© 2022 Zhengkai Wu

GUIDING CODE ANALYSIS AND GENERATION WITH PROBABILISTIC CONSTRAINTS

BY

ZHENGKAI WU

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois Urbana-Champaign, 2022

Urbana, Illinois

Doctoral Committee:

Professor Tao Xie, Chair and Director of Research
Professor Darko Marinov
Associate Professor Lingming Zhang
Research Scientist Rishabh Singh, Google

ABSTRACT

Different forms of constraints have been a useful kind of artifacts across multiple computer science domains, especially in code analysis and generation tasks. Existing constraint-based approaches (being guided by constraints) often work under the assumption that in the solution to the target problem, every constraint must be satisfied. However, we show that this assumption may limit the effectiveness of the existing constraint-based approaches.

To overcome the aforementioned limitation, we explore ideas to extract and maintain probabilistic constraints in this dissertation. Probabilistic constraints are defined as constraints assigned with a probability distribution denoting the likelihood of each constraint to be satisfied. The probability distribution allows us to incorporate a broader range of constraints with different levels of confidence reflected by the probability, thus helping with the tasks of code analysis and generation.

To effectively guide code analysis and generation with probabilistic constraints, we propose approaches in four specific tasks. In the first half of this dissertation, we investigate the code analysis tasks. In the first part (REINAM), we focus on how to maintain the probability distribution on the grammatical constraints to allow better generalization ability in the grammar-guided fuzzing problem. In the second part (Groot), we focus on how to design different types of constraints to more accurately locate the root cause when conducting root cause analysis in a complicated microservice system. In the second half, we investigate the code generation tasks. In the third part (NL2Viz), we focus on how to extract probabilistic constraints from ambiguous inputs such as natural language (NL) and data context in the task of NL-to-visualization (in short as NL2Visualization). In the fourth part (NRRanker), we focus on how to generate additional probabilistic constraints based on the given NL instruction to rank the initial candidate regular expressions in the task of NL-to-regular-expression (in short as NL2Regex).

Based on the empirical evaluations of our proposed approaches in the four tasks, we find that the guidance of probabilistic constraints substantially improves the effectiveness of constraint-based approaches on the tasks of code analysis and generation.

To my parents and friends, for their love and support.

ACKNOWLEDGMENTS

Six and a half years of Ph.D. have been the longest study period and the most important part of my life. I would not be able to complete this dissertation without the help of many people. In this acknowledgment, I would like to try my best to express my gratitude for all the support that I received over the years. I may inevitably forget to mention someone who has lent a helping hand, but you all have my deepest appreciation for all the help that you have offered.

I would first like to thank my advisor Prof. Tao Xie. It has been a great honor to work with him. Tao has provided me with great guidance in both research and life throughout my Ph.D. study. In research, it would not be possible for me to conduct projects in different domains without his open-minded attitude and encyclopedic knowledge about different research directions. Not only has he guided me with research skills, but he has also enlightened me on how to formulate the target problems, especially on how to think about the problems under a big picture. In life, his motto *work hard, work smart, work wise* has inspired me to apply these principles not only in research but also in facing difficulties in life to continuously improve myself.

I would also like to thank the other three members in my thesis committee, Prof. Darko Marinov, Prof. Lingming Zhang, and Dr. Rishabh Singh. They are all very supportive of my work. Darko has been selflessly providing much help during my Ph.D. study. His hard-working and sense of humor have been an icon for the UIUC software engineering family. Lingming has also offered me a lot of help especially on the NRRanker work. His deep understanding in the code generation domain is truly insightful for me to get a high level understanding of related problems. Although I did not get the opportunity to collaborate with Prof. Darko Marinov or Prof. Lingming Zhang, they are always willing to listen to my research updates and give their advice. I got to work with Rishabh three years ago. Although our project did not turn into a publication, it is a great pleasure to work with him. He also gave a lot of helpful feedback even on the projects that he did not participate in.

I would also like to thank my research advisor Prof. Yingfei Xiong during my undergrad. Yingfei has done amazing research in program synthesis and he is the one who led me into the gate of research.

I would also like to thank the group/lab mates with whom I am fortunate to spend time. First I want to thank the members of the ASE group led by Prof. Tao Xie in both UIUC and

PKU. (Although I have not had the opportunity to personally meet the PKU members.) Specifically I want to thank Angello Astorga, Liia Butler, Joey (Jiayi) Cao, Yingjie Fu, Chiao Hsieh, Junhao Hu, Lori Jia, Wing Lam, Linyi Li, Zhenwen Li, Xueqing Liu, Dezhi Ran, Luyao Ren, Siwakorn Srisakaokul, Wenyu Wang, Wei Yang, Hao Yu, Mingming Zhang, Neil (Zirui) Zhao, and Zexuan Zhong. Second, I would like to thank Sam Cheng, Saikat Dutta, Vimuth Fernando, Zixin Huang, Keyur Joshi, Owolabi Legunsen, August Shi, and Xudong Su for sharing the lab space with me and discussion about many things. I also want to thank Prof. Chandra Chekuri for helping my program of study.

I would also like to thank the collaborators through the years: Angello Astorga, Osbert Bastani, Joey (Jiayi) Cao, Yuetang Deng, Sumit Gulwani, Junhao Hu, Huai Jiang, Evan Johnson, Selcuk Kopru, Wing Lam, Vu Le, Oreoluwa Legunsen, Dengfeng Li, Zhenwen Li, Xueqing Liu, Hui Luo, Ray Ozzie, Bin Peng, Jian Peng, Arjun Radhakrishna, Ivan Radicek, Rishabh Singh, Gustavo Soares, Dawn Song, Siwakorn Srisakaokul, Ashish Tiwari, Hanzhang Wang, Jiamu Wang, Wenyu Wang, Xinyu Wang, Xusheng Xiao, Tao Xie, Peng Yan, Wei Yang, Haibin Zheng, and Zexuan Zhong. I especially want to thank the industry collaborators from Microsoft and eBay. Without their resources, we would not be able to finish the projects with real-world impacts in this dissertation.

I would also like to thank the CS department staff members at UIUC who have helped me through my Ph.D. program on miscellaneous activities. Specifically, I would like to thank Kimberly Bogle, Maggie Chappel, Jennifer Comstock, Dana Garard, Viveka Kudaligama, and Kara MacGregor.

I would also like to thank all of my friends. Specifically, I would like to thank Jingxiang Dao, Keyu Gan, Tian Gao, Yang He, Liming Huang, Chao Li, Zhen Li, Zhongyu Li, Mincong Pan, Yifan Sun, Bowen Wang, Qixin Wang, Yiming Wang, Linling Xun, Jingcheng Yu, Yuan Zhao, Rui Zhu, and Chen Zou. I also want to thank friends that I have met only over the Internet. Although I may not even get to know the real names of these online friends, they still bring much joy to my daily life nonetheless.

I would finally and most importantly like to thank my family, especially my parents, Chunchu Wu and Yunyu Xue, for their care and support since my childhood. They always have my back during my highs and lows. I cannot say enough thanks for everything that they have done for me.

TABLE OF CONTENTS

CHAPTER 1 INTRODUCTION 1
1.1 Thesis Statement 2
1.2 Contributions 4
1.3 Organization of Dissertation 5

CHAPTER 2 REINAM: REINFORCEMENT LEARNING FOR INPUT-GRAMMAR INFERENCE 7
2.1 Overview 7
2.2 Background 10
2.3 Motivating Examples 12
2.4 Approach 15
2.5 Evaluation 21
2.6 Summary 29

CHAPTER 3 GROOT: AN EVENT-GRAPH-BASED APPROACH FOR ROOT CAUSE ANALYSIS 30
3.1 Overview 30
3.2 Motivating Examples 33
3.3 Approach 35
3.4 Evaluation 42
3.5 Experience 45
3.6 Discussion 50
3.7 Summary 50

CHAPTER 4 NL2VIZ: NATURAL LANGUAGE TO VISUALIZATION VIA CONSTRAINED SYNTAX-GUIDED SYNTHESIS 52
4.1 Overview 52
4.2 Movivating Examples 54
4.3 Approach 58
4.4 Evaluation 64
4.5 Summary 72

CHAPTER 5 NRRANKER: TRANSLATING NATURAL LANGUAGE TO REGULAR EXPRESSION VIA INTEGRATING PRE-TRAINED MODEL AND TEST GENERATION 74
5.1 Overview 74
5.2 Background 76
5.3 Approach 79

5.4 Evaluation . . . 84
5.5 Discussion . . . 91
5.6 Summary . . . 91

CHAPTER 6 RELATED WORK . . . 93
6.1 Input Grammar Inference . . . 93
6.2 Root Cause Analysis . . . 94
6.3 NL2CODE . . . 96

CHAPTER 7 CONCLUSIONS AND FUTURE WORK . . . 100
7.1 Future Work . . . 101

REFERENCES . . . 103

CHAPTER 1: INTRODUCTION

Different forms of constraints have been a useful kind of artifacts across multiple computer science domains. One popular example for constraints is a context-free grammar (CFG), which can be seen as the constraints on a string input, and represented in a set of production rules in the form of $A \longrightarrow \alpha$, with A being a terminal symbol and α being a sequence of terminal and/or non-terminal symbols [1]. Other forms of constraints include property-value pairs [2], input-output examples [3, 4, 5] and so on.

In software engineering, particularly the tasks of code analysis and generation, different forms of constraints are widely used. In code analysis, constraints are usually extracted from the code under analysis and then used to guide downstream tasks. For example, in the task of fuzz testing [6, 7], an input grammar is often synthesized to guide the input string generation in fuzz testing. While in the task of root cause analysis [8, 9], constraints are usually not only about the input value under consideration, but also about different components of the whole system. In the tasks of code generation such as program synthesis [2, 3, 4, 5, 10, 11, 12], constraints are extracted from different sources of input such as natural language (NL) and then used by constrained-synthesis algorithm to produce the desired output program.

Existing constraint-based approaches with constraint guidance often work under the assumption that in the solution to the target problem, every constraint must be satisfied. However, this assumption may limit the effectiveness of the existing approaches. For example, Glade [13], an approach of inferring an input grammar, requires every production rule to be valid in its inferred CFG, indicating that the approach requires the inferred grammar to always produce a valid input string. Once any incorrect input string is produced from the inferred grammar, Glade would invalidate and remove latest added production rule. This "satisfied-or-invalid" assumption assures the overall validity of the CFG inferred by Glade. But as shown in Chapter 2, the generalization ability of Glade is limited by this assumption. In code generation, especially the tasks of NL to code (NL2Code), the users want to synthesize a program from their NL instructions. Due to the ambiguity of the NL input, it is often hard to extract constraints that must be satisfied [14].

To overcome the aforementioned limitation, we explore ideas to extract and maintain probabilistic constraints in this dissertation. Probabilistic constraints are defined as constraints assigned with a probability distribution denoting the likelihood of each constraint to be satisfied. The likelihood of each constraint reflects the confidence level that the constraint should be satisfied according to our estimation. In other words, these constraints may not always be satisfied at the same time. One example for such probabilistic constraints is

probabilistic context-free grammar (PCFG). PCFG is a CFG with a probability assigned to each production rule. The probability denotes the likelihood for each production rule to be applied when a string is generated from the grammar. As shown in Chapter 2, by using probabilistic constraints in the form of PCFG in the task of learning an input grammar, our proposed approach can infer the input grammar with substantially higher grammar coverage and the fuzzer guided by our inferred grammar could achieve higher code coverage.

Leveraging probabilistic constraints is a challenging task for three main reasons. First, in code analysis and generation, probability distributions are usually not given as part of the given input. To extract probabilistic constraints, we need to initialize and dynamically adjust the probability distribution or use other heuristics to resemble the probabilities. Second, since not every constraint needs to be satisfied, we need to consider all constraints with a non-zero probability to be satisfied. There are a large number of constraints with non-zero probability. To effectively conduct downstream tasks, we need to design mechanisms to filter out constraints with low probability without a large impact on the result. Third, when constraints are applied in later steps such as constrained synthesis, due to the increased number of probabilistic constraints to be considered and different combinations of constraints to be satisfied in the solution, an effective algorithm of pruning and ranking is also important to produce accurate results within reasonable time.

To tackle these challenges, we propose approaches to guide code analysis and generation with probabilistic constraints in four specific tasks. In the first half, we investigate the code analysis tasks. In the first part (Chapter 2), we focus on how to maintain the probability distribution on grammatical constraints to allow better generalization ability in the grammar-guided fuzzing problem. In the second part (Chapter 3), we focus on how to design different types of constraints to more accurately locate the root cause when conducting root cause analysis in a complicated microservice system. In the second half, we investigate the code generation tasks. In the third part (Chapter 4), we focus on how to extract probabilistic constraints from an ambiguous input such as NL and data context in the task of NL-to-visualization (in short as NL2Visualization). In the fourth part (Chapter 5), we focus on how to generate additional probabilistic constraints based on the given NL instruction to rank the initial candidate regular expressions in the task of NL to regular expression (in short as NL2Regex).

1.1 THESIS STATEMENT

In this dissertation, we make the following thesis statement:

Effective extraction and application of probabilistic constraints can improve the effectiveness of constraint-based approaches in the tasks of code analysis and generation.

In this dissertation, we first investigate the task of grammar-guided fuzz testing in the code analysis part. In this task, a program accepting a string as input and producing a YES/NO label as output is given. The task is to synthesize the context free grammar that would generate all input strings with the expected output label YES and then use the grammar to perform grammar-guided fuzz testing. We find that Glade [13], a state-of-the-art input grammar synthesizer has limited generalization ability. We introduce probabilistic constraints to the problem by generalizing the synthesized CFG into a PCFG. We use reinforcement learning (RL) techniques to adjust the probability distribution in the PCFG so that we can keep a balance between the ability to generalize and the synthesized grammar's accuracy. We find our approach REINAM substantially improve the coverage (recall) of the synthesized grammar without losing the accuracy of the grammar. The fuzzer guided by our synthesized grammar can also achieve higher code coverage compared to an industry fuzzer and other grammar-guided fuzzers.

We then investigate the task of root cause analysis in a large-scale microservice system. In this task, the dependency of the system and related logs, metrics are given. The task is to locate the root cause for a given anomaly report. We find existing approaches already leverage probabilistic constraints but they fail to address the operational, scale and monitoring complexity of the problem. We design event-based constraints to effectively tracking on the causality graph that is generated using the event-based constraints. We then apply root cause ranking algorithm based on the graph to locate the most probable root cause. We find our approach Groot substantially improve the accuracy of the root cause located.

In the part of the code generation, we first investigate the task of NL to visualization (NL2Visualization). In this task, an NL instruction and a dataset is given. The task is to generate a program to produce the visualization based on the dataset to match with the NL description. We find that the classical constraints extracted from the NL description are not enough to synthesize the desired visualization program. We extract additional probabilistic constraints from the data and program context. We then apply a constrained syntax-guided synthesis algorithm using the combination of classical and probabilistic constraints to synthesize the visualization programs. We find our approach NL2Viz achieve substantial higher accuracy compared to state-of-the-art approaches on the NL2Visualization task.

We then investigate the task of NL to regular expression (NL2Regex). In this task, an NL instruction and a set of sample strings input/output are given. The task is to find a regular expression that matches with the NL instruction. (The regular expression also needs

to pass the sample input/output pairs.) We find existing approaches only rely on constraints that are the example input/output pairs provided by users. Therefore existing approaches find it difficult to rank their list of candidate regular expressions being synthesized. We generate probabilistic constraints by generating input strings that are able to differentiate different candidate regular expressions and their corresponding expected output labels using a pre-trained model as the oracle. We use the feedback on the probabilistic constraints to rank the candidate regular expressions and substantially improve the ranking of the results.

Note that although we investigate the extraction and application of probabilistic constraints in this dissertation, some constraints we explored do not come with an explicit probability distribution. These constraints still have an implicit and inherent probability distribution denoting the likelihood of each constraint to be satisfied. But in certain scenarios, we are unable to precisely quantify the accurate probability distribution for the constraints. Therefore we have designed and implemented techniques to extract and apply the probabilistic constraints without an accurate probability distribution. For example, in NL2Viz, we treat some constraints as soft constraints that should be satisfied as many as possible.

