Performance Analysis of Program Synthesis Techniques

Hima Mahajan[1], Pranjal Kaler[1], Priyanshu Gautam[1], Siddharth Dangwal[1], Padmavati[1]

[1]Department of Computer Science and Engineering, Punjab Engineering College, Chandigarh

Abstract In recent years, program synthesis or mechanized construction of software, also dubbed as 'self-writing code' has gained tremendous importance in the field of computer science. Program synthesis has the potential to change the way general-purpose computational devices are used by enabling users to solve problems in an automated fashion without designing and implementing new algorithms. It can be used by various programmers to develop utility programs, improve and debug their codes. In this paper, we focused on reviewing various program synthesis techniques used to generate automated programs. We examined the intriguing evolution of various program generation models and present the detailed analysis of various state-of-the-art design methods along with accuracy, requirements and their future scope. We formulated a hierarchical classification to classify various techniques based on input parameters. A comparison table is presented based on parameters such as input, output, efficiency and a brief description of proposed techniques. Lastly, we present a validation table containing comparison of the results obtained at our end with the results suggested by the authors. We conclude with a comprehensive look at program synthesis and future research scope in this field.

***Index Terms*—Artificial intelligence, Automatic programming, Deep learning, Machine learning, Neural network, Program synthesis, Programming languages, Software diversity.**

I. Introduction

Program synthesis is the task of generating a program in a specified programming language to meet user intent. User intent can be captured in various forms including I/O examples, natural language description, flowcharts, partial source codes, logic specifications, demonstrations etc. The ultimate goal of the research community is to fully automate the process of synthesizing, debugging, refactoring and maintaining software programs. It can help non-programmers develop programs with greater ease, repair programs and transforming raw data.

With advancements in deep learning leading to faster and efficient computational power, program synthesis has seen exponential growth in interest and advances in research. This firstly, opens the door for research in deductive synthesis approaches in which user intent is specified in formal language. Later, the complexity of specifications led to inductive synthesis techniques where user intent is captured as examples, demonstrations etc. Two main challenges in the field are: (1) ambiguous user intent, (2) large program search space. This paper gives a comprehensive overview of state-of-the-art of this field to serve as an introduction to newcomers. We have included an overview of different works on program synthesis and provided an abridged explanation of the techniques used and comparison that conceives future scope of various approaches finally follows.

II. Literature Survey

Many techniques have been proposed to synthesize programs from various specifications. A number of research papers have been studied and briefly described in this section. Papers have been classified based on the input parameters.

A. Input based approaches

Numerous techniques followed in this section require some input specifications for the synthesis of programs.

1. Input as sample I/O examples

In [1], the author proposed an approach for solving programming competition style problems from I/O examples using deep learning. It trains a Neural Networks (NN) on inductive synthesis problems to predict cues from problem description to aid search-based technique. This approach reduces effect of failures of a neural network by searching over program space rather than relying on a single prediction and thus improves accuracy. It specifies a Domain Specific

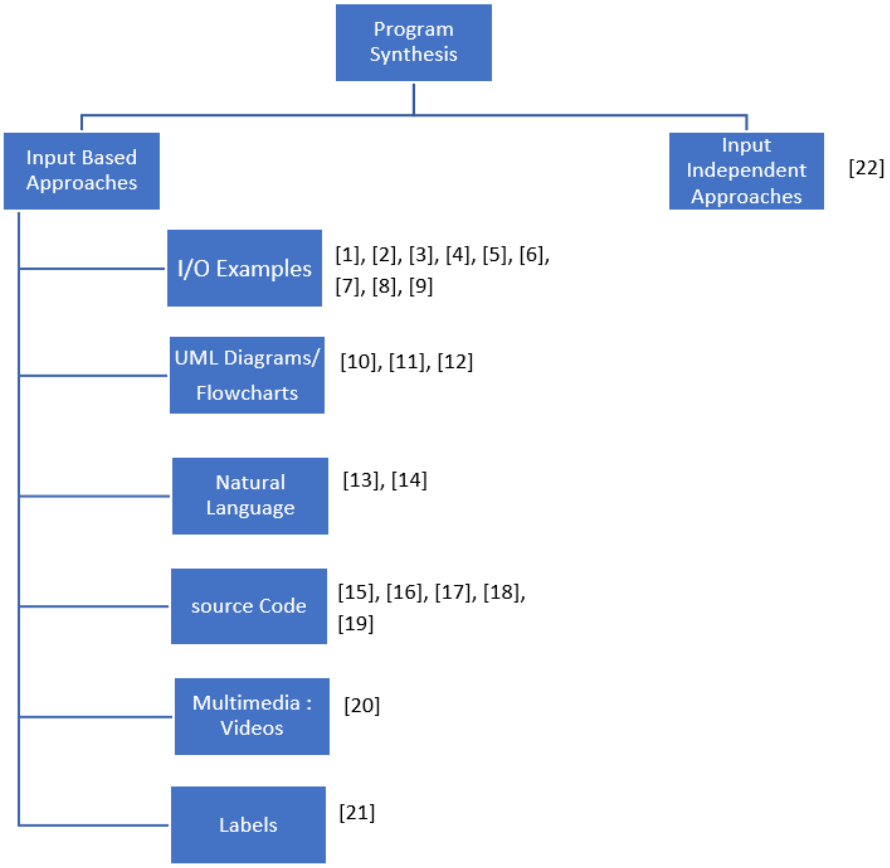

Figure 1. Hierarchical classification based upon input parameters.

