%	93.5 %	84.9%	91.2 %
Text Editing	492	74.0%	91.1 %	74.4%	91.8 %	82.3%	94.9 %

Table 7: Comparison of ranking using equal weights, gradient descent, and RankBoost. Top (Top-3) shows the percentage of benchmarks where the correct translation is ranked 1 (in top 3).

Score Combination Weights.

We used gradient descent to learn how much to weight each score in the computation of the final rank of a program. To evaluate the quality of the weights identified via the gradient descent we compared them with a naive selection of equal weights for all the component scores and with the results of boosting. Boosting [10] is a frequently technique which combines a set of weaker rankings, such as the individual component scores, to produce a single strong ranking. Table 7 shows the results of the rankings obtained with the three approaches.

Using gradient descent has improved the number of top ranked benchmarks significantly over the naive weight selection (as large as 15%). However, the improvement in the top 3 ranked benchmarks is much smaller. Similarly, the gradient descent approach produces substantially better results than RankBoost with an average difference of 9% in the top ranked benchmarks. Thus, we can conclude that the use of gradient descent for learning the combination weights is important for the overall quality of the results.

Our choice of the ranking functions is critical to the quality of results. As shown in Table 6, dropping any of the component functions results in a substantial loss of precision. Also, using a simpler method, such as equal weights or boosting [10], to compute the combination weights results in a loss of 9-15% in precision when compared to the use of gradient descent (Table 7).

In our system, most failures (i.e. the correct solution failing to rank in the top-three solutions) arise because some key information is left implicit in the English description, e.g. “I want to fly to Chicago on August 15”. In this case, the departing city should default to “CURRENT_CITY” , and the time should default to “ANY” . Such issues might be fixed either by having orders of magnitude larger training data or by building some specialized support for handling implicit contextual information in various domains.

As part of learning the weights for the component scores we used a shifted variant of the logistic function as our loss function (§5). Fig. 3 shows how the value of loss changes with iteration index and the corresponding number of top ranked benchmarks. It can be seen that as the loss value decreases, the number of top ranked benchmarks increases and vice-a-versa. Thus, as these values are negatively correlated as needed for optimal performance of the gradient descent algorithm, and even though our loss function contains

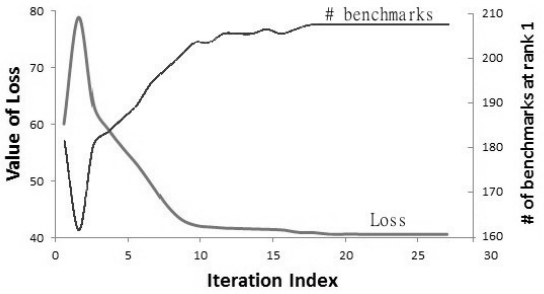

Figure 3: Behavior of Loss Function and Top Ranks.

Domain	% change on using weights learnt for ATIS	Automata	Text Editing
ATIS	0.0%	-1.5%	-5.0%
Automata	-2.0%	0.0%	-1.2%
Text Editing	-0.6%	-0.8%	0.0%

Table 8: Generalization of learning across domains.

the log-exponential approximation of the *max* operation it is well behaved for the gradient descent algorithm.

Since the weights learned in §4.2 are general purpose, we expect that weights learned from one domain are applicable to other domains, eliminating the the time and effort required to re-learn these values on each new domain. The results in Table 8 show that the weight vectors that are learned for one domain perform well when used to rank the results for a new domain. The average decrease in the number of top ranked programs is only 1.9% (with a maximum decrease of 5.0%). For the number of top-3 ranked programs the change is insignificant with a maximum decrease of less than 0.5% and thus we do not include them here. This result demonstrates that the learning of the component weights is highly domain independent and generalizes well, allowing it to be reused (or used as a starting point) for new domains.