1.2 CONTRIBUTIONS

This dissertation makes the following main contributions:

- An input-grammar inference framework namely REINAM using reinforcement learning. In REINAM, the task is to synthesize the grammar (which is in the form of CFG in this work) of the input string that can be accepted by a given program. In this work, we specifically focus on an approach to maintain a PCFG based on the input strings dynamically sampled during the inference process. We use reinforcement learning to leverage the feedback from the program under test to adjust the probability distribution for the PCFG. We evaluate REINAM on 11 real-world benchmarks with manually written grammars used in real scenarios. Our evaluation results show that REINAM outperforms Glade [13] (a state-of-the-art tool) in terms of precision, recall, and fuzz testing coverage for most of the benchmarks. In one of our benchmarks—namely, the input grammar encoding regular expressions that can be accepted by the GNU Grep [15]—REINAM improves recall from 0.02 to 1.0, indicating that the grammar inferred by REINAM actually covers the entire program input space.

- A graph-based root cause analysis approach namely Groot for microservice architecture. In Groot, the task is to find the root cause of a reported anomaly in a microser-

vice system. In this work, we specifically focus on incorporating domain knowledge from site reliability engineering (SRE) engineers into a set of constraints to define the events and causalities between the events in the system. We use such rule constraints to construct an event-based causality graph to perform root cause analysis (RCA) in a large distributed-system. We implement and evaluate GROOT on eBay's production system that serves more than 159 million users and features more than 5,000 services. Our experimental results show that GROOT is able to achieve 95% top-3 accuracy and 78% top-1 accuracy for 952 real production incidents collected over 15 months with the help of event-building rule constraints.

- An NL to visualization tool namely NL2VIZ. In NL2VIZ, the task is to generate a visualization according to the NL description given by users and its corresponding data set. In this work, we specifically focus on extracting constraints from the given NL input as the hard constraints and complement the hard constraints using constraints extracted from other sources of input such as code or data context as the soft constraints. We implement NL2VIZ as a plug-in tool in the popular Jupyter Notebook environment. Our evaluation results show NL2VIZ achieves the accuracy comparable to state-of-the-art tools, and data scientists find the tool easy to use in real world scenarios.

- An NL to Regular Expression (NL2Regex) tool namely NRRANKER. In NRRANKER, the task is to synthesize a regular expression from an NL description given by users. In this work, we specifically focus on generating probabilistic constraints consisting of additional test input strings that can differentiate behaviors of the candidate regular expressions being synthesized and their corresponding expected test outputs generated from the NL description. NRRANKER further integrates an existing NL2Regex system with the preceding test generation using pre-trained models. We have evaluated our approach on two most commonly used datasets to show the effectiveness of NRRANKER. NRRANKER achieves the top-1 accuracy of 89.8% and 90.1% on the two datasets, respectively, outperforming state-of-the-art approaches by 11.1% on average.

1.3 ORGANIZATION OF DISSERTATION

The remaining chapters of this dissertation are organized as follows.

Chapter 2: REINAM: Reinforcement Learning for Input-Grammar Inference

This chapter presents REINAM, a novel framework to synthesize program-input grammars based on reinforcement learning.

Chapter 3: GROOT: An Event-graph-based Approach for Root Cause Analysis

This chapter presents GROOT, an event-graph-based approach for root cause analysis tackling challenges in industrial settings.

Chapter 4: NL2VIZ: Natural Language to Visualization via Constrained Syntax-Guided Synthesis

This chapter presents NL2VIZ, a novel NL2CODE approach that aims to address challenges on the user inputs and interactions in the application domain of NL2VISUALIZATION.

Chapter 5: NRRANKER: Translating Natural Language to Regular Expression via Integrating Pre-Trained Model and Test Generation

This chapter presents NRRANKER, an approach that integrates an existing NL2Code system with novel test generation using a pre-trained model.

Chapter 6: Related Work

This chapter presents an overview of related work on the topics of input grammar inference, root cause analysis, and NL2CODE program synthesis.

Chapter 7: Conclusions and Future Work

This chapter summarizes the chapters covered in this dissertation, concludes the dissertation, and discusses possible future work that can be done following the work in this dissertation.

CHAPTER 2: REINAM: REINFORCEMENT LEARNING FOR INPUT-GRAMMAR INFERENCE

2.1 OVERVIEW

Many programs take a string of symbols as inputs. The set of such strings that a program accepts is called a language, which is represented by a program-input grammar. Program-input grammars facilitate understanding of the input structure and are essential for a wide range of applications such as symbolic execution [16, 17] (generally test-input generation), reverse engineering, protocol specification [18], delta debugging [19], prevention of exploits [20, 21], and identification of a software system's acceptability property [22]. Despite the importance of program-input grammars, acquiring them often demands a lot of manual effort, and they are often either not specified or specified in a machine-unfriendly form (*e.g.,* textual documents). For example, the full specification of the PDF format is available only in the form of textual document with over 1,300 pages [23]. For a program whose input grammar is not specified in a machine-friendly form, existing approaches attempt to infer the input grammar based on program analysis [24, 25, 26, 27], language induction [13, 28, 29, 30, 31], and machine learning [32, 33, 34].

However, these existing approaches of grammar inference are not able to produce grammars of sufficient quality (in terms of completeness and accuracy) for real-world software systems due to the following three main challenges.

Unanalyzable code. The existing approaches based on program analysis [24, 25, 26, 27] infer input grammars based on static-analysis information of the target program's code or from runtime program information collected via instrumentation of the program. However, these approaches cannot handle programs that cannot be instrumented (such as web services) or parts of programs that are too difficult for static program analysis to handle (such as native code or dynamic language features).

Low variety and quality of seed inputs. The existing approaches based on language induction [13, 28, 29, 30, 31] leverage the language-induction algorithms to synthesize input grammars given a set of seed inputs. However, the effectiveness of language-induction algorithms highly depends on the variety and quality of the seed inputs. For example, to infer an input grammar for a program that parses IP addresses, if the seed inputs contain only IPv4 addresses, the grammar inferred by language-induction algorithms could not capture the IPv6 formats.

Unavailability of a large number of seed inputs. Given a large number of seed inputs, the existing approaches based on machine learning [32, 33, 34] can train a machine-

learning model representing input grammars that can be used to generate inputs for fuzz testing. However, for many programs, there are not so many valid examples to learn from, and the machine-learning model is not human-comprehensible. As the model serves solely for the generation purpose, human analysts would not be able to have a similar level of understanding about the model as they understand regular languages or context-free grammars (CFGs) [35, 36].

To tackle the preceding challenges, we aim to address an inherent limitation in existing state-of-the-art approaches for grammar inference (e.g., Glade [13]). Existing approaches usually leverage active learning which includes an iterative process of generalization steps, each of which generates new grammar candidates from the given seed inputs (more details are described in Section 2.2). Such approach discards a grammar candidate if *any* of its generated strings is rejected by the program that implements the grammar check. Such design decision is common in active learning approaches, resulting in a rigid strategy of no-overgeneralization-allowed to ensure that grammar candidates in each generalization step are accurate (*i.e.,* all strings generated from a synthesized grammar are covered by the ideal input grammar). However, such design decision also misses the opportunity to potentially expand the coverage of the final synthesized grammar (*i.e.,* the overlapping scope of all the strings generated from the final synthesized grammar and all the ones generated by the ideal input grammar).

To retain the opportunity of expanding coverage in a generalization step while achieving high accuracy for the final synthesized grammar, our work leverages the probability distribution in the Probabilistic Context-Free Grammar (PCFG) model learned from the given seeded inputs to *improve* imperfect grammar candidates instead of *discarding* them as done by the rigid no-overgeneralization-allowed strategy used by the existing approaches [13]. To train a probabilistic generative model such as PCFG, a large dataset is usually needed. Because the available seed inputs are quite limited, we conduct reinforcement learning [37] to tune the probability distribution in PCFG. In reinforcement learning, the program that implements the grammar check can be used as a black-box oracle to give feedback for improving the PCFG model to gradually generate more data to further tune the probability distribution.

Centered around the PCFG model gradually enhanced via reinforcement learning, in this chapter, we propose, REINAM, a novel framework of reinforcement learning that synthesizes comprehensive and interpretable program-input grammars. Figure 2.1 shows how REINAM creatively formulates the grammar-synthesis task into a reinforcement learning problem including an agent (a PCFG) that interacts with an environment (the target program). The agent generates an action (a set of program inputs) that causes a state transition of the

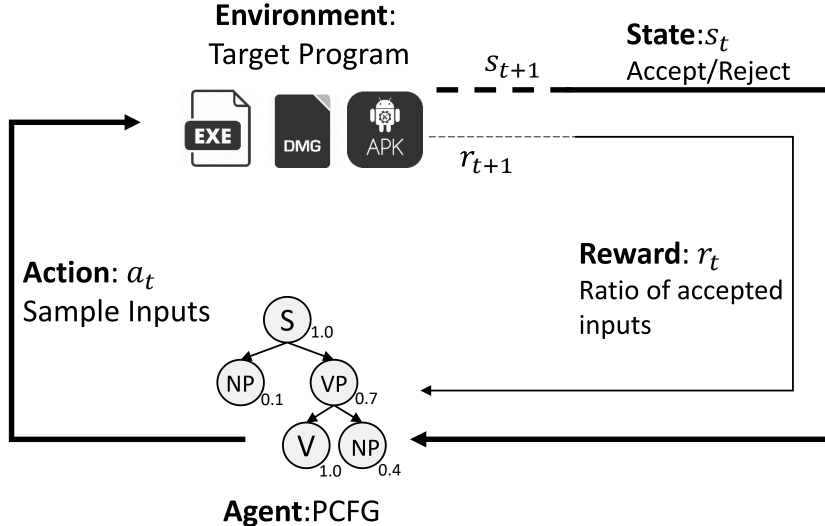


Figure 2.1: Modeling Grammar Synthesis as a Reinforcement Learning Process

environment. Upon each performed action, the agent observes the next state (the inputs being accepted or rejected) and receives a reward (the ratio of accepted inputs).

In REINAM, we compute probabilities in the PCFG, and then use the probabilities to determine whether we are keeping the candidates after applying certain generalization operators to evolve the structure of our agent. REINAM is in contrast to traditional reinforcement learning (*e.g.,* Deep Q-learning [37]), which tunes the parameters of the agent (*e.g.,* deep neural network) without changing the structure of the agent to maximize the corresponding reward. In our problem setting, we need to constantly evolve the structure of our agent (*i.e.,* PCFG) to achieve a more comprehensive and accurate grammar, as accomplished by REINAM.

We design REINAM based on our two key insights. First, REINAM enhances the completeness of the final synthesized grammar by *improving* the incorrect production rules instead of *discarding* the rules. Built upon the benefit of PCFG and reinforcement learning, REINAM gradually mutates incorrect production rules through the reinforcement learning agent to make the rules accurate. In contrast, the existing approaches [13] just discard these production rules if any membership query returns negative. Second, applying a single generalization operator often results in an inaccurate grammar, while as illustrated in Section 2.3, if we compose multiple such inaccurate generalizations, the composed generalizations may complement each other, forming an accurate composite generalization. REINAM represents the accuracy of each production rule as a probability, being a parameter of the reinforcement learning agent (the PCFG). Such representation allows inaccuracy to exist in single generalization step, and decreases/eliminates the inaccuracy through composite generalizations

later.

We evaluate REINAM on eleven real-world subjects with manually written grammars used in real scenarios. We measure the precision and recall of the synthesized grammars. We also evaluate the benefits of the synthesized grammars in assisting the task of fuzz testing. Our evaluation results show that REINAM outperforms Glade in terms of precision, recall, and fuzz-testing coverage for most of the subjects. In one subject whose input grammar is all regular expressions accepted by the GNU Grep [15], REINAM improves the recall of the grammar from 0.02 to 1.0, indicating that the grammar inferred by REINAM even covers the whole input space.

This chapter makes the following main contributions:

- A novel framework, REINAM, to synthesize program-input grammars based on reinforcement learning.
- Novel formulation of the grammar synthesis problem as a reinforcement learning problem where we use the PCFG as the agent, sampled inputs as the actions, and the ratio of accepted inputs on each production rule as the reward function. The training procedure decides to take the generalization operator in each iteration by adjusting the probability of the corresponding production rule.
- Evaluations on eleven real-world subjects showing that REINAM effectively synthesizes generalized and interpretable grammars, and evaluations on fuzz testing showing that REINAM is helpful in improving effectiveness of fuzz testing.

2.2 BACKGROUND

Glade [13] is an existing state-of-the-art approach for synthesizing a program-input grammar from a given set of seed inputs specified by the users for a program that includes input validation for accepting only the valid inputs. Glade requires only black-box access to the program and uses it as an oracle to determine whether a given input is valid or not. The algorithm used by Glade iteratively generates new candidate grammars by applying generalization operators (from a predefined set of operators) upon the given seed inputs. The algorithm then verifies the correctness of these candidate grammars by applying a set of checks on the new inputs generated from the candidate grammars against the program oracle. Glade's algorithm consists of two steps. The first step learns a regular grammar by applying generalization operators such as *repetition* and *alternation* as illustrated below on the given seed inputs. The second step transforms the learned regular grammar into a

context-free grammar by applying *merging* operators as illustrated below. In between the first and second step, the *character generalization* operator is applied to generalize over characters.

- *Repetition* is to repeat a certain substring in a regular grammar. For example, the regular grammar a can be generalized to $a(a)^*$ by using repetition.
- *Alternation* is to alternate a substring in a regular grammar into its decompositions in a string quoted by '*'. For example, the regular grammar $(ab)^*$ can be generalized to $(a|b)^*$.
- *Merging* is to equate two non-terminal symbols in the context-free grammar translated from the regular grammar resulted from Step 1. For example, we have a Context-Free Grammar (CFG) $S \rightarrow ('a'\ T\ 'a')^*; T \rightarrow ('b' \mid 'c')^*$ (note that this CFG is not a standard form of CFG). Then we merge S and T by substituting T with S. The grammar becomes $S \rightarrow ('a'\ S\ 'a')^*; S \rightarrow ('b' \mid 'c')^*$.
- *Character generalization* is the simplest operator. The idea is that for some terminal symbols, such as 'a' in "$(a)^*$", we are also testing other characters 'b','c',... in the alphabet other than 'a'. The intuition is that often the time if one letter can work, the others would probably also work.

Glade then performs a set of checks in order to avoid overgeneralization. It constructs a set Chk of strings and for each $\alpha \in Chk$, the string α has the production rule that we just mutate in its derivation. Then Glade executes the program with each input string in Chk to get the result of either accepted or rejected. If *any* string α gets rejected, Glade rejects this generalization.

Note that this mechanism is designed to follow the no-overgeneralization-allowed strategy mentioned in section 2.1, but overgeneralization could still happen in Glade. The reason is that the set Chk may not be sufficient to check all possible generated strings. It could be all strings in Chk are accepted but there still exists string generated from the grammar that would be rejected by the program. We can see overegeneralization also appear in Glades' synthesized grammar in the evaluation.

Probabilistic Context-Free Grammar (PCFG) is a CFG augmented with a probabilistic distribution. A PCFG G is defined by a tuple:

$$G = (M, T, R, S, P) \tag{2.1}$$

$$
\begin{aligned}
genchar &\rightarrow' 0' \mid 'A' \mid 'a' \\
nbchar &\rightarrow genchar \mid s \mid t \mid '!' \mid '"' \mid '\#' \mid '\$' \mid '\%' \mid '\&' \mid ''' \\
&\mid ',' \mid '.' \mid '/' \mid ':' \mid ';' \mid ' <' \mid ' =' \mid ' >' \\
&\mid '@' \mid '_' \mid '\backslash' \mid ' \sim' \\
npchar &\rightarrow nbchar \mid ' +' \mid ' *' \mid '?' \\
char &\rightarrow nbchar \mid nbchar\ npchar \mid char\ p\ nbchar \\
nums &\rightarrow numall \mid nums\ numall \\
tok &\rightarrow char \mid tok\ char \\
single &\rightarrow tok \mid single\ tok \\
single &\rightarrow single\ '\backslash'\ '\{'\ nums\ '\backslash'\ '\}' \\
&\mid single\ '\backslash'\ '\{'\ nums\ ','\ nums\ '\backslash'\ '\}' \\
single &\rightarrow single\ '['\ tok\ ']' \mid single\ '['\ ' \wedge'\ tok\ ']' \\
&\mid single\ '['\ ' ='\ tok\ ' ='\ ']' \mid single\ '['\ '.'\ tok\ '.'\ ']' \\
&\mid single\ '['\ ' :'\ tok\ ' :'\ ']' \\
regex &\rightarrow single \mid regex\ single \\
regex &\rightarrow regex\ '\backslash'\ '('\ regex\ '\backslash'\ ')'
\end{aligned}
$$

Figure 2.2: The ground-truth grammar of “GREP” program

where M is the set of non-terminal symbols, T is the set of terminal symbols, R is the set of production rules, S is the start symbol, and P is the probabilities of each production rule. The probabilities are defined as follow: for each non-terminal symbol A, suppose that it has k different production rules. They can be connected by the ‘ | ’ symbol to be $A \rightarrow Prod_1 \mid Prod_2 \mid ... \mid Prod_k$. Then when we want to apply rules on A, we have the probability $P_A(i)$ of choosing production rule $A \rightarrow Prod_i$. The probabilities should have the property of $\sum_{i=1}^{k} P_A(i) = 1$.