Language (DSL) for programs, such that each program can be mapped to a vector of attributes. Attributes can be presence or absence of high-level functions or control flow templates. Dataset of program, corresponding attributes and accompanying I/O examples are generated. Using feed-forward neural network model with negative cross entropy, the model learns mapping from I/O examples to attributes and predicts the presence or absence of individual functions of DSL. The search techniques integrate predicted attributes to search over program space. The model improves the runtime of a wide range of Inductive Program Synthesis (IPS) baselines by 10-1000 times.

In [2], the author addressed the problems that arise on using multiple predictors, by using Latent Predictor Model (LPN). This combines a character level SoftMax to generate language-specific tokens. Languages targeted in this model are Java and Python. A multi-pointer network is used to copy keywords from the input. Additionally, this algorithm also works on code compression by mapping commonly used programming keywords onto new symbols. For example, `public static void foo()` is converted to `X1, X2, X3, X4()`. This model uses Magic The Gathering (MTG, two player game) dataset for Java and HearthStone (HS, two player game) dataset for python. The bilingual evaluation understudy (BLEU) metric is used for determining accuracy.

In [3], the author presented an algorithm for example-guided synthesis of functional programs over recursive data structures. The synthesis algorithm targets a functional programming language that permits higher-order functions, combinators like map, fold and filter, pattern matching, recursion, and a flexible set of primitive operators and constants. The input to the algorithm is a set of I/O examples that define the behavior of the target program on certain small-sized instances. This approach combines three technical ideas: inductive generalization, deduction, and best-fit enumerative search. In inductive generalization, the user-provided examples are generalized into a set of hypothesis as per the structure of the target program. A hypothesis is either a concrete program or a “skeleton” that contains placeholders (“holes”) for unknown programs. For each hypothesis, deduction is used to infer new input/output examples for the missing subexpressions. This leads to a new subproblem where the goal is to synthesize expressions within each hypothesis. Best-fit enumerative search is used to identify the hypotheses that can be realized into programs that fit the top-level goals.

In [4], the author proposed attention based Recurrent Neural Network (RNN) architecture to synthesize programs which take input-output examples as input and generates a program as output for string transformations. Model is trained in a fully supervised manner on a large corpus of I/O

examples and program pairs. A program is generated in DSL for string transformations. Observed input-output pairs are encoded using a sequential RNN which then generates the program using another RNN, token by token. Multiple unordered I/O examples are encoded using Attention-C based RNN with late-pooling technique to incorporate powerful attention mechanisms into our model. The synthesis models are decoded using DP-Beam search decoder to get k output programs and only 1-best output is taken from k-best candidate programs which are tested one by one to determine consistency. Usage of RNN based approach and expressiveness of DSL makes it robust to moderate levels of noise in I/O examples and achieves 92% accuracy on real-world Programming by Example (PBE) task thereby outperforming previous best neural synthesis model by 58%.

In [5], the author addressed auxiliary problems that arise out of program synthesis, like program aliasing, that is, there being more than one semantically correct program for a given problem. Models today tend to penalize such semantically correct alternatives, thus affecting synthesizer performance. This approach targets this using Reinforcement Learning. Secondly, it targets the idea that for a program, by only using syntactically correct inputs, we limit the search space and can thus improve efficiency, especially when the training set is limited. Here the syntax checker is implemented as a LSTM (Long Short-Term Memory, a RNN technique) to provide greater penalties for using wrong tokens by use of an activation function. Also, it ensures that the checker models run on only the language syntax by learning only program tokens and not I/O pairs. The model is run on Karel DSL with 1 million training examples.

In [6], the author proposed Neural Guided Deductive Search (NGDS), a hybrid synthesis technique, that allows it to leverage qualities of both deductive framework and symbolic logic technique. This, in turn, builds upon PROgram Synthesis using Examples (PROSE) but consistently avoids the top-down enumeration problem while retaining principle attributes of a deductive search. This approach is effective in the PBE domain in that it learns the required program from only one input/output example in about 60% of the cases. Results are benchmarked against DeepCoder and RobustFill and are found to be better and more accurate in various aspects.

In [7], the author proposed an inductive synthesis algorithm to synthesize recursive programs. It accepts input-output examples from user to learn and is parametrized by a set of components (instructions) that can appear in a synthesized program. Component set must include a recursive call to synthesized program to allow recursive programs. It uses explicit search strategy alternating between two phases: (1) Forward: enumerates programs by augmenting new components with synthesized programs. (2) Conditional Inference: uses a goal graph data structure to detect the need to join two synthesized programs by conditional statement. The alteration is guided by a heuristic function for choosing rule to be applied based on size of program. The algorithm can be applied to different domains by instantiating it with different sets of components. Adoption of explicit search strategy does not restrict synthesis by supported Satisfiability Modulo Theories (SMT) theories unlike traditional SMT encoding. Use of goal graph increases efficiency in comparison to naive approach of using if-then-else component. The algorithm synthesizes broad range of programs including tail recursive, divide-and-conquer and recursive programs. It is shown to outperform state-of-art SAT-based synthesis tool at synthesizing recursive and conditional programs from components.

