# LEARNING-BASED AUTOMATED SOFTWARE CREATION

A Dissertation

by

SHANTANU MANDAL

Submitted to the Graduate and Professional School of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

| | |
|---|---|
| Chair of Committee, | Abdullah Muzahid |
| Committee Members, | Theodora Chaspari |
| | Guni Sharon |
| | Paul Gratz |
| Head of Department, | Scott Schaefer |

August 2023

Major Subject: Computer Science

# ABSTRACT

Increasing demands in software industry and scarcity of software engineers motivates researchers and practitioners to explore avenues for automating various aspects of software engineering. Automated software generation and the development of tools to streamline the software generation process are significant components among the various processes of software engineering. Therefore, we set out to investigate two major tasks of software engineering to make the process more easier and convenient for the software engineer. Our first task is automated software generation. Automated software generation, which is also known as program synthesis, is a complex and challenging task. We investigate two different sub-tasks for automatically synthesizing programs using provided specifications. In our research, we utilize input-output examples as the specification for both methods. The first sub-task involves the use of a genetic algorithm (GA) to synthesize programs. In this approach, we train a neural network-based fitness function with input-output specifications and program traces. For the second sub-task, we formulate the program synthesis process as a continuous optimization problem and leverage covariance matrix adaption evolutionary strategy (CMA-ES) to solve it. The second task of our study focuses on exploring the synthesis process of software configuration specifications from natural language text. Configurations for large software systems are typically configured by human, and due to the vast number of configurations involved, their specifications are often described in software manuals written in natural language specifically in English. Our research seeks to investigate the process of synthesizing specifications from natural language-based sources. To achieve this, we formulate the specification synthesis process as an end-to-end sequence-to-sequence learning process and integrate large language models to improve the understanding and extraction of specification from the text.

# DEDICATION

Dear Ma and Baba, thank you for everything you have done for me and for always supporting me.

This small piece of work is dedicated to you.

# ACKNOWLEDGMENTS

First and foremost, I want to express my gratitude to my amazing advisor, Abdullah Muzahid, for teaching me the fundamental principles of research. When I began my PhD journey, I had no prior research experience and felt lost navigating this new world. It is thanks to my advisor's guidance and mentorship that I am where I am today as a researcher. I am deeply grateful for everything you have done for me, Professor Muzahid.

From the beginning of my PhD journey, I have had the great opportunity to work with an amazing team from Intel, consists of Todd Anderson, Javier Turek, Justin Gottschlich, and Jesmin Jahan Tithi. I would like to express my gratitude to all of them for providing me with tremendous learning opportunities. Thank you.

I would also like to extend my appreciation to all of my lab mates. They were all incredibly helpful and made my experience in the lab more enjoyable. Additionally, I am grateful to my brothers, friends, and every person who has been there for me whenever I needed someone to listen, ask for help, or just wanted to talk. Thank you to everyone.

# CONTRIBUTORS AND FUNDING SOURCES

## Contributors

This work was supported by a dissertation committee consisting of Dr. Abdullah Muzahid (advisor), Dr. Theodora Chaspari, and Dr. Guni Sharon of the Department of Computer Science and Engineering, and Dr. Paul Gratz of the Department of Electrical and Computer Engineering.

We collaborated with a team from Intel consists of Dr. Todd Anderson, Dr. Javier Turek, Dr. Justin Gottschlich, and Dr. Jesmin Jahan Tithi. Todd Anderson and Javier Turek collaborated in Chapter 2, 3 and 4. Justin Gottschlich collaborated in Chapter 2 and 3. Jesmin Jahan Tithi collaborated in Chapter 4.

Apart from that, Adhrik Chethan, Vahid Janfaza, and S M Farabi Mahmud collaborated in Chapter 4

## Funding Sources

Graduate study was supported by Texas A&M University Faculty Startup Grant, Intel Research Grant, and NSF Grant No. 1931078.

# NOMENCLATURE

| | |
|---|---|
| ML | Machine Learning |
| NN | Neural Network |
| RNN | Recurrent Neural Network |
| LSTM | Long Short Term Memory |
| RL | Reinforcement Learning |
| GA | Genetic Algorithm |
| MP | Machine Problem |
| IO | Input Output |
| IPS | Inductive Program Synthesis |
| DSL | Domain Specific Language |
| FF | Fitness Function |
| DCE | Deadcode Elimination |
| CF | Common Function |
| LCS | Longest Common Subsequence |
| NS | Neighborhood Search |
| BFS | Breadth First Search |
| DFS | Depth First Search |
| CMA-ES | Covariance Matrix Adaption Evolutionary Strategy |
| NLP | Natural Language Processing |
| BERT | Bidirectional Encoder Representations from Transformers |
| MLM | Masked Language Model |

TABLE OF CONTENTS

Page

LIST OF FIGURES

FIGURE Page

LIST OF TABLES

# 1. INTRODUCTION

The importance of software in our daily lives cannot be overstated. However, the process of creating software is often complex and time-consuming due to the involvement of humans. The high level of human involvement in the software creation process can lead to errors and inefficiencies that can significantly impact the quality and speed of the software development lifecycle. As a result, the automation of different steps in the software creation process has become increasingly important. Automation tools can help to streamline the software creation process, reduce errors, and increase productivity. By automating repetitive and time-consuming tasks, developers can focus on more important aspects of software development, such as innovation and creativity. Therefore, the automation of software creation processes is essential to improve the efficiency and quality of software development, ultimately benefiting both developers and end-users.

To this end, in this dissertation we investigate two aspects of automatic software generation process.

- Generation: The generation aspect of software development involves the investigation of automatic software construction, also known as program synthesis.
- Facilitation: The facilitation aspect of software development involves investigating the creation of tools aimed at enabling the smooth running of a software system.

**Generation aspect:** In generation aspect, we study the automatic creation of software given some specification. We call this as automatic program synthesis. For our task we use input output example as the specification. Let $S^t = \{(I_j, O_j)\}_{j=1}^{m}$ be a set of $m$ input-output pairs. Let's assume there is a program $P^t$ which can take $I_j$ as input and produce $O_j$ as output. Assume, we do not know the target program $P^t$ but only the specification $S^t$. Now, program synthesis is defined as finding the program $P^t$ by only giving specification $S^t$.

We divide the software generation into two sub-tasks. In the first sub-task, we set out to use an off-the-self genetic algorithm for program synthesis. A genetic algorithm requires a ranking

function to rank all the solutions so that the best ones can be chosen to analyze further. We propose to use a trained neural network as a ranking function. Note that a *genetic algorithm* (GA) is a machine learning technique that attempts to solve a problem from a pool of candidate solutions. These generated candidates are iteratively evolved and mutated and selected for survival based on ranking function, often referred to as the *fitness function*. Fitness functions are usually hand-crafted heuristics that grade the approximate correctness of candidate solutions such that those that are closer to being correct are more likely to appear in subsequent generations. The project we investigated this is called as NetSyn and described in Chapter 2

For the second sub-task, we formulate the searching process of program synthesis from discrete domain to continuous domain and propose to solve it as a continuous optimization problem. Suppose a program $P$, consisting of $l$ tokens (instructions or functions), is denoted by $P = \langle P_1, ..., P_l \rangle$, where $P_i$ represents the $i$-th token for $1 \leq i \leq l$. $P_i$ can be any token from the set of all possible tokens, $\Sigma_{DSL}$. Instead of a discrete search process, we set out to investigate if an approach can be developed to map $P$ into continuous parameters that does not require any learned encoding for the formulation. Towards that end, we propose a *novel* formulation, where $P$ can be expressed as $P = \langle f_1(\cdot), f_2(\cdot), \ldots, f_M(\cdot) \rangle$. Here, each $f_i(\cdot)$, for $1 \leq i \leq M$, is a function that takes a number of continuous parameters as inputs and maps them into some tokens in $P$. With this formulation, an error function that compares the output produced by $P$ (for a specified input) with the specified output, essentially becomes a function of continuous parameters. Therefore, the problem of program synthesis amounts to minimizing the error function. In other words, program synthesis becomes a continuous optimization problem where the goal is to minimize the error function. We propose to solve this continuous optimization problem using Covariance Matrix Adaptation Evolution Strategy (CMA-ES). Commonly used error functions for comparing program outputs, such as edit or Manhattan distance [85, 14, 100], are non-smooth and ill-conditioned, i.e., a small change in the input can produce a large error. Therefore, CMA-ES is perfectly suited to solve such cases. It is a stochastic derivative-free algorithm for difficult (e.g., non-convex, ill-conditioned, multi-modal, rugged, noisy, etc.) optimization problems and considered as one of the most advanced optimization

algorithms. We investigate this in Chapter 3 and named the project as GeneSys.

**Facilitation aspect:** In the realm of facilitation aspect, our investigation focuses on the facilitation of the software creation process. This involves exploring the synthesis of software configuration specifications. Configuring a large-scale software requires the establishment of certain values or rules to accommodate various setup criteria. As the software grows in complexity, the number of configurations and criteria increases proportionally. Typically, the system administrator is responsible for setting up these configurations based on specific needs. The rules governing these configurations are presented in software manuals, which are often extensive and composed in natural language particularly in English. Consequently, system administrators may forego reading the manuals and rely on intuition to configure the software, potentially resulting in misconfigurations and software failure. Therefore, we investigate software configuration specification synthesis from software manuals that is written in natural language. We formulate the specification synthesis problem as an end-to-end squence-to-sequence learning problem. For better context understanding we also integrate large language models (LLM) for the specification synthesis purpose.

## 1.1 Contribution

Finally, in the pursuit of advancing automatic software generation, we have made the following contributions:

**Generation aspect:**

- We propose NetSyn, that use genetic algorithm with a trained neural network based fitness function to synthesize programs automatically using input output specifications.
- We propose GeneSys, that formulate program synthesis as a continuous optimization problem and use CMA-ES to synthesize programs automatically using input output specifications.

**Facilitation aspect:**

- We propose SpecSyn, that formulate software configuration specification synthesis as a sequence-to-sequence learning problem by integrating large language model.

## 1.2 Outline

We organize each of our projects into different chapters, with each chapter starting with an overview of the project. We then discuss the motivation behind the project before proceeding to the system design description. Following this, we present the results and provide a conclusion for each chapter.

# 2. NETSYN: LEARNING FITNESS FUNCTIONS FOR MACHINE PROGRAMMING

## 2.1 Overview

The problem of automatic software generation has been referred to as *machine programming*. In this work, we propose a framework based on genetic algorithms to help make progress in this domain. Although genetic algorithms (GAs) have been successfully used for many problems, one criticism is that hand-crafting GAs *fitness function*, the test that aims to effectively guide its evolution, can be notably challenging. Our framework presents a novel approach to *learn* the fitness function using neural networks to predict values of ideal fitness functions. We also augment the evolutionary process with a minimally intrusive search heuristic. This heuristic improves the framework's ability to discover correct programs from ones that are approximately correct and does so with negligible computational overhead. We compare our approach with several state-of-the-art program synthesis methods and demonstrate that it finds more correct programs with fewer candidate program generations.

## 2.2 Introduction

In recent years, there has been notable progress in the space of automatic software generation, also known as *machine programming* (MP) [42, 45, 109]. An MP system produces a program as output that satisfies some input specification to the system, often in the form of input-output examples. The previous approaches to this problem have ranged from formal program synthesis [46, 5] to machine learning (ML) [13, 153, 34, 114] as well as their combinations [40]. Genetic algorithms (GAs) have also been shown to have significant promise for MP [14, 100, 16, 73]. GA is a simple and intuitive approach and demonstrates competitive performance in many challenging domains [69, 126, 111]. Therefore, in this paper, we focus on GA - more specifically, a fundamental aspect of GA in the context of MP.

A *genetic algorithm* (GA) is a machine learning technique that attempts to solve a problem from a pool of candidate solutions. These generated candidates are iteratively evolved and mutated and

selected for survival based on a grading criteria, called the *fitness function*. Fitness functions are usually hand-crafted heuristics that grade the approximate correctness of candidate solutions such that those that are closer to being correct are more likely to appear in subsequent generations.

In the context of MP, candidate solutions are programs, initially random but evolving over time to get closer to a program satisfying the input specification. Yet, to guide that evolution, it is particularly difficult to design an effective fitness function for a GA-based MP system. The fitness function is given a candidate program and the input specification (e.g., input-output examples) and from those, must estimate how close that candidate program is to satisfying the specification. However, we know that a program having only a single mistake may produce output that in no *obvious* way resembles the correct output. That is why, one of the most frequently used fitness functions (i.e., edit-distance between outputs) in this domain [14, 100, 16, 73] will in many cases give wildly wrong estimates of candidate program correctness. Thus, it is clear that designing effective fitness functions for MP is difficult.

Designing simple and effective fitness functions is a unique challenge for GA. Despite many successful applications of GA, it still remains an open challenge to automate the generation of such fitness functions. An impediment to this goal is that fitness function complexity tends to increase proportionally with the problem being solved, with MP being particularly complex. In this paper, we explore an approach to automatically generate these fitness functions by representing their structure with a neural network. While we investigate this technique in the context of MP, we believe the technique to be applicable and generalizable to other domains. We make the following technical contributions:

- *Fitness Function:* Our fundamental contribution is in the automation of fitness functions for genetic algorithms. We propose to do so by mapping fitness function generation as a big data learning problem. To the best of our knowledge, our work is the *first* of its kind to use a neural network as a genetic algorithm's fitness function for the purpose of MP.

- *Convergence:* A secondary contribution is in our utilization of local neighborhood search to improve the convergence of approximately correct candidate solutions. We demonstrate its

efficacy empirically.

- *Generality:* We demonstrate that our approach can support different neural network fitness functions, uniformly. We develop a neural network model to predict the fitness score based on the given specification and program trace.

- *Metric:* We contribute a new metric suitable for MP domain. The metric, "search space" size (i.e., how many candidate programs have been searched), is an alternative to program generation time, and is designed to emphasize the algorithmic efficiency as opposed to the implementation efficiency of an MP approach.

### 2.3 Related Work

Machine programming can be achieved in many ways. One way is by using *formal program synthesis*, a technique that uses formal methods and rules to generate programs [88]. Formal program synthesis usually guarantees some program properties by evaluating a generated program's semantics against a corresponding specification [46, 5]. Although useful, such formal synthesis techniques can often be limited by exponentially increasing computational overhead that grows with the program's instruction size [58, 15, 121, 81, 26].

An alternative to formal methods for MP is to use machine learning (ML). Machine learning differs from traditional formal program synthesis in that it generally does not provide correctness guarantees. Instead, ML-driven MP approaches are usually only *probabilistically* correct, i.e., their results are derived from sample data relying on statistical significance [91]. Such ML approaches tend to explore software program generation using an objective function. Objective functions are used to guide an ML system's exploration of a problem space to find a solution.

More recently, there has been a surge of research exploring ML-based MP using neural networks (NNs). For example, in [12], the authors train a neural network with input-output examples to predict the probabilities of the functions that are most likely to be used in a program. Raychev et al. [110] take a different approach and use an n-gram model to predict the functions that are most likely to complete a partially constructed program. Robustfill [34] encodes input-output examples

using a series of recurrent neural networks (RNN), and generates the the program using another RNN one token at a time. Bunel et al. [17] explore a unique approach that combines reinforcement learning (RL) with a supervised model to find semantically correct programs. These are only a few of the works in the MP space using neural networks [114, 19, 21].

Significant research has been done in the field of genetic programming [100, 16, 73] whose goal is to find a solution in the form of a complete or partial program for a given specification. Prior work in this field has tended to focus on either the representation of programs or operators during the evolution process. Real et al. [112] recently demonstrated that genetic algorithms can generate accurate image classifiers. Their approach produced a state-of-the-art classifier for CIFAR-10 [71] and ImageNet [32] datasets. Moreover, genetic algorithms have been exploited to successfully automate the neural architecture optimization process [115, 126, 80, 72, 113]. Even with this notable progress, genetic algorithms can be challenging to use due to the complexity of hand-crafting fitness functions that guide the search. We claim that our proposed approach is the first of its kind to automate the generation of fitness functions.

## 2.4 Problem

## 2.5 Background

Let $S^t = \{(I_j, O_j^t)\}_{j=1}^m$ be a set of $m$ input-output pairs, such that the output $O_j^t$ is obtained by executing the program $P^t$ on the input $I_j$. Inherently, the set $S^t$ of input-output examples describes the behavior of the program $P^t$. One would like to synthesize a program $P^{t'}$ that recovers the same functionality of $P^t$. However, $P^t$ is usually unknown, and we are left with the set $S^t$, which was obtained by running $P^t$. Based on this assumption, we define equivalency between two programs as follows:

**Definition 2.5.1** (Program Equivalency)**.** Programs $P^a$ and $P^b$ are equivalent under the set $S = \{(I_j, O_j)\}_{j=1}^m$ of input-output examples if and only if $P^a(I_j) = P^b(I_j) = O_j$, for $1 \leq j \leq m$. We denote the equivalency by $P^a \equiv_S P^b$.

Definition 2.5.1 suggests that to obtain a program equivalent to $P^t$, we need to synthesize

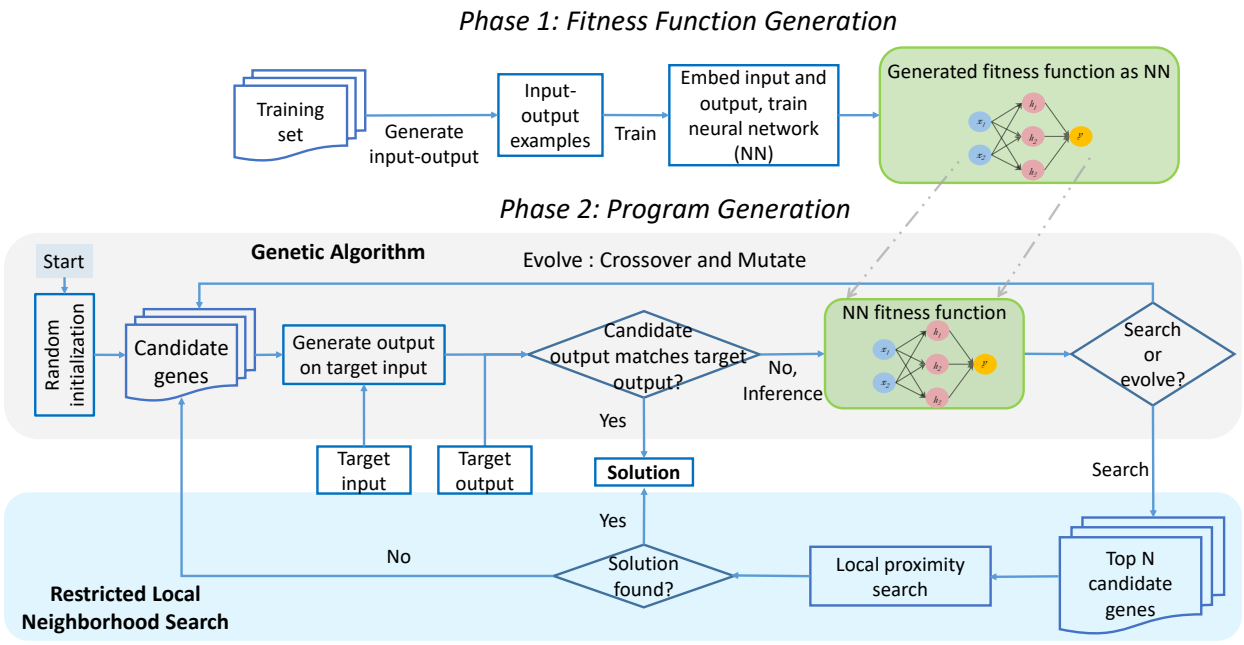



Figure 2.1: Overview of NetSyn. Phase 1 automates the fitness function generation by training a neural network on a corpus of example programs and their inputs and outputs. Phase 2 finds the target program for a given input-output example using the trained neural network as a fitness function in a genetic algorithm.

a program that is consistent with the set $S^t$. Therefore, our goal is to find a program $P^{t'}$ that is equivalent to the target program $P^t$ (which was used to generate $S^t$), i.e., $P^{t'} \equiv_{S^t} P^t$. This task is known as Inductive Program Synthesis (IPS). As suggested by [12], a machine learning based solution to the IPS problem requires the definition of some components. First, we need a programming language that defines the domain of valid programs. Second, we need a method to search over the program domain. The search method sweeps over the program domain to find $P^{t'}$ that satisfies the equivalency property. Optionally, we may want to define a ranking function to rank all the solutions found and choose the best ones. Last, as we plan to base our solution on machine learning techniques, we will need data to train models.

## 2.6 Design

## 2.7 NetSyn

Here, we describe our solution to IPS in more detail, including the choices and novelties for each of the proposed components. We name our solution NetSyn as it is based on neural networks for program synthesis.

### 2.7.1 Domain Specific Language

As NetSyn's programming language, we choose a domain specific language (DSL) constructed specifically for it. This choice allows us to constrain the program space by restricting the operations used by our solution. NetSyn's DSL follows the DeepCoder's DSL [12], which was inspired by SQL and LINQ [35]. The only data types in the language are *(i)* integers and *(ii)* lists of integers. The DSL contains 41 functions, each taking one or two arguments and returning one output. Many of these functions include operations for list manipulation. Likewise, some operations also require lambda functions. There is no explicit control flow (conditionals or looping) in the DSL. However, several of the operations are high-level functions and are implemented using such control flow structures. A full description of the DSL can be found in the supplementary material. With these data types and operations, we define a program $P$ as a sequence of functions. Table 3.1 presents an example of a program of 4 instructions with an input and respective output.

Arguments to functions are not specified via named variables. Instead, each function uses the output of the previously executed function that produces the type of output that is used as the input to the next function. The first function of each program uses the provided input $I$. If $I$ has a type mismatch, default values are used (i.e., 0 for integers and an empty list for a list of integers). The final output of a programs is the output of its last function.

As a whole, NetSyn's DSL is novel and amenable to genetic algorithms. The language is defined such that all possible programs are *valid by construction*. This makes the whole program space valid and is important to facilitate the search of programs by any learning method. In particular, this is very useful in evolutionary process in genetic algorithms. When genetic crossover occurs between

| [int] | Input: |
|---|---|
| FILTER (>0) | $[-2, 10, 3, -4, 5, 2]$ |
| MAP (*2) | |
| SORT | Output: |
| REVERSE | $[20, 10, 6, 4]$ |

Table 2.1: An example program of length 4 with an input and corresponding output.

two programs or mutation occurs within a single program, the resulting program will *always* be valid. This eliminates the need for pruning to identify valid programs.

### 2.7.2 Search Process

NetSyn synthesizes a program by searching the program space with a genetic algorithm-based method [130]. It does this by creating a population of random genes (i.e., candidate programs) of a given length $L$ and uses a learned neural network-based fitness function (NN-FF) to estimate the fitness of each gene. Higher graded genes are preferentially selected for crossover and mutation to produce the next generation of genes. In general, NetSyn uses this process to evolve the genes from one generation to the next until it discovers a correct candidate program as verified by the input-output examples. From time to time, NetSyn takes the top $N$ scoring genes from the population, determines their neighborhoods, and looks for the target program using a local proximity search. If a correctly generated program is not found within the neighborhoods, the evolutionary process resumes. Figure 3.3 summarizes the NetSyn's search process.

We use a value encoding approach for each gene. A gene $\zeta$ is represented as a sequence of values from $\Sigma_{DSL}$, the set of functions. Formally, a gene $\zeta = (f_1, \ldots, f_i, \ldots, f_L)$, where $f_i \in \Sigma_{DSL}$. Practically, each $f_i$ contains an identifier (or index) corresponding to one of the DSL functions. The encoding schemenet satisfies a one-to-one match between programs and genes.

The search process begins with a set $\Phi^0$ of $|\Phi^0| = T$ randomly generated programs. If a program equivalent to the target program $P^t$ is found, the search process stops. Otherwise, the genes are ranked using a learned NN-FF. A small percentage (e.g., 20%) of the top graded genes in $\Phi^j$ are passed in an unmodified fashion to the next generation $\Phi^{j+1}$ for the next evolutionary phase. This

guarantees that some of the top graded genes are identically preserved, aiding in forward progress guarantees. The remaining genes of the new generation $\Phi^{j+1}$ are created through crossover or mutation with some probability. For crossover, two genes from $\Phi^{j}$ are selected using the Roulette Wheel algorithm with the crossover point selected randomly [44]. For mutation, one gene is Roulette Wheel selected and the mutation point $k$ in that gene is selected based on the same learned NN-FF. The selected value $z_k$ is mutated to some other random value $z'$ such that $z' \in \Sigma_{DSL}$ and $z' \neq z_k$.

Crossovers and mutations can occasionally lead to a new gene with dead code. To address this issue, we eliminate dead code. Dead code elimination (DCE) is a classic compiler technique to remove code from a program that has no effect on the program's output [31]. Dead code is possible in our list DSL if the output of a statement is never used. We implemented DCE in NetSyn by tracking the input/output dependencies between statements and eliminating those statements whose outputs are never used. NetSyn uses DCE during candidate program generation and during crossover/mutation to ensure that the effective length of the program is not less than the target program length due to the presence of dead code. If dead code is present, we repeat crossover and mutation until a gene without dead code is produced.

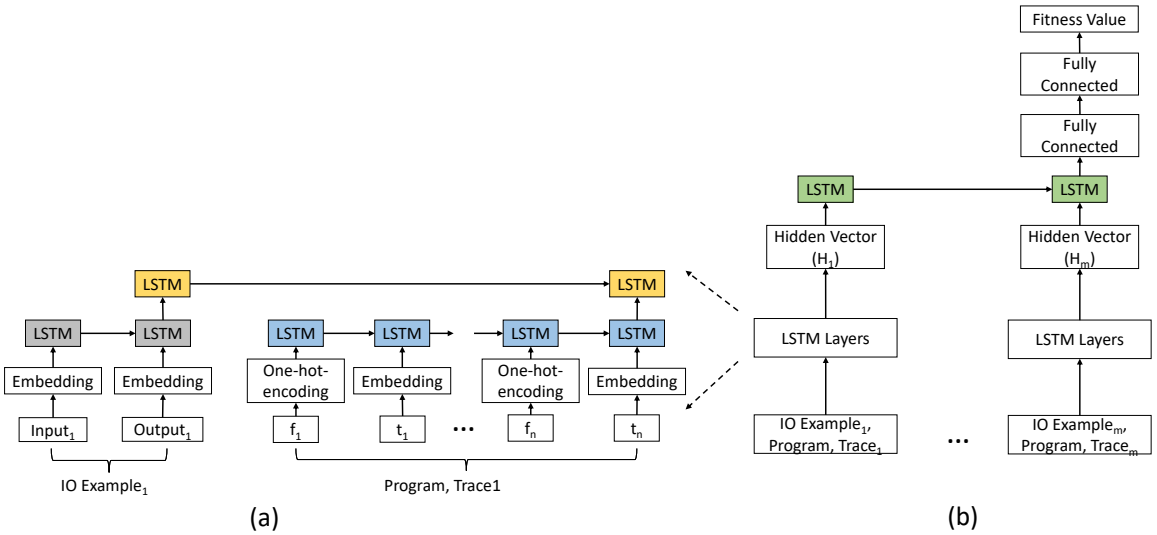



Figure 2.2: Neural network fitness function for (a) single and (b) multiple IO examples. In each figure, layers of LSTM encoders are used to combine multiple inputs into hidden vectors for the next layer. Final fitness score is produced by the fully connected layer.