2.3 MOTIVATING EXAMPLES

Despite its utility, Glade has a number of limitations shown by the motivating examples discussed in this section. According to the evaluation results of Glade [13], for some programs under evaluation, Glade synthesized a grammar that achieved high coverage of the valid input space, and a fuzzer equipped with the grammars inferred by Glade outperformed other fuzzers under comparison. However, we find that the “GREP” program used in the evaluation can achieve only very low coverage of the valid input space, and also in “GREP” even a naive fuzzer can outperform the fuzzer equipped with the grammars inferred by Glade.

Figure 2.2 shows that the grammar of the “GREP” program consists of many special char-

acters (*nbchar* and *npchar*). These special characters not only form the basic building blocks for the grammar (*char*) but also serve as the special control characters in the grammar (see the production rule of *single*). Therefore, in Glade's Step 1, which involves generalization of regular languages, the generalization would be likely to fail. For example, suppose the seed input is "$[\wedge a]$", then neither generalization operator in Step 1 are able to cover the grammar "$[= (a)* =]$". What we can do at best is that we first apply the character generalization operator, the grammar would become "$[(\wedge| =)a]$". Next we can apply the repetition operator, the grammar can be generalized into "$[(\wedge| =)(a)*]$". However, because in the character generalization step, "=" served as a simple character, we will not be able to generalize the grammar to be able to cover "$[= (a)* =]$", in which "=" is served as the special character in the production rule of the symbol *single*.

In this case, since the generalization in Step 1 would easily fail, the hope of synthesizing a grammar with good coverage lies on the quality of seed inputs, i.e., whether the set of seed inputs can contain the case of $[= a =]$. In the experiment setting of Glade, the author uses only 50 seed inputs that are randomly generated from the ideal grammar. The first concern here is that in the real world scenario, the developers using the tool are unlikely to know the ideal grammar. One possible case is that they only use the strings used in the tests as the seed inputs. Therefore we can expect the quality of seed inputs will be far worse than those sampled from the ideal grammar.

The second concern is as the grammar of "GREP" queries is very coarse in terms that special characters form both content of queries and format of the queries, randomly generated seed input won't be enough for Glade to synthesize the desired grammar.

Based on these concerns, the quality of seed input highly affects the performance of Glade. So we are inspired to use the power of test generation tools, for example, Pex [38, 39]. Pex is a white-box automated testing tool based on dynamic symbolic execution. It will explore possible program execution paths to generate test input that cover as largely as possible of the program. Our results show that we significantly increase both the precision and the recall on the "GREP" program with the help of Pex.

The second limitation of Glade is that, it does not allow overgeneralization. If the any string in the constructed set *Chk* gets rejected by the program, Glade would reject this generalization. (Overgeneralization could still happen since the set *Chk* is not complete, however it's not desired by design of Glade.) However, sometimes an overgeneralized intermediate grammar could increase the coverage of grammar. Moreover, if we further apply generalization on this intermediate grammar, the overgeneralization might be avoided. To see this, let's consider the grammar in Figure 2.2 again. For the example of grammar "$([= a =])^*$", we have claimed that any further alternation won't be able to cover the string "$[.a.]$" since it requires

$$
\begin{array}{ll}
S \rightarrow' a'\ T\ 'a' & \\
T \rightarrow P\ Q\ T \mid P\ Q & \\
P \rightarrow ... & \\
Q \rightarrow ... &
\end{array}
\Longrightarrow
\begin{array}{l}
S \rightarrow' a'\ S\ 'a' \\
S \rightarrow P \mid Q \\
P \rightarrow ... \\
Q \rightarrow ...
\end{array}
$$

Figure 2.3: An imaginary grammar: initial grammar(left) and the generalized grammar(right)

$$
\begin{array}{l}
S \rightarrow' a'\ T\ 'a' \\
T \rightarrow P\ T \mid Q\ T \mid P \mid Q \\
P \rightarrow ... \\
Q \rightarrow ...
\end{array}
$$

Figure 2.4: An imaginary grammar: perform alternation on the initial state

character generalization at two places at same time. However, if we allow overgeneralization, which means we keep the intermediate grammar "$([= a(= |.)])^*$" after applying character generalization at the second "=". Note this grammar will produce input like "$[= a.]$". which will be rejected by the program. But if we apply another character generalization at the first "=", the grammar can be transformed into "$([(= |.)a(= |.)])^*$", which both increases the coverage (now can cover string "$[.a.]$"), and avoids overgeneralization. Currently Glade is taking the strategy of "No Overgeneralizations Allowed" making the previous generalization steps impossible. We noted that Glade is using PCFG (Probabilistic Context-Free Grammar) in the seed input generation. We want to see what else can the probability in the PCFG help other than only generating the seed inputs. And by the help reinforcement learning, we are allowed to adjust the probability distribution of this PCFG to reflect its accuracy (how overgeneralized each rule is). So that we could keep some overgeneralization intermediate grammar to have larger coverage, while sacrificing small percentage of accuracy.

The third limitation of Glade is that, the generalization operators are divided stiffly into two steps, that is repetition and alternation in Step 1 and merging in Step 2. Glade only performs the repetition and alternation in Step 1 and latter transformed corresponding changes to the translated context-free grammar. However, the repetitions and alternations are still viable and actually needed in Step 2.

To show this, we can see in Figure 2.3. This is an imaginary grammar in the procedure of Glade execution. The left grammar is the intermediate state in the progress of Step 2 generalization. The right grammar is one possible generalization of the left one. We can see that actually it generalize the first two production rules by introducing a production rule $S \rightarrow' a'\ (P|Q)^*\ 'a'$ (this rule is not written in formal CFG format). Let's say the right

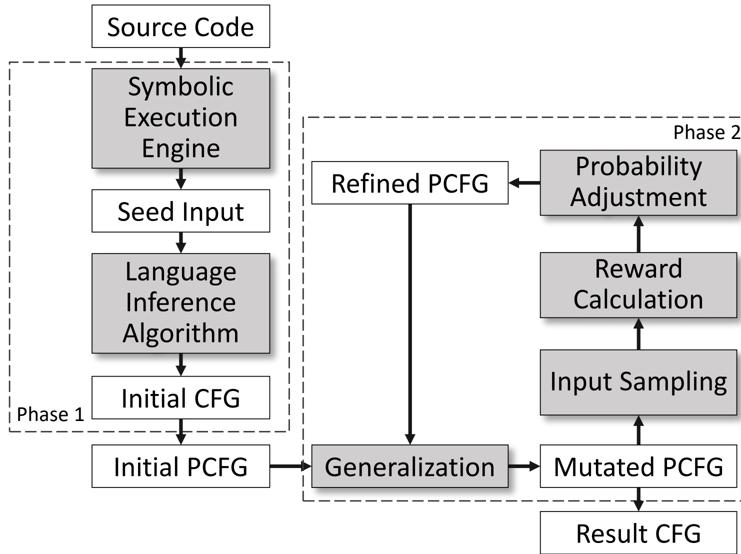


Figure 2.5: Workflow of REINAM

grammar is a subset of the ideal grammar, which means the generalization is correct and should increase the coverage of the resulting grammar. We show that this generalization can be done in two steps, with the first to be an alternation that changes the production rule of T from $P\ Q$ to $P \mid Q$. Then we can perform a merging on the symbol T and S. Finally we get the below grammar in Figure 2.3. However, in Glade, such generalization is not viable, because as we stated in the previous paragraph, no repetition and alternation is performed in phase 2. This example inspires us to combine the two phases of Glade. We can directly perform the four different kinds of generalization all on the context-free grammar, which also simplifies the generalization procedure as we do not need to split two steps.

2.4 APPROACH

We next present our REINAM approach, which consists of two phases: generating higher-variety, higher-quality, and higher-quantity seed inputs using Pex and conducting generalization on the PCFG using reinforcement learning.

As shown in Figure 2.5, the input to REINAM is the target program for which we want to synthesize input grammars. We first leverage symbolic execution engine Pex [38] to generate strings as the test input of program. We then use these strings as the initial inputs of language inference algorithm such as Glade. The resulting grammar is the output of Phase 1.

In Phase 2, we first initialize the PCFG from the CFG generated in Phase 1. Then we use reinforcement learning to refine our PCFG. The procedure of reinforcement learning consists of four steps: applying generalization operators, sampling strings from the PCFG, calculating the reward for each production rule, and adjusting the probability distribution based on the calculated reward. Noted that the generalization operators are applied to mutate the current PCFG. The operators can be specified according to mutant operators for different grammars. These pluggable operators bring flexibility to our approach. Our approach can be easily extended to grammar-transformation operators other than the ones discussed in this work.

2.4.1 Phase 1: Generating Seed Inputs using Pex

Phase 1 is a one-pass procedure that takes the assembly code of a program as the input to symbolic execution engine, uses the engine output as the initial inputs of Glade, and then runs Glade.

We use symbolic execution engine Pex [38] in phase 1. Pex performs path-bounded dynamic symbolic execution by repeatedly executing the program to generate path-based constraints and using SMT (satisfiability modulo theories) checker to solve these constraints to obtain the program inputs that would lead to different execution paths. The idea of using Pex to generate seed input is inspired by the fact that better seed inputs would likely produce better grammar. Such correlation is discussed both in the Glade paper [13] and in section 2.3. Additionally, as we point out in section 2.3, a set of seed inputs is good mostly determined by the different "categories" of input it can cover, which implies a high code coverage if we track the amount of code covered by executing the program with these inputs. Pex, by its design, is targeting to achieve high code coverage. Other test generation tools may also be used in this phase.

2.4.2 Phase 2: Generalization on the PCFG using Reinforcement Learning

Initialization of PCFG Between the phase 1 and 2, we have an intermediate step of PCFG initialization. For each nonterminal symbol S, we count the number n of production rules that expand S (S on the left side of $\rightarrow$), then each of these rules would be assigned a probability of $\frac{1}{n}$ equally.

Reinforcement Learning (RL) Reinforcement learning is an area of machine learning inspired by behaviorist psychology. The basic concerns are how software *agents* take *actions*

in the *environment* to maximize some metric called *reward*. In our context of the grammar synthesis problem, these concepts are:

- **Environment.** The program under test.
- **Action.** The set of input strings sampled from the PCFG.
- **State.** The program output (accept or reject) when taking the sampled input strings.
- **Reward.** The ratio of the number of accepted inputs out of total number of sampled input.
- **Agent.** The PCFG itself. Note that the purpose of general RL is to optimize the agent's ability to make better actions in terms of getting better reward. The purpose in our context is to increase the quality (precision and recall) of the grammar. We achieve our purpose by applying generalization operators to the existing grammar to generate a mutated grammar each time. The generalization will increase (at least not decrease) the recall. So our purpose of the reward function is to keep the precision to prevent overgeneralization. We use the maximum likelihood estimation to tune the probability distribution of the PCFG so that if any production rule's probability is lower than the threshold, we will remove the rule from the PCFG. By this means, we are improving the ability of our agent (grammar) to make actions (generating inputs) to get better reward (achieving higher coverage).

Generalization Operators

- **Character Generalization.** Same as the character generalization introduced in Section 2.2. The intuition is that often the time if one letter can work, the others would probably also work. We would assign the probability of $\frac{1}{\#(current\ characters)}$ to the newly added character and reduce other probabilities in proportion. Take the GREP program in section 2.3 for example, in the string "$[= a =]$", we are also testing other characters $'b',' c', ...$ in the alphabet other than $'a'$.
- **Repetition.** Repetition is to change part of the grammar from "p" into "$p(p)*$". In the CFG, we do the repetition by picking a production rule and trying to repeat the symbols on the right hand side. For example, for the production rule $S \rightarrow P\ Q$, we would try both $S \rightarrow P\ P*\ Q$ and $S \rightarrow P\ Q\ Q*$. To express $'*'$ in CFG, we'll translate it into $S \rightarrow P1\ Q;\ P1 \rightarrow P1\ P;\ P1 \rightarrow P$ and $S \rightarrow P\ Q1;\ Q1 \rightarrow Q1\ Q;\ Q1 \rightarrow Q$. The probability of $S \rightarrow P1\ Q$ remains same as the original $S \rightarrow P\ Q$, and the probability of $P1 \rightarrow P1\ P$ and $P1 \rightarrow P$ are both set to 0.5.

- **Alternation** Alternation is to change part of the grammar from "pq" into "$p|q$". In the CFG, we do the alternation by picking one production rule and randomly decomposing a substring of the production rule into two parts and list them as alternations. For example, for the production rule $S \to P\ Q\ 'a'$. Then we will try these new production rules $S \to P1\ 'a';\ P1 \to P \mid Q$. The probability of original $S \to P\ Q\ 'a'$ would be divided by 2 to be assigned to both $S \to P\ Q\ 'a'$ and $S \to P1\ 'a'$. The probability of $P1 \to P \mid Q$ would be 1.

- **Merging.** Merging two nonterminal symbols is to substitute all usage of one symbol into another. For example, if we have $P \to ...$ and $Q \to ...$, if we want to merge P and Q, we simply add one rule $P \to Q$. We would assign the probability of $\frac{1}{\#(production\ rules\ of\ P)}$ to $P \to Q$ and reduce other probabilities in proportion.

The reason why we choose these generalization operators is based on two factors. First, these rules substantially increases grammar coverage in Glade's evaluation, and the form of these rules is simple which provides convenience for both implementation and human understanding. Second, many existing works also used same or similar operators/mutants [27, 40]. Such fact implies the effectiveness and flexibility of the operators.

Input Sampling. We perform the following steps n times to sample n strings from our PCFG:

- Start with the starting symbol S, $\alpha = S$.
- For each non-terminal symbol A in α, randomly apply one production rule on A based on the probability of each rule.
- If there is still non-terminal symbol in α, repeat the previous step.
- Return the string $\alpha = \alpha_1 ... \alpha_k$, in which all α_i is a terminal symbol (character).

After sampling n inputs from the grammar, we execute the program on these n inputs and collect the output from the program. We record whether the program accept or reject our test inputs.

Probability Adjustment. The purpose of the probability adjustment is that, in our PCFG, the probability indicates the correctness of the production rule, that is whether this rule should exist in the ideal grammar. Remind that our model allows overgeneralization to exist. In glade, if a input string generated by a production rule is rejected by the program,

that rule is no longer viable, and the generalization step that produced such rule should be revoked. However, in our model, we may keep such rule if it can simultaneously generate strings with accept result. We achieve this by adjusting the probability of each production rule based on the reward function.

The reward function is calculated for each production rule. We track the production rule used in the sampling step for each input string. Therefore the reward function is defined on each production rule $r_i \in R$:

$$reward(r_i) = \frac{\#(accepted\ inputs\ using\ r_i)}{\#(inputs\ using\ r_i)} \tag{2.2}$$

The probability adjustment is using a maximum likelihood estimation [41]. We consider different non-terminal symbols separately since the probability is only related to different production rules of the same non-terminal symbol. Suppose we are considering the non-terminal symbol P and the probability distribution for all its production rules is θ. P has k production rules $r_1^P, ..., r_k^P$, and before the adjustment, the probability for each production rule is $\theta(r_i^P)$. The standard policy gradient [41] gives us the new θ' to be:

$$\theta' = \theta + \alpha \nabla_\theta log \pi_\theta(s_t, a_t) * v \tag{2.3}$$

in which $\pi_\theta(s_t, a_t)$ is the probability of the agent to produce action a_t under state s_t, α is a constant to be the learning rate and v is the reward. So here, the formula can be inferred to be:

$$\begin{aligned} \theta'(r_i^P) &= \theta(r_i^P) + \alpha \nabla_\theta log(\theta(r_i^P)) * reward(r_i^P) \\ &= \theta(r_i^P) + \alpha \frac{\#(inputusing\ r_i^P) * reward(r_i^P)}{\theta(r_i^P)} \end{aligned} \tag{2.4}$$

We normalize the new probability distribution $\theta'(r_i^P)$ to make the sum to be 1. We would run the sampling and adjustment steps multiple times for one generalization until the probability is stabilized. After we calculate the adjusted probability, we will remove the production rules with the probability lower than the threshold.

It is also an option to keep all the production rules and not to remove them as they have the probability of zero. But this would waste computations on generalizations of "dead" production rules to make the time complexity unscalable.