In [8], the author proposed a neural guided constraint logic programming system to solve PBE. It combines two state-of-art approaches of PBE: (1) symbolic techniques for restricted DSL (2) synthesizing program as a sequence and guide using neural model. The model takes input-output examples as input. Using dialect of List Processing (LISP) as constraint programming language, constraints are built from input-output examples to represent search space for PBE problem. Candidate programs contains unknowns whose values are restricted by constraints. Target programs are searched by alternating between partial programs with bias towards programs having more constraints already satisfied. Candidate programs are scored using RNN with bi-directional LSTM units. The approach overcomes the problem of scalability of symbolic system and uses neural guidance to navigate exponentially large search spaces. The approach is able to generalize to programs larger than training and can generalize to larger programs.

In [9], the author proposed a technique called Neuro-Symbolic Program Synthesis which automates computer programs in a DSL provided a set of input output examples. The approach is divided into two neural architecture modules. The first module, a cross correlation I/O network which produces a continuous set of I/O examples and helps the model discover input substrings that are copied to output. The second module, Recursive-Reverse-Recursive Neural Network (R3NN) for synthesizing a program by incrementally expanding partial programs. There are a number of ways in which the partial program tree (PPT) generation model can be conditioned using the I/O example encodings depending on where the I/O example information is inserted in the R3NN model - Pre-conditioning, Post-conditioning and Root-conditioning. R3NN model is able to construct programs from new input-output examples as well as construct new programs for tasks that it had never observed before training. This approach works well only for string-based transformations.

2. Input as flowcharts or UML diagrams

In [10], the author presented an approach for code generation from unified modeling language (UML) models based on class and sequence diagrams. The main sequence diagram is referenced to the method main and defines the start point for the behavioral code. From the class diagram, structural code is generated. From each class, a Java file is generated, describing its attributes and method signatures. Parameters passed in the constructor are used to initialize the parameters. The code generation phase considers relationship between classes or interfaces, the cardinality of attributes, and it generates get and set methods. The sequence of method invocations, their arguments and return values are captured from the sequence diagrams. Loops and conditionals are captured from sequence diagram to generate corresponding Java statements (for/while or switch/if-else). When a ref fragment is found, another sequence diagram is read to generate corresponding structure code. The approach cannot generate complete code, since the sequence diagram granularity is the method invocations, which in turn, is difficult to precisely process. However, it offers high abstraction and relevant automation for embedded software development.

In [11], the author presented an approach of automatic java code generation from UML models. All the control flow and data flows are depicted using activity diagrams and the method invocations (communication between objects) are depicted using sequence diagrams. The prototype generation includes four steps. First step, the workflow of a system is modelled using activity and sequence diagrams. In the second step, diagrams and object constraint language (OCL) statements are checked for errors. In the third step, diagrams and OCL statements are converted to Extensible MarkupLanguage (XML) files. In the final step, these XML files are converted to prototype of the system. The prototype generation algorithm, *Am_To_Prototype*, takes the object tree of activity graph (AG) and sequence diagram as input and the output is the prototype of the activity model. It generates one main class and one class for each type of object present in activity graph. Main class includes a main() method which initiates execution. This algorithm uses two sub procedures, *Excecution_Logic* and *Method_Body*. *Excecution_Logic* is used to implement the main() method in the Main class. *Method_Body* is used to implement the definition of all methods in activity graph. This method ensures more than 80% of completion of the prototype of system. Since activity models are associated with collaboration models it is able to generate more complete source code automatically.

In [12], the author presents a tool which takes UML class, sequence diagram and activity diagrams as an input to generate completely executable Java code. The architecture of tool is divided into three main components: (1) XML Parser: It takes UML diagrams in XML format as input and generates UML diagrams metamodel instances. (2) Code Generator: It generates isolated Java code from metamodel instances. UML class diagram metamodel instances generates Java structural code which contains classes, their attributes, method signatures, interface, methods and relationship between classes. UML activity diagram metamodel maps UML activity actions onto Java code. UML sequence diagram metamodel instance outputs class methods code which contains control flow, alternative and iterative blocks, and activity diagram references. (3) Code Merger: It replaces activity diagram references and generates the complete Java code for classes and interfaces which contain fully functional class methods. This approach generates accurate, maintainable, efficient, and complete Java code.

3. Input specified as natural language description

In [13], the author proposed source code generation, from natural language (here, English) in python is targeted. Thus, a query, sort a list in descending order gives *sorted(name of_list, reverse=true)* as code output. This approach uses a grammar model to generate a derivative Abstract Syntax Tree (AST) using standard parsers provided by python. It then applies either production rules or adds terminal tokens to update the AST. Algorithm is tested on HearthStone (HS), Django and IF This Then That (IFTTT) datasets. Since there can be several codes for complex problems and all of them may be correct, the authors use BLEU metric for measuring accuracy. The results are benchmarked against Latent Predictor Network (LPN) and SEQ2TREE[23] and are found to be better.

In [14], the author presented a complete machine learning approach for translating natural language to code, mainly bash scripts. In this approach, a system called Tellina is built that lets a programmer describe a desired operation in natural language, then automatically translates it to a programming language for review and approval by the programmer. The system does the translation using RNNs, a natural language processing technique that is augmented with slot (argument) filling and other enhancements. A three-step deep-learning approach is followed. (1) A user provides a natural language sentence X, which Tellina transforms to a template ˜X by recognizing and abstracting over domain-specific constants, such as file names or times. (2) A RNN encoder-decoder model translates ˜X into a ranked list of possible program templates ˜Yi, where each ˜Yi contains argument slots to be filled by entities recognized. (3) The slots in each ˜Yi are replaced by program literals to produce an output program Yj, using a k-nearest neighbor classifier which identities the corresponding source entities. The model is trained on a newly gathered corpus of over 5000 natural language and bash command pairs. The encoder-decoder is trained using pairs of natural language and program templates. The standard sequence-to-sequence training objective is used to maximize the likelihood of the ground truth program template given the natural language template. Tellina is

evaluated in the context of shell scripting. Tellina's RNNs are trained on textual descriptions of file system operations and bash one-liners, scraped from the web and achieves 80% top-3 (results are correctly guessed in the top 3 corrected classes that we count) accuracy for determining command structures and 36% top-3 accuracy for full commands.