7. RELATED WORK

Programming by demonstration (PBD) systems, which use a trace of a task performed by a user, and programming by example (PBE) systems, which learn from a set of input-output examples, have been used to enable end-user programming for a variety of domains. For PBD these domains include text manipulation [28] and table transformations [23] among others [7]. Recent work on PBE by Gulwani et. al. has included domains for manipulating strings [13, 48], numbers [49], and tables [21]. As mentioned earlier, both PBD and PBE based techniques struggle when the desired transformations involve conditional operations. In contrast the natural language based approach in this work performs well for both simple and conditional operations.

Keyword programming refers to the process of translating a set or sequence of keywords into function calls over some API. This API may consist either of operations in an existing programming language [34, 41, 54] or a DSL constructed for a specific class of tasks [32, 33]. These techniques use various program synthesis approaches to build expression trees from the elements of the underlying API, similar to the *Bag* algorithm in §4, and then use simple heuristics, such as words used and keyword-to-terminal weights, to rank the resulting expression trees. These systems have low precision when used on inputs with complex intents, 50%-60% accuracy in [34], and will frequently suggest incorrect programs. Conversely, due to the ranking methodology, the synthesizer in this paper is able to maintain high accuracy even on complex domains such as ATIS (where we achieve 88% accuracy).

Semantic parsing [39] is a sophisticated means of constructing a program from natural language using a specialized language parser. Several approaches including, syntax directed [25], NLP parse trees [11], SVM driven [24], combinatory categorical grammars [27, 56, 57], and dependency-based semantics [30, 42] have been proposed. These systems have high precision, usually suggesting the correct program, but recall rates below 85% [57] in ATIS. In contrast the technique in this paper achieves similar or higher levels of precision while providing a recall rate near 92%.

A number of natural language programming systems have been built around grammars, NLC [4], or templates, NaturalJava [45], which impose various constraints on input expressions. Such systems are sensitive to grammatical errors or extraneous words. There has been extensive research on developing natural language interfaces to databases (NLIDB) [2, 40]. While early systems were based on pattern matching the user's input to one of the known patterns, PRECISE [43, 44] translates *semantically tractable* NL questions into corresponding SQL queries. These systems depend heavily on the underlying data having a known schema which makes them impractical when the underlying data structure is unknown (or non-existent) as in the text-editing domain used in this work.

SmartSynth [29] is a system for synthesizing smartphone scripts from NL. The synthesis technique in SmartSynth is highly specialized to the underlying smartphone domain and uses a simple the ranking strategy for the programs that it produces. Similarly, the NLyze [20] system synthesizes spreadsheet formulas from NL. Again, NLyze is designed for a specific domain (spreadsheet formula) and uses a relatively simple ranking system consisting of only the equivalent of the coverage, mapping, and overlap features presented in our paper. In contrast, our technique is agnostic to the specifics of the target DSL, the ranking features are independent of the underlying DSL, and we automatically learn appropriate weights for the features. In addition, as the experimental results in Table 6 demonstrate, the use of the ranking methodologies described in SmartSynth or NLyze results in substantial reductions in recall/precision. Thus, the ranking methodology in this paper presents an improvement on those use in NLyze/SmartSynth and an opportunity to further improve their performance by integrating the advancements.

The work in [47] leverages natural language to enable compositional PBE (programming by examples). It does not apply any natural language learning techniques used in PBNL approaches, but only utilizes the natural language decomposition into phrases followed by asking the user to provide example based interpretation of those phrases.

8. CONCLUSION

This paper develops a meta approach for synthesizing programs from natural language descriptions that can be instantiated for a range of interesting DSL's including text-processing, automata construction, and information retrieval queries. Our approach takes three inputs from the synthesis designer, a suitable DSL definition, a basic training data set, and assistance in construction of a words-to-token dictionary, and from these inputs constructs a corresponding high-precision and high-recall NL-to-DSL synthesizer.

We aim to further generalize the framework to allow construction of synthesizers for a wider variety of domains. Another area of investigation is the addition of context-awareness when translating single-line intents (like ours) to provide interactive programming environment for programming in larger DSLs and accomplishing more complex tasks.

References

[1] R. Alur, L. D'Antoni, S. Gulwani, D. Kini, and M. Viswanathan. Automated grading of DFA constructions. In *IJCAI*, 2013.