2.4.3 Reasoning and Discussion on the Probability in REINAM

We note that in the probability adjustment formula (2.4), the gradient $\frac{\#(inputusing\ r_i^P)*reward(r_i^P)}{\theta(r_i^P)}$ is non-negative. To address such issue, we normalize the probability to make the sum of all $\theta'(r_i^P)$ to be 1. After the normalization, we can see that the fact that all gradient is non-negative is not a problem here. If a certain production rule r_i^P is higher in the reward function such that strings produced using this rule is more likely to be accepted, then the gradient added in formula (2.4) would be larger, which leads to larger normalized new probability. On the contrary, if certain rule has lower reward function value, it will lead to the normalized new probability to be less than the probability before the adjustment.

We can also deduce the time complexity of our approach by estimating the number of sampling and adjustment for the convergence. Suppose the threshold we used to determine stability of the probability is ϵ (*i.e.,* if no probability of the production rules in one symbol changes is more than ϵ, we will terminate this round of RL process and start the next generalization) and the set of sampled inputs producing the same reward each time, we can see the gradient difference each time is $\alpha\frac{\#(inputusing\ r_i^P)*reward(r_i^P)}{\theta(r_i^P)}$ in which $\alpha*\#(inputusing\ r_i^P)*reward(r_i^P)$ remains constant under same set of sampling strings. So the $\theta(r_i^P)$ would converge proportionally to the ratio of accepted strings among all string using that rule. The number of sampling and adjustment for the convergence is approximately $O(\frac{1}{\alpha*\epsilon})$ which is constant.

The time complexity of our approach is $O(n*m*RL(n))$, n denoting the number of symbols in the grammar and $RL(n)$ is the time needed for one RL process with a n-symbol grammar, m is the number of total production rules. The $O(n*m)$ is the upper bound of generalizations that may be tried, though each generalization would introduce new production rules, but we only do generalization on the initial production rules set. $RL(n)$ equals the time used for sampling strings and getting the results of the program execution using the sampling strings as input. The time used for probability adjustment is negligible. Per our previous discussion, each generalization takes $O(\frac{1}{\alpha*\epsilon})$ rounds of sampling and adjustment, so $RL(n) = O(\frac{\#(Total\ strings\ sampled)*average\ execution\ time}{\alpha*\epsilon})$.

Another thing need to clarify is that the probability in our PCFG model represents how likely the resulting string would be accepted by the program if we apply certain production rule, but the probability does not represent the probability of the actual frequency of certain production rules used as the input grammar. The reason is that we don't actually have the distribution of strings in the real usage of the input grammar. As we mentioned in the previous sections, we initially set the probability of each production rule r_i^P under same symbol P to be the same. Afterwards we are adjusting the probabilities such that they are converged proportionally to the ratio of accepted strings among all strings using that certain

rule. The probability is used to eliminate the production rules with too small probability to eliminate the unreasonable overgeneralizations.

2.5 EVALUATION

To evaluate the effectiveness of REINAM and how much each phase contributes to the effectiveness of the tool, we have conducted evaluations on eight subjects.

We seek to answer the following research questions:

- **RQ2.1**: How effective is the final grammar synthesized by REINAM in terms of precision and recall?
- **RQ2.2**: How effective is the final grammar synthesized by REINAM in the fuzz testing scenario?
- **RQ2.3**: How does the two phases of REINAM contribute to the grammar's precision/recall and performance in fuzz testing?
- **RQ2.4**: What's the execution time of REINAM?

In RQ2.1, we compare REINAM with Glade in terms of the precision and recall of the final synthesized grammar. We compute the two metrics using the manually written ideal grammar for the subjects as ground truth. In RQ2.2, we evaluate the capability of REINAM to learn a grammar for a fuzz testing task. We feed the synthesized grammar by REINAM to a grammar-based fuzzer to perform fuzz testing on programs. We compare our results with Glade and a industry fuzzer. In RQ2.3, we compare the grammar after our phase 1 (using Pex generated seed input for Glade) and the final grammar after our phase 2 (generalization using RL). We compare the difference in both precision/recall and fuzz testing coverage to see the different improvement that each part brings in. In RQ2.4, we measure the time used by each phase of our tool.

2.5.1 Study Subjects

We use the subjects of manually written grammars from Glade [13]. These subjects are listed as follows:

- A grammar used to match URLs [42]. Note that we separate this grammar into four subjects based on the protocols (*i.e.,* "http", "https", "mailto" and "nntp"). We will

test whether REINAM can infer a complete grammar starting from seed inputs only including one of the four protocols.

- A grammar for the regular expression accepted as input of GNU Grep [15]. This grammar is discussed in Figure 2.2 in section 2.3.
- A grammar for a simple Lisp parser [43], including support for quoted strings and comments.
- A grammar for a XML parser [44], including all XML constructs except that only a fixed number of tags are included (to ensure that the grammar is context-free).
- A grammar for CSS parser [45], Cascading Style Sheets (CSS) is a language to describe the presentation of a document written in a markup language like HTML.

In addition to the existing subjects in Glade, we also add two subjects that generated from ANTLR [46]. ANTLR is a widely used parser generator tool. We use the two subjects to evaluate our tool's performance on synthesizing a program that generated by a automatic parser generator. We select grammars on the official website of ANTLR [46] which has less than 100 non-terminal symbols.

- A grammar used to match CSV files [47].
- A grammar used to describe simple first order logic formulas [48].

2.5.2 Experiment Settings

Precision and Recall We measure precision in order to determine whether our final synthesized language L is overapproximating the ideal grammar L_*. We calculate precision as $\frac{|E_{prec} \cap L_*|}{|E_{prec}|}$, where E_{prec} is a set of 1,000 strings randomly sampled from L. The number is essentially how much percent of the strings generated from our final synthesized grammar would be covered by the ideal grammar. We measure recall in order to determine whether L is underapproximating L_*. We calculate recall as $\frac{|E_{rec} \cap L|}{|E_{rec}|}$ where E_{rec} is a set of 1,000 strings randomly sampled from a reference grammar created to the specification of L_*.

Precision in this context measures the probability that a randomly selected string from the grammar would be accepted by the target program. Recall in this context refers to the percent of the strings generated from the ideal grammar that belong to our final synthesized grammar.

Generate Seed Input We use Pex's existing capabilities for dynamic symbolic execution in order to generate input strings that the parser accepts. We then use this set of strings as input to Glade.

Sampling from PCFG We sample a production A from PCFG G as follows:

- First, randomly select a production A' belonging to A.
- Second, for all A_1 A_2 ... A_n that make up A', if A_i is a nonterminal, recursively sample A_i, else if A_i is a terminal, return A_i.

All choices made are made over a uniform distribution for simplicity.

Program used for evaluation Because Pex requires source code to do the dynamic symbolic execution in order to generate program inputs, we write programs in C# to parse the grammars we listed in Section 2.5.1. We use parsers in the .NET system library to parse the grammar of URL, IP, regular expression and XML. We also test against an open-source lisp parser and an open-source css parser. The programs of two ANTLR grammars are generated by ANTLR in C# mode.

Fuzz testing Fuzz testing or fuzzing is an automated software testing technique. An effective fuzzer will generate "valid enough" inputs and then monitor the execution of the program on these inputs. One of the purpose in using fuzz testing is to observe the program's behavior under various inputs, so we want the fuzzer to get a high code coverage.

Grammar-based Fuzzer From Glade and REINAM, we can synthesize a CFG L which approximates the ideal language L_*. We use a grammar-based fuzzer to adapt the synthesized grammar into the fuzzing task. The fuzzer is similar to the input sampling step we described in section 2.4. We randomly sample 1,000 strings from the grammar based on the probability. However, for both grammars synthesized by REINAM and Glade, we treat the probability as a uniform distribution. The reason why we use uniform distribution instead of using the distribution from our PCFG is that the distribution in the PCFG has no relation to the real distribution. Then we execute the program on the sampled input and use Visual Studio to measure the code coverage which is denoted as the blocks covered. We use the code coverage by the manually written ideal grammar as the upper bound.

We compared our fuzzer against a state of the art fuzzer that does not employ grammar-based methods, Radamsa. Radamsa combines random bit-flipping with domain independent heuristics meant to test edge cases in order to create high quality mutation.

The tool for Learn&Fuzz [32] is not available for us to obtain at the experiment time. So we cannot use it to compare with our tool.

Table 2.1: Precision (P) and Recall (R)

Subject	http		https		nntp		mailto		ip		css		xml		grep		lisp		csv		fol	
	P	R	P	R	P	R	P	R	P	R	P	R	P	R	P	R	P	R	P	R	P	R
LI	.79	.06	.77	.07	.80	.05	.99	.13	1.0	.50	.99	1.0	.32	1.0	.79	.02	.65	1.0	1.0	.24	.90	.30
LI+SE	.96	.50	.96	.64	.97	.49	.97	.42	.91	1.0	.99	1.0	.56	1.0	.98	.40	.91	1.0	1.0	.40	.88	.45
LI+SE+RL	.87	.77	.91	.70	.95	.82	.93	.77	.82	1.0	.86	1.0	.50	1.0	.78	1.0	.73	1.0	.94	1.0	.75	.45

Table 2.2: Coverage (numbers are basic blocks covered in code)

LI (**L**anguage **I**nference algorithm): results of Glade only
LI+SE (**S**ymbolic **E**xecution engine): results of REINAM phase 1
LI+SE+RL (**R**einforcement **L**earning): results of REINAM phase 1+2
Radamsa: code coverage of the fuzzer Radamsa.

Subject	http	https	nntp	mailto	ip	lisp	css	xml	grep	csv	fol
Radamsa	1579	1558	1461	1157	2272	131	737	1349	1819	82	216
LI	1625	1623	1618	1691	2241	175	1819	1349	1819	88	348
LI+SE	1750	1796	1673	1829	2318	176	1642	1349	1819	88	353
LI+SE+RL	1784	1811	1716	1863	2325	211	1809	1349	1819	92	353
Total	2164	2164	2164	2164	2421	211	1948	1349	1819	111	414

2.5.3 RQ2.1: Precision and Recall

We can see from table 2.1 that our tool in general is very effective in grammar inference. The row “LI” shows the result of the grammar synthesized by solely running Glade and the row “LI+SE+RL” shows the result of the final grammar synthesized by REINAM.

On average, we can see that the precision almost stays at the same level, with a less than 1% change (-0.36% on average), while the recall gets drastically improved by 49.2% on average. The number is calculated by subtracting the number on the third row by the first row and taking the average after. So among the 11 subjects we evaluate, REINAM can outperform Glade in recall which represents the coverage of the final synthesized grammar without losing precision which represents the accuracy of the final synthesized grammar.

If we look into details to compare the performance of REINAM and Glade in each of the subjects. We can find that in the three subjects “css”, “xml” and “lisp”, Glade can already achieve 1.0 recall which means the grammar it synthesized already cover the whole input space of the three programs. In this sense, REINAM cannot further improve the recall but it manages to keep the perfect score as expected. After taking a closer look into these three subjects, we can see they are all in the similar form of parentheses matching pattern. “lisp” is a sequence of statements quoted by parentheses while “css” and “xml” have to match the tag names like parentheses.

For the 4 subjects (“http”, “https”, “nntp” and “mailto”) separated from the “url” grammar, we can see a drastically increase in recall. As we mentioned in the section 2.5.1, we intentionally split the “url” grammar into 4 parts. As expected, Glade is performing poorly

with the recall lower than 0.15 in all 4 subjects. This is due to the limitation of Glade that the quality of its synthesized grammar highly depends on the quality of the seed input as we mentioned in section 2.1. So since we only feed strings starting with “http” (or “https”, “nntp”, “mailto” in other 3 subjects) as the seed input, Glade is not able to explore other possible url protocols. Meanwhile, REINAM could leverage the help of Pex in phase 1 to have a seed input set with higher coverage. Therefore the final synthesized grammar of our tool gets in these 4 subjects have the recall to be above .70 without losing precision. The “ip” subject is similar. Glade start with the seed input in IPv4 and it could not get enough generalization to cover the IPv6 case. These 5 subjects are more like following a protocol type grammar.

The performance on subjects “grep”, “csv” are similar. We already analyzed why Glade is only achieving .02 recall in section 2.3. And we can see that our tool can get 1.0 recall which means all possible strings that can be accepted by the program are covered by the final synthesized grammar of REINAM. Considering we have very close precision (.78 vs. .79), the improvement is huge. The “csv” subject is similar. Since the grammar describes all possible contents in csv file, there are also a lot of special characters which could be data field of the file as the terminal symbols, with a few characters serving both as the separating symbols and content symbols. Therefore REINAM also easily outperforms Glade by improving from .24 recall to 1.0 again. These two subjects are more in a miscellaneous pattern such that many special characters are used as the terminal symbol.

The case for “fol” subject is quite unique. Though somehow improved on recall from .3 to .45, the precision decreases from .90 to .75 and this is the only subject that REINAM cannot achieve a recall over 0.5. Intuitively first order logic formulas are harder to write. For example, to write a simple http address, it only needs to be started with “http://” and having a “.” between the site and domain part. However first order logic formulas are strictly formatted. So the intuition is that the ideal grammar of first order logic formulas is quite small, which means if we consider the whole input space of an FOL parser, it would be much smaller than the input space of a URL parser. Therefore the generalization operator in our tool may not be able to generalize to the complicated structure of the fol grammar.

2.5.4 RQ2.2: Application in Fuzzers

In the fuzz testing application, we can see that REINAM also improves largely from Table 2.2 and Figure 2.6. If we compare the row “Radamsa”, “LI” and “LI+SE+RL”, we can find that on average, our tool improves 18.4% in coverage compared to the Radamsa fuzzer

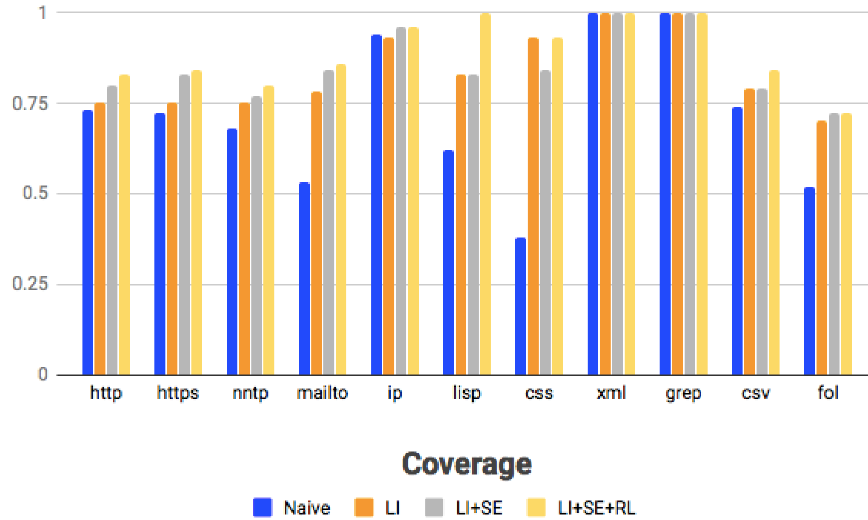


Figure 2.6: Code coverage of different fuzzers in percentage

and 4.9% compared to the Glade's grammar-based fuzzer.

In the two misc pattern subjects "xml" and "grep", all fuzzers achieve perfect code coverage. It may be strange that grammar synthesized by Glade itself ("LI") only achieves .02 recall in Table 2.1 but the fuzzer which uses this grammar can still get perfect code coverage. The reason is that different strings generated from the same grammar can share same or similar execution path that would cover similar basic blocks in code. Therefore the coverage metric is has different meaning from recall.

If we rule out the two subjects with perfect coverage for all the tools, REINAM beats the naive fuzzer in all other subjects. When compared with Glade's grammar-based fuzzer, REINAM only slightly loses at the "css" subject but outperforms in all other subjects. This advantage is due to the benefit we get from allowing overgeneralization. Our tool will produce more rejected inputs compared to glade. These rejected inputs are going to cover the part of source code in which the accepted inputs cannot cover, for example, the additional code used specifically to deal with the illegal cases.

Another thing worth notice is that the Radamsa fuzzer performs badly in the parentheses matching pattern subjects "lisp"and "css" and also in the subject "fol". Especially in "css" subject, it cannot even reach 40% code coverage. The reason is that as we discussed, the ideal grammar for "lisp", "css" and "fol" is more structured than other subjects. The character-level mutation used by a non grammar-based fuzzer cannot synthesizes these structured strings while the generalization operators REINAM uses can generalize to such structures.

2.5.5 RQ2.3: Comparison of Phase 1 and 2

From table 2.1, table 2.2 and figure 2.6, we find that phase 1 and phase 2 contribute differently to the improvements over Glade in terms of precision/recall and fuzzing.