4. Input as source code or partial programs

In [15], the author used an iterative optimization scheme, where a RNN is trained on a dataset of k-best programs stored in a priority queue. The training starts with an empty buffer and a randomly initialized RNN. New programs are generated and evaluated based on their reward function with penalties for the length of programs generated. These programs are then added to priority queue if their reward value is in the top k values in the queue. This algorithm is labeled as Priority Queue Training (PQT) and is benchmarked against genetic algorithms (GA) and reinforcement learning techniques like policy gradient and gives significantly better results over both.

In [16], the author targeted two auxiliary aspects of program generation, automatic program repair and mutation testing. Moving beyond conventional approaches which used search-based techniques in which search space is generally very large, this approach vouches for variational execution. Variational execution is wrapping believed-to-be-working pieces of code in conditionals and then executing them in a test suite to ascertain the working or the most optimal patch of code. For mutation testing as well, this approach works on generating relevant and diverse combination of mutants (a test case modified from some original one) and comes up with an exhaustive set of test cases.

In [17], the author presented an automated program repair technique named SketchFix to modify faulty programs and fix them with respect to given test suites. SketchFix generates candidate fixes on demand during the test execution. Instead of iteratively re-compiling and re-executing each actual candidate program, it translates faulty programs to sketches, i.e. partial programs with "holes", and compiles each sketch which may represent thousands of concrete candidates. This technique focuses on reducing number of candidate programs by pruning and generating on-demand candidates during test validation. SketchFix performs a systematic reduction of program repair to program synthesis by translating a faulty Java program to sketches, which will be completed by a synthesizer with respect to the given test suite. The input is a faulty Java program and test suite. The system introduces holes to suspicious statements based on the AST node level transformation schemas and uses the backtracking technique to fill in these holes. The experimental results show that system works well in manipulating expressions.

In [18], the author addressed the problem of learning complex programs from I/O examples. The input is a sequence of tokens *["a", "=", "1", "if", "x", "==", "y"]*, and the output is its parse tree. The approach is to train a differentiable neural program to operate a domain-specific non-differentiable machine. A two-phase reinforcement learning-based algorithm is adopted in which the problem is divided into two separate tasks: (1) Searching for possible traces; and (2) Training the model with the supervision of traces. To implement the idea of learning a neural program operating a non-differentiable machine, the LL machine is designed, which bakes in the concept of recursion in its design. An LL machine has a stack for recursive call s, and provides an instruction. It has instructions CALL and RETURN which can be used to simulate recursive calls. Then, neural parsing program is designed, such that every neural parsing program operating an LL machine is restricted to represent an LL parser. A TWO-PHASE TRAINING STRATEGY is adopted. First, for each I/O pair, a set of valid candidate instruction type traces with a preference toward shorter ones is searched. Second, to search for a satisfiable specification, which is a neural parsing program that can parse all inputs into their outputs using the corresponding instruction type traces in the specification. This approach is able to achieve 100% accuracy on test samples which are longer than training samples.

In [19], the author explored the possibility of using program synthesis approaches on program-diversity problems, that is, having more than one executable code for a particular problem. In this approach, the authors target algorithmic diversity as opposed to diversity by manipulating control flow graph by moving logical blocks of code around the source code. An example taken in the paper shows how a developer written code to sort a list of numbers is reimplemented in a different manner by the model. The paper proposes an automated program diversity framework using specific inference and program synthesis. Next, it assesses the security impact made by removing code reuse attacks. Finally, it assesses the resilience of the diversity technique to common software faults by fault injection techniques. Future scope of this paper involves generating reliable metrics for reliability and security implications of an automatically diversified system.

5. Input as set of multimedia videos

In [20], the author proposed a model that synthesizes programs from multiple demonstration videos exhibiting diverse behaviors. The model infers a program behind set of demonstrations by: (1) interpreting each demonstration video. This is achieved by LSTM to encode each state to capture actions and perceptions of agent. (2) discovering the branching conditions and subsequent action taken in demonstration. This is achieved by re-encoding the output of demonstration encoder and aggregating to get summarized representation. (3) Program is synthesized from summarized representation. LSTMs are initialized with summary and at each step gets previous token as an input and outputs probability of following program tokens. The method is

evaluated on a fully observable, third-person environment (Karel environment) and a partially observable, egocentric game (ViZDoom environment). The experiments demonstrate that the proposed model is able to reliably infer underlying programs and achieve satisfactory performances.

6. Input as labels

In [21], the author presented an approach to generate source code in strongly typed, Java-like language mainly API-heavy codes using conditional program generation. The problem is addressed with a combination of neural learning and type guided combinatorial search. In contrast to general program synthesis approach driven by I/O examples, author proposed the idea of using labels defined as finite sets: a set of possible API calls in ARC Macro Language (AML), a set of possible object types, and a set of keywords, defined as words, such as "read" and "file", that often appear in textual descriptions of what programs do. The learning is over sketches (tree-structured syntactic models) of programs rather than the source codes. Various sketches are sampled out using Gaussian Encoder-decoder and converted into type safe programs using combinatorial method for program synthesis. This approach is implemented for a system called BAYOU, which accepts an input as a label and gives API-heavy java code as output. System is able to generate complex method bodies from just few tokens available in label and learning at the level of sketches is the key to performing such generation effectively.