#### *2.7.2.1 Learning the Fitness Function*

Evolving the population of genes in a genetic algorithm requires a fitness function to rank the fitness (quality) of genes based on the problem being solved. Ideally, a fitness function should measure how close a gene is to the solution. Namely, it should measure how close a candidate program is to an equivalent of $P^t$ under $S^t$. Finding a good fitness function is of great importance to reduce the number of steps in reaching the solution and directing the algorithm in the right direction so that genetic algorithm are more likely to find $P^t$.

**Intuition:** A fitness function, often, is handcrafted to approximate some ideal function that is impossible (due to incomplete knowledge about the solution) or too computationally intensive to implement in practice. For example, if we knew $P^t$ beforehand, we could have designed an ideal fitness function that compares a candidate program with $P^t$ and calculates some metric of closeness (e.g., edit distance, the number of common functions etc.) as the fitness score. Since we do not know $P^t$, we cannot implement the ideal fitness function. Instead, in this work, we propose to approximate the ideal fitness function by learning it from training data (generated from a number of known programs). For this purpose, we use a neural network model. We train it with the goal of predicting the values of an ideal fitness function. We call such an ideal fitness function (that would always give the correct answer with respect to the actual solution) the *oracle* fitness function as it is impossible to achieve in practice merely by examining input-output examples. In this case, our models will not be able to approach the 100% accuracy of the *oracle* but rather will still have sufficiently high enough accuracy to allow the genetic algorithm to make forward progress. Also, we note that the trained model needs to generalize to predict for any unavailable solution and not a single specific target case.

We follow ideas from works that have explored the automation of fitness functions using neural networks for approximating a known mathematical model. For example, Matos Dias et al. [89] automated them for IMRT beam angle optimization, while Khuntia et al. [68] used them for rectangular microstrip antenna design automation. In contrast, our work is fundamentally different in that we use a large corpus of program metadata to train our models to predict how close a given,

incorrect solution could be from an *unknown* correct solution (that will generate the correct output). In other words, we propose to automate the generation of fitness functions using big data learning. To the best of our knowledge, NetSyn is the ***first*** proposal for automation of fitness functions in genetic algorithms. In this paper, we demonstrate this idea using MP as the use case.

Given the input-output samples $S^t = \left\{\left(I_j, O_j^t\right)\right\}_j$ of the target program $P^t$ and an ideal fitness function $fit(\cdot)$, we would like a model that predicts the fitness value $fit(\zeta, P^t)$ for a gene $\zeta$. In practice, our model predicts the values of $fit(\cdot)$ from input-output samples in $S^t$ and from execution traces of the program $P^\zeta$ (corresponding to $\zeta$) by running with those inputs. Intuitively, execution traces provide insights of whether the program $P^\zeta$ is on the right track.

In NetSyn, we use a neural network to model the fitness function, referred to as NN-FF. This task requires us to generate a training dataset of programs with respective input-output samples. To train the NN-FF, we randomly generate a set of example programs, $E = \{P^{e_j}\}$, along with a set of random inputs $I^j = \{I_i^{e_j}\}$ per program $P^{e_j}$. We then execute each program $P^{e_j}$ in $E$ with its corresponding input set $I^j$ to calculate the output set $O^j$. Additionally, for each $P^{e_j}$ in $E$, we randomly generate another program $P^{r_j} = (f_1^{r_j}, f_2^{r_j}, ..., f_n^{r_j})$, where $f_k^{r_j}$ is a function from the DSL i.e., $f_k^{r_j} \in \Sigma_{DSL}$. We apply the previously generated input $I_i^{e_j}$ to $P_j^r$ to get an execution trace, $T_i^{rj} = (t_{i1}^{rj}, t_{i2}^{rj}, ..., t_{in}^{rj})$, where $t_{ik}^{rj} = f_k^{rj}(t_{i(k-1)}^{rj})$ with $t_{i1}^{rj} = f_1^{rj}(I_i^{e_j})$ and $t_{in}^{rj} = f_n^{rj}(t_{i(n-1)}^{rj}) = P^{r_j}(I_i^{e_j})$. Thus, the input set $I^j = \{I_i^{e_j}\}$ of the program $P^{e_j}$ produces a set of traces $T^j = \{T_i^{r_j}\}$ from the program $P^{r_j}$. We then compare the programs $P^{r_j}$ and $P^{e_j}$ to calculate the fitness value and use it as an example to train the neural network.

In NetSyn, the inputs of NN-FF consist of input-output examples, generated programs, and their execution traces. Let us consider a single input-output example, $(I_i^{e_j}, O_i^{e_j})$. Let us assume that $P^{e_j}$ is the target program that NetSyn attempts to generate and in the process, it generates $P^{r_j}$ as a potential equivalent. NN-FF uses $(I_i^{e_j}, O_i^{e_j})$, and $\{(f_k^{r_j}, t_{ik}^{r_j})\}$ as inputs for this example. Each of $(I_i^{e_j}, O_i^{e_j})$, and $t_{ik}^{r_j}$ are passed through an embedding layer followed by an LSTM encoder. $f_k^{r_j}$ is passed as a one-hot-encoding vector. Figure 4.2(a) shows the details of how a single input-output example is processed. Two layers of LSTM encoders combines the vectors to produce a single vector, $H_i^j$. In

order to handle a set of input-output examples, $\{(I_i^{e_j}, O_i^{e_j})\}$, a set of execution traces, $T^j = \{T_i^{r_j}\}$, is collected from a single generated program, $P^{r_j}$. Each input-output example, $(I_i^{e_j}, O_i^{e_j})$, along with the corresponding execution trace produces a single vector, $H_i^j$. An LSTM encoder combines such vectors to produce a single vector, which is then processed by fully connected layers to predict the fitness value (Figure 4.2(b)).

**Example:** To illustrate, suppose the program in Table 3.1 is in $E$. Let us assume that $P^{r_j}$ is another program {[INT], FILTER (>0), MAP (*2), REVERSE, DROP (2)}. If we use the input in Table 3.1 (i.e., $[-2, 10, 3, -4, 5, 2]$) with $P^{r_j}$, the execution trace is $\{[10, 3, 5, 2]$, $[20, 6, 10, 4]$, $[4, 10, 6, 20]$, $[6, 20]\}$. So, the input of NN-FF is $\{[-2, 10, 3, -4, 5, 2], [20, 10, 6, 4]$, $Filter_v$, $[10, 3, 5, 2]$, $Map_v$, $[20, 6, 10, 4]$, $Reverse_v$, $[4, 10, 6, 20]$, $Drop_v$, $[6, 20]\}$. $f_v$ indicates the value corresponding to the function $f$.

There are different ways to quantify how close two programs are to one another. Each of these different methods then has an associated metric and ideal fitness value. We investigated three such metrics – common functions, longest common subsequence, and function probability – which we use as the expected predicted output for the NN-FF.

**Common Functions:** NetSyn can use the number of common functions (CF) between $P^\zeta$ and $P^t$ as a fitness value for $\zeta$. In other words, the fitness value of $\zeta$ is $f_{P^t}^{CF}(\zeta) = |\mathbf{elems}(P^\zeta) \cap \mathbf{elems}(P^t)|$. For the earlier example, $f^{CF}$ will be 3. Since the output of the neural network will be an integer from 0 to $\mathbf{len}(P_t)$, the neural network can be designed as a multiclass classifier with a softmax layer as the final layer.

**Longest Common Subsequence:** As an alternative to CF, we can use longest common subsequence (LCS) between $P^\zeta$ and $P^t$. The fitness score of $\zeta$ is $f_{P^t}^{LCS}(\zeta) = \mathbf{len}(LCS(P^\zeta, P^t))$. Similar to CF, training data can be constructed from $E$ which is then fed into a neural network-based multiclass classifier. For the earlier example, $f^{LCS}$ will be 2.

**Function Probability:** The work [12] proposed a probability map for the functions in the DSL. Let us assume that the probability map $\mathbf{p}$ is defined as the probability of each DSL operation to be in $P^t$ given the input-output samples. Namely, $\mathbf{p} = (p_1, \ldots, p_k, \ldots, p_{|\Sigma_{DSL}|})$ such that

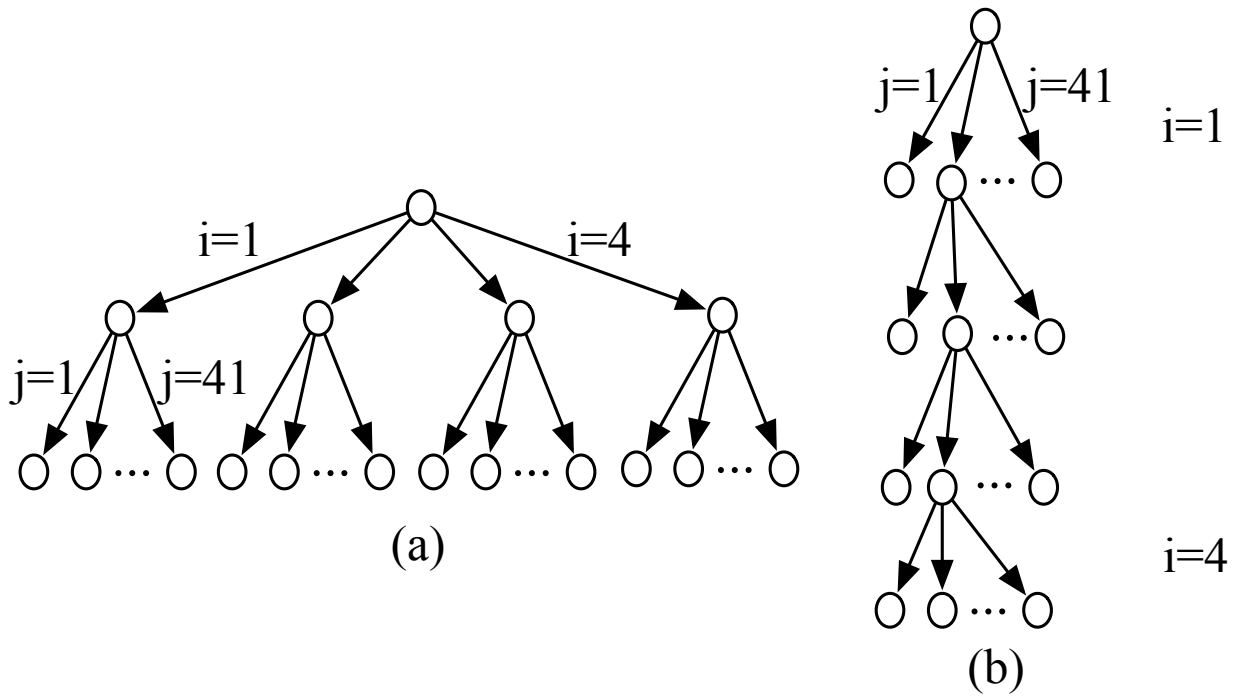



Figure 2.3: Example of neighborhood for a 4-length gene using (a) BFS- and (b) DFS-based approach.

$p_k = Prob(\text{op}_k \in \textbf{elems}(P^t)|\{(I_j, O_j^t)\}_{j=1}^m)$, where $\text{op}_k$ is the $k^{th}$ operation in the DSL. Then, a multiclass, multilabel neural network classifier with sigmoid activation functions used in the output of the last layer can be used to predict the probability map. Training data can be constructed for the neural network using $E$. We can use the probability map to calculate the fitness score of $\zeta$ as $f_{P^t}^{FP}(\zeta) = \sum_{k:\text{op}_k \in \textbf{elems}(P^\zeta)} p_k$. NetSyn also uses the probability map to guide the mutation process. For example, instead of mutating a function $z_k$ with $z'$ that is selected randomly, NetSyn can select $z'$ using Roulette Wheel algorithm using the probability map.

#### *2.7.2.2 Local Neighborhood Search*

Neighborhood search (NS) checks some candidate genes in the *neighborhood* of the $N$ top scoring genes from the genetic algorithm. The intuition behind NS is that if the target program $P^t$ is in that neighborhood, NetSyn may be able to find it without relying on the genetic algorithm, which would likely result in a faster synthesis time.

Let us assume that NetSyn has completed $l$ generations. Then, let $\mu_{l-w+1,l}$ denote the average fitness score of genes for the last $w$ generations (i.e., from $l-w+1$ to $l$) and $\mu_{1,l-w}$ will denote the average fitness score before the last $w$ generations (i.e., from 1 to $l-w$). Here, $w$ is the sliding window. NetSyn invokes NS if $\mu_{l-w+1,l} \leq \mu_{1,l-w}$. The rationale is that under these conditions, the search procedure has not produced improved genes for the last $w$ generations (i.e., saturating). Therefore, it should check if the neighborhood contains any program equivalent to $P^t$.

**Algorithm 1:** Defines and searches neighborhood based on BFS principle

**Input:** A set $G$ of top $N$ scoring genes
**Output:** $P^{t'}$, if found, or *Not found* otherwise

```
1 for Each ζ ∈ G do
2     NH ← ∅
3     for i ← 1 to len(ζ) do
4         for j ← 1 to|Σ_DSL| do
5             ζ_n ← ζ with ζ_i replaced with op_j   such that ζ_i ≠ op_j
6             NH ← NH ∪ {ζ_n}
7     if there is P^t' ∈ NH such that P^t' ≡_S^t P^t then
8         return P^t'
9 return Not found
```

**Neighborhood Definition:** Algorithm 1 shows how to define and search a neighborhood. The algorithm is inspired by the breadth first search (BFS) method. For each top scoring gene $\zeta$, NetSyn considers one function at a time starting from the first operation of the gene to the last one. Each selected operation is replaced with all other operations from $\Sigma_{DSL}$, and inserts the resultant genes into the neighborhood set $NH$. If a program $P^{t'}$ equivalent to $P^t$ is found in $NH$, NetSyn stops there and returns the solution. Otherwise, it continues the search and returns to the genetic algorithm. The complexity of the search is $\mathcal{O}(N \cdot \mathbf{len}(\zeta) \cdot |\Sigma_{DSL}|)$, which is significantly smaller than the exponential search space used by a traditional BFS algorithm. Similar to BFS, NetSyn can define and search the neighborhood using an approach similar to depth first search (DFS). It is similar to Algorithm 1 except $i$ keeps track of depth here. After the loop in line 4 finishes, NetSyn picks the best scoring gene from $NH$ to replace $\zeta$ before going to the next level of depth. The

algorithmic complexity remains the same. Figure 2.3 (a) and (b) show examples of neighborhood using BFS- and DFS-based approach.

## 2.8 Results

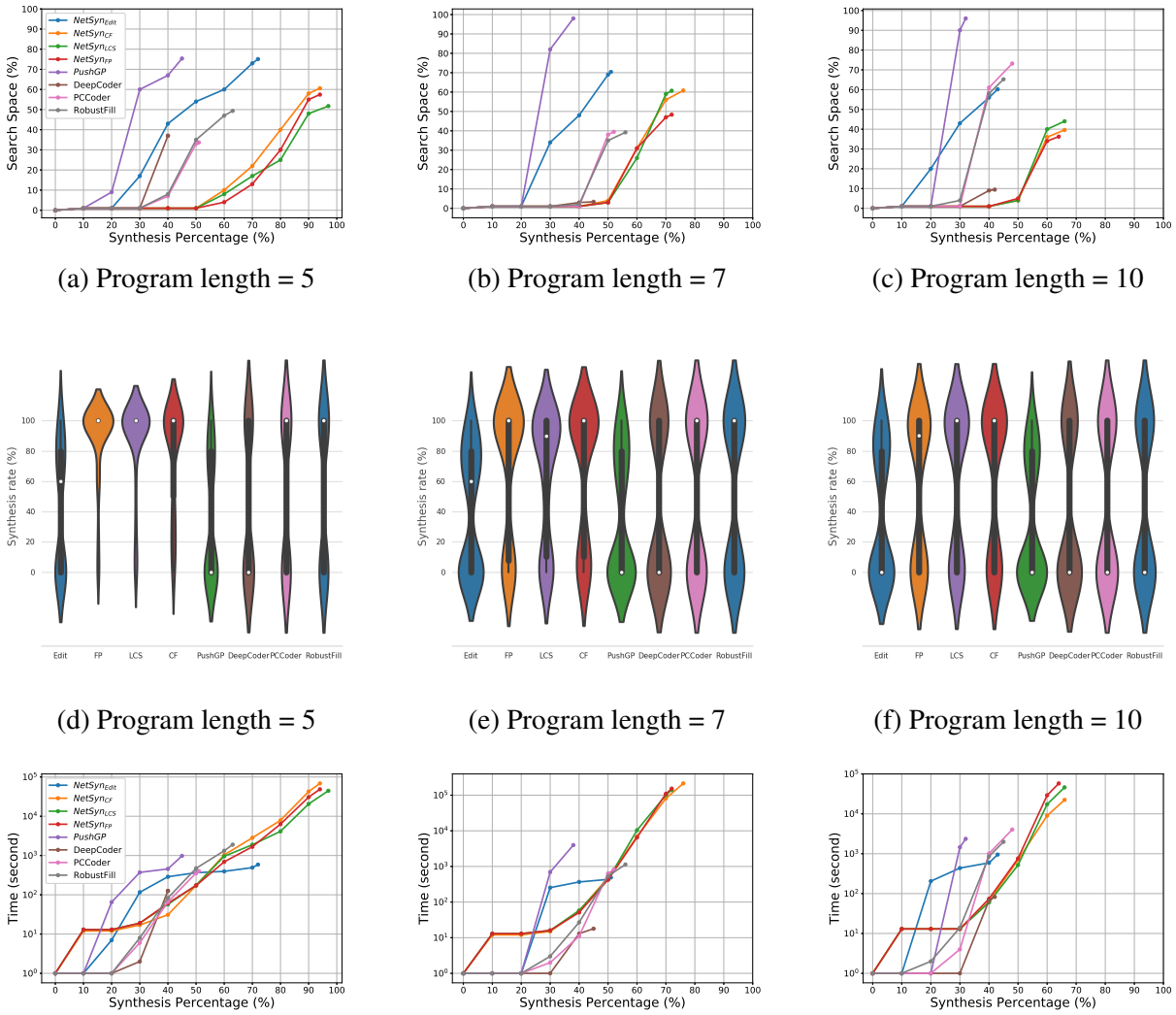
## 2.9 Experimental Results

We implemented NetSyn in Python with a TensorFlow backend [2]. We developed an interpreter for NetSyn's DSL to evaluate the generated programs. We used 4,200,000 randomly generated unique example programs of length 5 to train the neural networks. We used 5 input-output examples for each program to generate the training data. To allow our model to predict equally well across all possible CF/LCS values, we generate these programs such that each of the 0-5 possible CF/LCS values for 5 length programs are equally represented in the dataset. To test NetSyn, we randomly generated a total of 100 programs for each program length from 5 to 10. For each program length, 50 of the generated programs produce a singleton integer as the output; the rest produce a list of integers. We therefore refer to the first 50 programs as *singleton programs* and the rest as *list programs*. We collected $m = 5$ input-output examples for each testing program. When synthesizing a program using NetSyn, we execute it $K = 10$ times and average the results to eliminate any noise.

### 2.9.1 Demonstration of Synthesis Ability

We ran three variants of NetSyn - $NetSyn_{CF}$, $NetSyn_{LCS}$, and $NetSyn_{FP}$, each predicting $f^{CF}$, $f^{LCS}$, and $f^{FP}$ fitness functions, respectively. Each used $NS^{BFS}$ and FP-based mutation operation. We ran the publicly available best performing implementations of DeepCoder [12], PCCoder [153], and RobustFill [34]. We also implemented a genetic programming-based approach, PushGP [100]. For comparison, we also tested two other fitness functions: 1) edit-distance between outputs ($f^{Edit}$), and 2) the oracle ($f^{Oracle}$). For every approach, we set the maximum search space size to 3,000,000 candidate programs. If an approach does not find the solution before reaching that threshold, we conclude the experiment marking it as *"solution not found"*.

Figure 3.10(a) - (c) show comparative results using the proposed metric: *search space* used. For each test program, we count the number of candidate programs searched before the experiment

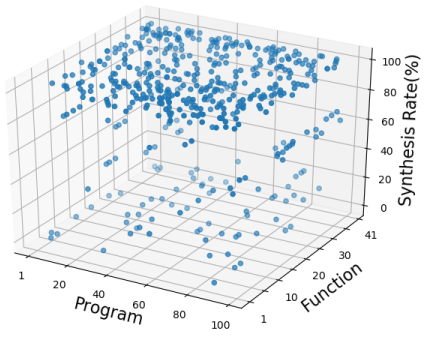



(a) Program length = 5 (b) Program length = 7 (c) Program length = 10

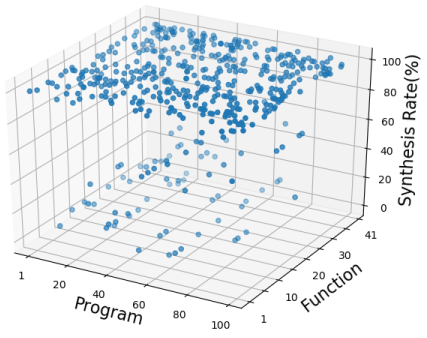
(d) Program length = 5 (e) Program length = 7 (f) Program length = 10

(g) Program length = 5 (h) Program length = 7 (i) Program length = 10

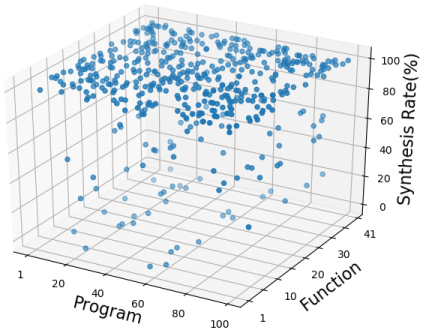
Figure 2.4: NetSyn's synthesis ability with respect to different fitness functions and schemes. When limited by a maximum search space, NetSyn synthesizes more programs than DeepCoder, PCCoder, RobustFill, and PushGP. Moreover for each program, NetSyn synthesizes a higher percentage of runs than other approaches.




(a) $NetSyn_{CF}$




(b) $NetSyn_{LCS}$




(c) $NetSyn_{FP}$

Figure 2.5: NetSyn's synthesis ability with respect to fitness functions and DSL function types. Programs producing a single integer output are harder to synthesize in all three variants of NetSyn.

has concluded by either finding a correct program or exceeding the threshold. The number of candidate programs searched is expressed as a percentage of the maximum search space threshold, i.e., 3,000,000 and shown in y-axis. We sort the time taken to synthesize the programs. A position *N* on the X-axis corresponds to the program synthesized in the *Nth* longest percentile time of all the programs. Lines terminate at the point at which the approach fails to synthesize the corresponding program. For all approaches, except for $f^{Edit}$-based NetSyn and PushGP, up to 30% of the programs can be synthesized by searching less than 2% of the maximum search space. Search space use increases when an approach tries to synthesize more programs. In general, DeepCoder, PCCoder, and RobustFill search more candidate programs than $f^{CF}$, $f^{LCS}$ or $f^{FP}$-based NetSyn. For example, for synthesizing programs of length 5, DeepCoder, PCCoder and RobustFill use 37%, 33%, and 47% search space to synthesize 40%, 50%, and 60% programs, respectively. In comparison, NetSyn can synthesize upwards of 90% programs by using less than 60% search space. NetSyn synthesizes programs at percentages ranging from 65% (in case of $NetSyn_{FP}$ for 10 length programs) to as high as 97% (in case of $NetSyn_{LCS}$ for 5 length programs). In other words, NetSyn is more efficient in generating and searching likely target programs. Even for length 10 programs, NetSyn can generate 65% of the programs using less than 45% of the maximum search space. In contrast, DeepCoder, PCCoder, and RobustFill cannot synthesize more than 60% of the programs even if they use the maximum search space. PushGP and edit distance-based approaches always use more search space than $f^{CF}$ or $f^{LCS}$.

Figure 3.10(d) - (f) show the distribution of synthesis rate (i.e., what percentage of $K = 10$ runs synthesizes a particular program) in violin plots. A violin plot shows interquartile range (i.e., middle 50% range) as a vertical black bar with the median as a white dot. Moreover, wider section of the plot indicates more data points in that section. For 5 length programs, NetSyn has a high synthesis rate (close to 100%) for almost every program (as indicated by one wide section). On the other hand, DeepCoder, PCCoder, RobustFill, and PushGP have bimodal distributions as indicated by two wide sections. At higher lengths, NetSyn synthesizes around 65% to 75% programs and therefore, the distribution becomes bimodal with two wide sections. However, the section at the top

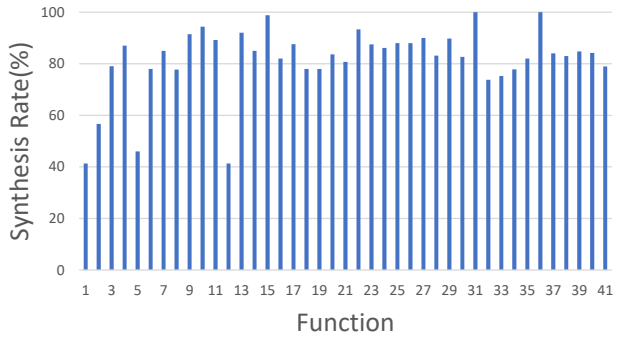
is wider indicating that NetSyn maintains high synthesis rate for the successful cases. DeepCoder, PCCoder, RobustFill, and PushGP have more unsuccessful cases than the successful ones. However, for the successful cases, these approaches also have high synthesis rates.

Figure 3.10(g) - (i) show comparative results using synthesis time as the metric. In general, DeepCoder, PCCoder, RobustFill and NetSyn can synthesize up to 20% programs within a few seconds for all program lengths we tested. As expected, synthesis time increases as an approach attempts to synthesize more difficult programs. DeepCoder, PCCoder, and RobustFill usually find solutions faster than NetSyn. It should be noted that the goal of NetSyn is to synthesize a program with as few tries as possible. Therefore, the implementation of NetSyn is not streamlined to take advantage of various parallelization and performance enhancement techniques such as GPUs, hardware accelerators, data parallel models etc. The synthesis time tends to increase for longer length programs.

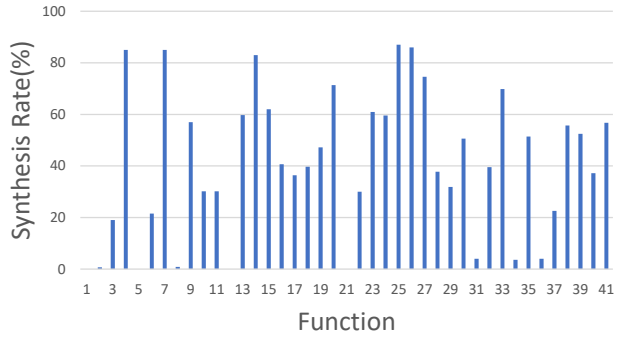
To asses generalizability of the proposed NN, we conducted several experiments - (i) we experimented with 2 new sets of test programs having different Gaussian distributions of functions, and (ii) we tested with a different order of IO examples. For 5-length programs, the synthesis rate is found to be within 73% to 92% with a search space of 8.5% to 28.6%. Although there is some variability in synthesis rate and search space utilization, no consistent patterns emerge across different configurations. It is difficult to interpret any difference in synthesis rate as an effect of generalization since different functions have differing synthesis difficulties. When we change the distribution, we also change synthesis difficulty. We also performed these tests for 7 and 10-length programs and found that the variability diminishes as we increase program length. Moreover, the average synthesis rates for the newer configurations tend to be higher. We believe this to be evidence that our approach generalizes and the complete range of results are still better than any prior state-of-the-art.