In RQ2.1, if we compare the the row “LI+SE” which shows the result of the grammar synthesized by running Glade on the Pex generated seed input (REINAM phase 1) against the row “LI” which shows the result of synthesized grammar of Glade itself, we can find phase 1 on average improves precision by 9.2% and recall by 29.1%.

In the three parentheses matching pattern subjects “css”, “xml” and “lisp” which Glade already achieves 1.0 recall. Obviously both phase 1 and phase 2 cannot improve the recall. However, phase 1 does improve precision in both “xml” and “lisp” subjects and remains the same precision in “css”. The reason is that Pex generate a set of seed input which has higher coverage. As we described in section 2.2, the first step of Glade directly synthesizes a regular grammar that only captures the seed inputs. Therefore a set of seed input with higher coverage would result in an initial regular grammar with higher coverage, so it needs fewer further generalizations to achieve the ideal grammar. Since imprecision happens during generalization, fewer generalization would result in a higher precision. Similar things can also be observed in other subjects especially when the precision of Glade grammar is not high.

However, for the three subjects “fol”, “ip” and “mailto”. We find that Glade’s synthesized grammar already achieves near or already perfect precision and the recall improves a lot after the phase 1 grammar. In this case, the precision after phase 1 does get decreased. This decrease is a sacrifice for the improvement in recall. The reason is that a seed input set with higher coverage brings more opportunity for generalization and also causes the checking mechanism in Glade to fail more so that some overgeneralization is not rejected, which decreases the precision.

Also in RQ2.1, if we compare the the row “LI+SE” which shows the result of the grammar synthesized by running Glade on the Pex generated seed input (REINAM phase 1) and the row “LI+SE+RL” which shows the result of the final synthesized grammar of REINAM, we can find that, on average, the results after phase 2 improves 20.1% in recall but worsens 9.5% in accuracy.

First thing we can observe is that the precision decreases for all subjects compared to the results of phase 1. This is expected as the reinforcement learning in phase 2 allows overgeneralization to further generalize possible grammar candidates. Compared to the no-overgeneralization-allowed strategy used in Glade, our allowance of overgeneralization would sure decrease the precision.

Table 2.3: Execution time breakdown of REINAM (number in seconds and percentage of total time)

Subject	http	https	nntp	mailto	ip	lisp	css	xml	grep	csv	fol
SE	900 (33.5%)	900 (32.3%)	900 (37.8%)	900 (35.3%)	14 (22.2%)	208 (20.5%)	900 (62.5%)	605 (32.2%)	44 (7.8%)	190 (50.5%)	307 (40.8%)
LI	624 (23.2%)	834 (29.9%)	84 (3.5%)	825 (32.3%)	14 (22.2%)	358 (35.2%)	49 (3.4%)	01 (0.0%)	167 (29.7%)	4 (1.1%)	11 (1.5%)
RL	1160 (43.2%)	1051 (37.7%)	1398 (58.7%)	827 (32.3%)	35 (55.6%)	450 (44.3%)	492 (34.1%)	1273 (67.8%)	351 (62.5%)	182 (48.4%)	435 (57.8%)
Total	2684	2785	2382	2612	63	1016	1441	1878	562	376	753

Then we observe that, in the subjects of url grammar ("http", "https", "nntp" and "mailto"), "grep" and "csv", after phase 1, the recall is still low . The highest is "https" with .64 recall, all others are less than .50. So the reinforcement learning in phase 2 could further generalize the grammar to achieve higher coverage. Table 2.1 shows that, among these six subjects, we observe an average improvement of 35.1% in recall compared to the phase 1 result. And the "grep" and "csv" even achieve 1.0 recall which means the final synthesized grammar of REINAM can perfectly cover the whole input space of the program. This result suggests that our reinforcement learning approach is effective especially in the case when the input space is large but existing approaches can only synthesize a grammar that covers part of the input space.

In the fuzz testing task, we calculate the average improvement of REINAM's synthesized grammar-based fuzzer's coverage (row "LI+SE+RL") against the coverage of grammar-based fuzzer using the phase 1 grammar (row "LI+SE") to be 3.2%. And the average improvement of row "LI+SE" against row "LI" is 1.7%. From the data we can see that phase 2 contributes more to the code coverage improvement. This comparison proves the benefits of keeping some overgeneralized rules in our RL algorithm. We see that for the subjects like "css", phase 2 does not increase the recall rate (recall rate remains 1.0). However the final synthesized grammar improves the coverage by 8.6% (1809 vs. 1642) compared to phase 1. This means that though phase 2 does not improve the overlapping scope of the all the strings generated from the final synthesized grammar and the ones generated by the ideal grammar., the grammar of phase 2 actually is able to generate strings that cover more basic blocks in the code. This further indicates that our reinforcement learning approach is effective not only in exploring the input space of all valid program input, but also in generating invalid program input similar to the valid ones which could cover different execution paths.

2.5.6 RQ2.4: Execution Time

We can see from Table 2.3 that the RL phase is taking in average 49.2% of the total execution time. The Pex is taking in average 34.2% of time in phase 1. The execution time of Pex can be tuned by setting the timeout parameter. The timeout in the evaluation is set

to 900 seconds as we can see actually in 5 subjects, Pex timed out.

Based on the discussion in Section 2.4.3, the execution time of the RL phase is mainly dependent on the size of the grammar and the execution time of the program on the sampling strings. The time overhead of RL phase here is not satisfying. However, the bottleneck of phase 2 is waiting for execution results of the program from the sampling strings. This step can be parallelized to execute the program on k sampling strings simultaneously. Currently we execute the program on 1500 sampling strings on 4 threads. The results can be improved on parallelizing the program execution on more threads.

2.6 SUMMARY

In this chapter, we have presented REINAM, a reinforcement learning approach for synthesizing a probabilistic context-free grammar that encodes the language of valid program inputs. To address the challenge of lacking high-variety, high-quality, and high-quantity seed inputs faced by the existing approaches, REINAM includes an industrial symbolic execution engine (Pex [38]) to generate initial seed inputs for the given target program, and includes a grammar-generalization loop to proactively generate additional inputs during grammar inference. In the grammar-generalization loop, instead of eliminating production rules (in a candidate grammar) that may not be accurate initially (as done by Glade [13], an existing state-of-the-art approach), REINAM keeps and evolves them, thus being able to infer ground-truth grammars whose inference requires *composite* generalizations from the initial seed inputs. To efficiently search for such composite generalizations in a huge search space of candidate generalization operators, REINAM includes a novel formulation of the search problem as the problem of reinforcement learning. Our evaluation results show that REINAM outperforms Glade, an existing state-of-the-art approach, on both precision and recall of the synthesized grammars, and fuzz testing based on REINAM substantially increases the coverage of the valid input space. REINAM is often able to synthesize a grammar covering the whole valid input space without decreasing the precision of the grammar.

CHAPTER 3: GROOT: AN EVENT-GRAPH-BASED APPROACH FOR ROOT CAUSE ANALYSIS

3.1 OVERVIEW

Since the emergence of microservice architecture [49], it has been quickly adopted by many large companies such as Amazon, Google, and Microsoft. Microservice architecture aims to improve the scalability, development agility, and reusability of these companies' business systems. Despite these undeniable benefits, different levels of components in such a system can go wrong due to the fast-evolving and large-scale nature of microservices architecture [49]. Even if there are minimal human-induced faults in code, the system might still be at risk due to anomalies in hardware, configurations, etc. Therefore, it is critical to detect anomalies and then efficiently analyze the root causes of the associated incidents, subsequently helping the system reliability engineering (SRE) team take further actions to bring the system back to normal.

In the process of recovering a system, it is critical to conduct accurate and efficient root cause analysis (RCA) [50], the second one of a three-step process. In the first step, anomalies are detected with alerting mechanisms [51, 52, 53] based on monitoring data such as logs [54, 55, 56, 57, 58], metrics/key performance indicators (KPIs) [59, 60, 61, 62, 63], or a combination thereof [9, 64]. In the second step, when the alerts are triggered, RCA is performed to analyze the root cause of these alerts and additional events, and to propose recovery actions from the associated incident [54, 65, 66]. RCA needs to consider multiple possible interpretations of potential causes for the incident, and these different interpretations could lead to different mitigation actions to be performed. In the last step, the SRE teams perform those mitigation actions and recover the system.

Based on our industrial SRE experiences, we find that RCA is difficult in industrial practice due to three complexities, particularly under microservice settings:

- ***Operational Complexity***. For large-scale systems, there are typically centered (aka infrastructure) SRE and domain (aka embedded) SRE engineers [67]. Their communication is often ineffective or limited under the microservice scenarios due to a more diversified tech stack, granular services, and shorter life cycles than traditional systems. The knowledge gap between the centered SRE team and the domain SRE team gets further enlarged and makes RCA much more challenging. Centered SRE engineers have to learn from domain SRE engineers on how the new domain changes work to update the centralized RCA tools. Thus, adaptive and customizable RCA is required

instead of one-size-fits-all solutions.

- ***Scale Complexity***. There could be thousands of services simultaneously running in a large microservice system, resulting in a very high number of monitoring signals. A real incident could cause numerous alerts to be triggered across services. The inter-dependencies and incident triaging between the services are proportionally more complicated than a traditional system [63]. To detect root causes that may be distributed and many steps away from an initially observed anomalous service, the RCA approach must be scalable and very efficient to digest high volume signals.

- ***Monitoring Complexity***. A high quantity of observability data types (metrics, logs, and activities) need to be monitored, stored, and processed, such as intra-service and inter-service metrics. Different services in a system may produce different types of logs or metrics with different patterns. There are also various kinds of activities, such as code deployment or configuration changes. The RCA tools must be able to consume such highly diversified and unstructured data and make inferences.

To overcome the limited effectiveness of existing approaches [50, 51, 62, 64, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78] (as mentioned in Section 6.2) in industrial settings due to the aforementioned complexities, we propose GROOT, an event-graph-based RCA approach. In particular, GROOT constructs an event causality graph, whose basic nodes are monitoring events such as performance-metric deviation events, status change events, and developer activity events. These events carry detailed information to enable accurate RCA. The events and the causalities between them are constructed using specified rules and heuristics (reflecting domain knowledge). In contrast to the existing fully learning-based approaches [51, 58, 70], GROOT provides better transparency and interpretability. Such interpretability is critical in our industrial settings because a graph-based approach can offer visualized reasoning with causality links to the root cause and details of every event instead of just listing the results. Besides, our approach can enable effective tracking of cases and targeted detailed improvements, e.g., by enhancing the rules and heuristics used to construct the graph.

GROOT has two salient advantages over existing graph-based approaches:

- ***Fine granularity*** (events as basic nodes). First, unlike existing graph-based approaches, which directly use services [72] or hosts (VMs) [77] as basic nodes, GROOT constructs the causality graph by using monitoring events as basic nodes. Graphs based on events from the services can provide more accurate results to address the monitoring complexity. Second, for the scale complexity, GROOT can dynamically create hidden events or additional dependencies based on the context, such as adding

dependencies to the external service providers and their issues. Third, to construct the causality graph, Groot takes the detailed contextual information of each event into consideration for analysis with more depth. Doing so also helps Groot incorporate SRE insights with the context details of each event to address the operational complexity.

- ***High diversity*** (a wide range of event types supported). First, the causality graph in Groot supports various event types such as performance metrics, status logs, and developer activities to address the monitoring complexity. This multi-scenario graph schema can directly boost the RCA coverage and precision. For example, Groot is able to detect a specific configuration change on a service as the root cause instead of performance anomaly symptoms, thus reducing triaging efforts and time-to-recovery (TTR). Second, Groot allows the SRE engineers to introduce different event types that are powered by different detection strategies or from different sources. For the rules that decide causality between events, we design a grammar that allows easy and fast implementations of domain-specific rules, narrowing the knowledge gap of the operational complexity. Third, Groot provides a robust and transparent ranking algorithm that can digest diverse events, improve accuracy, and produce results interpretable by visualization.

To demonstrate the flexibility and effectiveness of Groot, we evaluate it on eBay's production system that serves more than **159** million active users and features more than **5,000** services deployed over three data centers. We conduct experiments on a labeled and validated data set to show that Groot achieves 95% top-3 accuracy and 78% top-1 accuracy for 952 real production incidents collected over 15 months. Furthermore, Groot is deployed in production for real-time RCA, and is used daily by both centered and domain SRE teams, with the achievement of 73% top-1 accuracy in action. Finally, the end-to-end execution time of Groot for each incident in our experiments is less than 5 seconds, demonstrating the high efficiency of Groot.

We report our experiences and lessons learned when using Groot to perform RCA in the industrial e-commerce system. We survey among the SRE users and developers of Groot, who find Groot easy to use and helpful during the triage stage. Meanwhile, the developers also find the Groot design to be desirable to make changes and facilitate new requirements. We also share the lessons learned from adopting Groot in production for SRE in terms of technology transfer and adoption.

In summary, this chapter makes four main contributions:

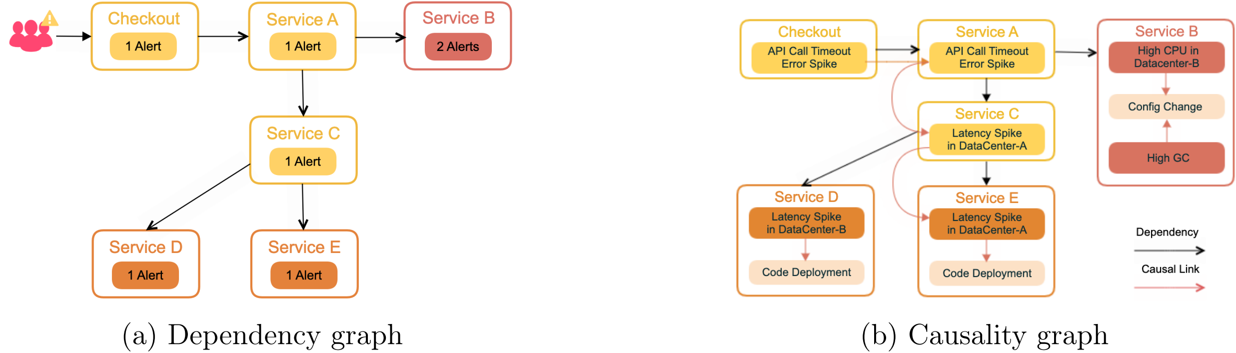


Figure 3.1: Motivating example of event causality graph

- An event-graph-based approach named GROOT for root cause analysis tackling challenges in industrial settings.
- Implementation of GROOT in an RCA framework for allowing the SRE teams to instill domain knowledge.
- Evaluation performed in eBay's production environment with more than 5,000 services, for demonstrating GROOT's effectiveness and efficiency.
- Experiences and lessons learned when deploying and applying GROOT in production.

3.2 MOTIVATING EXAMPLES

In this section, we demonstrate the effectiveness of event-based graph and adaptive customization strategies with two motivating examples.

Figure 3.1 shows an abstracted real incident example with the dependency graph and the corresponding causality graph constructed by GROOT. The *Checkout* service of our e-commerce system suddenly gets an additional latency spike due to a code deployment on the *Service-E*. The service monitor is reporting *API Call Timeout* detected by the ML-based anomaly detection system. The simplified sub-dependency graph consisting of 6 services is shown in Figure 3.1a. The initial alert is triggered on the *Checkout* (entrance) service. The other nodes *Service-** are the internal services that the *Checkout* service directly or indirectly depends on. The color of the nodes in Figure 3.1a indicates the severity/count of anomalies (alerts) reported on each service. We can see that *Service-B* is the most severe one as there are two related alerts on it. The traditional graph-based approach [72, 77] usually takes into account only the graph between services in addition to the severity information on each service. If the traditional approach got applied on Figure 3.1a, either *Service-B*, *Service-D*, or

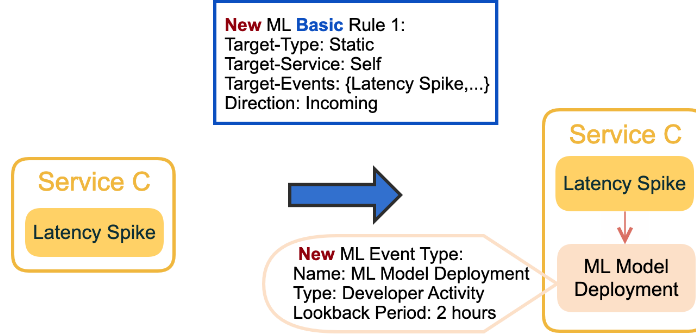


Figure 3.2: Example of event type addition

Service-E could be a potential root cause, and *Service-B* would have the highest possibility since it has two related alerts. Such results are not useful to the SRE teams.