B. Input independent approaches

In [22], the author used Genetic Algorithms, where 8 instructions set and Turing complete language brain-fuck (BF) is used. No I/O samples are provided. The model starts training with a random sequence of its instructions. Then, for the second iteration some bits of instructions are retained (crossover) while some bits are replaced to generate a new program (mutation). Progress here is determined using a fitness function that does a character-by-character comparison of the generated output with the expected output while training. Simple programs like printing "Hello World!", multiplying by 3 to little complex ones like XML to JSON converter were tried.

III. COMPARISON OF EXISTING TECHNIQUES FOR PROGRAM SYNTHESIS

TABLE I: COMPARISON OF EXISTING TECHNIQUES FOR PROGRAM SYNTHESIS

S.No.	*Title*	*Input Parameters*	*Output*	*Proposed Techniques*	*Accuracy/ Efficiency*	*Future Scope/ Drawbacks*
1.	DeepCoder [1]	Set of input and output examples (5 for each test case) E.g.: Input: [-17, -3, 4, 11, 0, -5, -9, 13, 6, 6, -8, 11] Output: [-12, -20, -32, -36, -68].	Program in DSL.	Learning Inductive Programming synthesis (LIPS) Model 1: Feed-forward neural network (negative cross entropy loss) Model 2: RNN Search techniques: SMT; sort and add enumeration.	Proposed technique improves the runtime of a wide range of IPS baselines by 1-3 orders.	DSL can't express solutions to many problems. Application of these techniques to Turing complete language so as to include more complex problems.
2.	Latent Predictor Model for Code Generation [2]	Two different datasets used: MTG in Java and HS in Python.	Source code in Java and python.	Latent Prediction Network (LPN), loosely based 'on NN.	Proposed system obtain significant improvements over phase based, retrieval based and hierarchical based system and achieved upto 62.3 accuracy.	Efficiency is the only aspect that can be worked on.
3.	Synthesizing Data Structure Transformations from I/O Examples * [3]	Input/output examples: [[1,3,5], [5, 3, 2]] = [3, 5], [5, 3]	Output source code in DSL.	1. Type-aware inductive generalization 2. Search over hypotheses about program structure 3. Use of deduction to guide the search.	Median runtime is 0.43s and can synthesize 88% of benchmarks under a minute.	Using the same method for synthesizing the proofs.
4.	Neural Program learning under noisy I/O[4]	Input -Output examples.	Output in DSL specific to string manipulations.	Attention-C based RNN with DP-beam search for decoder.	92% compared to 34% of previous experiments.	Scope is limited to string manipulation.
5.	Leveraging Grammar and Reinforcement	Karel Dataset (~ 1 million I/O examples).	Source code in Karel. Karel is DSL (Domain	Reinforcement learning	100% accuracy in generating syntactically correct programs.	NA

	Learning for NPS[5]		Specific Language)			
6.	NGDS for real time program synthesis from examples [6]	Input/output Examples: ["(612) 8729128"]} --> "612-872-9128".	Formatted string.	a) Neural Guided Deductive Search (NGDS) b) Inductive Program Synthesis (IPS).	Results. Speed up: ~20% over FlashFill (state of the art till date). And generalization with single example in 60% cases.	Improving accuracy.
7.	Recursive program synthesis[7]	I/O examples Corpus of possible statements to be used in a program.	Source code in DSL.	Inductive heuristic search as a means to search the corpus to synthesis valid recursive program.	Time used is very less compared to other techniques.	Extend the work to generate loops along with recursion and include SMT-based search to improve synthesis of constants-based program.
8.	Leveraging Constraint Logic Programming for Neural Guided Program[8]	I/O examples.	Source code in LISP.	Uses constraint-based search to search for a program and neural network to rate candidate programs and thus, guide the search.	The model is able to solve 99% of test problems	Extend the approach for recursive problems.
9.	Neuro-Symbolic Program Synthesis[9]	Input/output Examples Example: Input: William Henry Charles ---> Charles, W.	Source code in DSL.	a) Cross correlation I/O network b) Recursive Reverse-Recursive Neural Network (R3NN)	High accuracy for string manipulations.	Extend the domain of programs generated.
10.	Generating Java code from UML Class and Sequence Diagrams[10]	UML Diagrams: class diagrams and sequence diagrams.	Partial program in java.	From each class diagram, a java file is created. Sequence of method invocation and loops and conditional statements are captured from sequence diagrams.	NA.	Complete code is not generated, as the sequence diagram granularity is method invocations only.
11.	Automatic code generation using unified modeling language activity and sequence models[11]	UML Diagrams: activity diagrams and sequence diagrams.	Nearly complete source code in Java.	a) The workflow of a system is modelled using activity and sequence diagrams. b) The diagrams and OCL statements are checked for errors. c) The diagrams and OCL statements are converted to XML files. d) XML files are converted to prototype of the system.	NA.	Complete working code in java language for all problems is not guaranteed.
12.	Automatic Generation of Java Code from UML Diagrams using UJECTOR[12]	UML Diagrams (class, activity, sequence) in XML format.	Completely executable Java code.	(1) XMLParser: takes UML diagrams in XML format and generates UML diagrams metamodel instances. (2) CodeGenerator: generates isolated Java code from metamodel instances. (3) CodeMerger: Replaces activity diagram references and generates the complete Java code for classes.	Fully functional and consistent.	Need highly detailed UML which is highly human centric work.
13.	Syntactic Neural Model for General	Natural Language i.e. sentences in	Source code in python.	a) Grammar model: Abstract Segment Tree (AST).	BLEU scores for datasets - 75.8 - HS 84.5 – Django.	Working on datasets of languages other than Python