### 2.9.2 Characterization of NetSyn

Next, we characterize the effect of different components of NetSyn. We show the results in this section based on programs of length 5. However, we found our general observations to be true for




(a) CF




(b) FP

Figure 2.6: Synthesis percentage across different functions. Functions 1 to 12 tend to have a lower synthesis rate because they produce a single integer output. Moreover, $f^{CF}$ has a higher synthesis rate.

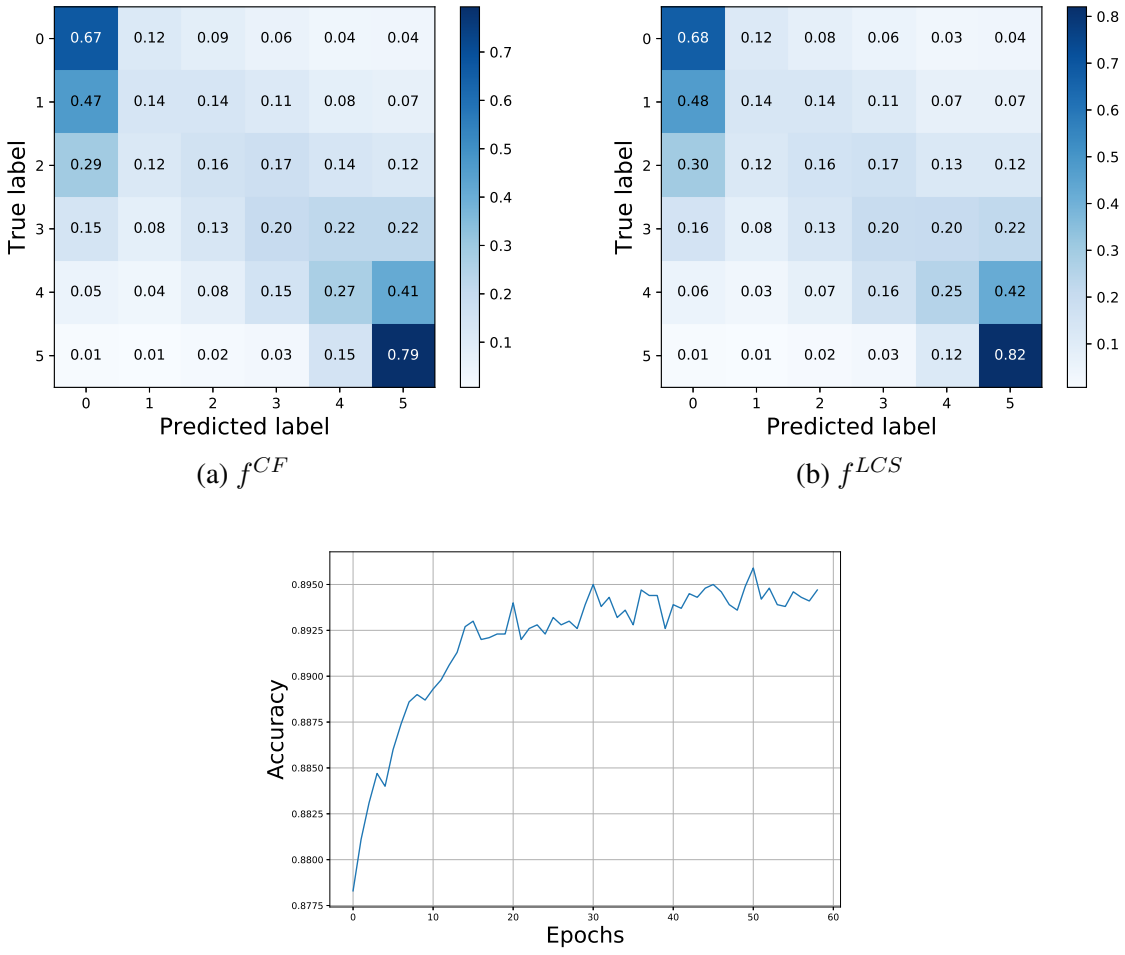
longer length programs also.

| Approach | Programs Synthesized | Avg Generation | Avg Syn. Rate (%) |
|---|---|---|---|
| $GA + f^{CF}$ | 92 | 3273 | 74 |
| $GA + f^{CF} + NS^{BFS}$ | 94 | 2953 | 77 |
| $GA + f^{CF} + NS^{DFS}$ | 94 | 3026 | 76 |
| $GA + f^{CF} + Mutation^{FP}$ | 93 | 2726 | 83 |
| $GA + f^{CF} + NS^{BFS} + Mutation^{FP}$ | 94 | 2275 | 85 |

Table 2.2: Programs synthesized for different settings of NetSyn. GA stands for genetic algorithm.

Table 2.2 shows how many unique programs of $length = 5$ (out of a total of 100 programs) that the different approaches were able to synthesize. It also shows the average generations and synthesis rate for each program. NetSyn synthesized the most number of programs in the lowest number of generations and at the highest rate of synthesis when both the NS and improved mutation based on function probability ($Mutation^{FP}$) are used in addition to the the NN-FF. We note that BFS-based NS performs slightly better than DFS-based NS. Moreover, $Mutation^{FP}$ has some measurable impact on NetSyn. Figure 2.5(a) - (c) show the synthesis rate for different programs and fitness functions. Program 1 to 50 are singleton programs and have lower synthesis rate in all three fitness function choices. Particularly, the $f^{FP}$-based approach has a low synthesis rate for singleton programs. Functions 1 to 12 produce singleton integer and tend to cause lower synthesis rate for any program that contains them. This implies that singleton programs are relatively harder to synthesize.

To shed more light on this issue, Figure 2.6 shows synthesis rate across different functions. The synthesis rate for a function is at least 40% for the $f_{CF}$-based approach, whereas for the $f_{FP}$-based approach, four functions cannot be synthesized at all. Details of functions are in the appendix.




(a) $f^{CF}$ (b) $f^{LCS}$

(c) $f^{FP}$

Figure 2.7: Confusion matrix of (a) $f^{CF}$ (b) $f^{LCS}$ neural network fitness functions. (c) shows accuracy of $f^{FP}$ over epochs. All graphs are based on the validation data. Overall, $f^{CF}$ and $f^{LCS}$ are capable identifying of close-enough solutions as well as mostly mistaken solutions. $f^{FP}$ reaches close to 90% accuracy after 40 epochs.

### 2.9.3 Characterization of Neural Networks

Figure 2.7(a), (b), and (c) show the prediction ability of our proposed neural network fitness functions on validation data. Figure 2.7(a) & (b) show the confusion matrix for $f^{CF}$ and $f^{LCS}$ neural network fitness functions. The confusion matrix is a two dimensional matrix where $(i, j)$ entry indicates the probability of predicting the value $i$ when the actual value is $j$. Thus, each row of the matrix sums up to 1.0. We can see that when a candidate program is close to the solution (i.e., the fitness score is 4 or above), each of $f^{CF}$ and $f^{LCS}$-based model predicts a fitness score of 4 or higher with a probability of 0.7 or higher. In other words, the models are very accurate in identifying potentially close-enough solutions. Similar is the case when the candidate program is mostly mistaken (i.e., a fitness score is 1 or less). Thus, the neural networks are good at identifying both close-enough solutions and mostly wrong solutions. If a candidate program is some what correct (i.e., the candidate program has few correct functions but the rest of the functions are incorrect), it is difficult to identify them by the proposed models.

$f^{FP}$ model predicts probability of different functions given the IO examples. We assume a function probability to be correct if the function is in the target program and the neural network predicts its probability as 0.5 or higher. Figure 2.7(c) shows the accuracy of $f^{FP}$ model. With enough epochs, it reaches close to 90% accuracy on the validation data set.

#### *2.9.3.1 Additional Models and Fitness Functions*

We tried several other models for neural networks and fitness functions. For example, instead of a classification problem, we treated fitness scores as a regression problem. We found that the neural networks produced higher prediction error as the networks had a tendency to predict values close to the median of the values in the training set. With the higher prediction errors of the fitness function, the genetic algorithm performance degraded.

We also experimented with training a network to predict a correctness ordering among a set of genes. We note that the ultimate goal of the fitness score is to provide an order among genes for the Roulette Wheel algorithm. Rather than getting this ordering indirectly via a fitness score for each

gene, we attempted to have the neural network predict this ordering directly. However, we were not able to train a network to predict this relative ordering whose accuracy was higher than the one for absolute fitness scores. We believe that there are other potential implementations for this relative ordering and that it may be possible for it to be made to work in the future.

Additionally, we tried a two-tier fitness function. The first tier was a neural network to predict whether a gene has a fitness score of 0 or not. In the event the fitness score was predicted to be non-zero, we used a second neural network to predict the actual non-zero value. This idea came from the intuition that since many genes have a fitness score of 0 (at least for initial generations), we can do a better job predicting those if we use a separate predictor for that purpose. Unfortunately, mispredictions in the first tier caused enough good genes to be eliminated that NetSyn's synthesis rate was reduced.

Finally, we explored training a *bigram* model (i.e., predicting pairs of functions appearing one after the other). This approach is complicated by the fact that over 99% of the $41 \times 41$ (i.e., number of DSL functions squared) bigram matrix are zeros. We tried a two-tiered neural network and principle component analysis to reduce the dimensionality of this matrix [75]. Our results using this bigram model in NetSyn were similar to that of DeepCoder, with up to 90% reduction in synthesis rate for singleton programs.

## 2.10 Conclustion

In this paper, we presented a genetic algorithm-based framework for program synthesis called NetSyn. To the best of our knowledge, it is the first work that uses a neural network to automatically generate an genetic algorithm's fitness function in the context of machine programming. We proposed three neural network-based fitness functions. NetSyn is also novel in that it uses neighborhood search to expedite the convergence process of a genetic algorithm. We compared our approach against several state-of-the art program synthesis systems - DeepCoder [12], PCCoder [153], RobustFill [34], and PushGP [100]. NetSyn synthesizes more programs than each of those prior approaches with fewer candidate program generations. We believe that our proposed work could open up a new direction of research by automating fitness function generations for

genetic algorithms by mapping the problem as a big data learning problem. This has the potential to improve any application of genetic algorithms.

# 3. GENESYS: SYNTHESIZING PROGRAMS WITH CONTINUOUS OPTIMIZATION

## 3.1 Overview

Automatic software generation based on some specification is known as *program synthesis.* Most existing approaches formulate program synthesis as a search problem with discrete parameters. In this paper, we present a *novel* formulation of program synthesis as a continuous optimization problem and use a state-of-the-art evolutionary approach, known as Covariance Matrix Adaptation Evolution Strategy to solve it. We then propose a mapping scheme to convert the continuous formulation into actual programs. We compare our system, called GeneSys, with several recent program synthesis techniques (in both discrete and continuous domains) and show that GeneSys synthesizes more programs within a fixed time budget than those existing schemes. For example, for programs of length 10, GeneSys synthesizes 28% more programs than those existing schemes within the same time budget.

## 3.2 Introduction

Program synthesis, a subset of the field of machine programming (MP) [45, 109], aims to automatically generate a software program from some specification and has the potential to revolutionize how we develop programs. Such systems typically produce a program as an output that satisfies some input specifications, classically in the form of input-output examples. At a high level, most existing techniques formulate program synthesis as a search problem in the discrete domain. Typically, the search problem is solved either through formal methods [46, 5, 15, 26, 58, 81, 121] , machine learning approaches [12, 153, 34, 114, 85, 100, 22, 17, 16, 73, 100] or a combination of both [40, 95]. Also, some prior art [120, 118] has framed the problem in the continuous domain. Si et al. [120] formulates discrete semantics to a continuous one by annotating program rules with different weights and solve it using Markov Chain Monte Carlo technique. Recently, Neural Program Optimization (NPO) [79] formulate program synthesis in the continuous domain using an autoencoder [70] generated latent representations. Although these continuous approaches have

shown promising early results for a simple problem set (e.g., programs similar to simple digital circuits such as comparator, multiplexer, etc. in case of NPO), they fail to deliver any significant benefit for a complex problem set (see Section 3.7.4). Therefore, in this paper, we set out to investigate how these promising results in the continuous domain can be extended to more complex problems.

Suppose a program $P$, consisting of $l$ tokens (instructions or functions), is denoted by $P = \langle P_1, ..., P_l \rangle$, where $P_i$ represents the $i$-th token for $1 \leq i \leq l$. $P_i$ can be any token from the set of all possible tokens, $\Sigma_{DSL}$. Given some input-output examples as a specification, most existing works find a program $P$ that satisfies the specification by searching through different programs resulting from different combinations of various tokens. Since $P_i$ is limited to a fixed number of distinct tokens from $\Sigma_{DSL}$, this formulation can be categorized as a discrete search problem. Various approaches differ in the ways they prune the search space while still being able to find $P$ [12, 153, 23, 60]. Recently, NPO proposed converting $P$ into a latent representation in the continuous domain using the encoder of an autoencoder model [79]. In this paper, we propose a *novel* formulation where $P$ can be expressed as $P = \langle f_1(\cdot), f_2(\cdot), \ldots, f_M(\cdot) \rangle$. Here, each $f_i(\cdot)$, for $1 \leq i \leq M$, is a function that takes a number of continuous parameters as inputs and maps them into some tokens in $P$. With this formulation, an error function that compares the output produced by $P$ (for a specified input) with the specified output, essentially becomes a function of continuous parameters. Therefore, the problem of program synthesis amounts to minimizing the error function but as a *continuous optimization* problem.

We propose to solve this continuous optimization problem using Covariance Matrix Adaptation Evolution Strategy (CMA-ES). Commonly used error functions for comparing program outputs, such as edit or Manhattan distance [85, 14, 100], are non-smooth and ill-conditioned, i.e., a small change in the input can produce a large error. Therefore, CMA-ES is perfectly suited to solve such cases. It is a stochastic derivative-free algorithm for difficult (e.g., non-convex, ill-conditioned, multi-modal, rugged, noisy, etc.) optimization problems and considered as one of the most advanced optimization algorithms with many successful applications [53]. We refer to the proposed program

synthesis framework as GeneSys.

GeneSys uses a multivariate normal distribution for generating potential solutions. During each generation, GeneSys takes a number of samples from the distribution. Using the proposed *novel* mapping scheme, GeneSys converts the samples of continuous parameters into actual programs. These candidate programs are used to evaluate the error function. If a candidate program has no error, the target program is found. Otherwise, GeneSys starts the next generation of evolution. For this generation, a new mean for the distribution is calculated based on the error from the past generation. The covariance matrix, which captures dependencies among the continuous parameters, is adapted with the new mean. GeneSys evaluates the error function with new samples and the process continues. Since CMA-ES is a local policy, GeneSys can converge to a local minima. GeneSys restarts the algorithm with new samples to escape from the minima. During restarts, GeneSys can restart CMA-ES while retaining some prior information related to the continuous parameters. GeneSys stops when the target program is found or some maximum time limit has exceeded. Note that NPO also uses CMA-ES to solve the optimization problem (similar to GeneSys). However, NPO suffers from poor problem formulation as well as local minima due to the use of autoencoder and absence of any restart policy.

In summary, we make the following contributions:

- We investigate how program synthesis can be formulated with continuous parameters that can lead to a better synthesis algorithm. Towards that end, we propose a *novel formulation* of program synthesis as a continuous optimization problem where a program is expressed as a tuple of functions such that each function maps continuous parameters into one or more discrete tokens.

- We introduce a number of *novel mapping* schemes to convert continuous parameters into actual programs (and describe several other less performant mapping schemes). We propose GeneSys, a CMA-ES-based program synthesis framework that uses that mapping scheme. We investigate different restart policies in the context of GeneSys to escape from local minima.

- We compare GeneSys with six prior program synthesis techniques - DeepCoder [12], PCCoder [153], RobustFill [34], NetSyn [85], PushGP [100], and NPO [79]. These techniques cover a wide spectrum of program synthesis approaches from discrete search and learning-based schemes to continuous optimization. Our results show that GeneSys synthesizes similar or more programs within a fixed time budget than those existing schemes. Specially at higher length (e.g., 10-length), GeneSys, on average, synthesizes 28.1% more programs than those existing schemes.

### 3.3 Background

CMA-ES was initially proposed in [50, 54] with a more recent version described by [51]. It is a variation of the evolutionary algorithm. An evolutionary algorithm repeatedly selects and mutates genes in multiple generations to find a solution. These genes are selected based on an objective (fitness) function to evaluate and create the next generation of genes. In CMA-ES, new genes (i.e., candidate solutions) are sampled from a multivariate normal distribution. A perturbation of zero mean is added for variations. The pairwise dependencies between variables can be represented by a covariance matrix. CMA-ES updates the covariance matrix through generations to learn a second order model of the objective function. This method is particularly useful when the objective function is ill-conditioned.

The main goal of CMA-ES is to minimize an objective function $f$. CMA-ES samples $\lambda$ points in each generation $k$ from a multivariate normal distribution $\mathcal{N}(m_k, \sigma_k^2.C_k)$ and adapts the parameters $C_k \in^{d \times d}$, $m_k \in^d$ and $\sigma_k \in^+$ by evaluating $f$. For a minimization task, $\lambda$ points are ranked by $f$ such that $f(x_1, k) \leq f(x_2, k) \leq ... \leq f(x_\lambda, k)$. The distribution mean is set to the weighted average $m_{k+1} = \sum_{i=1}^{\mu} \omega_i x_{i,k}$. The weights do not depend on the evaluated function value, rather on the ranking of different $\lambda$ points. Typically $\mu \leq \lambda/2$ and the weights are chosen such that $\mu_w = 1/\sum_{i=1}^{\mu} w_i^2 \approx \lambda/\ 4$ [51].

Step size $\sigma_k$ is updated using cumulative step-size adaption (CSA), also known as path length control [51]. The evolution path, $p_\sigma$, is updated first using Equation 3.1. Then, the step size $\sigma_{k+1}$ is updated using Equation 3.2.

$$
\begin{aligned}
p_\sigma =&(1-c_\sigma)p_\sigma + \sqrt{1-(1-c_\sigma)^2}\sqrt{\mu_w}C_k^{-1/2} \\
&\cdot \frac{m_{k+1}-m_k}{\sigma_k}
\end{aligned} \tag{3.1}
$$

$$
\sigma_{k+1} = \sigma_k \cdot e^{\frac{c_\sigma}{d_\sigma}\left(\frac{p_\sigma}{E\mathcal{N}(0,I)}-1\right)}. \tag{3.2}
$$

Here, $c_\sigma^{-1} \approx n/3$ is the backward time horizon for the evolution path $p_\sigma$ and larger than one, $\mu_w = (\sum_{i=1}^{\mu} w_i^2)^{-1}$ is the variance effective selection mass and $1 \leq \mu_w \leq \mu$ holds true by the definition of $w_i$, $C_k^{-1/2}$ is the unique symmetric square root of the inverse of $C_k$, $d_\sigma$ is the damping parameter usually close to one and $E$ is the expected values of $\mathcal{N}(0, I)$.

Finally the covariance matrix is updated, where again the respective evolution path is updated first.

$$
\begin{aligned}
p_c =&(1-c_c)p_c + \mathbf{1}_{[0,\alpha\sqrt{n}]}(p_\sigma)\sqrt{1-(1-c_c)^2} \\
&\cdot \sqrt{\mu_w}\frac{m_{k+1}-m_k}{\sigma_k}
\end{aligned} \tag{3.3}
$$

$$
\begin{aligned}
C_{k+1} =&(1-c_1-c_\mu+c_s)C_k + c_1 p_c p_c^T \\
&+ c_\mu \sum_{i=1}^{\mu} w_i \frac{x_{i:\lambda}-m_k}{\sigma_k}\left(\frac{x_{i:\lambda}-m_k}{\sigma_k}\right)^T
\end{aligned} \tag{3.4}
$$

The parameters used here are as follows:

- $C_c^{-1} \approx n/4$ is the backward time horizon for the evolution path $p_c$ and larger than one.
- $\alpha \approx 1.5$.
- The indicator function $\mathbf{1}_{[0,\alpha\sqrt{n}]}(p_\sigma)$ evaluates to $p_\sigma \in [0, \alpha\sqrt{n}]$.
- $c_s = (1 - \mathbf{1}_{[0,\alpha\sqrt{n}]}(p_\sigma)^2)c_1 c_c(2 - c_c)$ makes partly up for the small variance loss in case the indicator is zero.
- $c_1 \approx 2/n^2$ is the learning rate for the rank-one update of the covariance matrix and $c_\mu \approx \sigma_w/n^2$ is the learning rate for the rank-$\mu$ update of the covariance matrix. The covariance matrix update

tends to increase the likelihood for $p_c$ and for $(x_{i:\lambda} - m_k)/\sigma_k$ to be sampled from $\mathcal{N}(0, C_{k+1})$. CMA-ES is attractive as a powerful black box optimization technique due to the fact that it adjusts all of its parameters automatically based on the objective function and covariance matrix. Interested readers should consult [51] for more details about CMA-ES.

### 3.4 Problem Statement

Let $S = \{(I_j, O_j)\}_{j=1}^{s}$ be a set of $s$ input-output pairs, such that the output $O_j$ is obtained by executing the program $P^t$ on the input $I_j$. Inherently, the set $S$ of input-output examples describes the behavior of the program $P^t$. One would like to synthesize a program $P$ that recovers the same functionality of $P^t$. However, $P^t$ is unknown, and we are provided with the set $S$ as the specification. Based on this assumption, we define equivalency between two programs as follows:

**Definition 1** (Program Equivalency). Programs $P^a$ and $P^b$ are equivalent under the set $S = \{(I_j, O_j)\}_{j=1}^{s}$ of input-output examples if and only if $P^a(I_j) = P^b(I_j) = O_j$, for $1 \leq j \leq s$. We denote the equivalency by $P^a \equiv_S P^b$.

Definition 1 suggests that to obtain a program equivalent to $P^t$, we need to synthesize a program that satisfies $S$. Therefore, our goal is to find a program $P$ that is equivalent to the target program $P^t$ (which was used to generate $S$), i.e., $P \equiv_S P^t$. This task is known as Inductive Program Synthesis. Any solution to this problem requires the definition of two components. First, we need a programming language that defines the domain of valid programs. Second, we need a method to find $P$ from the valid program domain.

### 3.5 Problem Statement

### 3.6 Program Synthesis Framework

Here, we describe our choice of programming language, proposed formulation of program synthesis as a continuous optimization problem followed by various mapping and restart schemes and finally, the proposed framework, GeneSys.

| | |
|---|---|
| [int] | Input: |
| MAP (+1) | [5, 0, -3, 1, 4] |
| SORT | |
| FILTER (EVEN) | Output: |
| REVERSE | [6, 2, -2] |

Table 3.1: An example program of length 4 with an input and corresponding output.

### 3.6.1 Domain Specific Language

As GeneSys's programming language, we chose a domain specific language (DSL) used in earlier work, such as NetSyn [85] and DeepCoder [12]. This choice allows us to constrain the program space by restricting the operations used by our solution. The only data types in the language are *(i)* integers and *(ii)* lists of integers. The DSL contains 41 functions, each taking one or two arguments and returning one output. We will refer to DSL functions as *Tokens* to avoid ambiguity with later terminology. We use $\Sigma_{DSL} = \{D_i\}_{i=1}^{|\Sigma_{DSL}|}$ to indicate the set of all tokens in the DSL, with $|\Sigma_{DSL}| = 41$. Many of the DSL tokens include operations for list manipulation. Likewise, some operations also require lambda functions. There is no explicit control flow (conditionals or looping) in the DSL. However, several of the operations are high-level functions and are implemented using such control flow structures. A full description of the DSL can be found in NetSyn [85]. With these data types and operations, we define a program $P$ as an ordered sequence of DSL tokens i.e., $P = \langle P_i \rangle_{i=1}^{l}$ where $l$ is the length and $P_i$ is the $i$-th token of the program. Table 3.1 presents an example of a program of 4 tokens with an input and respective output. This program can be expressed as $P = \langle Map(+1), Sort, Filter(Even), Reverse \rangle$.

### 3.6.2 Program Synthesis as a Continuous Optimization Problem - A Novel Formulation

We propose to model a program as an $l$-tuple, where each element of the tuple is a function of continuous parameters. Intuitively, each function takes one or more continuous parameters and maps them to 0 or more DSL tokens in the program. A program $P$ can be expressed as $P = \left\langle f_1(x_1^1, ..., x_M^1), ..., f_N(x_1^N, ..., x_M^N) \right\rangle$ where, $f_i(x_1^i, ..., x_M^i) : \mathbb{R}^M \rightarrow \Sigma_{DSL}^{C_i}$ for $1 \leq i \leq N$, $0 \leq C_i \leq l$, and $N, M > 0$. Namely, each function $f_i$ maps $M$ continuous random variables into

$C_i$ DSL tokens in the program. When $C_i = 0$, it indicates the special case when $f_i$ does not map its parameters to any DSL token. In other words, $f_i$ becomes a NULL function. Figure 3.1(a) shows the mapping from the continuous variables to DSL tokens. Since the length of the program $P$ is $l$, we can infer $\Sigma_i C_i = l$.

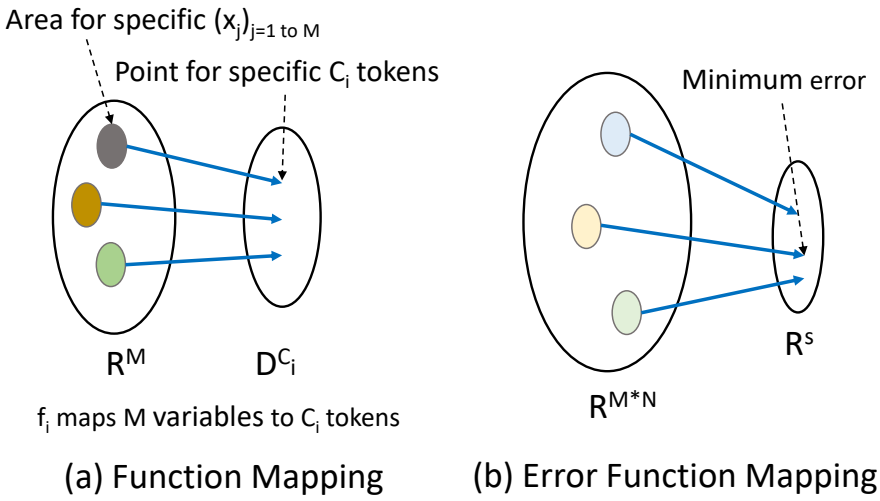



Figure 3.1: Pictorial representation of how (a) each function maps the continuous parameters to DSL tokens in the program and (b) the error function maps the continuous parameters.