Groot constructs the event-based causality graph as shown in Figure 3.1b. The events in each service are used as the nodes here. We can see that the *API Call Timeout* issue in *Checkout* is possibly caused by *API Call Timeout* in *Service-A*, which is further caused by *Latency Spike* in *DataCenter-A* of *Service-C*. Groot further tracks back to find that it is likely caused by *Latency Spike* in *Service-E*, which happens in the same data center. Finally Groot figures out that the most probable root cause is a recent *Code Deployment* event in *Service-E*. The SRE teams then could quickly locate the root cause and roll back this code deployment, followed by further investigations.

There are no casual links between events in *Service-B* and *Service-A*, since no causal rules are matched. The *API Call Timeout* event is less likely to depend on the event type *High CPU* and *High GC*. Therefore, the inference can eliminate *Service-B* from possible root causes. This elimination shows the benefit of the event-based graph. Note that there is another event *Latency Spike* in *Service-D*, but not connected to *Latency Spike* in *Service-C* in the causality graph. The reason is that the *Latency Spike* event in *Service-C* happens in *DataCenter-A*, not *DataCenter-B*.

Figures 3.2 and 3.3 show how SRE engineers can easily change Groot to adapt to new requirements, by updating the events and rules. In Figure 3.2, SRE engineers want to add a new type of deployment activity, *ML Model Deployment*. Usually, the SRE engineers first need to select the anomaly detection model or set their own alerts and provide alert/activity data sources for the stored events. In this example, the event can be directly fetched from the ML model management system. Then Groot also requires related properties (e.g., the detection time range) to be set for the new event type. Lastly, the SRE engineers add the rules for building the causal links between the new event type and existing ones. The blue box in Figure 3.2 shows the rule, which denotes the edge direction, target event, and target

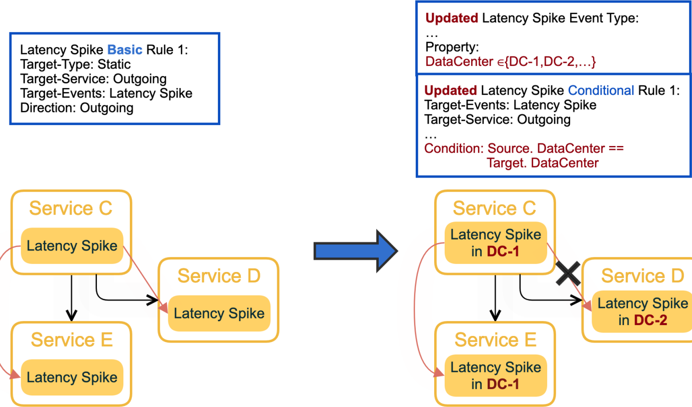


Figure 3.3: Example of event and rule update

service (self, upstream, and downstream dependency).

Figure 3.3 shows a real-world example of how GROOT is able to incorporate SRE insights and knowledge. More specifically, SRE engineers would like to change the rules to allow GROOT to distinguish the latency spikes from different data centers. As an example in Figure 3.1b, *Latency Spike* events propagate only within the same data center. During GROOT development, SRE engineers could easily add new property *DataCenter* to the *Latency Spike* event. Then they add the corresponding "conditional" rules to be differentiated with the "basic" rules in Figure 3.3. In conditional rules, links are constructed only when the specified conditions are satisfied.

3.3 APPROACH

Figure 3.4 shows the overall workflow of GROOT. The triggers for using GROOT are usually alert(s) from automated anomaly detection, or sometimes an SRE engineer's suspicion. There are three major steps: constructing the service dependency graph, constructing the event causality graph, and root cause ranking. The outputs are the root causes ranked by the likelihood. To support fast human investigation experience, we build an interactive UI as shown in Figure 3.8: the service dependency, events with causal links and additional details such as raw metrics or the developer contact (of a code deployment event) are presented to the user for next steps. As an offline part of human investigation, we label/collect a data set, perform validation, and summarize the knowledge for further improvement on all incidents on a daily basis.

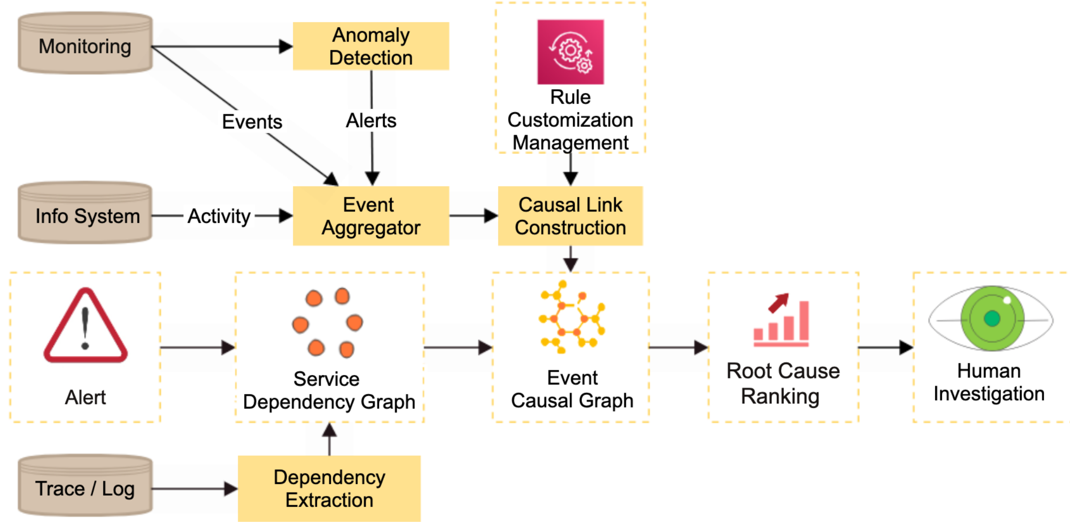


Figure 3.4: Workflow of GROOT

3.3.1 Constructing Service Dependency Graph

The construction of the service dependency graph starts with the initial alerted or suspicious service(s), denoted as I. For example, in Figure 3.1a, $I = \{Checkout\}$. I can contain multiple services based on the range of the trigger alerts or suspicions. We maintain domain service lists where domain-level alerts can be triggered because there is no clear service-level indication.

At the back end, GROOT maintains a global service dependency graph G_{global} via distributed tracing and log analysis. The directed edge from nodes A to B (two services or system components) in the dependency graph indicates a service invocation or other forms of dependency. In Figure 3.1a, the black arrows indicate such edges. Bi-directional edges and cycles between the services can be possible and exist. In this work, the global dependency graph is updated daily.

The service dependency (sub)graph G is constructed using G_{global} and I. An extended service list L is first constructed by traversing each service in I over G_{global} for a radius range r. Each service $u \in L$ can be traversed by at least one service $v \in I$ within r steps: $L = \{u | \exists v \in I,\ dist(u, v) \leq r\ or\ dist(v, u) \leq r\}$. Then, the service dependency subgraph G is constructed by the nodes in L and the edges between them in G_{global}. In our current implementation, r is set to 2, since this dependency graph may be dynamically extended in the next steps based on events' detail for longer issue chains or additional dependencies.

3.3.2 Constructing Event Causality Graph

In the second step, GROOT collects all supported events for each service in G and constructs the causal links between events.

Collecting Events Table 3.1 presents some example event types and detection techniques for GROOT's production implementation. For detection techniques, "De Facto" indicates that the event can be directly collected via a specific API or storage. The detection either runs passively in the back end to reduce delay and improve accuracy, or runs actively for only the services within the dependency graph range to save resources.

There are three major categories of events: performance metrics, status logs, and developer activities:

- *Performance metrics* represent an anomaly of monitored time series metrics. For example, high CPU usage indicates that the service is causing high CPU usage on a certain machine. In this category, most events are continuously and passively detected and stored.

- *Status logs* are caused by abnormal system status, such as spike of HTTP error code metrics while accessing other services' endpoints. Different types of error metrics are important and supported in GROOT, including third-party APIs. For example, Bad Host indicates abnormal patterns on some machines running the service, and can be detected by a clustering-based ML approach.

- *Developer activities* are the events generated when a certain activity of developers is triggered, such as code deployment and config change.

In Groot, there are more than a dozen event types such as *Latency Spike* as listed in the column 2 of Table 3.1. Each event type is characterized by three aspects: $Name$ indicates the name of this event type; *LookbackPeriod* indicates the time range to look back (from the time when the use of GROOT is triggered) for collecting events of this event type; *PropertyType* indicates the types of the properties that an event of this event type should hold. *PropertType* is characterized by a vector of pairs, each of which indicates the string type for a property's name and the primitive type for the property's value such as string, integer, and float. Formally, an event type is defined as a tuple: $ET =< Name, LookbackPeriod, PropertyType >$ where $PropertyType =< (string, type_1), ...,$ $(string, type_n) >$ (n is the number of properties that an event of this event type holds).

Each event of a certain event type *ET* is characterized by four aspects: *Service* indicates the service name that the event belongs to; *Type* indicates *ET*'s *Name*; *StartTime* indicates

Table 3.1: List of example event types used in GROOT

Type	Event Type	Detection Technique
Performance Metrics	High GC (Overhead)	Rule-based
	High CPU Usage	Rule-based
	Latency Spike	Statistical Model
	TPS Spike	Statistical Model
	Database Anomaly	ML Model
	Business Metric Anomaly	ML Model
Status Logs	WebAPI Error	Statistical Model
	Internal Error	Statistical Model
	ServiceClient Error	Statistical Model
	Bad Host	ML Model
Developer Activities	Code Deployment	De Facto
	Configuration Change	De Facto
	Execute URL	De Facto

the time when the event happens; *Properties* indicates the properties that the event holds. Formally, an event is defined as a tuple: $e =< Service, Type, StartTime, Properties >$ where *Properties* is an instantiation of *ET*'s *PropertyType*.

For example, in Figure 3.1, the generated event for *Latency Spike in DataCenter-A* in *Service-C* would be $<$ "*Service-C*", "*Latency Spike*", *2021/08/01-12:36:04*, $<$ ("*DataCenter*", "*DC-1*"), ... $>>$.

Constructing Causal Link After collecting all events on all services in G, in this step, causal links between these events are constructed for RCA ranking. The causal links (red arrows) in Figure 3.1b are such examples. A causal link represents that the source event can possibly be caused by the target event. SRE knowledge is engineered into rules and used to create causal links between the pairs of events.

A rule for constructing a causal link is defined as a tuple: $Rule =< Target\text{-}Type, Source\text{-}Events, Target\text{-}Events, Direction, Target\text{-}Service, Condition >$ (*Condition* can be optionally specified). *Target-Type* indicates the type of the rule, being either *Static* or *Dynamic* (explained further later). *Source-Events* indicates the type of the causal link's source event (*Source-Events* are listed in the names of the rules shown in Figures 3.2, 3.3 and 3.5). *Target-Events* indicates the type of the causal link's target event. *Direction* indicates the direction of the casual link between the target event and source event. *Target-Service* indicates the service that the target event should belong to. Note that *Target-Service* in *Static* rules can be $Self$, which indicates that the target event would be within the same service as the source event, or $Outgoing/Incoming$, which indicates that the target event would belong to the

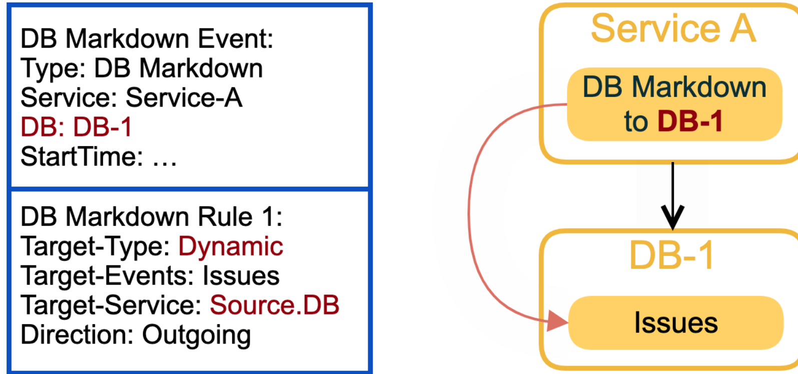


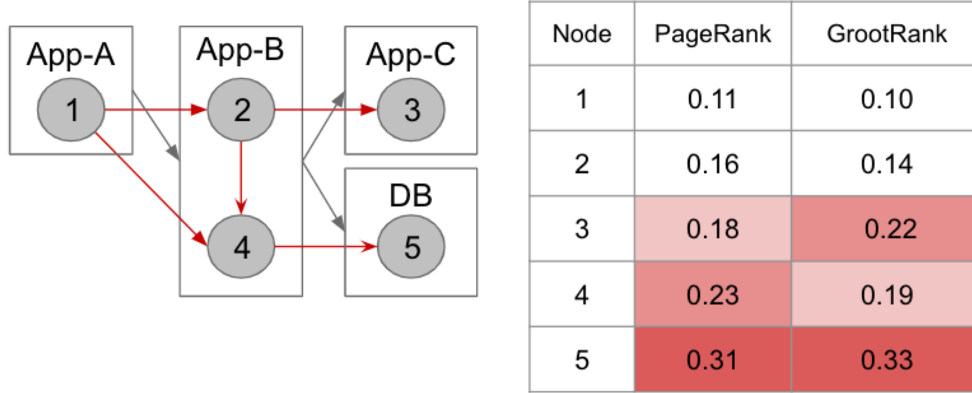
Figure 3.5: Example of dynamic rule

downstream/upstream services of the service that the source event belongs to in G.

There are two categories of special rules. The first category is *dynamic* rules (i.e., rules whose *Target-Type* is set to *Dynamic*) to support dynamic dependencies. Here *Target-Service* does not indicate any of the three possible options listed earlier but indicates the name of the target service that Groot would need to create. For example, live DB dependencies are not available due to different tech stacks and high volume. In Figure 3.5, a DB issue (DB Markdown) is shown in *Service-A*. Based on the listed *dynamic* rule, Groot creates a new "service" *DB-1* in G, a new event "Issues" that belongs to *DB-1*, and a causal link between the two events. In practice, the SRE teams use dynamic rules to cover a lot of third-party services and database issues since the live dependencies are not easy to maintain.

The second category of special rules is *conditional* rules. *Conditional* rules are used when some prerequisite conditions should be satisfied before a certain causal link is created. In these rules, *Condition* is specified with a boolean predicate. As shown in Figure 3.3, the SRE teams believe *Latency Spike* events from different services are related only when both events happen within the same data center. Based on this observation, Groot would first evaluate the predicate in *Condition* and build only the causal link when the predicate is true. A conditional rule overwrites the basic rule on the same source-target event pair.

When constructing causal links, Groot first applies the *dynamic* rules so that dynamic dependencies and events are first created at once. Then for every event in the initial services (denoted as I), if the rule conditions are satisfied, one or many causal links are created from this event to other events from the same or upstream/downstream services. When a causal link is created, the step is repeated recursively for the target event (as a new origin) to create new causal links. After no new causal links are created, the construction of the event causality graph is finished.



Node	PageRank	GrootRank
1	0.11	0.10
2	0.16	0.14
3	0.18	0.22
4	0.23	0.19
5	0.31	0.33

Figure 3.6: Example of personalization vector customization

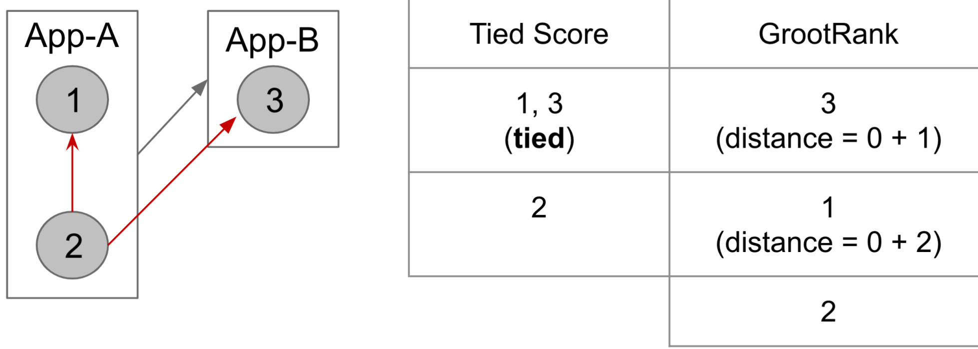
3.3.3 Root Cause Ranking

Finally, GROOT ranks and recommends the most probable root causes from the event causality graph. Similar to how search engines infer the importance of pages by page links, we customize the PageRank [79] algorithm to calculate the root cause ranking; the customized algorithm is named as GrootRank. The input is the event causality graph from the previous step. Each edge is associated with a weighted score for weighted propagation. The default value is set as 1, and is set lower for alerts with high false-positive rates.