	Purpose Code Generation[13]	English for what to code. Example: **Input:** *for every i in range of integers from 0 to length of result, not included.* **Prediction**: *for i in range (0, len(result)):*		b) BLEU metric for accuracy grading since there are multiple ways of writing complex codes.		and establishing effectiveness of model there as well.
14.	Program Synthesis from Natural Language Using Recurrent Neural Networks[14]	Natural language sentences in English. **Example:** I have a bunch of ".zip" files in several directories "dir1/dir2", "dir3", "dir4/dir5". How would I move them all to a common base folder? Output: find dir*/ -type f -name "*.zip" -exec mv { } "basedir" \;	Shell Scripts.	a) Recurrent neural networks b) Slot argument filling technique c) Natural Language processing (NLP).	a) Model achieves top-3 accuracy of 80.0% for determining program structure that is, ignoring constant values. b) Model achieves top-3 accuracy of 36.0% for full commands.	a) To extend the neural architecture so that the entire framework for entity recognition, template translation and argument filling can be learned end-to-end. b) To extend the approach to cover more programming languages.
15.	Neural Program Synthesis[15]	Training dataset: Code of remove-char and reverse a string to train the hyperparameters. No elaborate dataset of source codes or test cases.	Program in BF.	Two approaches Used: a) Policy Gradient to train the neural network one token at a time. b) Priority Queue Training on BF i.e. Sample new programs from the RNN and update the queue iteratively.	22.8% Programs are synthesized 5.7/25 programs ran during evaluation.	Mostly, similar to GA approach. Working on enhanced problems.
16.	Using variational execution for automatic program repair and higher order mutation testing[16]	Source code in DSL.	Source code in DSL.	Variational execution (uses if-then-else to execute test cases in order).	NA.	Improving Accuracy.
17.	Towards Practical Program Repair with On-Demand Candidate Generation[17]	Input: a faulty program Output: Program sketch generated by sketch-fix.	Fixed source code.	a) AST node-level transformation b) Candidate generation and the sketching using backtracking search.	Fixes 19 out of 357 defects in 23 minutes on average using the default setting. Finds the first repair with 1.6% of re-compilations and 3.0% of re-executions out of all repair candidates.	Currently, the model works well only for manipulating expressions. To increase the scope of model to cover broader problems.
18.	Towards Synthesizing Complex Programs from Input-Output Examples[18]	Input: A sequence of tokens Example: *["a", "=", "1", "if", "x", "==", "y"].*	A parse tree.	Two-phase reinforcement learning-based algorithm and policy gradient technique to improve the search strategy.	Almost 100% accuracy for simple programs.	Extending approach to learn algorithms on more complex data structures such as trees and graphs.

19.	Automated program diversity using Program Synthesis[19]	Input: Sample code. Example: Counting sort written by developer: *for (i = 0 ; i < (length - 1); i++) { 6 for (j = 0 ; j < length - i - 1; j++).*	Output: Code that implements the same business logic. Written by model: *(forall i,0<i< length - 1, arr[i] < arr[i+1]) 5 (forall i, arr[i] exists in arr').*	NA.	Validation used to ascertain if code generated is logically correct. So, full accuracy is being assumed.	Generate evaluation metrics to specify workability of code.
20.	Neural Program Synthesis from Diverse Demonstration Videos[20]	Input: Diverse set of demonstration videos	Output: Source code in DSL.	1. Interprets each video to identify actions and perceptions of agent 2. Summarizes set of videos to identify branching conditions and subsequent actions and information is stored in LSTMs 3. Program is synthesized from LSTM	The model has been evaluated in two different environments: 1. Karel: visually simple environment The model gives execution accuracy of 18.9% as compared to synthesis baseline with accuracy of 42.4% 2. Viz Doom It is a 3D first-person shooter game environment. It gives execution accuracy of 78.4% compared to 48.2% of synthesis baseline.	NA.
21.	Neural Sketch Learning for Conditional Program Generation[21]	Input: label containing a small amount of information about a program's code (function calls, type, libraries, API's).	Java generated code.	a) Maximum Conditional Likelihood Estimation (CLE) b) Bayes net approach c) Combinatorial mathematics for clustering.	Higher accuracy as compared to I/O examples approach.	To extend the domain of programs as currently, works well only for API heavy java code.
22.	Genetic Algorithms for Autonomously building Simplistic Programmes[22]	None.	Source code in BF language.	Genetic Algorithms using BF.	Source code for all basic programs was generated.	Generating code for more arithmetic intensive problems like Prime Numbers, Google Code Jam questions etc.