[2] I. Androutsopoulos, G. Ritchie, and P. Thanisch. Natural language interfaces to databases - an introduction. *CoRR*, 1995.

[3] D. Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, 1987.

[4] B. W. Ballard and A. W. Biermann. Programming in natural language: "NLC" as a prototype. In *Annual conference*, ACM, 1979.

[5] D. P. Bertsekas. *Nonlinear Programming: 2nd Edition*. Athena Press, 2004.

[6] C. M. Bishop and N. M. Nasrabadi. *Pattern recognition and machine learning*, volume 1. springer New York, 2006.

[7] A. Cypher. *Watch What I Do: Programming by Demonstration*. MIT Press, 1993.

[8] D. A. Dahl, M. Bates, M. Brown, W. Fisher, K. Hunicke-Smith, D. Pallett, C. Pao, A. Rudnicky, and E. Shriberg. Expanding the scope of the atis task: The atis-3 corpus. In *HLT*, 1994.

[9] A. Desai, S. Gulwani, V. Hingorani, N. Jain, A. Karkare, M. Marron, R. Sailesh, and S. Roy. Benchmarks for program synthesis using natural language, January 2016. https://sites.google.com/site/nl2pgm/.

[10] Y. Freund, R. Iyer, R. E. Schapire, and Y. Singer. An efficient boosting algorithm for combining preferences. *J. Mach. Learn. Res.*, 4, Dec. 2003.

[11] R. Ge and R. J. Mooney. A statistical semantic parser that integrates syntax and semantics. In *CoNLL*, 2005.

[12] S. Gulwani. Dimensions in program synthesis. In *PPDP*, 2010.

[13] S. Gulwani. Automating string processing in spreadsheets using input-output examples. In *POPL*, 2011.

[14] S. Gulwani. Synthesis from examples: Interaction models and algorithms. In *SYNASC*, 2012.

[15] S. Gulwani. Flash Fill: Excel 2013 Feature , 2013. http://research.microsoft.com/en-us/um/people/sumitg/flashfill.html.

[16] S. Gulwani. Example-based learning in computer-aided stem education. In *CACM*, 2014.

[17] S. Gulwani, W. Harris, and R. Singh. Spreadsheet data manipulation using examples. *CACM*, 2012.

[18] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan. Synthesis of loop-free programs. In *PLDI*, 2011.

[19] S. Gulwani, V. A. Korthikanti, and A. Tiwari. Synthesizing geometry constructions. In *PLDI*, 2011.

[20] S. Gulwani and M. Marron. NLyze: Interactive programming by natural language for spreadsheet data analysis and manipulation. In *SIGMOD*, 2014.

[21] W. R. Harris and S. Gulwani. Spreadsheet table transformations from examples. In *PLDI*, 2011.

[22] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu. On the naturalness of software. In *ICSE*, 2012.

[23] S. Kandel, A. Paepcke, J. Hellerstein, and J. Heer. Wrangler: Interactive visual specification of data transformation scripts. In *CHI*, 2011.

[24] R. J. Kate and R. J. Mooney. Using string-kernels for learning semantic parsers. In *ACL*, 2006.

[25] R. J. Kate, Y. W. Wong, and R. J. Mooney. Learning to transform natural to formal languages. In *AAAI*, 2005.

[26] D. Klein and C. Manning. Accurate unlexicalized parsing. In *ACL*, 2003.

[27] T. Kwiatkowski, L. Zettlemoyer, S. Goldwater, and M. Steedman. Lexical generalization in CCG grammar induction for semantic parsing. In *EMNLP*, 2011.

[28] T. Lau, S. Wolfman, P. Domingos, and D. Weld. Programming by demonstration using version space algebra. *Machine Learning*, 53(1-2), 2003.

[29] V. Le, S. Gulwani, and Z. Su. Smartsynth: Synthesizing smartphone automation scripts from natural language. In *MobiSys*, 2013.