Based on the observation that dangling nodes are more likely to be the root cause, we customize the personalization vector as $P_n = f_n$ or $P_d = 1$, where P_d is the personalization score for dangling nodes, and P_n is for the remaining nodes; and f_n is a value smaller than 1 to enhance the propagation between dangling nodes. In our work, the parameter setting is $f_n = 0.5$, $\alpha = 0.85$, $max_{iter} = 100$ (which are parameters for the PageRank algorithm). Figure 3.6 illustrates an example. The grey circles are the events collected from three services and one database. The grey arrows are the dependency links and the red ones are the causal links with the weight of 1. Both of the PageRank and GrootRank algorithms detect *event*5 (DB issue) as the root cause, which is expected and correct. However, the PageRank algorithm ranks *event*4 higher than *event*3. But *event*3 of *Service-C* is more likely to be the second most possible root cause (besides *event*5), because the scores on dangling nodes are propagated to all others equally in each iteration. We can see that *event*3 is correctly ranked as second using the GrootRank algorithm.

The second step of GrootRank is to break the tied results from the previous step. The tied results are due to the fact that the event graph can contain multiple disconnected sub-graphs with the same shape. We design two techniques to untie the ranking:

1. For each joint event, the access distance (sum) is calculated from the initial anomaly



Tied Score	GrootRank
1, 3 (**tied**)	3 (distance = 0 + 1)
2	1 (distance = 0 + 2)
	2

Figure 3.7: Example of using access distance to untie the ranking results

service(s) to the service where the event belongs to. If any "access" is not reachable, the distance is set as $d_m + 1$ where d_m is the maximum possible distance. The one with shorter access distance (sum) would be ranked higher and vice versa. Figure 3.7 presents an example, where *Service-A* and *Service-B* are both initial anomaly services. Since GROOT suspects that *event*2 is caused by either *event*3 or *event*1 with the same weight. The scores of *event*3 and *event*1 are tied. Then, *event*3 has a score of 1 (i.e., $0 + 1$) and *event*1 has a score of 2 (i.e., $0 + 2$), since it is not reachable by *Service-B*). Therefore, *event*3 is ranked first and logical.

2. For the remaining joint results with the same access distances, GROOT continues to untie by using the historical root cause frequency of the event types under the same trigger conditions (e.g., checkout domain alerts). This frequency information is generated from the manually labeled dataset. A more frequently occurred root cause type is ranked higher.

3.3.4 Rule Customization Management

While GROOT users create or update the rules, there could be overlaps, inconsistencies, or even conflicts being introduced such as the example in Figure 3.3. GROOT uses two graphs to manage the rule relationships and avoid conflicts for users. One graph is to represent the link rules between events in the same service (*Same-Graph*) while the other is to represent links between different services (*Diff-Graph*). The nodes in these two graphs are the event types defined in Section 3.3.2. There are three statuses between each (directional) pair of event types: (1) no rule, (2) only basic rule, and (3) conditional rule (since it overwrites the basic rule). In *Same-Graph*, GROOT does not allow self-loop as it does not build links between an event and itself.

When rule change happens, existing rules are enumerated to build edges in *Same-Graph* and *Diff-Graph* based on *Target-Events* and *Target-Service*. Based on the users' operation of (1) "remove a rule", GROOT removes the corresponding edge on the graphs; (2) "add/update a rule", GROOT checks whether there are existing edges between the given event types, and then warns the users for possible overwrites. If there are no conflicts, GROOT just adds/updates edges between the event types.

After all changes, GROOT extracts the rules from the graphs by converting each edge to a single rule. These rules are automatically implemented, and then tested against our labeled data set. The GROOT users need to review the changes with validation reports before the changes go online.

3.4 EVALUATION

We evaluate GROOT in two aspects: (1) *effectiveness (accuracy)*, which assesses how accurate GROOT is in detecting and ranking root causes, and (2) *efficiency*, which assesses how long it takes for GROOT to derive root causes and conduct end-to-end analysis in action. Particularly, we intend to address the following research questions:

- **RQ3.1.** What are the accuracy and efficiency of GROOT when applied on the collected dataset?
- **RQ3.2.** How does GROOT compare with baseline approaches in terms of accuracy?
- **RQ3.3.** What are the accuracy and efficiency of GROOT in an end-to-end scenario?

3.4.1 Evaluation Setup

To evaluate GROOT in a real-world scenario, we deploy and apply GROOT in eBay's e-commerce system that serves more than 159 million active buyers. In particular, we apply GROOT upon a microservice ecosystem that contains over 5,000 services on three data centers. These services are built on different tech stacks with different programming languages, including Java, Python, Node.js, etc. Furthermore, these services interact with each other by using different types of service protocols, including HTTP, gRPC, and Message Queue. The distributed tracing of the ecosystem generates 147B traces on average per day.

Data Set The SRE teams at eBay help collect a labeled data set containing 952 incidents over 15 months (Jan 2020 - Apr 2021). Each incident data contains the input required by

Groot (e.g., dependency snapshot and events with details) and the root cause manually labeled by the SRE teams. These incidents are grouped into two categories:

- *Business domain incidents.* These incidents are detected mainly due to their business impact. For example, end users encounter failed interactions, and business or customer experience is impacted, similar to the example in Figure 3.1.
- *Service-based incidents.* These incidents are detected mainly due to their impact on the service level, similar to the example in Figure 3.5.

An internal incident may get detected early, and then likely get categorized as a service-based incident or even solved directly by owners without records. On the other hand, infrastructure-level issues or issues of external service providers (e.g., checkout and shipping services) may not get detected until business impact is caused.

There are 782 business domain incidents and 170 service-based incidents in the data set. For each incident, the root cause is manually labeled, validated, and collected by the SRE teams, who handle the site incidents everyday. For a case with multiple interacting causes, only the most actionable/influential event is labelled as the root cause for the case. These actual root causes and incident contexts serve as the ground truth in our evaluation.

Groot Setup The Groot production system is deployed as three microservices and federated in three data centers with nine 8-core CPUs, 20GB RAM pods each on Kubernetes.

Baseline Approaches In order to compare Groot with other related approaches, we design and implement two baseline approaches for the evaluation:

- *Naive Approach.* This approach directly uses the constructed service dependency graph (Section 3.3.1). The events are assigned a score by the severeness of the associated anomaly. Then a normalized score for each service is calculated summarizing all the events related to the service. Lastly, the PageRank algorithm is used to calculate the root cause ranking.
- *Non-adaptive Approach.* This approach is not context-aware. It replaces all special rules (i.e., conditional and dynamic ones) with their basic rule versions. Its other parts are identical to Groot.

The non-adaptive approach can be seen as a baseline for reflecting a group of graph-based approaches (e.g., CauseInfer [69] and Microscope [80]). These approaches also specify certain service-level metrics but lack the context-aware capabilities of Groot. Because the tools

Table 3.2: Accuracy of RCA by Groot and baselines

	Groot		Naive		Non-adaptive	
	Top 3	Top 1	Top 3	Top 1	Top 3	Top 1
Service-based	92%	74%	25%	16%	84%	62%
Business domain	96%	81%	2%	1%	28%	26%
Combined	95%	78%	6%	3%	38%	33%

for these approaches are not publicly available, we implement the non-adaptive approach to approximate these approaches.

3.4.2 Evaluation Results

RQ3.1 Table 3.2 shows the results of applying Groot on the collected data set. We measure both top-1 and top-3 accuracy. The top-1 and top-3 accuracy is calculated as the percentage of cases where their ground-truth root cause is ranked within top 1 and top 3, respectively, in Groot's results. Groot achieves high accuracy on both incident categories. For example, for business domain incidents, Groot achieves 96% top-3 accuracy.

The unsuccessful cases that Groot ranks the root cause after top 3 are mostly caused by missing event(s). More than one-third of these unsuccessful cases have been addressed by adding necessary events and corresponding rules over time. For example, initially, we had only an event type of general error spike, which mixes different categories of errors and thus causes high false-positive rate. We then have designed different event types for each category of the error metrics (including various internal and client API errors). In many cases that Groot ranks the root cause after top 1, the labeled root cause is just one of the multiple co-existing root causes. But for fairness, the SRE teams label only a single root cause in each case. According to the feedback from the SRE teams, Groot still facilitates the RCA process for these cases.

Our results show that the runtime cost of applying Groot is relatively low. For a service-based incident, the average runtime cost of Groot is 1.06s while the maximum is 1.69s. For a business domain incident, the average runtime cost is 0.98s while the maximum is 1.14s.

RQ3.2 We additionally apply the baseline approaches on the data set. Table 3.2 also shows the evaluation results. The results show that the accuracy of Groot is substantially higher than that of the baseline approaches. In terms of the top-1 accuracy, Groot achieves 78% compared with 3% and 33% of the naive and non-adaptive approaches, respectively. In terms of the top-3 accuracy, Groot achieves 95% compared with 6% and 38% of the naive

Table 3.3: Comparison of Groot results on the dataset and end-to-end scenario

	Service-based		Business Domain	
	Dataset	End-to-End	Dataset	End-to-End
Top-1 Accuracy	74%	73%	81%	73%
Top-3 Accuracy	92%	91%	96%	87%
Average Runtime Cost	1.06s	3.16s	0.98s	2.98s
Maximum Runtime Cost	1.69s	4.56s	1.14s	3.61s

and non-adaptive approaches, respectively.

The naive approach performs worst in all settings, because it blindly propagates the score at service levels. The accuracy of the non-adaptive approach is much worse for business domain incidents. The reason is that for a business domain incident, it often takes a longer propagation path since the incident is triggered by a group of services, and new dynamic dependencies may be introduced during the event collection, causing more inaccuracy for the non-adaptive approach. There can be many non-critical or irrelevant error events in an actual production scenario, aka "soft" errors. We suspect that these non-critical or irrelevant events may be ranked higher by the non-adaptive approach since they are similar to injected faults and hard to be distinguished from the actual ones. Groot uses dynamic and conditional rules to discover the actual causal links, building fewer links related to such non-critical or irrelevant events for leading to higher accuracy.

RQ3.3 To evaluate Groot under an end-to-end scenario, we apply Groot upon actual incidents in action. Table 3.3 shows the results. The accuracy has a decrease of up to 9 percentage points in the end-to-end scenario, with some failures caused by production issues such as missing data and service/storage failures. In addition, the runtime cost is increased by up to nearly 3 seconds due to the time spent on fetching data from different data sources, e.g., querying the events for a certain time period.

3.5 EXPERIENCE

Groot currently supports daily SRE work. Figure 3.8 shows a live Groot's "bird's eye view" UI on an actual simple checkout incident. Service C has the root cause ($ErrorSpike$) and belongs to an external provider. Although the domain service A also carries an error spike and gets impacted, Groot correctly ignores the irrelevant deployment event, which has no critical impact. The events on C are virtually created based on the dynamic rule. Note that all causal links (yellow) in the UI indicate "is cause of", being the opposite of "is caused by" as described in Section 3.3.2 to provide more intuitive UI for users to navigate through.

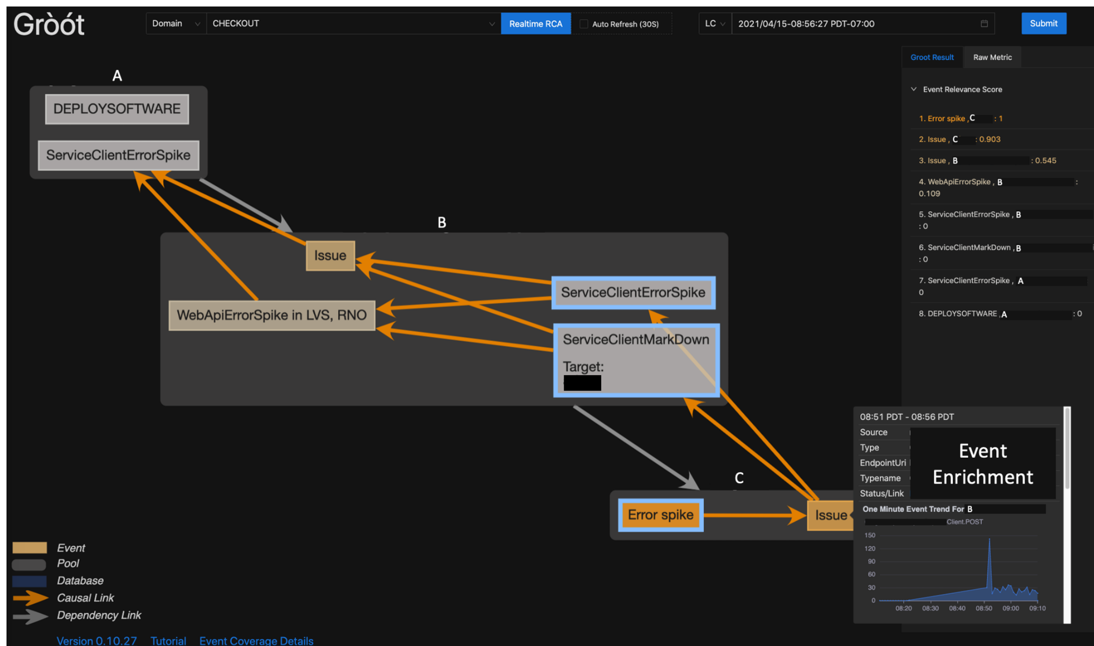


Figure 3.8: GROOT UI in production

GROOT visualizes the dependency and event causality graph with extra information such as an error message. The SRE teams can quickly comprehend the incident context and derived root cause to investigate GROOT further. A mouseover can trigger "event enrichment" based on the event type to present details such as raw metrics and other additional information.

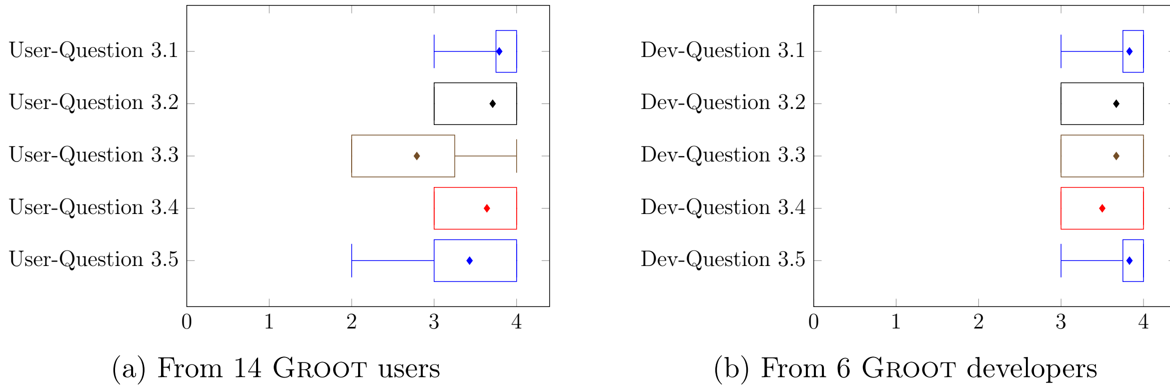
We next share two major kinds of experience:

- **Feedback from GROOT users and developers**, reflecting the general experience of two groups: (1) domain SRE teams who use GROOT to find the root cause, and (2) a centered SRE team who maintains GROOT to facilitate new requirements.
- **Lessons learned**, representing the lessons learned from deploying and adopting GROOT in production for the real-world RCA process.

3.5.1 Feedback from GROOT Users and Developers

We invite the SRE members who use GROOT for RCA in their daily work to the user survey. We call them users in this section. We also invite different SRE members responsible for maintaining GROOT to the developer survey. We call them developers in this section. In total, there are 14 users and 6 developers who respond to the surveys.

For the user survey, we ask 14 users the following 5 questions (User-Question 3.4-3.5 have the same choices as User-Question 3.1):



(a) From 14 GROOT users

(b) From 6 GROOT developers

Figure 3.9: Survey results

- **User-Question 3.1.** When GROOT correctly locates the root cause, how does it help with your triaging experience? Answer choices: Helpful(4), Somewhat Helpful(3), Not Helpful(2), Misleading(1).
- **User-Question 3.2.** When GROOT correctly locates the root cause, how does it save/extend your or the team's triaging time? (Detection and remediation time not included) Answer choices: Lots Of Time Saved(4), Some Time Saved(3), No Time Saved(2), Waste Time Instead(1).
- **User-Question 3.2.** Based on your estimation, how much triage time GROOT would save on average when it correctly locates the root cause? (Detection and remediation time not included) Answer choices: More than 50%(4), 25-50%(3), 10-25%(2), 0-10%(1), N/A(0).
- **User-Question 3.4.** When GROOT correctly locates the root cause, do you find that the result "graph" provided by GROOT helps you understand how and why the incident happens?
- **User-Question 3.5.** When GROOT does not correctly locate the root cause, does the result "graph" make it easier for your investigation of the root cause?