IV. Experimental Validations

The results of [7] and [22] are validated using a system with Intel Xeon W-2155 Processor having 13.75 MB Cache, 10 cores, 20 Threads and 4.50 Ghz Max Turbo Frequency.

a) Recursive Program Synthesis [7]

Table I TypedEscher Performance

	Local results/ Expected results				***Deviation (Percent Error)***			
Name	Cost	Depth	Examples	Time (in ms)	Cost	Depth	Examples	Time (in ms)
Sorted list	29/35	12/13	11/7	403,482.25/ 123,855	-17.72	-7.69	57.14	225.77
reverse	12/12	8/8	4/3	118.5/500	0	0	33.33	-76.3
length	8/8	4/4	3/3	14.37/19	0	0	0	-24.37
compress	25/25	9/9	14/8	176.51/415	0	0	75	-57.47
stutter	13/13	9/9	32/3	100.95/ 79	0	0	966.67	27.78
squarelist	14/14	9/9	6/6	3,258.86/799	0	0	0	307.8
insert	19/19	9/9	8/8	6,492.25/554	0	0	0	1071.88
contains	14/14	5/5	7/7	9.91/10	0	0	0	-0.9
lastInList	14/14	5/5	5/5	10.04/6	0	0	0	67.33
shiftLeft	12/12	8/8	5/5	45.93/23	0	0	0	99.69
maxInList	20/24	10/8	7/8	13,123.30/233	-16.67	25	-12.5	5532.32
dropLast	15/15	6/6	5/5	12.77/22	0	0	0	-41.95
evens	16/16	7/7	5/5	27.29/24	0	0	0	13.71
cartesian	32/32	11/12	4/4	3,463.95/174	0	-8.33	0	1890.77
fib	15/15	8/8	8/8	2,048.36/228	0	0	0	798.40
sumUnder	10/10	5/5	5/5	8.36/9	0	0	0	-7.11
times	11/22	6/8	6/19	298.00/59,827	-50	-25	-68.42	-99.50
flattenTree	14/14	10/10	3/3	85.49/28	0	0	0	205.32
tConcat	15/15	11/11	6/6	50,626.05/14,899	0	0	0	239.7
nodesAtLevel	27/27	11/11	12/11	271,612.84/1,751	0	0	9.09	15411.87
Total	335/429	163/196	127/168	755,016/343,111	-21.91	-16.8	-24.40	120.05

Table II AscendRec Performance

	Local results/ Expected results				***Deviation (Percent Error)***			
Name	Cost	Depth	Examples	Time (in ms)	Cost	Depth	Examples	Time (in ms)
reverse	12/12	8/8	3/4	358.93/508	*0*	0	-25	-29.34
length	8/8	4/4	3/3	20.11/19	*0*	0	0	5.84
compress	25/25	9/9	8/14	19.68/295	*0*	0	-42.86	-93.32
stutter	13/13	9/9	3/3	51.56/149	*0*	0	0	-65.39
squarelist	14/14	9/9	6/6	569.95/3,279	*0*	0	0	-82.62
insert	19/19	9/9	8/8	396.74/6,600	*0*	0	0	-93.98
contains	14/14	5/5	7/7	9.59/8	*0*	0	0	19.87
lastInList	14/14	5/5	5/5	5.39/5	*0*	0	0	7.8
shiftLeft	12/12	8 /8	5/5	22.28/43	*0*	0	0	-48.19
maxInList	24 /20	8/10	16/7	3,400.71/13,500	*20*	-20	128.57	-74.81
dropLast	15/15	6/6	5/5	6.27/9	*0*	0	0	-30.33
evens	16/16	7/7	5/5	10.61/15	*0*	0	0	-29.26
cartesian	32/32	12/11	4/4	133.41/3,569	*0*	9.09	0	-96.26
fib	15/15	8/8	8/8	183.56/2,080	*0*	0	0	-91.17
sumUnder	10/10	5/5	6/5	10.97/7	*0*	0	20	56.71
times	22 /11	8/6	19/6	58,191.86/273	*100*	33.33	216.66	21215.69
flattenTree	14/14	10/10	3/3	24.38/81	*0*	0	0	-69.90
tConcat	15/15	11/11	6/6	13,525.30/54,045	*0*	0	0	-74.97
nodesAtLevel	27/27	11/11	11/12	1,981.65/238,356	*0*	0	-9.09	-99.16
Total	429/306	196/151	177/116	401,422/322,842	*40.19*	29.80	52.58	24.34

b) BF-Programmer[22]

Table III BF-Programmer Performance

Name	*Expected Duration (s)*	*Expected Generations*	*Local Duration (s)*	*Local Generations*	*Deviation in Duration (%)*	*Deviation in Generations (%)*
hi	52	5,700	160	17,200	207	201.7
Hi!	7,644	1,219,400	87	16,900	- 98.8	- 98.6
reddit	1,362	195,000	4,114	12,93,000	202	563
hello	1,713	252,000	897	1,71,600	- 47.6	- 31.9
Addition	2,698	92,400	929	77,200	65.5	- 16.4
Fibonacci	21,862	151,900	859	21,800	- 96	- 85.6
If/then conditionals	8,313	46,200	1,864	84,600	- 77.5	83.1
Bottles of Beer on the Wall	2,957	61,400	11,788	2,00,200	298	226
Reverse string	49	2,600	697	31,300	1322	1103
Countdown	-	-	15	800	-	-
Logical AND	-	-	56	3,600	-	-
Less than equal to 3	-	-	275	13,000	-	-

V. Conclusion and Future Scope

This performance review considers model training for program synthesis with and without use of inputs. General overview reveals that for models that do not require input, simple programs like printing strings on the screen and adding two numbers have been performed. Significant progress has been obtained using DSL where a class of problems has been tackled. The review also covers work in specific post program generation jobs like automatic program repair, mutation testing and ranking of possible solutions generated. We came across a class of inputs that models today accept where training using I/O examples remains the most favored type. Since the current state-of-the-art models are either problem specific or if are generic, can only solve a very simple set of problems, we feel the real challenge in this field is resolving the tradeoff between accuracy of code produced and generality of the algorithm. Two paths could lead from here. One, extend the programming language base, that is, work on generating source code in multiple languages. Most approaches suggested here have tackled the problem in one programming language. Second, work on solving more complex problems.