We define an error over the given specification, $S = \{(I_j, O_j)\}$ as $\mathbb{E}(P, S) = \langle E(P(I_j), O_j)\rangle_{j=1}^{s}$. Here, $E(P(I_j), O_j)$ could be any commonly used distance function in program synthesis such as edit distance, Manhattan distance, etc. [65]. With the formulation of $P$ as an $l$-tuple, $\mathbb{E}$ becomes a function mapping $\mathbb{R}^{M\times N}$ to $\mathbb{R}^s$. Therefore, program synthesis becomes a continuous optimization problem where the goal is to find values of $M \times N$ continuous random variables that minimize $\mathbb{E}$ (i.e., the minimum point in the $s$-dimensional space). Figure 3.1(b) depicts this formulation.

Figure 3.2: Representation of the bin mapping scheme.

### 3.6.3 Mapping Scheme: Bin Mapping

Based on the problem formulation in the prior Section, we propose a mapping scheme to construct an $l$-length program $P$. We choose $l$ functions to represent the program. Each function takes one continuous parameter. Thus, $P = \langle f_1(x_1^1), f_2(x_1^2), \ldots, f_l(x_1^l) \rangle$ where $f_i(x_1^i) : \mathbb{R} \rightarrow \Sigma_{DSL}$. Each function $f_i$ must be able to map the continuous parameter to one of the DSL tokens. This is shown in Figure 3.2. We conceptually divide the range of each variable into $|\Sigma_{DSL}|$ bins. That is why, we refer to this scheme as *Bin Mapping*. The size of each bin can be equal (except the bin at each end) or proportional to the probability of the corresponding token being present in the program $P$. Prior work [85, 12] showed how to infer such probability. When GeneSys samples from the continuous variables to create a candidate program, it takes each real sampled value and determines into which of the bins the value falls. The token used in that position in the candidate program is the one corresponding to the bin number into which the sampled value fell. This sampling process occurs for each of the $l$ continuous parameters. Thus, the definition of $f_i$ and the corresponding selection of program token is as follows:

$$f_i(x_1^i) = D_j \qquad \text{if } x_1^i \in Bin_{D_j} \text{ for } 1 \leq j \leq |\Sigma_{DSL}|$$
$$P_i = D_j,$$

where $Bin_{D_j}$ represents the range of values corresponding to token $D_j$, and $P_i$ represents the $i$-th token of program $P$. This mapping scheme is robust and flexible and it requires the least number of variables to represent a program. This makes the optimization problem easier and helps GeneSys to find more programs within a fixed time budget.

### 3.6.4 Alternative Mapping Schemes

To experienced CMA-ES users, bin mapping is perhaps the most intuitive mapping but initially it was unclear whether other obviously possible mapping schemes like Multi-Group Mapping might

be superior. Thus, we did explore a number of other possible mappings before determining that bin mapping was the most effective. Each of these other mapping schemes use one or more continuous random variables to map to $0$ or more DSL tokens of a program. Table 3.2 summarizes different mapping schemes and their characteristics.

#### *3.6.4.1 Single Group Mapping*

In this mapping, $|\Sigma_{DSL}|$ functions map to $l$-length program, $P$. Each function represents a DSL token and takes one continuous parameter. Each function will map its parameter to at most one DSL token. Thus, $P = \left\langle f_1(x_1^1), f_2(x_1^2), \ldots, f_{|\Sigma_{DSL}|}(x_1^{|\Sigma_{DSL}|}) \right\rangle$ where $f_i(x_1^i) : \mathbb{R} \rightarrow \Sigma_{DSL}^{0|1}$. GeneSys samples each continuous parameter and then constructs the program by choosing the tokens corresponding to the $l$ largest sampled values. The token for the largest sampled value occurs first in the program. The token corresponding to the next largest value occurs second, and so on. This is shown in Table 3.2. A significant limitation of this approach is that each DSL token could occur at most once in a program and thus is not applicable if the length of the program, $l$ is larger than $|\Sigma_{DSL}|$. The formal definition of $f_i$ and the corresponding selection of program token is as follows:

$$
\begin{aligned}
f_i(x_1^i) = \quad & D_i \quad && if\ x_1^i = Top_j(\{x_1^i\}_{i=1}^{|\Sigma_{DSL}|}) \text{ for } 1 \leq j \leq l \\
& None && otherwise \\
P_j = \quad & D_i && when\ f_i(x_1^i) = D_i,
\end{aligned}
$$

#### *3.6.4.2 Multi-Group Mapping*

In this mapping, GeneSys uses $l$ functions, each taking $|\Sigma_{DSL}|$ continuous parameters to express the program $P$. Each position in the program is thus represented by one function. Each parameter of that function corresponds to a DSL token. GeneSys samples the continuous parameters of the first function and selects the largest sampled parameter. The corresponding DSL token becomes the first token in $P$. Then, GeneSys samples the parameters of the second function and the token

| Mapping Scheme | Domain | Range | Characteristics |
|---|---|---|---|
| Single Group | $\mathbb{R}$ | $\Sigma_{DSL}^{0\|1}$ | $f_i$ maps 1 variable to 0 or 1 token for $1 \leq i \leq \|\Sigma_{DSL}\|$. |
| Multi-Group | $\mathbb{R}^{\|\Sigma_{DSL}\|}$ | $\Sigma_{DSL}$ | $f_i$ maps $\|\Sigma_{DSL}\|$ variables to 1 token for $1 \leq i \leq l \times \|\Sigma_{DSL}\|$. |
| Dynamic Multi-Group | $\mathbb{R}^{\|\Sigma_{DSL}\|}$ | $\Sigma_{DSL}^{l/k}$ | $f_i$ maps $\|\Sigma_{DSL}\|$ variables to $l/k$ tokens for $1 \leq i \leq k$. |

Table 3.2: Characterization summary of different alternative mapping schemes.

corresponding to the largest sampled parameter becomes the second token in $P$. This process repeats for each of the $l$ functions. This mapping scheme does allow the same DSL token to be used multiple times in a program but since it requires a large number of variables to represent a program, it may cause CMA-ES to take a long time to find a solution or may not find a solution. The formal definition of $f_i$ and the corresponding selection of program token is as follows:

$$f_i(x_1^i, \ldots, x_{|\Sigma_{DSL}|}^i) = D_j \quad \text{if } x_j^i = Top_1\left(\{x_j^i\}_{j=1}^{|\Sigma_{DSL}|}\right)$$
$$P_i = D_j.$$

#### *3.6.4.3 Dynamic Multi-Group Mapping*

Dynamic multi-group mapping is a hybrid of the single and multi-group mappings in which there are $k$ functions, each taking $|\Sigma_{DSL}|$ continuous parameters and choosing $l/k$ DSL tokens for $P$. GeneSys randomly chooses $k$ for each program and treats $k$ as a variable that CMA-ES can evolve, thus allowing GeneSys to evolve to find the optimal $k$. If $k$ is less than the program length then the continuous parameters corresponding to groups larger than $k$ will be unused. However, since $k$ itself may increase, those continuous parameters must still be there and hence this scheme requires one more variable (i.e., $k$) than the Multi-Group Mapping.The formal definition is as follows:

$$f_i(x^i_1, \ldots, x^i_{|\Sigma_{DSL}|}) = \left\{ D_j |\ x^i_j = Top_{\frac{l}{k}} \left( \{x^i_j\}_{j=1}^{|\Sigma_{DSL}|} \right) \right\}$$

$$P_{(i-1)\frac{l}{k}+t} = D_j \quad \text{if } x^i_j = Top_t \left( \{x^i_j\}_{j=1}^{|\Sigma_{DSL}|} \right)$$

$$\text{for } 1 \leq t \leq \frac{l}{k}.$$

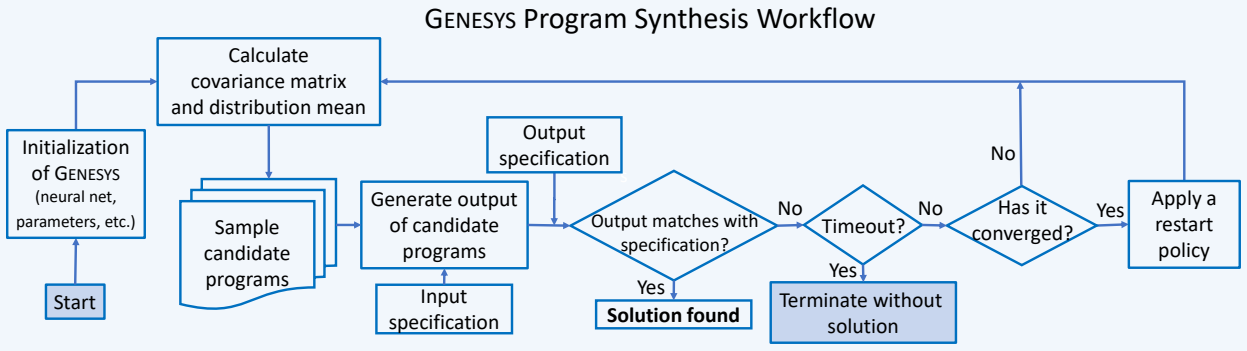



Figure 3.3: Overview of GeneSys.

#### *3.6.4.4 Dynamic Bin Mapping*

It is sometimes possible to find a program of a lower length (than the length of the target program) that also satisfies the given specification. This mapping scheme is the same as bin mapping but tries to take advantage of this property by including a function to choose the length, in other words the number of other bin mapping functions to turn into program statements. The length function conceptually divides the range of $x^0_{l+1}$ into $l$ equal sized bins. When GeneSys samples this parameter, it determines which bin the parameter's value fall into. The length corresponding to that bin is chosen as the program length. In practice, however, we found it to be more flexible to test all subsets of a generated program of maximum length for correctness rather than evolving the length variable as this length variable is not always accurate. The formal definition is as follows:

$$f_i(x_1^i) = D_j \qquad \text{if } x_1^i \in Bin_{D_j} \text{ and } x_1^{l+1} \in Bin_k$$
$$\text{for } 1 \leq j \leq |\Sigma_{DSL}| \text{ and } 1 \leq k \leq l \text{ and } i \leq k$$
$$P_i = D_j,$$

### 3.6.5 Restart Policy

GeneSys can stop its evolutionary process when the potential solutions are converging due to some local minima. This is checked by determining if a change in one or multiple axes does not affect the distribution mean, or if the condition number of the covariance matrix is too high (i.e., ill-conditioned), or the evolutionary path's step size is too small to reach the solution [51]. To escape from such situation, we investigate several restart policies where GeneSys restarts with fresh initial parameters. We explore all policies resulting from the combinations of 3 core restart policies: population-based (PB), mean-based (MB), and covariance matrix-based (CB). For the PB restart, GeneSys doubles the size of the population from its previous size. For the MB restart policy, GeneSys resets the mean vector to randomly initialized values (using the uniform random distribution) after a restart instead of keeping its current value. For the CB restart policy, GeneSys re-initializes the covariance matrix to the identity matrix after a restart instead of keeping its current values. Additional restart policies can be constructed by any combination of these core restart policies. Some of these combinations have been proposed in earlier work such as IPOP [10] and BIPOP [52] where they re-initialize mean vector, covariance matrix, and increase population size simultaneously.

### 3.6.6 Integration with Learning Approaches

Usually, CMA-ES initializes the genes by sampling continuous variables from a multivariate normal distribution and the bin mapping scheme uses a uniform size for bins to map the continuous variables. We investigated whether GeneSys would improve with the integration of a learning-based approach to bootstrap CMA-ES by means of better gene initialization or improve the core

performance of CMA-ES by unequal bin widths based on some learned-attributes such as token probability. To do so, we used a neural network model similar to that of NetSyn [85]. The model is a $LSTM$-based network where latent hidden state is generated as, $h_i = LSTM(h_{i-1}, S)$ where $S$ is the input-output example from the specification. Multiple input-output examples create different hidden states. These hidden states are aggregated together to get the token probability by passing through a $Softmax$ layer. Details of the model and its training results are presented in Section 3.7.2.2. GeneSys can use the token probabilities from this model when initializing the genes at the beginning of CMA-ES. Likewise, GeneSys can use bin widths that are proportional to the token probabilities from this model for the bin mapping scheme.

### 3.6.7 Putting It All Together

GeneSys follows the high level workflow shown in Figure 3.3. GeneSys starts by initializing a multivariate normal distribution and the population size. The continuous variables of the distribution are determined based on the bin mapping scheme in Section 3.6.3. In each generation, GeneSys samples $\lambda$ points from the multivariate distribution and maps each sample to a program. Each of these candidate programs could potentially be the solution. Therefore, GeneSys applies the error function $\mathbb{E}$. The error function essentially applies inputs from the given specification to a candidate program and checks if the produced output matches with the given output specification. If the outputs match, the candidate program is returned as the solution. Otherwise, GeneSys checks if convergence has reached. If not, GeneSys updates the mean, covariance matrix and distribution path based on the error function and starts a new generation. If, on the other hand, convergence is reached, GeneSys applies a restart policy and repeats the whole process with new parameters. The synthesis process continues until a maximum time limit has passed or a solution is found.

## 3.7 Results

### 3.7.1 Methodology

We implemented GeneSys in Python, starting from a base CMA-ES implementation from `https://github.com/srom/cma-es`. We developed an interpreter for our DSL in Sec-

tion 3.6.1 to evaluate a DSL program. We randomly generated a total of 600 programs from length 5 to 10, with 100 programs from each length. We checked those programs extensively to ensure that no program contains any token (or sequence of tokens) which does not affect the input data at all. For example, the token sequence $Map(+1), Map(-1)$ does not affect the input data at all. We performed this checking to ensure a shorter length program is unlikely to satisfy the specification of a particular length program. We used these 600 programs as test programs that GeneSys tries to synthesize. For each program, we used $s = 5$ input-output examples as its specification. To evaluate synthesis ability, we used a synthesis time limit of 3 hours (10,800 seconds) for each program. If any program can not be found within that time limit, we conclude that the program was "not found".

To run all the experiments we used SLURM batch processing system. All the experiments are run in a cluster with Intel(R) Xeon(R) CPU E5-2650 v2 @ 2.60GHz processor and 62GB of memory. For implementing various learning-based schemes, we use Nvidia Tesla K80 GPU with 128 GB RAM to train the neural networks. For training any learning-based schemes, we used 420,000 randomly generated unique programs of length 5 to train the model. We checked the programs to remove duplicates. For each of the program we used 5 IO examples as the specification. For biasing bin width with token probability (Section 3.7.2.2), GeneSys uses a neural network similar to the one used in NetSyn [85] to predict the probability of various DSL functions in a target program.

### 3.7.2 Characterization of GeneSys

We first explore the characterization of restart policies, impact of applying learning on top of GeneSys followed by other setups from different combinations of policies. Our main objective is to assess various settings of GeneSys.

#### *3.7.2.1 Restart Policies*

The synthesis percentage and time for different combinations of the three core restart policies are presented in Figure 3.4. Program synthesis percentages are shown (green) at the left $y$-axis and their synthesis times in the right $y$-axis as box plots for each of the setups. Each boxplot represents minimum, 25% and 75% quartiles, median and maximum synthesis times. Circles indicate outliers.

Among all policies, synthesis percentage can be as low as 49% for PB and as high as 88% for PB+CB case. Without any restart, GeneSys can synthesize 58% of programs. Among the individual policies, MB and CB both synthesize $1.5\times$ more programs than that of PB. On the other hand, when we combine PB+CB, GeneSys synthesizes 88% programs followed by PB+MB+CB with a synthesis percentage of 86%. Note that PB+MB+CB is essentially the BIPOP policy [52]. The results suggest that it is better to re-initialize the covariance matrix during restarts. This is because the covariance matrix may become ill-conditioned and lead to the same local minima. Doubling the population is generally beneficial because more potential solutions can be checked. Two best performing policies (i.e., PB+CB and PB+MB+CB) differ by whether to re-initialize the mean vector or not. PB+CB being the best policy indicates that keeping the mean vector as it is during a restart provides some advantage by accumulating information about the continuous parameters.

#### *3.7.2.2 Impact of Learning*

Figure 4.2 shows the neural network model that we used in GeneSys. This model is inspired from prior work [85]. The model has two components mainly. Figure 4.2(a) shows how to feed single IO example to the network. Input and output is a list of integer values. They go through embedding layer and each element in the list is represented by 20 length long latent representation. After that, embedded input and output is fed through LSTM layer. We use 5 IO examples in the specification. Each of the IO create their own hidden vector from their LSTM block. Those vectors got through a different LSTM block and some fully connected layer before giving the token probability distribution for the given specification. Figure 3.9 shows the training accuracy of the model across epochs. The model is trained until it converges. We use same number of training data as other baselines.

Next we explore the effects of biasing gene initialization and bin widths (using the neural network model) on GeneSys (see Section 3.6.6). Figure 3.5 shows results for different combinations of gene initialization and bin width selection. Note from this figure that using biased bin widths is always superior in terms of synthesis percentage compared to uniform bin widths. Also note that there is little difference between initial gene creation using a normal distribution versus a learned

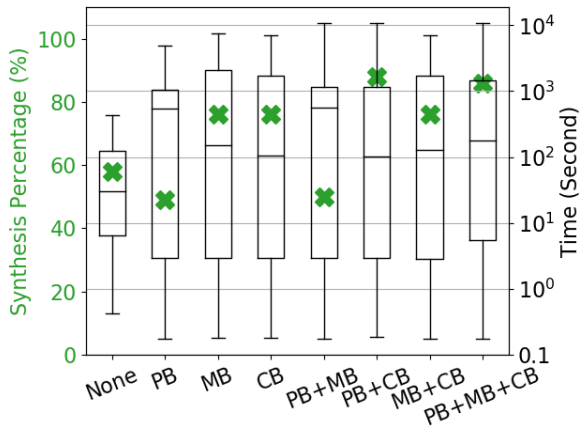



Figure 3.4: Effect of restart policies on GeneSys.

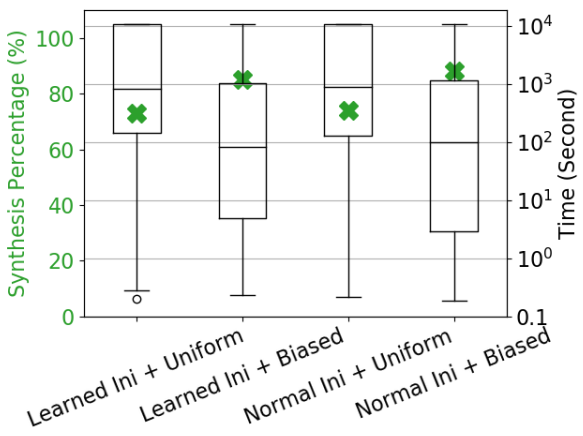



Figure 3.5: Impact of learning on GeneSys.

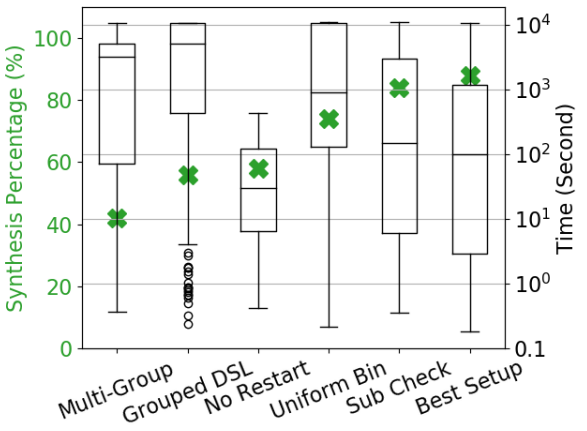



Figure 3.6: Different setups for GeneSys.

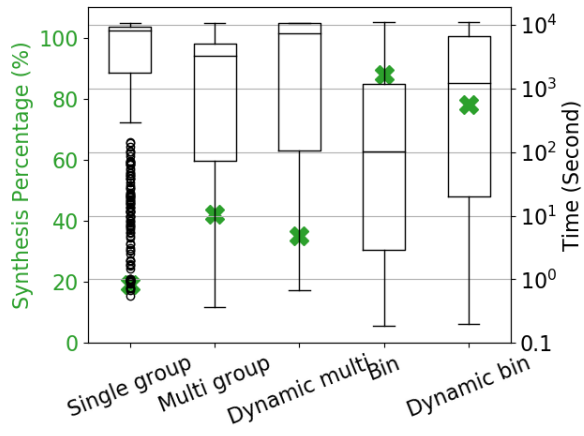



Figure 3.7: Effect of mapping schemes on GeneSys.

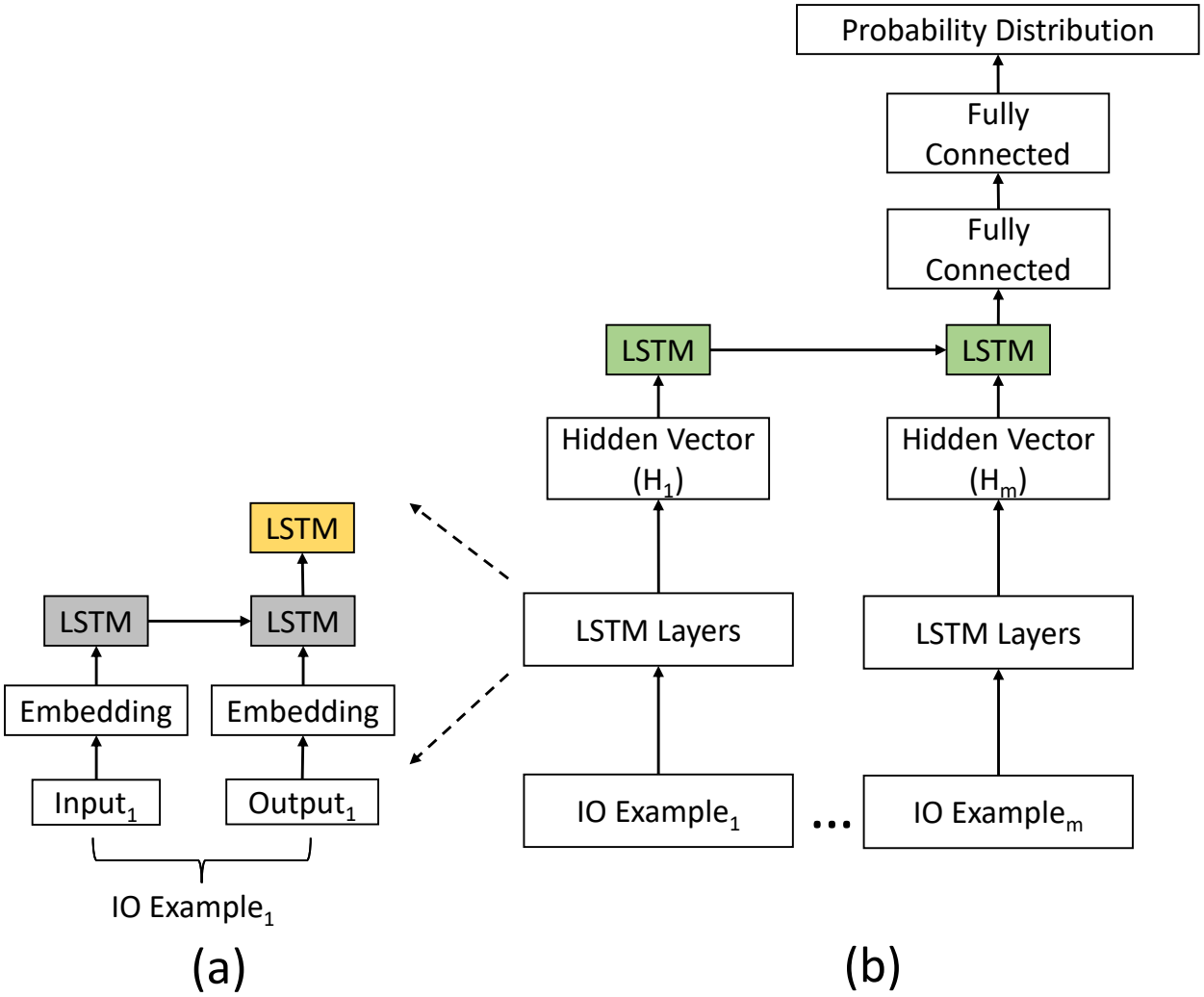



Figure 3.8: Neural network design for (a) single and (b) multiple IO examples. In each figure, layers of LSTM encoders are used to combine multiple inputs into hidden vectors for the next layer. Token probability distribution is produced by the fully connected layer.

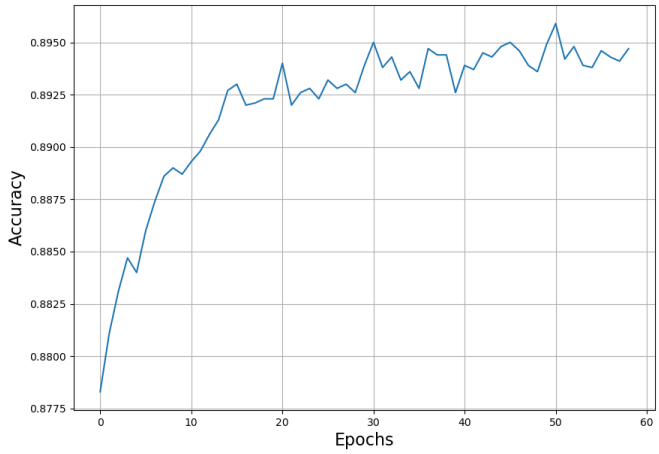



Figure 3.9: Training accuracy of the neural network used in GeneSys.

distribution. Thus, we conclude that learned attributes like token probabilities are better utilized in bin width selection as opposed to the gene initialization. This is intuitive since the effect of gene initialization diminishes after a few initial iterations of GeneSys. Thus, the default GeneSys setup uses *Normal Ini + Biased* as it had a 88% synthesis rate compared to *Learned Ini + Biased* which was slightly lower at 85%.

#### *3.7.2.3 Characterization of Mapping Schemes*

Although bin mapping is the default scheme for GeneSys, we investigated how other mapping schemes worked. Figure 3.7 shows the percentage of all programs synthesized and synthesis time for different mapping schemes. For programs of length 10, bin mapping can synthesize up to 88% programs whereas synthesis percentage is 78%, 19%, 42% and 35% for dynamic bin, single group, multi-group and dynamic multi-group mapping respectively. In terms of synthesis time, the bin mapping scheme is at least $1.5\times$ faster than other mapping schemes. Thus, bin mapping scheme works the best both in terms of synthesis percentage and time. This is because bin mapping has the lowest number of continuous parameters (i.e., $O(l)$). Other mapping schemes require more continuous parameters making the optimization domain much larger to find the solution within the given time limit.

| Setup Name | Mapping | DSL Type | Restart | Bin Type | Check |
|---|---|---|---|---|---|
| Multi-Group | Multi-group | Regular | Yes | Biased | Full |
| Grouped DSL | Bin | Grouped | Yes | Biased | Full |
| No Restart | Bin | Regular | No | Biased | Full |
| Uniform Bin | Bin | Regular | Yes | Uniform | Full |
| Sub Check | Bin | Regular | Yes | Biased | Sub |
| Best Setup | Bin | Regular | Yes | Biased | Full |

Table 3.3: Different configurations for GeneSys framework. For restart, we used PB+CB policy.