[30] P. Liang, M. I. Jordan, and D. Klein. Learning dependency-based compositional semantics. In *ACL*, 2011.

[31] H. Lieberman. *Your Wish Is My Command: Programming by Example*. Morgan Kaufmann, 2001.

[32] G. Little, T. A. Lau, A. Cypher, J. Lin, E. M. Haber, and E. Kandogan. Koala: Capture, share, automate, personalize business processes on the web. In *CHI*, 2007.

[33] G. Little and R. C. Miller. Translating keyword commands into executable code. In *UIST*, 2006.

[34] G. Little and R. C. Miller. Keyword programming in Java. *Autom. Softw. Eng.*, 16(1):37–71, 2009.

[35] Z. Manna and R. J. Waldinger. A deductive approach to program synthesis. *ACM TOPLAS*, 2(1), 1980.

[36] M. Manshadi, J. F. Allen, and M. D. Swift. A corpus of scope-disambiguated english text. In *ACL (Short Papers)*, 2011.

[37] M. Mayer, G. Soares, M. Grechkin, V. Le, M. Marron, O. Polozov, R. Singh, B. G. Zorn, and S. Gulwani. User interaction models for disambiguation in programming by example. In *UIST*, 2015.

[38] G. A. Miller. Wordnet: A lexical database for english. *CACM*, 2012.

[39] R. J. Mooney. Learning for semantic parsing. In *CICLing*, 2007.

[40] N. Nihalani, S. Silakari, and M. Motwani. Natural language interfce for database: A brief review. *IJCSI*, 8(2), 2011.

[41] D. Perelman, S. Gulwani, T. Ball, and D. Grossman. Type-directed completion of partial expressions. In *PLDI*, 2012.

[42] H. Poon. Grounded unsupervised semantic parsing. In *ACL*, 2013.

[43] A.-M. Popescu, A. Armanasu, O. Etzioni, D. Ko, and A. Yates. Modern natural language interfaces to databases: Composing statistical parsing with semantic tractability. In *COLING*, 2004.

[44] A.-M. Popescu, O. Etzioni, and H. A. Kautz. Towards a theory of natural language interfaces to databases. In *IUI*, 2003.

[45] D. Price, E. Riloff, J. L. Zachary, and B. Harvey. NaturalJava: A natural language interface for programming in Java. In *IUI*, 2000.

[46] Ranking. Modified competition ranking ("1334" ranking). http://en.wikipedia.org/wiki/Ranking, 2015.

[47] M. Raza, S. Gulwani, and N. Milic-Frayling. Compositional program synthesis from natural language and examples. In *IJCAI*, 2015.

[48] R. Singh and S. Gulwani. Learning semantic string transformations from examples. *PVLDB*, 5, 2012.

[49] R. Singh and S. Gulwani. Synthesizing number transformations from input-output examples. In *CAV*, 2012.

[50] A. Solar-Lezama. Program synthesis by sketching, 2008.

[51] A. Solar-Lezama, L. Tancau, R. Bodík, S. A. Seshia, and V. A. Saraswat. Combinatorial sketching for finite programs. In *ASPLOS*, 2006.

[52] S. Srivastava, S. Gulwani, and J. Foster. From program verification to program synthesis. In *POPL*, 2010.

[53] N. G. Stanford. Stanford parser - 2.0.2. http://nlp.stanford.edu/software/lex-parser.shtml/, 2014.

[54] D. M. L. Xu, R. Bodík, and D. Kimelman. Jungloid mining: Helping to navigate the API jungle. In *POPL*, 2005.

[55] K. Yessenov, S. Tulsiani, A. Menon, R. C. Miller, S. Gulwani, B. Lampson, and A. Kalai. A colorful approach to text processing by example. In *UIST*, 2013.

[56] L. Zettlemoyer and M. Collins. Learning to map sentences to logical form: Structured classification with probabilistic categorial grammars. In *UAI*, 2005.

[57] L. S. Zettlemoyer and M. Collins. Online learning of relaxed CCG grammars for parsing to logical form. In *EMNLP-CoNLL*, 2007.