The scope of automatic program synthesis is very impactful in that it can relieve humans of labor-intensive tasks of having to write code which is almost always on similar lines, like, writing a get API. Instead, they could get involved in more thinking critical jobs like product design and requirements analysis. It has a scope to revolutionize software development.

VI. References

[1] Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, Daniel Tarlow, “DEEPCODER: LEARNING TO WRITE PROGRAMS” in *ICLR*, 2017.

[2] W. Ling, E. Grefenstette, K. M. Hermann, T. Kocisky, A. Senior, F. Wang, P. Blunsom, “Latent predictor networks for code generation”, 2016.

[3] John K. Feser, Swarat Chaudhuri, Isil Dillig, “Synthesizing Data Structure Transformations from Input-Output Examples” in *ACM*, 2015.

[4] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, Pushmeet Kohli. “RobustFill: Neural Program Learning under Noisy I/O”, in *ICML*, 2017.

[5] R. Bunel, M. Hausknecht, J. Devlin, R. Singh, P. Kohli, “Leveraging grammar and reinforcement learning for neural program synthesis”, 2018.

[6] A. K. Vijayakumar, D. Batra, A. Mohta, P. Jain, O. Polozov, S. Gulwani, “Neural-guided deductive search for realtime program synthesis from examples”, *CoRR, abs/1805.04276*, 2018.

[7] Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid, “Recursive Program Synthesis” in *CAV'13 Proceedings of the 25th international conference on Computer Aided Verification*, 2013.

[8] Lisa Zhang, Gregory Rosenblatt, Ethan Fetaya, Renjie Liao, William E. Byrd, Raquel Urtasun and Richard Zemel, “LEVERAGING CONSTRAINT LOGIC PROGRAMMING FOR NEURAL GUIDED PROGRAM SYNTHESIS” in *Sixth International Conference on Learning Representations*, 2018.

[9] Emilio Parisotto, Abdel-rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, Pushmeet Kohli, “NEURO-SYMBOLIC PROGRAM SYNTHESIS” in *ICLR*, 2017.

[10] Abilio G. Parada, Eliane Siegert, Lisane B. de Brisolara, “Generating Java code from UML Class and Sequence Diagrams” in *Brazilian Symposium on Computing System Engineering*, 2011.

[11] Sunitha Edacheril Viswanathan, Philip Samuel, “Automatic code generation using unified modeling language activity and sequence models” in *Department of Computer Science, Cochin University of Science and Technology, Kochi, India, Division of Information Technology, SOE, Cochin University of Science and Technology, Kochi, India*, 2016.

[12] Muhammad Usman, and Aamer Nadeem, “Automatic Generation of Java Code from UML Diagrams using UJECTOR” in *International Journal of Software Engineering and Its Applications*, 2009.

[13] P. Yin, G. Neubig,” A Syntactic Neural Model for General-Purpose Code Generation”, 2017.

[14] Xi Victoria Lin, Chenglong Wang, Deric Pang, Kevin, Luke Ze, Michael D. Ernst, "Program Synthesis from Natural Language Using Recurrent Neural Networks" *University of Washington Department of Computer Science and Engineering*, 2017.

[15] D. A. Abolafia, M. Norouzi, J. Shen, R. Zhao, Q. V. Le, "Neural program synthesis with priority queue training", 2018.

[16] C. Wong, J. Meinicke, C. Kästner, "Beyond Testing Configurable Systems: Applying Variational Execution to Automatic Program Repair and Higher Order Mutation Testing", 2018.

[17] Jinru Hua, Mengshi Zhang, Kaiyuan Wang and Sarfraz Khurshid, "Towards practical program repair with on-demand candidate generation" in ACM/IEEE 40th International Conference on Software Engineering, 2018.

[18] Xinyun Chen, Chang Liu, Dawn Song, "TOWARDS SYNTHESIZING COMPLEX PROGRAMS FROM INPUT-OUTPUT EXAMPLES" in *ICLR*, 2018.

[19] A. Chan, "Automated Program Diversity using Program Synthesis", in *47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2017.

[20] Shao-Hua Sun, Hyeonwoo Noh, Sriram Somasundaram, Joseph J. Lim , "Neural Program Synthesis from Diverse Demonstration Videos" in *ICML*, 2018.

[21] Vijayaraghavan Murali, Letao Qi, Swarat Chaudhuri, and Chris Jermaine, "NEURAL SKETCH LEARNING FOR CONDITIONAL PROGRAM GENERATION" in *ICLR*, 2018.

[22] K. Becker, J. Gottschlich, BF-Programmer: "A counterintuitive approach to autonomously building simplistic programs using genetic algorithms", 2017.

[23] W. Ma, Z. Ni, K. Cao, X. Li, S. Chin, "Seq2Tree: A Tree-Structured Extension of LSTM Network", 2017.