### 3.7.3 Other Setups

We experimented with additional configurations of GeneSys. Table 3.3 presents each configuration with a name. The configurations are based on 5 criteria. *Mapping* and *Restart* indicate the mapping schemes and restart policies. *DSL Type* indicates whether we used our regular or grouped DSL. In grouped DSL, similar tokens are grouped into a single *super* token in an effort to reduce the optimization domain. *Bin Type* indicates whether the bin size is uniform or biased based on token probability. *Check* indicates whether GeneSys checks for specification after a full-length program or after each token in the program (i.e., sub-program). Checking after each token allows GeneSys to find an equivalent lower-length program at the expense of more checking operations. Figure 3.6 compares synthesis percentages and times for the different configurations. The best setup synthesizes 88% programs combining bin mapping with regular DSL, a restart policy (PB+CB), biased bins, and full program check. Token by token checking and grouped DSL did not provide as much benefit (either in terms of time or synthesis percentage) as expected due to additional checking overhead or additional selection requirements for each super token (i.e., selecting the exact token within a super token). No restart policy performs worse due to premature convergence. Multi-group setup synthesizes the least programs because a higher number of continuous parameters lead to higher synthesis time. Therefore, many programs are not found within the given time limit. Finally, compared to the uniform bin, biased bins increase synthesis percentage by 18.9% in the best setup.

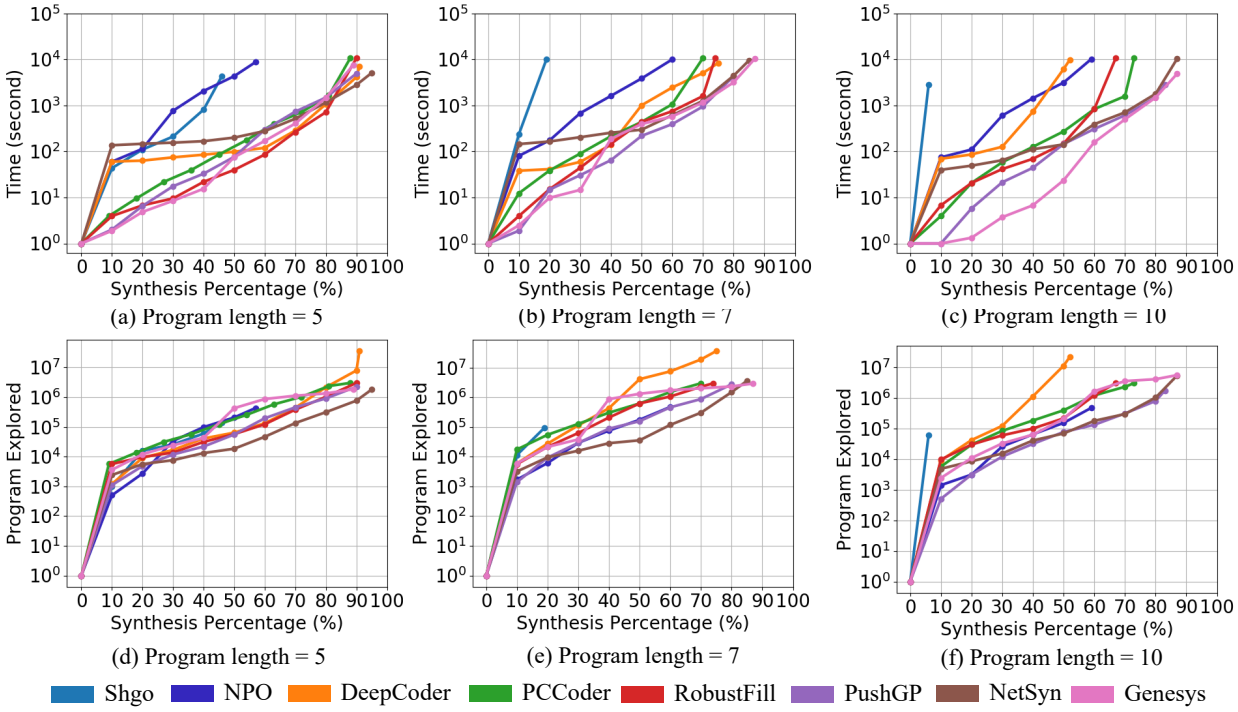



Figure 3.10: GeneSys's synthesis ability for different program lengths with respect to Shgo, NPO, DeepCoder, PCCoder, RobustFill, PushGP, and NetSyn.

### 3.7.4 Comparison of Synthesis Ability

#### *3.7.4.1 Overview and Methodology of Previous Schemes*

**Shgo:** Shgo [37] is simplicial homology global optimization technique. We used Shgo as a blackbox continuous optimization technique to justify our choice of CMA-ES in GeneSys. We used Shgo implementation from `https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.shgo.html`.

**NPO:** In NPO [79], an autoencoder based neural network model is used to encode and decode programs to and from latent continuous representations. In one end the encoder takes program and and creates a vector representation of the program. CMA-ES works with such vectors to find the optimal vector. That vector is passed through a decoder to produce the target program. We used stack based LSTM for the encoder and decoder. The encoder takes the program and tries to produce the same programs on the decoder part. We used 420K programs to train the autoencoder. The model took $\sim$12 hours to train.

**DeepCoder:** In Deepcoder [12], a feed forward neural network model is used to predict function probability for a given specification. The model passes IO examples through an embedding layer and produces a 20 length long vector. Then it passes through multiple layer of feed forward block. All the IO examples are passed through their individual block and aggregated together with a pooling layer. A softmax function is used to predict the probability distribution of the functions. We used all the 420K random programs to train the model (Section 3.7.1). It took $\sim$2 days for the model to converge.

**PCCoder:** PCCoder [153] is an encoder-decoder based neural network model. It takes the specification and creates state embedding. Those state embedding is passed through the dense block of neural networks (NN). For multiple IO examples, it passes through different NN blocks and max pooling is used to aggregate the result. Then it uses softmax to get the probability distribution. Like DeepCoder, we used all 420K programs and trained for $\sim$2 days.

**RobustFill:** RobustFill [34] is a seq-to-seq neural network model that uses a encoder-decoder

architecture where the encoder takes the specification and the decoder produce the program. It generates the probability distribution of different functions and a beam search is used to search for the program. The hidden representation $h_t$ is an LSTM hidden unit given by $E_t = Attention(h_{t-1}, e(X))$, $h_t = LSTM(h_{t-1}, E_t)$. Here $e(X)$ is the sequence of hidden states after processing the specification with an LSTM encoder. For multiple IO examples, all of them are passed with different network and max pooling is used to aggregate the output. Then they are passed through a softmax layer to get the probability distribution. We used all 420K programs and trained the model for $\sim$2 days to converge.

**PushGP:** PushGP is a genetic programming based approach [100]. It works in a stack based evaluator. Each of the data type has it's own stack. And recent result is always on top of the stack. Push instruction acts by pushing and popping various elements on and off the stacks. The program interpreter maintain it's own exec stack that maintain the control flow of the program. Our PushGP implementation follows the technique mentioned in [100].

**NetSyn:** NetSyn [85] uses genetic algorithm with a LSTM based recurrent neural network as fitness function. Fitness function give some score based on the genes from gene pool about how good or bad those genes are. Based on the score, genetic mutation happens and new genes are created for next generation. The LSTM model takes IO examples and program traces as input and produce fitness score as output. It took $\sim$3 days to train the model. This LSTM model is used in one of the schemes (i.e., bin mapping with biasing (Section 3.6.3)) of GeneSys.

In summary, we used the same training set to train each of the prior schemes. We trained the models until they converged. Thus, we gave the best effort for a faithful reproduction of prior schemes.

#### *3.7.4.2 Synthesis Ability of Different Schemes*

We compared GeneSys(the best setup) with other program synthesis techniques mentioned in Table 3.4 . Figure 3.10 shows comparative results of these methods for different program lengths. Synthesis time is measured in seconds as a function of the percentage of programs synthesized within the corresponding time. Lines terminate when an approach fails to synthesize additional

programs. Among the approaches, some are learning-based while others are not. Keep in mind that GeneSys uses a learning model to determine the bin width. For all approaches, except Shgo and NPO, 90% of programs can be synthesized within same amount of time for program length 5. However, as program length increases, GeneSys synthesizes more programs in less time compared to other approaches. For example, GeneSys synthesizes 50% of length-10 programs in less than half of the time for other approaches. Compared to the recent techniques (DeepCoder, PCCoder, RobustFill, PushGP, and NetSyn), GeneSys synthesizes at least 1.1% and up to 31.3% more programs across all lengths. *For higher length (e.g., 10-length), GeneSys, on average, synthesizes 28.1% more programs than those existing schemes*. As expected, Shgo performs the worst since it is not used in the literature for program synthesis but is included here to show CMA-ES performs better than other black-box optimization techniques. NPO is the closest work to GeneSys. However, NPO has a consistently lower synthesis rate than GeneSys for all lengths. This is due to the fact that NPO uses autoencoder generated latent variables to represent a program in the continuous domain. Even a small change in one of the latent variables can map the representation to a wildly different program leading to a large error. This characteristic makes the continuous optimization a much harder problem. On the other hand, our mapping scheme is resilient to small changes in continuous variables making the overall optimization problem easier than that of NPO. That is why, even uniform bin mapping (a scheme that does not rely on any learning model at all) can synthesize 73% of all programs, thereby outperforming NPO, which synthesizes 58% of all programs. Figure 3.10 (d) to (f) shows programs explored before finding the target program. Although GeneSys searches through more programs than NetSyn, PCCoder, PushGP, and RobustFill, it searches those programs quickly. That is why, GeneSys synthesizes more programs within fixed time.

#### *3.7.4.3 Stability Comparison*

The use of autoencoder in NPO makes the problem formulation unstable. If an optimization problem is unstable, it becomes less tolerant to random noises leading to a sharp decline in solutions. In order to compare stability of GeneSys with NPO, we introduce random noises in the continuous variables. For example, we insert some random noises to each of the CMA-ES variable. Similarly,

| Scheme | Main Idea |
|---|---|
| Shgo | Simplicial global optimization |
| NPO | Autoencoder embedding + CMA-ES |
| DeepCoder | Function probability (DNN) + DFS search |
| PCCoder | Encoder decoder (DNN) + Beam search |
| RobustFill | Encoder decoder (LSTM) + Beam search |
| PushGP | Genetic algorithm (GA) |
| NetSyn | Fitness function (LSTM) + GA |
| **GeneSys** | Function formulation + CMA-ES + Token probability (LSTM) |

Table 3.4: Different schemes compared.

| Noise | Baseline | Length | | |
|---|---|---|---|---|
| | | 5 | 7 | 10 |
| Without | GeneSys | 89% | 87% | 87% |
| | NPO | 57% | 60% | 59% |
| With | GeneSys | 79% | 79% | 78% |
| | NPO | 51% | 45% | 42% |

Table 3.5: Stability comparison between GeneSys and NPO in terms of synthesis percentage.

we introduce random noises in the latent representations of NPO. Table 3.5 shows synthesis percentage between GeneSys and NPO with added noise. It shows that synthesis rate for GeneSys drops 10% whereas, it drops 20% for NPO on average across different length programs. This is expected as the decoder takes the latent representation from CMA-ES with noise added and produces significantly different programs. This makes it unstable. On the other hand, adding a little noise does not make GeneSys wildly jump to the other bins. Rather it is kept in the same bin and GeneSys becomes noise tolerant.

### 3.8 Related Works

Machine programming as known as program synthesis has been widely studied with various applications. *Formal program synthesis* uses formal and rule based methods to generate programs[88, 135]. Formal program synthesis usually guarantees some program properties by evaluating a generated program's semantics against a corresponding specification [46, 5]. In [40], authors use a SMT based solver to find different constraints in the program and learn useful lemma that helps to prune away large parts of the search space to synthesize programs faster. However, these techniques can often be limited by exponentially increasing computational overhead that grows with the program's instruction size [58, 15].

Another way to formal methods for MP is to use machine learning (ML). Machine learning differs from traditional formal program synthesis in that it generally does not provide correctness guarantees. Instead, ML-driven MP approaches are usually only *probabilistically* correct, i.e., their results are derived from sample data relying on statistical significance [91]. Such ML approaches tend to explore software program generation using an objective function. Objective functions are used to guide an ML system's exploration of a problem space to find a solution. Other deep learning based program synthesizer [17, 98, 25] also tried different approaches such as reinforcement learning to solve the problem. Most of these works focus on synthesizing programs in domain specific languages, such as FlashFill [34, 66] for string transformation problem, simulated robot navigation, such as Karel [119, 20] or LIST manipulation [12, 85] work.

Among the ML-based MP, in Deepcoder[12], the authors train a neural network with input-output examples to predict the probabilities of the functions that are most likely to be used in a program. Raychev et al. [110] take a different approach and use an n-gram model to predict the functions that are most likely to complete a partially constructed program. Robustfill [34] encodes input-output examples using a series of recurrent neural networks (RNN), and generates the the program using another RNN one token at a time. Bunel et al. [17] explore a unique approach that combines reinforcement learning (RL) with a supervised model to find semantically correct programs. These are only a few of the works in the MP space using neural networks [85, 144, 114, 19, 21].

Significant research has been done in the field of genetic programming [100, 16, 73, 97, 131, 57, 43, 56] too, where the goal is to find a solution in the form of a complete or partial program for a given specification. Prior work in this field has tended to focus on either the representation of programs or operators during the evolution process. Real et al. [112] recently demonstrated that genetic algorithms can generate accurate image classifiers. Their approach produced a state-of-the-art classifier for CIFAR-10 [71] and ImageNet [32] datasets. Moreover, genetic algorithms have been exploited to successfully automate the neural architecture optimization process [115, 126, 80, 72, 113].

However, all these works formulate program synthesis as a search problem of discrete parameters. On the other hand, GeneSys tries to formulate program synthesis as a continuous optimization problem and uses a well established derivative free method, CMA-ES, to solve it. There exists many other evolutionary strategy-based continuous optimization techniques [18]. Each of these techniques has it's own limitations. For instance, differential evolution suffers drastically when the objective function is not linearly separable [29]. Hence, CMA-ES is best suited for our problem formulation in program synthesis domain. Previously, CMA-ES was used in machine learning [90], aerospace engineering [83], mechanical engineering [122], health [137] and various other engineering fields [53]. Some recent works explored different aspects of program synthesis [67, 123, 99] with CMA-ES. However, they are not in the field of synthesizing complex programs.

## 3.9 Other Discussion

### 3.9.1 DSL grouping

Apart from the regular DSL we tried one different DSL grouping approach. In our DSL we have two type of functions, regular function and higher order function. For example, Map is a higher order function that can have multiple regular functions like Map(+1), Map(*2) etc. In DSL group mechanism we group such higher order function together and while constructing program we tried all possible combination of functions from the same group. Although this approach might increase the synthesis time, the intuition was synthesis rate might also be increased.

Figure 3.11 shows result for two different DSL setup that we tried. For multi group mapping, synthesis rate is higher for group DSL compared to regular DSL as 69% vs 42%. But when binning mapping is used we get as high as 88% synthesis rate with regular DSL compare to 56% for Group DSL. Synthesis time also follows the same pattern.

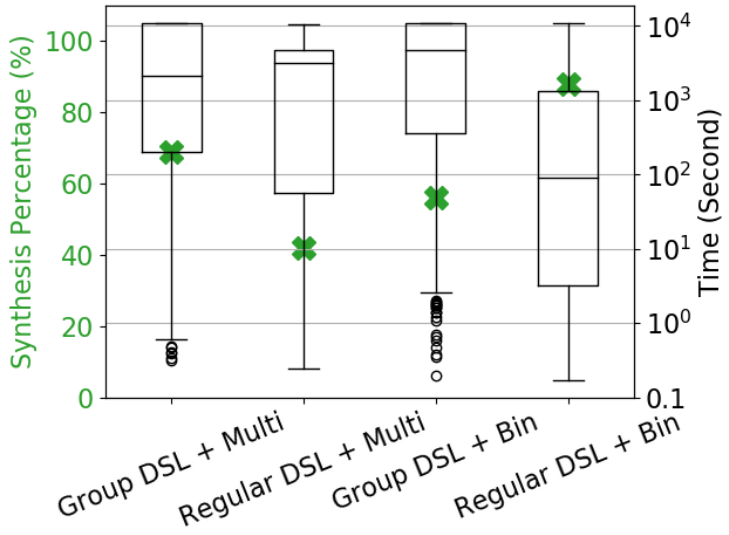



Figure 3.11: Choice of DSL with Multi group and Bin mapping

### 3.9.2 Restart Number

On average GeneSys restarts 5-15 times for different restart policies except *PB* based. In *PB* policy, only the population get doubled up but the mean vector and covariance-matrix remain same. After reaching some local minima, despite the population increase it always stuck in the same minima for every restart as mean or covariance doesn't change. Thus, in *PB* policy once GeneSys reaches some local minima it restarts in every generation and needs to stop as the increased population make the system out of memory.

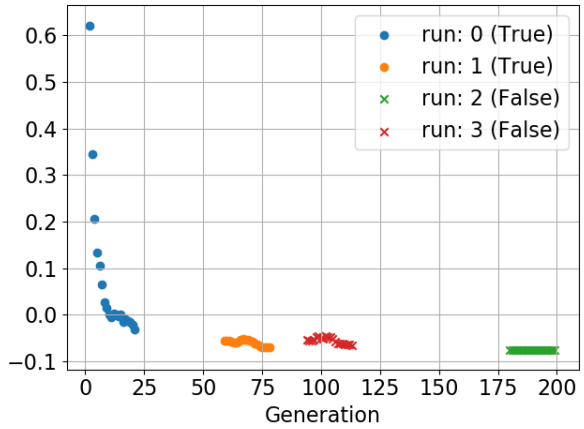



Figure 3.12: Covariance matrix

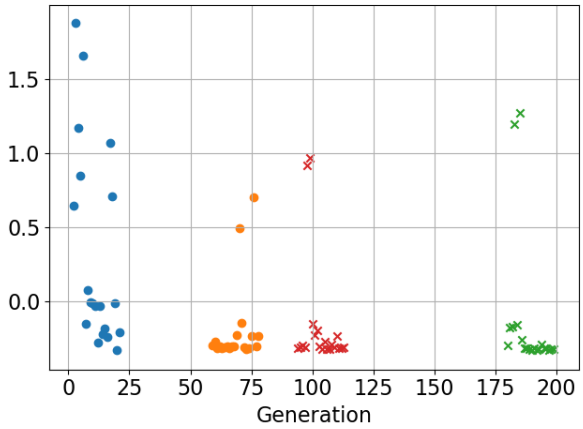



Figure 3.13: Eigenvectors

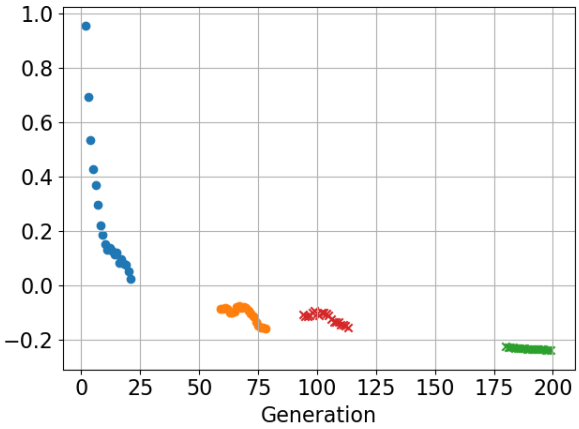



Figure 3.14: Eigenvalues

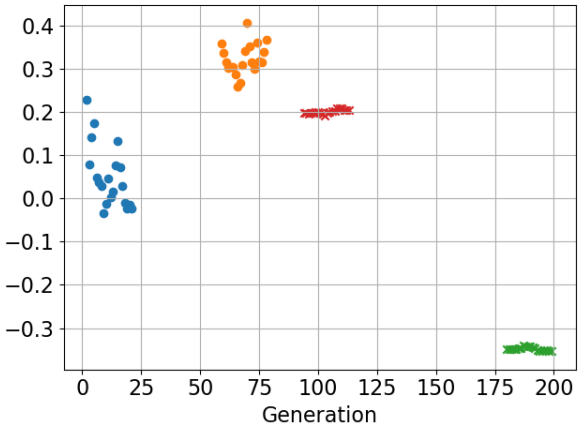



Figure 3.15: Mean

### 3.9.3 Characterizing CMA-ES Internals

CMA-ES starts by initializing five major components which are called state variables. Three of them are related to covariance matrix, one is for mean vector and the last one is step size, called sigma. A sample size is pre-determined to set the number of sampled out genes from the state variables. Now, at the start of each generation, CMA-ES samples out genes. To do that, it first multiplies the eigenvectors and eigenvalues with the sampled out values from normal distribution. Mean vector is added after multiplying with sigma. After getting the fitness score for each of the genes, these state variables updated based on the weighted fitness score for the next generation.

Figure 3.12, 3.13, 3.14 and 3.15 shows CMA-ES characteristics for these five major components. We choose one program to synthesize and run with GeneSys with different random seeds. Then we took 2 runs from *found* and *not found* case to see how CMA-ES behaves internally. As covariance matrix (C), eigenvectors (B) and eigenvalues (D) are all squared matrix, for 2d visualization we convert them into scalar value using principal component analysis (PCA) [75]. We do the same for the mean vector. Mean vector dimension is same as the number of variables in CMA-ES. Step size is a scalar value. We run GeneSys without any restart policy and took last N generations values for visualization purpose. For the *not found* cases, different CMA-ES component values become stagnant and hence, GeneSys is not able to find the solution. We notice that the mean vector is the most affected one by the stagnant values in the *not found* cases. The variance of the mean vector and step size is high for the *found* cases over generations. However, for the *not found* cases, it reduces significantly. For both cases, covariance matrix converges guided by the fitness score. Due to the ill fitness scores, it may stuck in a local minima where the solution is not present. This observation motivated us to use the restart mechanism and find the best restart policy for GeneSys.

## 3.10 Conclusion

In this paper, we presented a novel formulation of program synthesis as a continuous optimization problem. Essentially, it expresses a program with a tuple of functions, each taking a number of continuous parameters and mapping them to zero or more DSL tokens. We showed that a state-

of-the-art evolutionary approach, CMA-ES, can be used to solve the optimization problem very efficiently. Our proposed framework, GeneSys, consists of a mapping scheme to convert the continuous formulation into actual programs and restart policies for the evolutionary strategy to escape local minima. We compared the proposed framework with several recent program synthesis techniques and showed that GeneSys synthesizes, on average, 28.1% more programs within a fixed time budget at higher lengths. We envision GeneSys to be a stepping stone towards a completely new domain of research in program synthesis.

# 4. SPECSYN: AUTOMATIC SYNTHESIS OF SOFTWARE SPECIFICATIONS BASED ON LARGE LANGUAGE MODELS

## 4.1 Overview

Software configurations play a crucial role in determining the behavior of software systems. In order to ensure safe and error-free operation, it is necessary to identify the correct configuration, along with their valid bounds and rules, which are commonly referred to as software specifications. As software systems grow in complexity and scale, the number of configurations and associated specifications required to ensure the correct operation can become large and prohibitively difficult to manipulate manually. Due to the fast pace of software development, it is often the case that correct software specifications are not thoroughly checked or validated within the software itself. Rather, they are frequently discussed and documented in a variety of external sources, including software manuals, code comments, and online discussion forums. Therefore, it is hard for the system administrator to know the correct specifications of configurations due to the lack of clarity, organization, and a centralized unified source to look at. To address this challenge, we propose SpecSyn a framework that leverages a state-of-the-art large language model to automatically synthesize software specifications from natural language sources. Our approach formulates software specification synthesis as a sequence-to-sequence learning problem and investigates the extraction of specifications from large contextual texts. This is the **first** work that uses a large language model for end-to-end specification synthesis from natural language texts. Empirical results demonstrate that our system outperforms prior state-of-the-art specification synthesis tool by 21% in terms of F1 score and can find specifications from single as well as multiple sentences.

## 4.2 Introduction

Software configurations represent an essential component of software systems. System failures induced by software misconfigurations (i.e., not setting various configuration parameters according certain specifications) have become increasingly common [107, 145, 47]. Such misconfigurations

give rise to a range of issues, including application outages, security vulnerabilities, and inaccuracies in program execution [140, 38, 143]. The adverse impact of software misconfigurations is demonstrated in several high-profile cases [93, 28, 27, 41]. For instance, a configuration error caused by an internet backbone company, resulted in a nationwide network outage in 2017 [28]. Similarly, in 2019, millions of Facebook users were affected by a server misconfiguration outage [27]. Furthermore, a system configuration change led to a five-hour outage of AT&T's 911 service, preventing numerous callers from accessing the emergency line [41]. Therefore, the development of effective tools to help prevent software misconfigurations is of utmost importance.

The research community has recognized the criticality of software misconfiguration and proposed numerous efforts to address it by developing techniques to check, troubleshoot and diagnose configuration errors [8, 1, 143]. However, the majority of the approaches can generally only be applied after an error has occurred. Primarily, the root cause of software misconfiguration is attributed to human errors. Thus, a more effective strategy for avoiding such misconfigurations would be to guide or enforce the correct usage of configuration practices, rather than identifying and correcting them after a failure has already occurred. Typically, software configurations are set by software administrators. To aid these administrators, software vendors often release user manuals that describe different configuration specifications. These manuals, typically available in PDF or HTML format, provide guidance on the correct and recommended setup of configurations to system administrators, containing detailed textual descriptions of configuration parameters, their descriptions, usage, and constraints. Nevertheless, as these manuals are exceedingly voluminous, many administrators tend not to read them in detail and instead rely on intuition to configure software [142, 94], frequently leading to misconfiguration and subsequent software failures. Hence, it is crucial to develop an automatic tool that extracts configuration specifications from these sources, to provide guidance to administrators, or integrate automated tools that suggest best practices. Thus, this paper investigates the feasibility of building an automatic tool capable of extracting specifications from unstructured sources of configuration descriptions, which are predominantly written in a natural language such as English.

Specifications refer to a set of legitimate rules or guidelines that dictate the configuration of software. Failure to adhere to these specifications by configuring the software with invalid parameters may result in software malfunction. Extensive research has been conducted to extract software specifications from unstructured specification sources [139, 116, 92]. For instance, in PracExtractor [139], Xiang et al. utilize the Universal Dependency algorithm [30] to synthesize specifications from software manuals. This technique establishes a syntactic mapping between various parts of speech within a sentence. To construct the set of syntactic relationship trees, they initially collected samples from software manuals that contain valid specifications. They then attempted to match other sentences' syntactic relation with the collected syntactic relation tree. If a match was found, it implied that the samples also contained specifications and it was extracted based on the relation. The authors of ConfigV [116] have employed a rule-based approach to synthesize specifications from configuration files. This approach involves initially parsing a training set of configuration files, which may be partially correct, to create a well-structured and probabilistically-typed intermediate representation. A learner that utilizes an association rule algorithm is then employed to translate this intermediate representation into a set of rules. These rules are subsequently refined and ranked through rule graph analysis to synthesize specifications. Researchers have also tried to synthesize specifications from programming source code by employing static analysis [92]. However, specifications collected through this approach need more analysis and expert knowledge for refinement by humans.

The majority of prior methods for synthesizing specifications from unstructured sources have relied on rule-based approaches. However, such approaches are known to have limited generalizability and may require human intervention during certain steps of the synthesis process. Alternatively, learning-based approaches are better suited for discovering relationships in unstructured data, particularly those utilizing deep learning techniques that have shown significant success in various unstructured data domains, including natural languages, images, and videos. As the main objective of this paper is to synthesize specifications from a natural language source, we explore the potential of deep learning-based approaches to address this problem.

To this end, we have framed the specification synthesis problem as an **end-to-end sequence-to-sequence** learning problem. The resulting system is referred to as **SpecSyn**, which takes texts as an input and produces specifications as an output. The synthesis process involves two steps of prediction. First, it checks whether the input texts contain any specification. If they do, secondly, SpecSyn performs end-to-end synthesis of the specifications. Given that the input for specification synthesis is in the form of a natural language text, we have incorporated a large pre-trained language model to enhance the model's understanding of the context. One such widely used model is BERT [33], a deep learning-based model that pre-trains on a large corpus of English texts to learn the latent representations (i.e., a vector) of words and sentences within their respective contexts. We have fine-tuned [127] the BERT model for our specification synthesis task using a custom decoder (a decoder is a part of the language model that produces outputs based on the latent representations). The incorporation of this large language model has enabled us to synthesize not only simple single-sentence specifications but also complex sets of specifications from texts consisting of multiple sentences and parameters. Figure 4.1 shows the high level workflow of SpecSyn.

### 4.2.1 Contributions

We make the following technical contributions in this paper.

**Problem Formulation:** We formulate the task of software specification extraction from natural language texts as an end-to-end sequence-to-sequence learning problem. This approach involves mapping an input sequence, consisting of natural language texts, to an output sequence that comprises of single or a set of relevant software specifications.

**Contextual Model Integration:** In order to achieve accurate and effective extraction of software specifications from natural language texts, we propose to use the state-of-the-art BERT [33] model. By leveraging the acquired knowledge and advanced language processing capabilities of a pre-trained BERT model, we aim to extract relevant specifications from the text in a manner that is both accurate and efficient. Through empirical analysis, we demonstrate the effectiveness of SpecSyn in the context of software specification extraction from natural language texts. To the best of our knowledge, this is the **first** paper that uses a large language model for end-to-end synthesis of

configurations specifications from natural language texts.

**Complex Dependency Modeling:** The current investigation presents a model that is able to process text consisting of multiple sentences. Specifically, our proposed model is designed to effectively capture complex specification relations within longer text. Notably, the model is capable of discerning the relationships between multiple specifications contained within a single sentence, as well as extracting individual specifications that are connected in a meaningful manner within a text. The efficacy of our model is demonstrated through empirical analyses, which provide evidence of its ability to accurately identify and extract relevant information from longer text data.

**Generality:** The framework, SpecSyn, is capable of processing any natural language text, thereby rendering it independent of the source of textual data. Consequently, it does not solely rely on software manuals for the extraction of software specifications. Rather, it can be utilized to extract relevant specifications from a wide range of sources including software codebase comments, as well as online resources such as StackOverflow and discussion forums. This flexibility allows for the construction of a more comprehensive and robust set of specifications, thereby enhancing the overall effectiveness of the framework.

**Dataset:** We also contribute with a domain-specific dataset for the purpose of training and testing software specification extraction models. The dataset is composed of software specifications and is designed to be used in the context of natural language processing tasks related to software specification synthesis. By making this dataset publicly available, we aim to facilitate further progress in the field of software specification extraction and encourage the development of more accurate and effective models for this task.

### 4.2.2 Outline

The remainder of this paper is organized as follows. Section 4.3 describes the details of SpecSyn framework. First, the section lays out the specification definition, followed by discussion on specification extraction types and specification categories. Then, different specification sources and dataset construction approaches are described. After that, we describe the model development and integration of a large language model with our framework. Section 4.4 describes the experimental

setup and experimental results. Section 4.5 introduces the background and related work. Section 4.6 discusses potential future work. Finally, Section 4.7 concludes this paper.

## 4.3 Specification Synthesis Framework

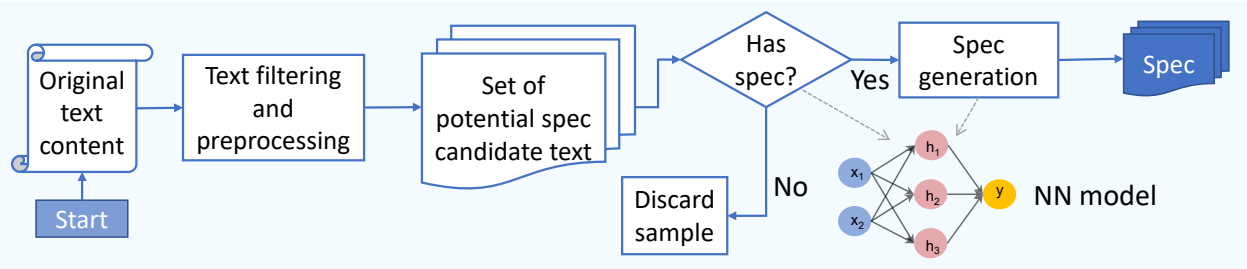



Figure 4.1: Overview of SpecSyn

### 4.3.1 Software Specification

Software specification is a set of rules that consists of relation between various keywords, numbers or pre-formatted string. Here, keywords represent different configuration parameters. Specification defines how a configuration should be presented by outlining the specific rules and requirements for configuration format and structure. Formally, a specification can be defined as Definition 1.

**Definition 1:** A specification $S$ is defined as $S = \{R_i\}$, where $R_i$ is a rule represented by a tuple, $R_i = \langle K_i, V_i, L_i \rangle$. Here,

$K_i \in$ *Keyword*

$V_i \in \{\mathbb{R}, Keyword, S\}$ *where* $\mathbb{R} = Set\ of\ Real\ Numbers, S = String$

$L_i \in \{\emptyset, =, \neq, >, <, AND, OR, Interval, Set, Use, With, String\ Format\}$

The goal of SpecSyn is to analyze the natural language texts from various sources and produce a specification, if there is any in the texts. Next we are going to classify types of specification

extraction as well as various categories of specifications.

#### *4.3.1.1 Specification Extraction Type:*

Based on the presence of specification in the text, we categorize specification extractions into two major types: *Simple* and *Complex*. By distinguishing between these two types, we are able to better evaluate the performance of our extraction process. Previous research has focused exclusively on the *Simple* extraction category, while the *Complex* category demands more sophisticated extraction process and modeling techniques. Table 4.1 shows examples of text for different specification extraction types.

| **Type** | **Example** |
|---|---|
| Simple | It is necessary to use a number greater than 1500 for $user\_port$ |
| $\text{Complex}_{\text{Single}}$ | The default pointer size in bytes is used when $max\_rows$ option is specified. This variable should be between 2 and 7. |
| $\text{Complex}_{\text{Multi}}$ | $have\_ssl$ and $have\_open\_ssl$ need to be set True to enable secured connection. |

Table 4.1: Examples of specification extraction types

**Simple Extraction:** *Simple* extraction refers to specifications extraction process where the specification is located within a single sentence containing only one specification. The majority of specification extractions fall within this category. The task of modeling *Simple* extractions is comparatively less intricate due to the concise nature of general sentences. The specifications are also defined in a more straightforward manner. We observed that fewer training examples are required to train a model for extracting *Simple* categories as opposed to *Complex* categories.

**Complex Extraction:** The second type, known as *Complex* Extraction, consists of cases where multiple sentences are required to extract specifications from the text or when there are multiple specifications intertwined in a text that need to be extracted. *Complex* extraction has been further subdivided into two distinct subcategories, namely $Complex_{Single}$ and $Complex_{Multi}$. The $Complex_{Single}$ category is applicable when only one specification is present in the text, but the

extraction process requires examining multiple sentences. Typically, in such cases, the first sentence identifies the parameter keyword, while the subsequent sentences describe the specification. The sentences usually connected by some pronoun. In order to extract the specification and identify the keyword to which it refers, all sentences are necessary to look at. On the other hand, $Complex_{Multi}$ refers to situations where multiple specifications are defined within a text, and multiple parameter keywords are used to refer to a single specification. Basically in this category multiple specifications of multiple keywords are described in a compact intertwined fashion in a natural language.

| **Category** | **Example** |
|---|---|
| Quantitative | It is recommended to raise the *ulimit* to 10,000, but more likely 10,240 because the value is usually expressed in multiples of 1024. |
| Utilization | Mount option *sync* is strongly recommended since it can minimize or avoid reordered writes, which results in more predictable throughput. |
| Interrelation | If you are having problems with the service, it is suggested you follow the instructions below to try starting *httpd.exe* from a console window, and work out the errors before struggling to start it as a service again. |
| Attribute | To avoid the ambiguity, users can specify the plugin option as *–pluginsql-mode*. Use of the *–plugin* prefix for plugin options is recommended to avoid any question of ambiguity. |
| Generic | It is recommended but not required that *–ssl-ca* also be specified so that the public certificate provided by the server can be verified. |

Table 4.2: Different categories of specifications with examples

#### *4.3.1.2 Specification Categories:*

We categorize the specification into five major categories based on the underlying specification definition and property. Among them, *Quantitative* represents specifications that are quantifiable, such as numerical or boolean comparisons. *Utilization* categorizes any usage of keyword for any particular task, *Interrelation* categorizes correlation between keywords, *Attribute* categorizes different attribute such as path, domain etc., and *Generic* categorizes general suggestions without any specific criteria. The categories and their examples are described in table 4.2.

The majority of the specifications identified in our study can be categorized as *Quantitative*. *Quantitative* specification can have multiple definitions. The *Generic* category lacks a specific definition and is primarily characterized by suggestions or recommendations. The remaining three categories are infrequent and each have only one specification definition. Table 4.3 describes the definition and pattern of each categories.

The significance of specification categorization lies in its potential to facilitate more effective utilization of detected specifications during later stages. Even though in our work the initial detection of specifications is a binary prediction process independent of their categories, the ability to categorize specifications can provide valuable insights for optimizing their deployment in later stages such as suggesting them to system administrators or incorporating them to other tools. Therefore, in our work, we also demonstrate the capability of our framework to accurately predict specification categories with different decoders. Although these categories are intuitive and general to notice, we are inspired about them from prior work [139]. However, previous research has not explored the detection capabilities of various categories of specification as we have done in our framework.

| **Category** | **Definition** | **Pattern** |
|---|---|---|
| Quantitative | p == v, p < v \| p > v, p ∈ [v, v'], p ∈ {v, v'}, | $v_{<value>}$, $less_{syn}$ \| $more_{syn}$ than $v_{value}$, $between_{syn}$ $v'_{<value>}$ to $v2_{<value>}$, $v_{<value>}$ or $v'_{<value>}$ |
| Utilization | use(p) | $used_{syn}$ \| $useful_{syn}$ |
| Interrelation | with(p, p'), prefer(p, p') | $along_{syn}$ with $p'_{para}$ $prefer_{syn}p'_{para}$ |
| Attribute | format(p,f) | $f_{<format>}$ |
| Generic | {recommended, prioritize, ...} | |

Table 4.3: Specification definition and patterns

### 4.3.2 Data Collection

Data is the lifeline of any deep learning-based solutions, and collecting data to train and test any deep learning-based system is one of the most significant and challenging tasks. For specification synthesis, it becomes even more difficult since it requires highly specific and domain-dependent data. Since there is no standard dataset available for specification synthesis, we have to collect and create our own datasets, which is a time-consuming and resource-intensive task. However, despite the challenges, there are several sources where software specifications can be found. The main source of specifications is the software manual. However, it can also be derived from comments embedded in the software code base, particularly when the source code is publicly available. It may be obtained from various other sources also such as Stack Overflow, online discussion forums, and other community-driven platforms. In the following section, we will describe different sources of software specifications and the specification collection process.

| **Name** | **Software type** | **Format** | **Pages** | **Keyword** |
|---|---|---|---|---|
| MySQL | Database | PDF | 6644 | Yes |
| PostgreSQL | Database | PDF | 3055 | Yes |
| HDFS | Distributed Storage | HTML | 2331 | Yes |
| HBase | Distributed Storage | HTML | 787 | Yes |
| Cassandra | Distributed Storage | HTML | 50 | No |
| Spark | Distributed Computing | PDF | 66 | Yes |
| HTTPD | Web Server | HTML | 1009 | Yes |
| NGINX | Proxy | HTML | 900 | No |
| Squid | Proxy | HTML | 330 | Yes |
| Flink | Stream Processing | HTML | 1434 | Yes |

Table 4.4: Description of software manuals

#### *4.3.2.1 Data Source:*

We mainly collect specifications from three major sources. The most important source of the specification is software manuals. The major portion of specifications is collected from manuals.

We also collected specifications from source code comments and other online sources.

**Software Manuals** The main source of software specification is the software manual. These manuals are typically provided by the corresponding software vendor and contain detailed information about how to use the software, including its features, functions, and limitations. Software manuals are often available in different formats such as PDF, HTML, or in online. Table 4.4 details the summary of software manuals that we used to collect specifications. In order to construct our dataset, we gathered specifications from 10 diverse software manuals, spanning a range of software domains. Typically, software manuals are extensive in nature, containing a considerable amount of information. For instance, MySQL's manuals consist of a total of around 6500 pages. Due to the sheer volume of information contained in these manuals, it is impractical to read them line by line. Therefore, a keyword-based filtering method is employed to extract only the relevant sentences for closer examination. These keywords typically correspond to configuration parameters, which can be located within the manuals. The majority of software packages typically have their keywords listed either in the manuals or on their configuration page. However, in our study, we were unable to find such a listing for NGINX and Cassandra. In the case of these two software packages, we solicited human assistance to aid us in extracting the specifications. While simple specifications can often be derived from individual sentences, more complex specifications may require analysis of neighboring sentences. As complex specifications may be embedded within the text, the focus is on the section of the text where the keyword is present.

**Other Sources** In addition to the manual-based specification, we also collected specifications from two other additional sources: software code base comments and online discussion forums. As our method operates with natural language, it is independent to the source of the data, provided that it is composed in a natural language. The inclusion of these alternative sources is primarily intended to demonstrate the generalizability of our framework across a broad range of text-based specification descriptions.

To demonstrate the source code base specification generalizability, we develop and integrate a parser into our framework and apply the parsing to MySQL source code only. However, the same

methodology can be applied to parsing other software as well. For the purpose of parsing comments from MySQL source code, we use a Python-based parser. It is essential to be cautious when parsing the source code in this manner, as the codebase also contains commented-out codes. But we are only interested in text-based comments. Therefore, commented out code needs to be discarded for a better-refined dataset. In addition, a crawler is also developed to extract software keyword-specific posts from StackOverflow to augment the specification-related dataset.

| **Tag** | **Pattern** |
|---|---|
| <bool> | "enable" \| "on" \| "true" \| "disable" \| "false" \| "off" \| ... |
| <num> | $\forall$ w $\in \mathbb{R}$ |
| <unit> | “byte” \| “MB” \| “ms” \| “%” \| ... |
| <keyword> | $\forall$ w $\in$ Configuration Parameter Name |
| <format> | “email address” \| “absolute path” \| “domain name” \| ... |

Table 4.5: Description of pattern for data composition

#### *4.3.2.2 Data Composition:*

Prior to generating the candidate specification texts from the original contents, our system refines the dataset through a series of preprocessing steps. First, the texts are divided into smaller candidate texts based on the extraction type. Then it verifies the presence of the keyword in the candidate texts. If the keyword is absent, the candidate texts are discarded and not considered as a potential sample for further processing. Both the extraction types (i.e., *Simple* and *Complex*) require the presence of relevant keywords in the candidate texts. The keywords can easily be found for each of the software in different places (e.g., manual, web). Upon identifying potential candidates, the system proceeds to search for predefined patterns within the candidate text as specified in Table 4.5. These patterns are then replaced with corresponding tags. In instances where identical patterns are present multiple times within the text, tags are differentiated through the use of different identifier. This process enhances the generality of the potential candidates, thereby enabling easier detection

and synthesis of specifications by the model. Thus, each candidate comprises of tuple $\langle C, T \rangle$, where $C$ represents the candidate text and $T$ represents the associated pattern tag set. After detecting and synthesizing specification based on $C$, the system can reconstruct the original representation of the specification by replacing the tag in synthesized specification of $C$ with the corresponding pattern in $T$.

One may argue that passing the keyword set for finding potential specification candidate text may introduce more redundancy. Rather, the system should identify the keyword itself. However, knowing the keyword set for a particular software is necessary. Multiple software can have same keyword with different specification rule. Therefore, if it is not known for which software the specifications are being synthesized, then the system can synthesize a syntactically correct but potentially wrong specification for a software. This is especially true if SpecSyn synthesizes specification from other sources. Therefore, knowing for which software the specifications are being synthesized and corresponding keyword set is quite important. Also, Keyword-based filtering enables us to accomplish two goals. First, it eliminates a significant portion of the samples that are irrelevant in nature. Since a model can easily recognize these and accurately classify them as non-specification, the model's performance would be heavily skewed towards making accurate predictions of true negatives. Therefore, considering sentences that has keywords or solely considering neighboring sentences with keywords will allow for a fairer performance comparison. Second, keyword based filtering also discards a major portion of false positives. For example text like "See page 157 for details of MySQL 11.7.8" does not hold any specification, but this can be detected as specification if the model is not trained with larger number of samples. In our study, we discovered that a model can also be trained for all cases even without implementing keyword-based filtering. However, such approach requires a larger number of samples to be used for model training. Also, identifying syntactically valid but semantically incorrect specifications requires additional failure checks through software execution. Therefore, due to resource and time constraint, in this project, we pursue a keyword-based filtering process and keep the other ideas as potential future work.

### 4.3.3 Model Development

#### *4.3.3.1 Contextual Model Integration: BERT*

Contextual model integration refers to the process of combining language models to improve the performance of natural language processing tasks. The idea is to leverage the strengths of different models and combine them in a way that captures the complex relationships between words and their contexts in a given text. One of the approaches of contextual model integration is to use a hierarchical model, where one model is used to capture the overall context of the input sequence and another model is used to make more specific predictions based on the context. In our case, we use BERT (Bidirectional Encoder Representations from Transformers), a pre-trained large language model [33], that is trained on a large dataset of natural language texts and we fine-tune it with our task-specific custom decoder.

BERT [33] is a powerful neural network model that has revolutionized the field of natural language processing (NLP) in recent years. It was first introduced by Google in 2018 and has since become one of the most widely used and effective language models in NLP. The core idea behind BERT is to leverage the power of Transformer-based architectures [132] to create a deep bidirectional language model that can capture contextual information from both directions of the input sequence. Unlike traditional language models, which are trained in a left-to-right or right-to-left fashion, BERT is trained using a masked language model (MLM) objective that randomly masks certain tokens in the input sequence and requires the model to predict the missing word based on the surrounding context. One of the key innovations of BERT is the use of multiple layers of self-attention to capture complex relationships between words in the input sequence. Each layer of the model contains a self-attention [132] mechanism that allows the model to attend to different parts of the input sequence and capture dependencies between words that are far apart in the input. Additionally, BERT uses a combination of word embeddings, positional embeddings, and segment embeddings to capture both the meaning and position of words in the input sequence. All of these embeddings are self-learned through back-propagation while training on a larger dataset.

BERT has proven to be highly effective for a wide range of NLP tasks, including text classification, question-answering, language translation, etc . [103]. In order to adapt BERT to specific tasks, researchers typically fine-tune [127] the model by adding a task-specific output layer and training the model on a task-specific dataset. The fine-tuning process allows the model to learn task-specific features and improve its performance on the target task.

#### *4.3.3.2 Specification Detection and Generation*

The specification synthesis process is a two-step procedure that involves specification detection and generation. In the first step, the text is examined to ascertain the presence or absence of a specification. If a specification is detected, in the second step the specification is synthesized. Figure 4.2 shows the neural network model for SpecSyn. Specification detection is a binary classification problem, where a BERT encoder is utilized. An encoder is a sequence of layers to convert the input texts in a hidden vector $h_c$. A special token $[CLS]$ is added at the beginning of the text to generate $h_c$ of the entire sequence for classification. This hidden state is subsequently passed to the decoder. A decoder is a sequence of layers to produce the final output from $h_c$. The softmax layer in the decoder produces the binary prediction of the presence or absence of a specification in the input text. Basically, the specification classification process can be described as Equation 4.1-4.3, where $X = \{x_{CLS}, x_1, x_2, ..., x_n\}$ is the tokenize set of input text, $x_{CLS}$ is the special token, $W$ is the weight matrix for the custom decoder and $c$ is the expected output prediction. We fine-tune $W$ to maximize the log probability of the correct class. For specification classification tasks, BERT takes the final hidden state $h_{\text{BERT}}$ of the first token $[CLS]$ as the representation of the whole sequence. A simple softmax layer-based classifier is added as the custom decoder on the top of BERT encoder to predict the probability of label c.

$$h_{\text{BERT}} = f_{\text{BERT}}(X) \tag{4.1}$$

$$\boldsymbol{h_c} = f_1(W_1 h_{\text{BERT}}) \tag{4.2}$$

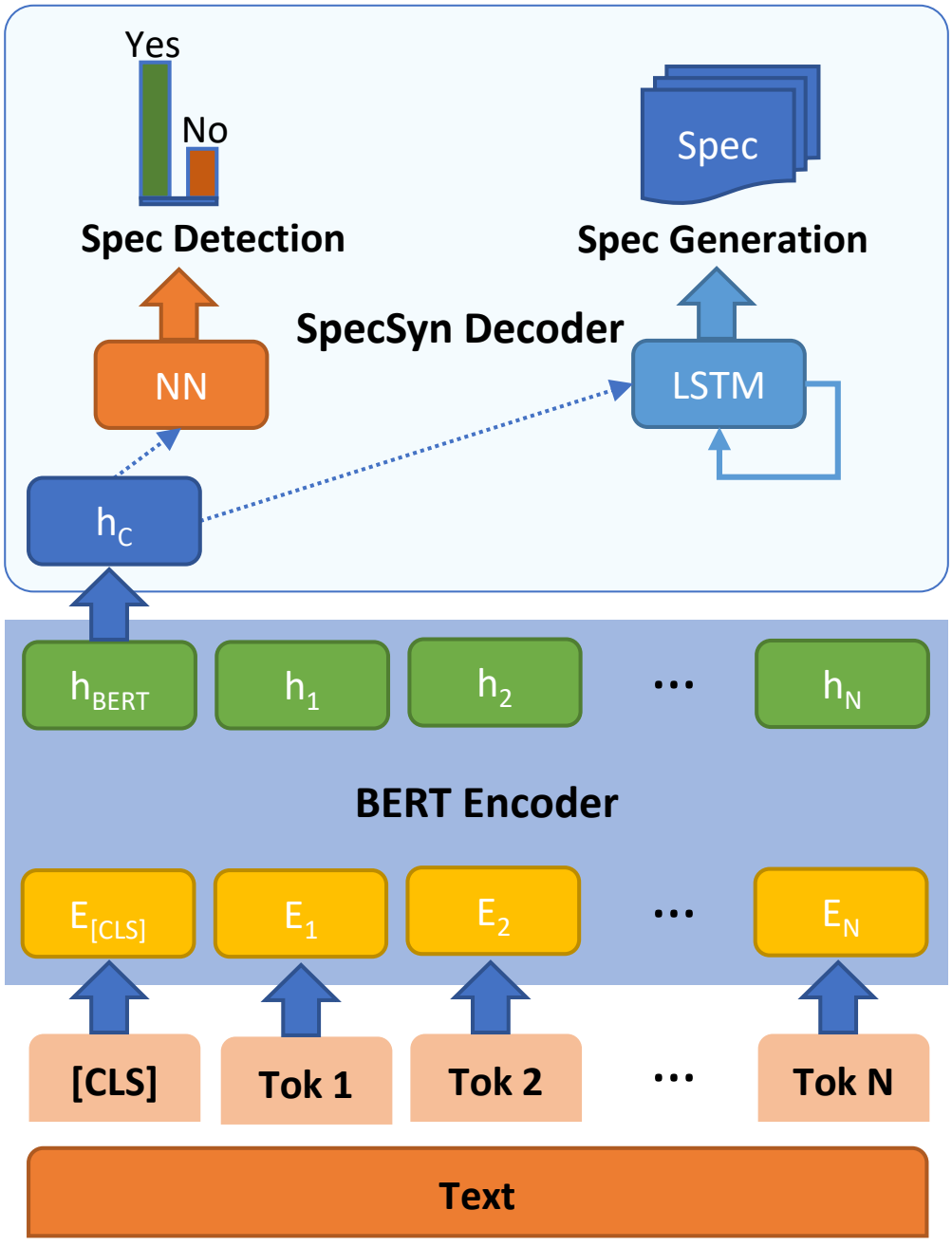



Figure 4.2: SpecSyn model architecture

$$p(c|\boldsymbol{h_c}) = \text{softmax}(W_2\boldsymbol{h_c}) \tag{4.3}$$

For the specification synthesizing process, we use the same hidden state $h_c$ and pass it to a different sequence generation decoder. This decoder synthesizes the specification according to the pattern defined in Table 4.3. The synthesizing decoder is an LSTM-based [59] recurrent neural network defined as Equation 4.5. It takes $h_c$ as the hidden state and a start token to start the synthesis process and construct the specification as it goes.

$$h_0 = \boldsymbol{h_c} \tag{4.4}$$

$$o_i = \text{LSTM}(h_{i-1}, o_{i-1}) \tag{4.5}$$

$$o_{\text{Specification}} = \{o_1, ..., o_m\} \tag{4.6}$$

A potential argument in favor of utilizing a single decoder for both the specification classification and generation tasks may be put forth. However, given that the majority of texts does not contain specifications and the generation of specifications is a complex task in comparison to binary classification, we found in our study that the use of two separate decoders yield better results in classifying and generating specifications within a text.

In our study, we categorize specifications into different categories. We can predict different classes of these categories with the same method. Upon detection of a specification, a decoder with a softmax layer having the desired number of classes can be used to classify different categories. A detailed analysis of the result is presented in the results section for this.

#### *4.3.3.3 Loss Function*

In the context of specification detection, a binary classifier is utilized to determine whether a given text contains any specifications or not. For this, we use weighted cross entropy loss function [6] defined by Equation 4.7 where $M$ is the number of classes, $\log$ is natural log, $y$ is binary indicator (0 or 1) if class label $c$ is the correct classification for observation $o$, and $p$ is the predicted probability observation $o$ is of class $c$.

$$L = -\sum_{c=1}^{M} w_c y_{o,c} \log(p_o, c) \tag{4.7}$$

Weighted cross entropy loss is a commonly used loss function in classification problems when dealing with imbalanced datasets [78, 55]. This method assigns a higher weight to the minority class samples to improve the performance of the model. The weight is determined based on the distribution of the minority class samples in the dataset. By assigning a higher weight to this class samples, the model is encouraged to focus more on correctly classifying these. Weighted cross entropy loss has been shown to be effective in improving the performance of models on imbalanced datasets, particularly in text classification tasks. Its usage can lead to a more accurate and reliable classification of the minority class. In our specific dataset, it has been observed that the number of texts containing specifications is significantly lower than those that do not contain any specifications, leading to an imbalanced dataset. In order to address this issue, we have utilized weighted cross entropy loss function. The weight is determined based on the distribution of specification-containing texts in the training set.

## 4.4 Results

In total, we collected 300 specifications from various sources, including 10 software manuals and other resources. The majority of the specifications were extracted from two sizable software manuals, due to their extensive page count. We categorize the software manuals into 3 groups according to their software types. From Table 4.4, first 2 softwares are categorized as Database

software, next 4 as Distributed System and last 4 are categorized as Proxy.

The quantity of available specifications are limited, as evidenced by prior research [139] too . Given that the dataset used in the previous study is not publicly accessible, we collected same number of specification as them. A deep learning based network requires a moderate number of training examples in order to train. However, given the lower number of available specification, it becomes challenging to create training and testing sets out of them. Therefore we generate synthetic training examples by sampling 50 real specifications from our collected set and create the training set. Since we know the specification of these random samples, we put random text inside them to create synthetic samples. That way we create our training set that consists of 3000 samples in total. Thus, we were able to create a good amount of samples for the training and we use rest 250 samples as the testing set to evaluate.

In our model, we used a pretrained BERT model available online at https://huggingface.co/. Although, the BERT model is large in nature, since this is pretrained, we do not need to train them from scratch. We design our own decoder neural network for our own task. We use a feed forward neural network consisting of 2 hidden layer of 50 neurons each with a softmax function on top for specification detection. Weighted cross entropy loss is used for this decoder. To generate specifications, we use an LSTM-based recurrent neural network with 20 hidden units. Given the limited number of training examples, our designed neural network demonstrates sufficient capacity to achieve better prediction performance. We train our model for 100 epochs until the training converges.

| Software Type | PracExtractor | | | SpecSyn | | |
|---|---|---|---|---|---|---|
| | Precision | Recall | F1 Score | Precision | Recall | F1 Score |
| Database | 0.84 | 0.58 | 0.68 | 0.95 | 0.90 | 0.92 |
| Distributed System | 0.79 | 0.50 | 0.61 | 0.90 | 0.75 | 0.82 |
| Proxy | 0.81 | 0.52 | 0.64 | 0.92 | 0.79 | 0.85 |
| **Total** | 0.81 | 0.54 | 0.65 | 0.92 | 0.81 | 0.86 |

Table 4.6: SpecSyn's specification synthesis ability compare to PracExtractor

### 4.4.1 Demonstration of Synthesis Ability

We compare our work with state-of-the-art specification synthesis tool named PracExtractor [139]. PracExtractor uses Universal Dependency algorithm. This tool only work for *Simple* extraction type specification. On average a *Simple* extracted type specification is 3 tokens long. Therefore, given the short length of specifications, our model is capable of synthesizing them accurately once they are detected. PracExtractor does not have any detection mechanism. Therefore, when they are able to synthesize a specification they are counted as successfully detected.

Table 4.6 shows the specification detection performance comparison between PracExtractor and SpecSyn. It showed result with three different software groups - Database, Distributed System and Proxy. PracExtractor performs poorly compared to SpecSynin all three cases. In case of Database software, SpecSynhas higher Precision $0.95$ compared to $0.84$ Precision of PracExtractor. The Recall score is $0.90$ which is significantly better than Recall for PracExtractor $(0.58)$. Consequently F1 score for SpecSyn$(0.92)$ is significantly better than that of PracExtractor $(0.68)$ For Distributed System software type, SpecSynhas better Precision, Recall and f1 score compared to that of PracExtractor. In this case, Precision for SpecSynis $0.90$ vs Precision of PracExtractor is$0.79$, Recall for SpecSynis $0.75$ compared to $0.50$ Recall value for PracExtractor. In case of F1 Score, SpecSynhas higher value $0.82$ compared to the state of the art PracExtractor model $(0.61)$ In case of Proxy applications, we see similar trends in the result. Precision for SpecSynis $0.92$ and Precision for PracExtractor is $0.81$ only. Recall is also much better for SpecSyn$(0.79)$ compared to Recall value for PracExtractor $(0.52)$. So F1 score for Proxy type software is higher in SpecSyn$(0.86)$ compared to F1 score of PracExtractor $(0.65)$.

For combined results of all three types of software, we can see that SpecSynhas higher Precision $(0.92)$ compared to PracExtractor that has $(0.81)$ Precision only. This indicates that $92\%$ of the predicted relevant results were accurate for SpecSynwhile only $81\%$ of the predicated relevant results were accurate for PracExtractor. SpecSynhas Recall value of $0.81$ compared to a mere $0.54$ Recall value for PracExtractor. Which signifies that SpecSyncould correctly identify $81\%$ of the relevant cases where PracExtractor could only correctly identify $54\%$ showing a huge improvement

($1.5\times$) in detecting relevant cases. Overall, SpecSynhas a F1 score of $0.86$ compared to F1 score of $0.65$ by PracExtractor which clearly explains that SpecSynperformed much better than the existing state of the art model.

| | True Positive | False Positive | False Negative |
|---|---|---|---|
| PracExtractor | 82% | 18% | 73% |
| SpecSyn | 94% | 6% | 21% |

Table 4.7: Confusion matrix of specification detection capabilities

Table 4.7 shows confusion matrix for specification detection between SpecSyn and PracExtractor. It shows SpecSynprovides better true positive and false positive compare to PracExtractor. In terms of true positive it is 12% better in prediction. PracExtractor gives much high false positive (3x) prediction compare to SpecSyn. In terms of false negative detection, SpecSyn gives 21% compare to 73% of other tool.

| Extraction Type | Detection | | | Generation |
|---|---|---|---|---|
| | Precision | Recall | F1 Score | |
| Simple | 0.92 | 0.81 | 0.86 | 100% |
| $\text{Complex}_{\text{Single}}$ | 0.81 | 0.70 | 0.75 | 87% |
| $\text{Complex}_{\text{Multi}}$ | 0.73 | 0.64 | 0.68 | 83% |

Table 4.8: SpecSyn's Synthesis capability across different specification extraction type

Table 4.8 demonstrates the specification synthesis results across different specification extraction type. The F1 score for specification detection in *Simple* extraction type is 0.86 where the maximum F1 score is 0.75 for the *Complex* category. The Precision in *Simple* extraction type is 0.92 that implies it is very good at detecting specification. Among *Complex* type $Complex_{Single}$ can detect specification with Precision of 0.81 where $Complex_{Multi}$ can detect specification with a Precision

score of 0.73. However, the later category has higher F1 score than that of other. In terms of specification synthesis, 100% specification can be synthesized in *Simple* category given they are correctly detected. Since, when the model is accurate at detecting specification the chances of synthesize a specification is higher. Moreover, in *Simple* cases average synthesized specification length is 3. Therefore, it is able to achieve 100% accuracy in terms of specification generation. On the other hand, $Complex_{Single}$ synthesize 4% more specification compare to $Complex_{Multi}$.

| **Category** | **Precision** | **Recall** | **F1 score** |
|---|---|---|---|
| Quantitative | 0.95 | 0.82 | 0.88 |
| Utilization | 0.8 | 0.89 | 0.84 |
| Interrelation | 0.8 | 0.89 | 0.84 |
| Attribute | 0.95 | 0.90 | 0.92 |
| Generic | 1.0 | 0.71 | 0.83 |

Table 4.9: SpecSynSynthesis ability across different specification categories

### 4.4.2 Performance of Specification Categorization

Table 4.9 shows Precision, Recall and F1 score across different categories of specifications - namely for Quantitative, Utilization, Interrelation, Attribute, Generic categories. For Quantitative category, SpecSynhad a Precision of $0.95$ and Recall of $0.82$ which gave a F1 score of $0.88$. For both Utilization and Interrelation categories, Precision value was $0.8$ while Recall was 0.89 and F1 score was 0.84. For Attribute category, SpecSynperformed better than previous categories with very high Precision of $0.95$ and Recall value of $0.9$ which gave the highest F1 score of $0.92$ across any categories. However, the highest Precision was achieved for Generic categories with $1.0$ value that is all the predicted relevant results were accurate for SpecSyn. However, for Generic category Recall value was a bit lower than other categories with 71% of the relevant cases detected. So, F1 score was lower than other categories with a value of $0.83$

### 4.4.3 Characterization of Models

| Language Model | Precision | Recall | F1 score |
|---|---|---|---|
| $\text{BERT}_{\text{Tiny}}$ | 0.91 | 0.82 | 0.86 |
| GPT[108] | 0.91 | 0.80 | 0.85 |
| BERT | 0.92 | 0.81 | 0.86 |

Table 4.10: Model characterization with pre-trained language models

We have used three different pretrained models and collected their Precision, Recall and F1 score. As we can see in Table 4.10, BERT [33] have the highest Precision among the models we have tested. BERT has Precision value of $0.92$, compared to Precision value of $0.91$ for both $\text{BERT}_{\text{Tiny}}$ and GPT [108] models. In case of identifying the fraction of relevant items that can be retrieved, i.e. the Recall score, $\text{BERT}_{\text{Tiny}}$ performs slightly higher $0.82$ than the BERT model $0.81$. GPT model has the lowest Recall score of $0.80$ among these three models. Combining these two score, the harmonic mean of Precision and Recall, known as F1 score, shows that both BERT and $\text{BERT}_{\text{Tiny}}$ models perform better with F1 score of $0.86$ compared to the GPT model which has a F1 score of $0.85$. However, the performance of each of these three models is not significantly different from one another.

## 4.5 Related Works

*Software configuration and specification extraction with NLP.* Numerous studies have addressed the challenge of effectively diagnosing and solving software configuration problems. One line of research focuses on using static analysis techniques to identify configuration errors before they result in system failures [39, 106, 92]. Other works, such as [61] and [7], aim to enhance system observability to detect configuration errors in situ. Some approaches propose proactive methods to detect and troubleshoot customer issues, such as [64] and [141], which introduce early detection systems to prevent configuration errors from causing significant damage. Online

error detection systems based on context have been proposed in [147], while [149] proposes a misconfiguration detection system based on system environment and correlation information. Shared knowledge or event traces have been used to diagnose misconfigurations in [3] and [146], respectively. An automated troubleshooting approach based on dynamic information flow analysis is presented in [9], while [104] proposes a parallelized approach to information flow queries. Precomputing approaches to possible configuration error diagnoses have been proposed in [105]. [133] proposes a troubleshooting approach based on peer pressure, while [134] suggests a state-based approach to change and configuration management. On the other hand, [136] proposes a search-based approach to configuration debugging, and [150] introduces an automated diagnosis system for configuration errors. In addition to technical approaches, some papers emphasize the importance of not blaming users for configuration errors and instead focusing on developing better tools and processes to prevent them, such as in [142]. Probabilistic approaches have also been proposed to learn configuration file languages and identify and diagnose configuration problems, as demonstrated in [117]. Similarly, [116] presents an association rule learning-based approach to synthesize configuration file specifications from a set of example configurations. Also, [125] proposes a causality analysis-based technique to identify the root cause of configuration errors and automate the configuration management process.

One particularly influential work in this area is the PracExtractor [139], which employs natural language processing to analyze specific configurations and convert them into specifications to identify potential system admin flaws. However, PracExtractor has limitations, such as inflexibility and low generalization ability due to the specific format required for accurate extraction. Additionally, PracExtractor struggles with large paragraph settings, which our SpecSyn system seeks to improve upon. Other research efforts have also explored methods for inferring specifications from text [128, 129, 96, 148, 151, 152, 138, 36, 82, 24], but they too have limitations. Our work builds on these prior efforts by leveraging their insights to provide a more accurate and effective model for specification extraction.

*Specification extraction using Knowledge Base.* The use of a Knowledge Base (KB) has been

explored in prior research for specification extraction, as seen in ConfSeer[101]. However, similar to PracExtractor, this approach has its limitations. ConfSeer relies on a KB to analyze potential configuration issues, which can make it feel more like a search engine. While this approach has its benefits, it can also limit flexibility. SpecSyn aims to address this limitation by analyzing unstructured data to provide a greater variety of specifications. Other systems have been proposed to assist with finding configurations, such as[133, 136], but they come with their own issues, such as high overheads and requirements for large datasets. Associated rule learning has been used in previous studies to address dataset issues[84, 11, 49, 74].

*BERT based extraction and NLP* In contrast to previous studies, we employed and fine-tuned the BiDirectional Encoder Representations and Transformers model, or BERT[33], to gain a better understanding and improve the training of our dataset. BERT is a new natural language processing model that offers a more in-depth and refined fine-tuning technique. However, there are some limitations associated with using BERT. As it was developed in 2018, it is still a relatively new model and may not be fully developed with regard to training sets. Additionally, it can be expensive to train and result in slower training times due to its many weights[102]. Despite these limitations, BERT does an excellent job of processing specific input and producing output with new specifications, providing great flexibility as it eliminates the need for a KB as used in ConfSeer and can expand upon the findings of the PracExtractor system.

In recent times, deep learning is heavily used to address diverse system-related issues, such as program synthesis [85, 86], partial program correction [48], bug fixing [77] etc. Several studies have employed natural language processing (NLP) to inspect and analyze software configurations, particularly in the context of security analysis[124]. This work focused on analyzing two security systems, CIS and Siemens, and trained two models, LDA and BERT, which is the model used in our study. However, this study had certain limitations, such as the narrow focus on only two types of security systems, which could limit the diversity of the training data. Moreover, the study found that the BERT model, according to their metric of measurement, was not accurate enough to detect misconfigurations. Nevertheless, this study provided a useful baseline for identifying configuration

issues using NLP and made their dataset publicly available on Kaggle, facilitating further research in the field.

Recent studies have also investigated software configuration and misconfiguration using state analysis techniques[76]. In[76], the authors developed ConfDetect, a system that analyzes log files and ranks them based on specific criteria, which can effectively predict misconfiguration errors. The system also utilizes NLP to extract logs. The authors reported substantial accuracy in diagnosing configuration errors. However, the system was only tested on three different systems, whereas our study has more variety in terms of data testing. Additionally, ConfDetect utilizes a knowledge base to detect configuration errors, which could limit its flexibility. Despite these limitations, the study provides valuable insights into finding proper software misconfiguration.

### 4.6 Discussion and future works

In this section, we aim to highlight some potential observations and discuss them in detail.

*Why use LSTM base decoder for generating specifications while Transformer base decoder is considered as the state-of-the-art?*

The Transformer architecture offers advantages over the LSTM-based architecture in two scenarios: firstly, when dealing with a tremendously large dataset (i.e. 100GB) that requires parallel training without any previous recurrence dependency, and secondly, when handling very long sequences [132]. In our specific case, the dataset that we use is comparatively very small, and the sequence to be generated is also relatively short. As a result, the utilization of a Transformer-based decoder over an LSTM-based decoder will not bring any benefits.

*Why applying data composition instead of passing original text?*

Data composition helps the model to generalize better. While it is feasible to train a model without data composition and achieve similar performance results, doing so would require a larger quantity of data and can be left for future exploration.

*How long sequence has been used to synthesize specification?*

In this study, we limit ourselves to check one sentence for *Simple* extraction type and two sentences for *Complex* extraction type. Our findings indicate that two sentences adequately cover

the majority of *Complex* extraction type specifications. Furthermore, our sample sentences have an average length of approximately 150 characters, which implies that we check a maximum of around 300 characters for *Complex* extraction. We also train our model with a maximum of two sentences long samples. As a result, the model's performance will be poor for longer sequences than it is trained on. Hence, it would require more data samples and a potentially larger parametric model architecture to overcome this issue. Therefore, we keep this as a future work. It would be valuable to investigate very long sequences and conduct a sequence length sensitivity analysis.

Furthermore, regarding comments written in software source code, it is possible to parse a greater number of software programs. Additionally, examining the neighboring source code can provide a better understanding of the context of the comments. For extracting specification from other online discussion forums, a more exhaustive search could be conducted to mine a larger set of specifications. However, this is beyond the scope and context of our current project.

### 4.7 Conclusion

In this paper we introduces a deep learning-based framework for automatic specification synthesis using a large language model. To the best of our knowledge, this is the **first** work to utilize a large language model to understand natural language context and synthesize specifications. We formulate specification synthesis task as a sequence learning problem and integrate BERT, a pre-trained large language model for this purpose. Our proposed framework SpecSyn outperforms prior state-of-the-art by 21% in terms of F1 score. This work opens up new direction to synthesize specification and can be extended for other similar system-related works as well.

# 5. OTHER WORKS

Throughout my PhD journey, I actively engaged in collaborative research projects with fellow scholars. Below, I have provided a compilation of abstracts from other works that I have collaborated during my PhD. For more comprehensive information about each publication, I encourage to access the respective paper sources.

## 5.1 XMeter: Finding Approximable Functions and Predicting Their Accuracy

Approximate computing has significant potential to improve the efficiency of a computing system. Numerous techniques have been proposed in literature. Virtually, all of them require programmers to either experiment with every instance of a specific type of code region exhaustively to find approximable code regions or annotate such regions manually. Both approaches are error-prone and can lead to missed opportunities. Therefore, we propose XMeter to automatically find and quantify approximable code regions. XMeter, first, analyzes the application code statically using a novel algorithm based on memory location updates. Also, XMeter provides a deep learning-based predictor to predict the accuracy of the application when different code regions are approximated. Our proposed scheme does not require the programmer to experiment exhaustively for all possible error rates and types of approximation techniques. Moreover, the scheme does not require any domain knowledge and is not specific to any approximation technique. Therefore, it is general enough to be applicable for any approximation technique. We developed XMeter using LLVM and experimented with 10 applications. We analyzed 43 approximable functions and found 21 to be highly tolerant of errors. We validated our results using 4 well-known approximation techniques and showed that XMeter can predict an application's accuracy accurately.

## 5.2 MERCURY: Accelerating DNN Training By Exploiting Input Similarity

Deep Neural Networks (DNN) are computationally intensive to train. It consists of a large number of multidimensional dot products between many weights and input vectors. However, there can be significant similarities among input vectors. If one input vector is similar to another, its

computations with the weights are similar to those of the other and, therefore, can be skipped by reusing the already-computed results. We propose a novel scheme, called MERCURY, to exploit input similarity during DNN training in a hardware accelerator. MERCURY uses Random Projection with Quantization (RPQ) to convert an input vector to a bit sequence, called Signature. A cache (MCACHE) stores signatures of recent input vectors along with the computed results. If the Signature of a new input vector matches that of an already existing vector in the MCACHE, the two vectors are found to have similarities. Therefore, the already-computed result is reused for the new vector. To the best of our knowledge, MERCURY is the first work that exploits input similarity using RPQ for accelerating DNN training in hardware. The paper presents a detailed design, workflow, and implementation of the MERCURY. Our experimental evaluation with twelve different deep learning models shows that MERCURY saves a significant number of computations and speeds up the model training by an average of 1.97× with an accuracy similar to the baseline system.

### 5.3 ADA-GP: Adaptive Gradient Prediction for DNN Training

Neural network training is inherently sequential where the layers finish the forward propagation in succession, followed by the calculation and back-propagation of gradients (based on a loss function) starting from the last layer. The sequential computations significantly slow down neural network training, especially the deeper ones. Prediction has been successfully used in many areas of computer architecture to speed up sequential processing. Therefore, we propose ADA-GP, that uses gradient prediction adaptively to speed up deep neural network (DNN) training while maintaining accuracy. ADA-GP works by incorporating a small neural network to predict gradients for different layers of a DNN model. ADA-GP uses a novel tensor reorganization to make it feasible to predict a large number of gradients. ADA-GP alternates between DNN training using backpropagated gradients and DNN training using predicted gradients. ADA-GP adaptively adjusts when and for how long gradient prediction is used to strike a balance between accuracy and performance. Last but not least, we provide a detailed hardware extension in a typical DNN accelerator to realize the speed up potential from gradient prediction. Our extensive experiments with fourteen DNN models

show that ADA-GP can achieve an average speed up of 1.47x with similar or even higher accuracy than the baseline models. Moreover, it consumes, on average, 34% less energy due to reduced off-chip memory accesses compared to the baseline hardware accelerator.

**Respective bibliography of all the works during my PhD:**

All my works are available online and can be found in as NetSyn [85], GeneSys [86], Spec-Syn [87], XMeter [4], MERCURY [63], and ADA-GP [62].

# REFERENCES


[1] *SOSP '07: Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*. New York, NY, USA: Association for Computing Machinery, 2007.

[2] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems," 2015. [Online]. Available: http://download.tensorflow.org/paper/whitepaper2015.pdf

[3] B. Agarwal, R. Bhagwan, T. Das, S. Eswaran, V. N. Padmanabhan, and G. M. Voelker, "Netprints: Diagnosing home network misconfigurations using shared knowledge." in *NSDI*, vol. 9, 2009, pp. 349–364.

[4] R. Akram, S. Mandal, and A. Muzahid, "Xmeter: Finding approximable functions and predicting their accuracy," *IEEE Transactions on Computers*, vol. 70, no. 7, pp. 1081–1093, 2021.

[5] R. Alur, R. Bodík, E. Dallal, D. Fisman, P. Garg, G. Juniwal, H. Kress-Gazit, P. Madhusudan, M. M. K. Martin, M. Raghothaman, S. Saha, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa, "Syntax-Guided Synthesis," in *Dependable Software Systems Engineering*, ser. NATO Science for Peace and Security Series, D: Information and Communication Security, M. Irlbeck, D. A. Peled, and A. Pretschner, Eds. IOS Press, 2015, vol. 40, pp. 1–25. [Online]. Available: https://doi.org/10.3233/978-1-61499-495-4-1

[6] R. Arumugam and R. Shanmugamani, *Hands-On Natural Language Processing with Python: A practical guide to applying deep learning architectures to your NLP applications*. Packt

Publishing, 2018. [Online]. Available: https://books.google.com/books?id=ipplDwAAQBAJ

[7] M. Attariyan, M. Chow, and J. Flinn, "X-ray: Automating root-cause diagnosis of performance anomalies in production software," in *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*, 2012, pp. 307–320.

[8] M. Attariyan and J. Flinn, "Automating configuration troubleshooting with dynamic information flow analysis," in *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*. Vancouver, BC: USENIX Association, Oct. 2010. [Online]. Available: https://www.usenix.org/conference/osdi10/automating-configuration-troubleshooting-dynamic-information-flow-analysis

[9] ——, "Automating configuration troubleshooting with dynamic information flow analysis." in *OSDI*, vol. 10, no. 2010, 2010, pp. 1–14.

[10] A. Auger and N. Hansen, "A restart cma evolution strategy with increasing population size," *IEEE CEC*, 2005.

[11] J. Ayres, J. Flannick, J. Gehrke, and T. Yiu, "Sequential pattern mining using a bitmap representation," in *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, 2002, pp. 429–435.

[12] M. Balog, A. L. Gaunt, M. Brockschmidt, S. Nowozin, and D. Tarlow, "DeepCoder: Learning to Write Programs," in *International Conference on Learning Representations*, April 2017.

[13] ——, "DeepCoder," https://github.com/dkamm/deepcoder, 2017.

[14] K. Becker and J. Gottschlich, "AI Programmer: Autonomously Creating Software Programs Using Genetic Algorithms," *CoRR*, vol. abs/1709.05703, 2017. [Online]. Available: http://arxiv.org/abs/1709.05703

[15] R. Bodík and B. Jobstmann, "Algorithmic Program Synthesis: Introduction," *International Journal on Software Tools for Technology Transfer*, vol. 15, pp. 397–411, 2013.

[16] M. Brameier, "On Linear Genetic Programming," Ph.D. dissertation, Dortmund, Germany, 2007.

[17] R. Bunel, M. J. Hausknecht, J. Devlin, R. Singh, and P. Kohli, "Leveraging Grammar and Reinforcement Learning for Neural Program Synthesis," in *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018. [Online]. Available: https://openreview.net/forum?id=H1Xw62kRZ

[18] M. L. J. C. Basilan, https://orcid.org/0000-0003-3105-2252, M. Padilla, and https://orchid.org/0000-0001-5025-12872, maleticiajose.basilan@deped.gov.ph, maycee.padilla@deped.gov.ph, Department of Education- SDO Batangas Province, Batangas, Philippines, "Assessment of teaching english language skills: Input to digitized activities for campus journalism advisers," *International Multidisciplinary Research Journal*, vol. 4, no. 4, Jan. 2023.

[19] J. Cai, R. Shin, and D. Song, "Making Neural Programming Architectures Generalize via Recursion," in *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017. [Online]. Available: https://openreview.net/forum?id=BkbY4psgg

[20] X. Chen, C. Liu, and D. Song, "Execution-guided neural program synthesis," in *ICLR*, 2018.

[21] ——, "Towards synthesizing complex programs from input-output examples," in *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018. [Online]. Available: https://openreview.net/forum?id=Skp1ESxRZ

[22] ——, "Towards Synthesizing Complex Programs From Input-Output Examples," in *International Conference on Learning Representations*, 2018. [Online]. Available: https://openreview.net/forum?id=Skp1ESxRZ

[23] X. Chen, D. Song, and Y. Tian, "Latent execution for neural program synthesis beyond domain-specific languages," *arXiv preprint arXiv:2107.00101*, 2021.

[24] X. Chen, Y. Mao, Z. M. Mao, and J. Van der Merwe, "Declarative configuration management for complex and dynamic networks," in *Proceedings of the 6th International COnference*, 2010, pp. 1–12.

[25] Y. Chen, C. Wang, O. Bastani, I. Dillig, and Y. Feng, "Program synthesis using deduction-guided reinforcement learning," in *ICCAV*, 2020.

[26] A. Cheung, A. Solar-Lezama, and S. Madden, "Using Program Synthesis for Social Recommendations," *ArXiv*, vol. abs/1208.2925, 2012.

[27] Cloudflare, "Facebook blames a server configuration change for yesterday's outage." 2019. [Online]. Available: https://blog.cloudflare.com/october-2021-facebook-outage/

[28] CNN, "Here's why you may have had internet problems today," 2017. [Online]. Available: https://money.cnn.com/2017/11/06/technology/business/internet-outage-comcast-level-3/index.html

[29] S. Das and P. N. Suganthan, "Differential evolution: A survey of the state-of-the-art," *IEEE Transactions on Evolutionary Computation*, vol. 15, pp. 4–31, 2011.

[30] M.-C. de Marneffe, T. Dozat, N. Silveira, K. Haverinen, F. Ginter, J. Nivre, and C. D. Manning, "Universal Stanford dependencies: A cross-linguistic typology," in *Proceedings of the Ninth International Conference on Language Resources and Evaluation (LREC'14)*. Reykjavik, Iceland: European Language Resources Association (ELRA), May 2014, pp. 4585–4592. [Online]. Available: http://www.lrec-conf.org/proceedings/lrec2014/pdf/1062_Paper.pdf

[31] S. K. Debray, W. Evans, R. Muth, and B. De Sutter, "Compiler Techniques for Code Compaction," *ACM Trans. Program. Lang. Syst.*, vol. 22, no. 2, pp. 378–415, Mar. 2000. [Online]. Available: http://doi.acm.org/10.1145/349214.349233

[32] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A Large-Scale Hierarchical Image Database," in *CVPR09*, 2009.

[33] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," 2018. [Online]. Available: https://arxiv.org/abs/1810.04805

[34] J. Devlin, J. Uesato, S. Bhupatiraju, R. Singh, A. Mohamed, and P. Kohli, "RobustFill: Neural Program Learning under Noisy I/O," in *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, 2017, pp. 990–998. [Online]. Available: http://proceedings.mlr.press/v70/devlin17a.html

[35] K. Dinesh, B. Luca, W. Matt, H. Anders, and G. Kit, "LINQ to SQL: .NET Language-Integrated Query for Relational Data," 2007. [Online]. Available: https://docs.microsoft.com/en-us/previous-versions/dotnet/articles/bb425822(v=msdn.10)

[36] W. Enck, T. Moyer, P. McDaniel, S. Sen, P. Sebos, S. Spoerel, A. Greenberg, Y.-W. E. Sung, S. Rao, and W. Aiello, "Configuration management at massive scale: System design and experience," *IEEE Journal on Selected Areas in Communications*, vol. 27, no. 3, pp. 323–335, 2009.

[37] S. C. Endres, C. Sandrock, and W. W. Focke, "A simplicial homology algorithm for lipschitz optimisation," *Journal of Global Optimization*, 2018.

[38] B. Eshete, A. Villafiorita, and K. Weldemariam, "Early detection of security misconfiguration vulnerabilities in web applications," in *2011 Sixth International Conference on Availability, Reliability and Security*. IEEE, 2011, pp. 169–174.

[39] N. Feamster and H. Balakrishnan, "Detecting bgp configuration faults with static analysis," in *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*, 2005, pp. 43–56.

[40] Y. Feng, R. Martins, O. Bastani, and I. Dillig, "Program Synthesis Using Conflict-driven Learning," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming*

*Language Design and Implementation*, ser. PLDI 2018. New York, NY, USA: ACM, 2018, pp. 420–435. [Online]. Available: http://doi.acm.org/10.1145/3192366.3192382

[41] fiercewireless, "At&t failure was caused by a system configuration change," 2017. [Online]. Available: https://www.fiercewireless.com/wireless/at-t-s-911-outage-result-mistakes-made-by-at-t-fcc-s-pai-says

[42] H. Finkel and I. Laguna, "Program Synthesis for Scientific Computing," https://www.anl.gov/cels/program-synthesis- for-scientific-computing-report, August 2020.

[43] S. Forstenlechner, D. Fagan, M. Nicolau, and M. O'Neill, "A grammar design pattern for arbitrary program synthesis problems in genetic programming," in *European Conference on Genetic Programming*, 2017.

[44] D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*, 1st ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1989.

[45] J. Gottschlich, A. Solar-Lezama, N. Tatbul, M. Carbin, M. Rinard, R. Barzilay, S. Amarasinghe, J. B. Tenenbaum, and T. Mattson, "The Three Pillars of Machine Programming," in *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, ser. MAPL 2018. New York, NY, USA: ACM, 2018, pp. 69–80. [Online]. Available: http://doi.acm.org/10.1145/3211346.3211355

[46] S. Gulwani, W. R. Harris, and R. Singh, "Spreadsheet Data Manipulation Using Examples," *Commun. ACM*, vol. 55, no. 8, pp. 97–105, Aug. 2012. [Online]. Available: http://doi.acm.org/10.1145/2240236.2240260

[47] H. S. Gunawi, M. Hao, R. O. Suminto, A. Laksono, A. D. Satria, J. Adityatama, and K. J. Eliazar, "Why does the cloud stop computing? lessons from hundreds of service outages," in *Proceedings of the Seventh ACM Symposium on Cloud Computing*, ser. SoCC '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 1–16. [Online]. Available: https://doi.org/10.1145/2987550.2987583

[48] R. Gupta, S. Pal, A. Kanade, and S. Shevade, "Deepfix: Fixing common c language errors by deep learning," in *Proceedings of the aaai conference on artificial intelligence*, vol. 31, no. 1, 2017.

[49] J. Han, H. Cheng, D. Xin, and X. Yan, "Frequent pattern mining: current status and future directions," *Data mining and knowledge discovery*, vol. 15, no. 1, pp. 55–86, 2007.

[50] N. Hansen and A. Ostermeier, "Adapting arbitrary normal mutation distributions in evolution strategies: the covariance matrix adaptation," in *ICEC*, 1996.

[51] N. Hansen, "The CMA Evolution Strategy: A Tutorial," 2005, arXiv e-prints, arXiv:1604.00772, 2016, pp.1-39. [Online]. Available: https://hal.inria.fr/hal-01297037

[52] ——, "Benchmarking a bi-population cma-es on the bbob-2009 function testbed," ser. GECCO '09, 2009.

[53] ——, "CMA-ES Applications," http://www.cmap.polytechnique.fr/~nikolaus.hansen/cmaapplications.pdf, 2009.

[54] N. Hansen and A. Ostermeier, "Completely derandomized self-adaptation in evolution strategies," 2001.

[55] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *CoRR*, vol. abs/1512.03385, 2015. [Online]. Available: http://arxiv.org/abs/1512.03385

[56] T. Helmuth and P. Kelly, "Psb2: The second program synthesis benchmark suite," in *Proceedings of the Genetic and Evolutionary Computation Conference*, ser. GECCO '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 785–794. [Online]. Available: https://doi.org/10.1145/3449639.3459285

[57] E. Hemberg, J. Kelly, and U.-M. O'Reilly, "On domain knowledge and novelty to improve program synthesis performance with grammatical evolution," in *Proceedings of the Genetic and Evolutionary Computation Conference*, ser. GECCO '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 1039–1046. [Online]. Available: https://doi.org/10.1145/3321707.3321865

[58] S. Heule, E. Schkufza, R. Sharma, and A. Aiken, "Stratified Synthesis: Automatically Learning the x86-64 Instruction Set," *SIGPLAN Not.*, vol. 51, no. 6, pp. 237–250, Jun. 2016. [Online]. Available: http://doi.acm.org/10.1145/2980983.2908121

[59] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, p. 1735–1780, nov 1997. [Online]. Available: https://doi.org/10.1162/neco.1997.9.8.1735

[60] J. Hong, D. Dohan, R. Singh, C. Sutton, and M. Zaheer, "Latent programmer: Discrete latent codes for program synthesis," in *ICML*, 2021.

[61] P. Huang, C. Guo, J. R. Lorch, L. Zhou, and Y. Dang, "Capturing and enhancing in situ system observability for failure detection," in *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, 2018, pp. 1–16.

[62] V. Janfaza, S. Mandal, F. Mahmud, and A. Muzahid, "Adaptive gradient prediction for dnn training," 2023, arXiv 2305.13236.

[63] V. Janfaza, K. Weston, M. Razavi, S. Mandal, F. Mahmud, A. Hilty, and A. Muzahid, "Mercury: Accelerating dnn training by exploiting input similarity," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2023, pp. 638–650.

[64] Y. Jin, N. Duffield, A. Gerber, P. Haffner, S. Sen, and Z.-L. Zhang, "Nevermind, the problem is already fixed: proactively detecting and troubleshooting customer dsl problems," in *Proceedings of the 6th International COnference*, 2010, pp. 1–12.

[65] D. Jurafsky and J. H. Martin, *Speech and language processing : an introduction to natural language processing, computational linguistics, and speech recognition*, 2009.

[66] A. Kalyan, A. Mohta, O. Polozov, D. Batra, P. Jain, and S. Gulwani, "Neural-guided deductive search for real-time program synthesis from examples," *arXiv*, 2018.

[67] M. Karmelita and T. P. Pawlak, "Cma-es for one-class constraint synthesis," in *Proceedings of the 2020 Genetic and Evolutionary Computation Conference*, ser. GECCO '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 859–867. [Online]. Available: https://doi.org/10.1145/3377930.3389807

[68] B. Khuntia, S. Pattnaik, D. Panda, D. Neog, S. Devi, and M. Dutta, "Genetic algorithm with artificial neural networks as its fitness function to design rectangular microstrip antenna on thick substrate," *Microwave and Optical Technology Letters*, vol. 44, pp. 144 – 146, 01 2005.

[69] M. F. Korns, *Accuracy in Symbolic Regression*. New York, NY: Springer New York, 2011, pp. 129–151. [Online]. Available: https://doi.org/10.1007/978-1-4614-1770-5_8

[70] M. A. Kramer, "Nonlinear principal component analysis using autoassociative neural networks," *AIChE journal*, vol. 37, no. 2, pp. 233–243, 1991.

[71] A. Krizhevsky, "Learning Multiple Layers of Features from Tiny Images," Tech. Rep., 2009.

[72] S. Labs, "Evolv Delivers Autonomous Optimization Across Web & Mobile," https://www.evolv.ai/.

[73] W. B. Langdon and R. Poli, *Foundations of Genetic Programming*, 1st ed. Springer Publishing Company, Incorporated, 2010.

[74] P. Langley and H. A. Simon, "Applications of machine learning and rule induction," *Communications of the ACM*, vol. 38, no. 11, pp. 54–64, 1995.

[75] C. Li and B. Wang, "Principal Components Analysis," 2014. [Online]. Available: http://www.ccs.neu.edu/home/vip/teach/MLcourse/5_features_dimensions/lecture_notes/PCA/PCA.pdf

[76] K. Li, Y. Xue, Y. Shao, B. Su, Y.-a. Tan, and J. Hu, "Software misconfiguration troubleshooting based on state analysis," in *2021 IEEE Sixth International Conference on Data Science in Cyberspace (DSC)*. IEEE, 2021, pp. 361–366.

[77] Y. Li, S. Wang, and T. N. Nguyen, "Dear: A novel deep learning-based approach for automated program repair," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 511–523.

[78] T. Lin, P. Goyal, R. B. Girshick, K. He, and P. Dollár, "Focal loss for dense object detection," *CoRR*, vol. abs/1708.02002, 2017. [Online]. Available: http://arxiv.org/abs/1708.02002

[79] P. Liskowski, K. Krawiec, N. E. Toklu, and J. Swan, "Program synthesis as latent continuous optimization: Evolutionary search in neural embeddings," ser. GECCO, 2020.

[80] H. Liu, K. Simonyan, O. Vinyals, C. Fernando, and K. Kavukcuoglu, "Hierarchical Representations for Efficient Architecture Search," *CoRR*, vol. abs/1711.00436, 2017. [Online]. Available: http://arxiv.org/abs/1711.00436

[81] C. Loncaric, M. D. Ernst, and E. Torlak, "Generalized Data Structure Synthesis," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE 2018. New York, NY, USA: ACM, 2018, pp. 958–968. [Online]. Available: http://doi.acm.org/10.1145/3180155.3180211

[82] B. T. Loo, J. M. Hellerstein, I. Stoica, and R. Ramakrishnan, "Declarative routing: extensible routing with declarative queries," *ACM SIGCOMM Computer Communication Review*, vol. 35, no. 4, pp. 289–300, 2005.

[83] T. Lutz, S. Wagner, T. Lutz, and S. Wagner, "Drag reduction and shape optimization of airship bodies," in *12th Lighter-Than-Air Systems Technology Conference*, 1997.

[84] N. R. Mabroukeh and C. I. Ezeife, "A taxonomy of sequential pattern mining algorithms," *ACM Computing Surveys (CSUR)*, vol. 43, no. 1, pp. 1–41, 2010.

[85] S. Mandal, T. Anderson, J. Turek, J. Gottschlich, S. Zhou, and A. Muzahid, "Learning fitness functions for machine programming," in *Proceedings of Machine Learning and Systems*, A. Smola, A. Dimakis, and I. Stoica, Eds., vol. 3, 2021, pp. 139–155. [Online]. Available: https://proceedings.mlsys.org/paper_files/paper/2021/file/32bb90e8976aab5298d5da10fe66f21d-Paper.pdf

[86] S. Mandal, T. A. Anderson, J. Turek, J. Gottschlich, and A. Muzahid, "Synthesizing programs with continuous optimization," 2022, arXiv 2211.00828.

[87] S. Mandal, A. Chethan, V. Janfaza, S. M. F. Mahmud, T. A. Anderson, J. Turek, J. J. Tithi, and A. Muzahid, "Large language models based automatic synthesis of software specifications," 2023, arXiv 2304.09181.

[88] Z. Manna and R. Waldinger, "Knowledge and Reasoning in Program Synthesis," *Artificial Intelligence*, vol. 6, no. 2, pp. 175 – 208, 1975.

[89] J. Matos Dias, H. Rocha, B. Ferreira, and M. d. C. Lopes, "A genetic algorithm with neural network fitness function evaluation for IMRT beam angle optimization," *Central European Journal of Operations Research*, vol. 22, 09 2014.

[90] S. D. Mueller, M. Milano, and P. Koumoutsakos, "Application of machine learning algorithms to flow modeling and optimization," in *CTR ARB*, 1999.

[91] K. P. Murphy, *Machine Learning: A Probabilistic Perspective*. The MIT Press, 2012.

[92] S. Nadi, T. Berger, C. Kästner, and K. Czarnecki, "Mining configuration constraints: Static analyses and empirical results," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 140–151. [Online]. Available: https://doi.org/10.1145/2568225.2568283

[93] K. Nagaraja, F. Oliveira, R. Bianchini, R. P. Martin, and T. D. Nguyen, "Understanding and dealing with operator mistakes in internet services," in *6th Symposium on Operating Systems Design & Implementation (OSDI 04)*. San Francisco, CA: USENIX Association, Dec. 2004. [Online]. Available: https://www.usenix.org/conference/osdi-04/understanding-and-dealing-operator-mistakes-internet-services

[94] D. G. Novick and K. Ward, "Why don't people read the manual?" in *Proceedings of the 24th Annual ACM International Conference on Design of Communication*, ser. SIGDOC '06. New York, NY, USA: Association for Computing Machinery, 2006, p. 11–18. [Online]. Available: https://doi.org/10.1145/1166324.1166329

[95] M. Nye, L. Hewitt, J. Tenenbaum, and A. Solar-Lezama, "Learning to infer program sketches," 2019.

[96] R. Pandita, X. Xiao, H. Zhong, T. Xie, S. Oney, and A. Paradkar, "Inferring method specifications from natural language api descriptions," in *2012 34th international conference on software engineering (ICSE)*. IEEE, 2012, pp. 815–825.

[97] E. Pantridge, T. Helmuth, and L. Spector, "Functional code building genetic programming," in *Proceedings of the Genetic and Evolutionary Computation Conference*, ser. GECCO '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1000–1008. [Online]. Available: https://doi.org/10.1145/3512290.3528866

[98] E. Parisotto, A.-r. Mohamed, R. Singh, L. Li, D. Zhou, and P. Kohli, "Neuro-symbolic program synthesis," *arXiv preprint arXiv:1611.01855*, 2016.

[99] T. P. Pawlak and K. Krawiec, "Synthesis of constraints for mathematical programming with one-class genetic programming," *IEEE Transactions on Evolutionary Computation*, vol. 23, no. 1, pp. 117–129, 2018.

[100] T. Perkis, "Stack-based genetic programming," in *Proceedings of the First IEEE Conference on Evolutionary Computation. IEEE World Congress on Computational Intelligence*, 1994, pp. 148–153 vol.1.

[101] R. Potharaju, J. Chan, L. Hu, C. Nita-Rotaru, M. Wang, L. Zhang, and N. Jain, "Confseer: leveraging customer support knowledge bases for automated misconfiguration detection," *Proceedings of the VLDB Endowment*, vol. 8, no. 12, pp. 1828–1839, 2015.

[102] ProjectPro, "Bert nlp model explained for complete beginners," Nov 2022. [Online]. Available: https://www.projectpro.io/article/bert-nlp-model-explained/558

[103] X. Qiu, T. Sun, Y. Xu, Y. Shao, N. Dai, and X. Huang, "Pre-trained models for natural language processing: A survey," *Science China Technological Sciences*, vol. 63, no. 10, pp. 1872–1897, Sep. 2020. [Online]. Available: https://doi.org/10.1007/s11431-020-1647-3

[104] A. Quinn, D. Devecsery, P. M. Chen, and J. Flinn, "Jetstream: Cluster-scale parallelization of information flow queries." in *OSDI*, vol. 16, 2016, pp. 451–466.

[105] A. Rabkin and R. Katz, "Precomputing possible configuration error diagnoses," in *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. IEEE, 2011, pp. 193–202.

[106] ——, "Static extraction of program configuration options," in *Proceedings of the 33rd International Conference on Software Engineering*, 2011, pp. 131–140.

[107] A. Rabkin and R. H. Katz, "How hadoop clusters break," *IEEE Software*, vol. 30, no. 4, pp. 88–94, 2013.

[108] A. Radford and K. Narasimhan, "Improving language understanding by generative pre-training," 2018.

[109] A. Ratner, D. Alistarh, G. Alonso, D. G. Andersen, P. Bailis, S. Bird, N. Carlini, B. Catanzaro, E. Chung, B. Dally, J. Dean, I. S. Dhillon, A. G. Dimakis, P. Dubey, C. Elkan, G. Fursin, G. R. Ganger, L. Getoor, P. B. Gibbons, G. A. Gibson, J. E. Gonzalez, J. Gottschlich, S. Han, K. M. Hazelwood, F. Huang, M. Jaggi, K. G. Jamieson, M. I. Jordan, G. Joshi, R. Khalaf, J. Knight, J. Konecný, T. Kraska, A. Kumar, A. Kyrillidis, J. Li, S. Madden, H. B. McMahan, E. Meijer, I. Mitliagkas, R. Monga, D. G. Murray, D. S. Papailiopoulos, G. Pekhimenko, T. Rekatsinas, A. Rostamizadeh, C. Ré, C. D. Sa, H. Sedghi, S. Sen, V. Smith, A. Smola, D. Song, E. R. Sparks, I. Stoica, V. Sze, M. Udell, J. Vanschoren, S. Venkataraman, R. Vinayak, M. Weimer, A. G. Wilson, E. P. Xing, M. Zaharia, C. Zhang, and A. Talwalkar, "SysML: The New Frontier of Machine Learning Systems," *CoRR*, vol. abs/1904.03257, 2019. [Online]. Available: http://arxiv.org/abs/1904.03257

[110] V. Raychev, M. Vechev, and E. Yahav, "Code Completion with Statistical Language Models," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14. New York, NY, USA: ACM, 2014, pp. 419–428. [Online]. Available: http://doi.acm.org/10.1145/2594291.2594321

[111] E. Real, A. Aggarwal, Y. Huang, and Q. V. Le, "Regularized Evolution for Image Classifier Architecture Search," *CoRR*, vol. abs/1802.01548, 2018. [Online]. Available: http://arxiv.org/abs/1802.01548

[112] ——, "Regularized Evolution for Image Classifier Architecture Search," in *Thirty-Third AAAI Conference on Artificial Intelligence*, February 2019.

[113] E. Real, C. Liang, D. R. So, and Q. V. Le, "Automl-zero: Evolving machine learning algorithms from scratch," 2020.

[114] S. E. Reed and N. de Freitas, "Neural Programmer-Interpreters," in *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2016. [Online]. Available: http://arxiv.org/abs/1511.06279

[115] T. Salimans, J. Ho, X. Chen, S. Sidor, and I. Sutskever, "Evolution Strategies as a Scalable Alternative to Reinforcement Learning," *CoRR*, vol. abs/1703.03864, 2017. [Online]. Available: https://arxiv.org/abs/1703.03864

[116] M. Santolucito, E. Zhai, R. Dhodapkar, A. Shim, and R. Piskac, "Synthesizing configuration file specifications with association rule learning," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, oct 2017. [Online]. Available: https://doi.org/10.1145/3133888

[117] M. Santolucito, E. Zhai, and R. Piskac, "Probabilistic automated language learning for configuration files," in *Computer Aided Verification: 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II 28*. Springer, 2016, pp. 80–87.

[118] E. Schkufza, R. Sharma, and A. Aiken, "Stochastic superoptimization," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 305–316. [Online]. Available: https://doi.org/10.1145/2451116.2451150

[119] E. C. Shin, I. Polosukhin, and D. Song, "Improving neural program synthesis with inferred execution traces," *NIPS*, 2018.

[120] X. Si, M. Raghothaman, K. Heo, and M. Naik, "Synthesizing datalog programs using numerical relaxation," 08 2019, pp. 6117–6124.

[121] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat, "Combinatorial Sketching for Finite Programs," *SIGOPS Oper. Syst. Rev.*, vol. 40, no. 5, pp. 404–415, Oct. 2006. [Online]. Available: http://doi.acm.org/10.1145/1168917.1168907

[122] T. Sonoda, Y. Yamaguchi, T. Arima, M. Olhofer, B. Sendhoff, and H.-A. Schreiber, "Advanced high turning compressor airfoils for low reynolds number condition—part i: Design and optimization," *Journal of Turbomachinery*, 2004.

[123] D. Sroka and T. P. Pawlak, "One-class constraint acquisition with local search," in *Proceedings of the Genetic and Evolutionary Computation Conference*, ser. GECCO '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 363–370. [Online]. Available: https://doi.org/10.1145/3205455.3205480

[124] P. Stöckle, T. Wasserer, B. Grobauer, and A. Pretschner, "Automated identification of security-relevant configuration settings using nlp," in *37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–5.

[125] Y.-Y. Su, M. Attariyan, and J. Flinn, "Autobash: Improving configuration management with operating system causality analysis," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6, pp. 237–250, 2007.

[126] F. P. Such, V. Madhavan, E. Conti, J. Lehman, K. O. Stanley, and J. Clune, "Deep Neuroevolution: Genetic Algorithms Are a Competitive Alternative for Training Deep Neural Networks for Reinforcement Learning," *CoRR*, vol. abs/1712.06567, 2017. [Online]. Available: http://arxiv.org/abs/1712.06567

[127] C. Sun, X. Qiu, Y. Xu, and X. Huang, "How to fine-tune BERT for text classification?" *CoRR*, vol. abs/1905.05583, 2019. [Online]. Available: http://arxiv.org/abs/1905.05583

[128] L. Tan, D. Yuan, G. Krishna, and Y. Zhou, "/* icomment: Bugs or bad comments?*," in *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, 2007, pp. 145–158.

[129] L. Tan, Y. Zhou, and Y. Padioleau, "acomment: mining annotations from comments and code to detect interrupt related concurrency bugs," in *Proceedings of the 33rd international conference on software engineering*, 2011, pp. 11–20.

[130] Thomas, *Global Optimization Algorithms-Theory and Application*, 2009, http://www.it-weise.de/projects/book.pdf.

[131] S. Tseng, E. Hemberg, and U.-M. O'Reilly, "Synthesizing programs from program pieces using genetic programming and refinement type checking," in *Genetic Programming*, E. Medvet, G. Pappa, and B. Xue, Eds. Cham: Springer International Publishing, 2022, pp. 197–211.

[132] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," *CoRR*, vol. abs/1706.03762, 2017. [Online]. Available: http://arxiv.org/abs/1706.03762

[133] H. J. Wang, J. C. Platt, Y. Chen, R. Zhang, and Y.-M. Wang, "Automatic misconfiguration troubleshooting with peerpressure." in *OSDI*, vol. 4, 2004, pp. 245–257.

[134] Y.-M. Wang, C. Verbowski, J. Dunagan, Y. Chen, H. J. Wang, C. Yuan, and Z. Zhang, "Strider: A black-box, state-based approach to change and configuration management and support," *Science of Computer Programming*, vol. 53, no. 2, pp. 143–164, 2004.

[135] Y. Wang, X. Wang, and I. Dillig, "Relational program synthesis," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–27, 2018.

[136] A. Whitaker, R. S. Cox, S. D. Gribble *et al.*, "Configuration debugging as search: Finding the needle in the haystack." in *OSDI*, vol. 4, 2004, pp. 6–6.

[137] S. Winter, B. Brendel, A. Rick, M. Stockheim, K. Schmieder, and H. Ermert, "Registration of bone surfaces, extracted from CT-datasets, with 3d ultrasound," *Biomedizinische Technik/Biomedical Engineering*, 2002.

[138] E. Wong, L. Zhang, S. Wang, T. Liu, and L. Tan, "Dase: Document-assisted symbolic execution for improving automated software testing," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 620–631.

[139] C. Xiang, H. Huang, A. Yoo, Y. Zhou, and S. Pasupathy, "PracExtractor: Extracting configuration good practices from manuals to detect server misconfigurations," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, Jul. 2020, pp. 265–280. [Online]. Available: https://www.usenix.org/conference/atc20/presentation/xiang

[140] T. Xu, L. Jin, X. Fan, Y. Zhou, S. Pasupathy, and R. Talwadker, "Hey, you have given me too many knobs!: Understanding and dealing with over-designed configuration in system software," in *2015 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE 2015 - Proceedings*, ser. 2015 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE 2015 - Proceedings. Association for Computing Machinery, Inc, Aug. 2015, pp. 307–319, publisher Copyright: © 2015 ACM.; 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE 2015 ; Conference date: 30-08-2015 Through 04-09-2015.

[141] T. Xu, X. Jin, P. Huang, Y. Zhou, S. Lu, L. Jin, and S. Pasupathy, "Early detection of configuration errors to reduce failure damage." in *OSDI*, vol. 10, 2016, pp. 3 026 877–3 026 925.

[142] T. Xu, J. Zhang, P. Huang, J. Zheng, T. Sheng, D. Yuan, Y. Zhou, and S. Pasupathy, "Do not blame users for misconfigurations," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 244–259. [Online]. Available: https://doi.org/10.1145/2517349.2522727

[143] T. Xu and Y. Zhou, "Systems approaches to tackling configuration errors: A survey," *ACM Comput. Surv.*, vol. 47, no. 4, pp. 70:1–70:41, 2015. [Online]. Available: https://doi.org/10.1145/2791577

[144] X. Ye, Q. Chen, I. Dillig, and G. Durrett, "Optimal neural program synthesis from multimodal specifications," *CoRR*, 2020.

[145] Z. Yin, X. Ma, J. Zheng, Y. Zhou, L. N. Bairavasundaram, and S. Pasupathy, "An empirical study on configuration errors in commercial and open source systems," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, ser. SOSP '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 159–172. [Online]. Available: https://doi.org/10.1145/2043556.2043572

[146] C. Yuan, N. Lao, J.-R. Wen, J. Li, Z. Zhang, Y.-M. Wang, and W.-Y. Ma, "Automated known problem diagnosis with event traces," *ACM SIGOPS Operating Systems Review*, vol. 40, no. 4, pp. 375–388, 2006.

[147] D. Yuan, Y. Xie, R. Panigrahy, J. Yang, C. Verbowski, and A. Kumar, "Context-based online configuration-error detection," in *Proceedings of the 2011 USENIX conference on USENIX annual technical conference*, 2011, pp. 28–28.

[148] J. Zhai, J. Huang, S. Ma, X. Zhang, L. Tan, J. Zhao, and F. Qin, "Automatic model generation from documentation for java api functions," in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 380–391.

[149] J. Zhang, L. Renganarayana, X. Zhang, N. Ge, V. Bala, T. Xu, and Y. Zhou, "Encore: Exploiting system environment and correlation information for misconfiguration detection," in *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*, 2014, pp. 687–700.

[150] S. Zhang and M. D. Ernst, "Automated diagnosis of software configuration errors," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 312–321.

[151] H. Zhong, L. Zhang, T. Xie, and H. Mei, "Inferring resource specifications from natural language api documentation," in *2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2009, pp. 307–318.

[152] Y. Zhou, R. Gu, T. Chen, Z. Huang, S. Panichella, and H. Gall, "Analyzing apis documentation and code to detect directive defects," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 27–37.

[153] A. Zohar and L. Wolf, "Automatic Program Synthesis of Long Programs with a Learned Garbage Collector," *CoRR*, vol. abs/1809.04682, 2018. [Online]. Available: http://arxiv.org/abs/1809.04682