Figure 3.9a shows the results of the user survey. We can see that most users find GROOT very useful to locate the root cause. The average score for User-Question 3.1 is 3.79, and 11 out of 14 participants find GROOT very helpful. As for User-Question 3.3, GROOT saves the triage time by 25-50%. Even in cases that GROOT cannot correctly locate the root cause, it is still helpful to provide information for further investigation with an average score of 3.43 in User-Question 3.5.

For the developer survey, we ask the 6 developers the following 5 questions (Dev-Question 3.2-3.5 have the same choices as Dev-Question 3.1):

- **Dev-Question 3.1.** Overall, how convenient is it to change and customize events/rules/-domains while using GROOT? Answer choices: Convenient(4), Somewhat Convenient(3), Not Convenient(2), Difficult(1).
- **Dev-Question 3.2.** How convenient is it to *change/customize event models* while using GROOT?
- **Dev-Question 3.3.** How convenient is it to *add new domains* while using GROOT?
- **Dev-Question 3.4.** How convenient is it to *change/customize causality rules* while using GROOT?
- **Dev-Question 3.5.** How convenient is it to change/customize GROOT compared to other SRE tools?

Figure 3.9b shows the results of the developer survey. Overall, most developers find it convenient to make changes on and customize events/rules/domains in GROOT.

3.5.2 Lessons learned

In this section, we share the lessons learned in terms of technology transfer and adoption on using GROOT in production environments.

Embedded in Practice. To build a successful RCA tool in practice, it is important to embed the R&D efforts in the live environment with SRE experts and users. We have a 30-minute routine meeting daily with an SRE team to manually test and review every site incident. In addition, we actively reach out to the end users for feedback. For example, the users found our initial UI hard to understand. Based on their suggestions, we have introduced alert enrichment with the detailed context of most events, raw metrics, and links to other tools for the next steps. We also make the UI interactive and build user guides, training videos, and sections. As a result, GROOT has become increasingly practical and well adopted in practice. We believe that R&D work on observability should be incubated and grown within daily SRE environments to bring developers with rich RCA experience into the R&D team.

Vertical Enhancements. High-confidence and automated vertical enhancements can empower great experiences. GROOT is enhanced and specialized in critical scenarios such as grouped related alerts across services or critical business domain issues, and large-scale scenarios such as infrastructure changes or database issues. Furthermore, the end-to-end

automation is also built for integration and efficiency with anomaly detection, RCA, and notification. For notification, domain business anomalies and diagnostic results are sent through communication apps (e.g., slack and email) for better reachability and experience. Groot now supports 18 business domains and sub-domains of the company. On average, Groot UI supports more than 50 active internal users, and the service sends thousands of results every month. Most of these usages are around the vertical enhancements.

Data and Tool Reliability. Reliability is critical to Groot itself and requires a lot of attention and effort. For example, if a critical event is missing, Groot may infer a totally different root cause, which would mislead users. We estimate the alert accuracy to be greater than 0.6 in order to be useful. Recall is even more important since Groot can effectively eliminate false positive alerts based on the casual ranking. Since there are hundreds of different metrics supported in Groot, we spend time to ensure a robust back end by adding partial and dynamic retry logic and high-efficiency cache. Groot's unsuccessful cases can be caused by imperfect data, flawed algorithms, or simply code defects. To better trace the reason behind each unsuccessful case, we add a tracing component. Every Groot request can be traced back to atomic actions such as retrieving data, data cleaning, and anomaly detection via algorithms.

Trade-off among Models. The accuracy and scalability trade-off among anomaly detection models should be carefully considered and tested. In general, some algorithms such as deep-learning-based or ensemble models are more adaptive and accurate than typical ones such as traditional ML or statistical models. However, the former requires more computation resources, operational efforts, and additional system complexities such as training or model fine-tuning. Due to the actual complexities and fast-evolving nature of our context, it is not possible to scale each model (e.g., deep-learning-based models), nor have it deeply customized for every metric at every level. Therefore, while selecting models, we must make careful trade-off in aspects such as accuracy, scalability, efficiency, effort, and robustness. In general, we first set different "acceptance" levels by analyzing each event's impact and frequency, and then test different models in staging and pick the one that is good enough. For example, a few alerts such as "high thread usage" are defined by thresholds and work just fine even without a model. Some alerts such as "service client error" are more stochastic and require coverage on every metric of every service, and thus we select fast and robust statistical models and actively conduct detection on the fly.

Phased Incorporation of ML. In the current industrial settings, ML-powered RCA products still require effective knowledge engineering. Due to the higher complexity and lower "signal to noise ratio" of real production incidents, many existing approaches cannot be applied in practice. We believe that the knowledge engineering capabilities can facilitate adoption of

technologies such as AIOps. Therefore, Groot is designed to be highly customizable and easy to infuse SRE knowledge and to achieve high effectiveness and efficiency. Moreover, a multi-scenario RCA tool requires various and interpretable events from different detection strategies. Auto-ML-based anomaly detection or unsupervised RCA for large service ecosystems is not yet ready in such context. As for the path of supervised learning, the training data is tricky to label and vulnerable to potential cognitive bias. Lastly, the end users often require complete understanding to fully adopt new solutions, because there is no guarantee of correctness. Many recent ML algorithms (e.g., ensemble and deep learning) lack interpretability. Via the knowledge engineering and graph capabilities, Groot is able to explain diversity and causality between ML-model-driven and other types of events. Moving forward, we are building a white-box deep learning approach with causal graph algorithms where the causal link weights are parameters and derivable.

3.6 DISCUSSION

In this section, we discuss some future extension or improvements on the system.

Many related works for anomaly detection are discussed in Section 6.2. Some approaches are using adaptive concept drifting, which can self-adapt to the target event without manual tuning [81]. One major burden in our event construction is that we need to build different detection strategies/algorithms for different events, while some events are still needed to be manually tuned or detected by rules. Auto-ML is yet to come for anomaly detection, and the interpretability of the result is also a big challenge for end-to-end usability. The filter range for some events also needs manual tuning which requires large amount of domain expertise and is not robust. As of now, we are investing self-adaptive approaches and building time series metrics anomaly detection platform to reduce the event on-boarding cost.

Also, our approach, which requires a fair amount of “one-time" effort, utilizes rules to build links between events. A lot of the rules have similar patterns. Despite that our users prefer to have their own control, rules can be inferred from historical patterns. A great extension to this work is the use of machine learning or grammar inference techniques [82] to automatically or semi-automatically infer the rules.

3.7 SUMMARY

In this chapter, we have presented our work around root cause analysis (RCA) in industrial settings. To tackle three major RCA challenges (complexities of operation, system scale,

and monitoring), we have proposed a novel event-graph-based approach named Groot that constructs a real-time causality graph for allowing adaptive customization. Groot can handle diversified anomalies and activities from the system under analysis and is extensible to different approaches of anomaly detection or RCA. We have integrated Groot into eBay's large-scale distributed system containing more than **5,000** microservices. Our evaluation of Groot on a data set consisting of 952 real production incidents shows that Groot achieves high accuracy and efficiency across different scenarios and also largely outperforms baseline graph-based approaches. We also share the lessons learned from deploying and adopting Groot in production environments.

CHAPTER 4: NL2VIZ: NATURAL LANGUAGE TO VISUALIZATION VIA CONSTRAINED SYNTAX-GUIDED SYNTHESIS

4.1 OVERVIEW

Recent development in Natural Language to Code (NL2Code) research allows the end-user, especially novice programmers to create a concrete implementation of their ideas by providing natural language (NL) instructions. While code generation for general-purpose languages such as Python is still challenging [83], NL2Code for a domain-specific language (DSL) such as SQL [84, 85, 86] or NL2Code in a specific application domain such as competitive programming [87] has witnessed major advances. Given that data science has seen tremendous growth in recent years, data visualization has become a great application domain of NL2Code. The main reason is that data scientists need to frequently produce visualization to help them perform exploratory data analysis (EDA) to discover useful information from data and draw insights to support decision making. Yet it is quite a burden on data scientists as they have to memorize names of data visualization APIs and their many parameters [88]. Indeed, in our user study (Section 4.4.4), data scientists confirm that they could not memorize all the API options and have to look into API documentation frequently.

It is difficult for an NL2Code tool to achieve its goal due to three major challenges. First, in the stated NL instruction, the user may use words whose semantics can be determined only in the context. For example, different NL2Sql approaches include different strategies to handle schema encoding to create a mapping from the user input to the entities in the database, while few approaches have achieved good adaptability [86]. In visualization, an NL2Code task gets more complicated as the user may already write some code to process the data and then use the Natural Language to Visualization (NL2Visualization) tool. So the tool also needs to understand the program context. Second, the user may not include all details (needed for code generation) in the NL instruction. For example, when trying to produce a line plot, the user may not specify the name of the data column used for the x-axis especially when there is a data column with index or time value. Third, the results of NL2Code are often imperfect and thus require the user's further interaction or update to fix issues in the results. Especially in data analysis, data scientists often need to make quick changes to visualization as data analysis is a data-driven process. The first two challenges on the user input may make the user's further interaction even more necessary.

To address the aforementioned three challenges, in this chapter, we propose a new approach and its supporting tool named NL2Viz in the domain of NL2Visualization with three salient features. First, we leverage not only the user's NL input but also the contextual

input, i.e., data and program context that the underlying query is upon. The data context includes the data tables that the user is working on and also the intermediate data vectors that the user has produced. The program context includes the previous code and existing plots that the user has produced (including the previous instruction-plot pairs produced by our NL2Viz tool for the user). For example, when the user creates her own filtering function and refers to the function name in her instruction, we would be able to understand and use the function in the generated plotting code by leveraging the program context. Second, to better fill the missing or ambiguous details in the NL input, we differentiate between hard constraints and soft constraints retrieved from the NL input and contextual input. The hard constraints are the ones that we have high confidence in and could be explicitly specified by the user. For example, the user states that she wants a scatterplot, and then the plot type to be scatterplot would be a hard constraint. Meanwhile the soft constraints are the ones that we do not have high confidence in and could be inferred from the context. For example, the user does not mention the data column for the x-axis but the column with time information would be the likely x-axis data. Third, we provide the user interface to allow iterative refinement for the user to further fix or change the results. We allow the user to give additional NL instructions to make changes to the visualization. Moreover, we are not only having the plot as the output but also the working code snippet that produces the plot. The user can also directly make changes on the code snippet; data scientists find making code changes convenient as shown in our user study (Section 4.4.4).

We implement our approach as a tool named NL2Viz in the Jupyter Notebook environment [89]. NL2Viz is directly embedded into the user's daily workflow without the burden to switch between different environments. At a high level, NL2Viz first parses the NL instruction (given by the user) using semantic parsing [90, 91] into *symbolic constraints* that the target visualization program needs to satisfy. NL2Viz also generates such constraints after retrieving the data, program, and existing plot context in the current notebook. Next, NL2Viz uses a novel syntax-guided program synthesis algorithm to generate a complete visualization program from these hard/soft constraints. During this process, NL2Viz keeps multiple candidates at each synthesis state and assigns a heuristic fitness score to help prioritize the most likely structure. Finally, NL2Viz can take a further refinement NL instruction to change the generated visualization program or the user can choose to directly use or apply changes to the generated program.

We assess NL2Viz using four evaluations. First, we assess the synthesis accuracy in a one-shot scenario. Our benchmark contains 295 NL instructions collected from data scientists and online homework assignments. Overall, NL2Viz is able to achieve an overall accuracy of 74.6%. Second, we assess NL2Viz's accuracy in interactive scenarios. Given an initial

plot and an instruction for describing a small change, NL2Viz achieves 62.5% accuracy in 40 scenarios. Third, we also assess NL2Viz in a public dataset [92]. NL2Viz outperforms a state-of-the-art approach [92] in easy to medium categories while achieving comparable overall accuracy of 55.0%. Fourth, we assess the usability of NL2Viz via a user study, where we ask 6 data scientist professionals to use NL2Viz to complete 5 visualization tasks. The participants are able to successfully complete 4.17 out of the 5 tasks on average. Most participants like NL2Viz and are willing to use it before writing actual visualization code.

This chapter makes the following main contributions:

- We propose a novel NL2Code approach that aims to address challenges on the user input and interactions in the application domain of NL2Visualization by leveraging the data/program context, retrieving hard/soft constraints, and providing interactive refinement support.
- We present NL2Viz, an end-to-end synthesis tool implemented in the Jupyter Notebook environment for helping data scientists visualize their data using an NL interface. NL2Viz shows both the plot and the readable code snippet for generating that plot, allowing the user to modify, extend, and reuse the code snippet.
- We evaluate NL2Viz on a real-world visualization benchmark and a public dataset to show its applicability. We also conduct a user study with data scientist professionals on real world scenarios, finding that NL2Viz is easy to use and helpful for generating not only plots, but also readable code that could be extended and reused.

In the rest of this chapter, Section 4.2 illustrates our overall approach using a motivating example. Section 4.3 discusses the implementation of NL2Viz. Section 4.4 presents our evaluation results and Section 4.5 summarize this part of work.

4.2 MOVIVATING EXAMPLES

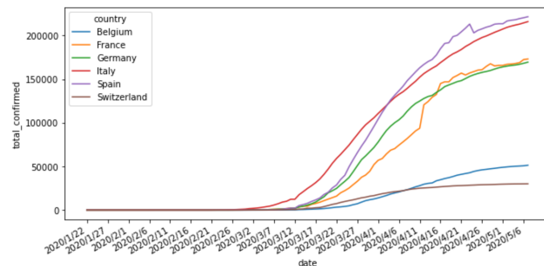
This section provides a high-level overview of NL2Viz via a motivating example. In this example, a data scientist named Alice wants to study the trend of COVID-19 infection in Europe. She opens Jupyter Notebook [89], a popular platform among data scientists, to load a COVID-19 dataset[1] (4.1). Each row in the dataset reports the numbers of daily confirmed cases and deaths for a country in a certain day, along with the accumulated numbers of confirmed cases and deaths until that date.

[1]https://data.humdata.org/dataset/novel-coronavirus-2019-ncov-cases

	continent	country	date	daily_confirmed	total_confirmed	daily_death	total_death
915	Europe	Belgium	3/27/2020	1049	7284	69	289
832	Asia	Iran	3/21/2020	966	20610	123	1556
1328	North America	Canada	4/25/2020	678	19245	55	989
749	Europe	Germany	3/15/2020	1210	5795	2	11
801	Asia	Japan	3/19/2020	35	924	0	29
833	Europe	Germany	3/21/2020	2365	22213	17	84
429	Europe	Spain	2/21/2020	0	2	0	0
1378	Asia	Iran	4/29/2020	1073	93657	80	5957
492	Asia	South Korea	2/26/2020	284	1261	2	12
162	Europe	Italy	2/2/2020	0	2	0	0

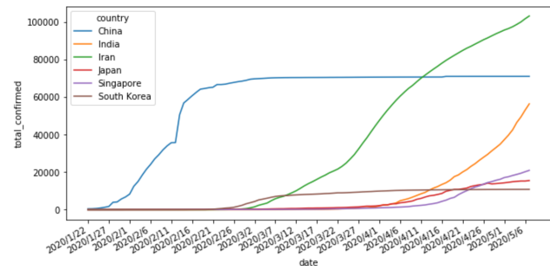
Figure 4.1: The "COVID-19" dataset (sampled 10 rows).

```
In [3]:  1 %plot line showing total confirmed cases for countries in Europe

df_tt0 = df.loc[df['continent']=='Europe']
fig, ax = plt.subplots(figsize=(10,5))
df_tt = df_tt0.groupby('country')
for cc, df_tt1 in df_tt:
    df_tt2 = df_tt1.reset_index()
    ax.plot(df_tt2['date'], df_tt2['total_confirmed'], label=str(cc))
ax.set_xticks(np.arange(0, df_tt1['date'].size,5))
fig.autofmt_xdate()
plt.xlabel('date')
plt.ylabel('total_confirmed')
plt.legend(title='country')
```



(a) The user invokes NL2Viz to obtain a plot with code.

```
In [4]:  1 %plot change to Asia

df_tt0 = df.loc[df['continent']=='Asia']
fig, ax = plt.subplots(figsize=(10,5))
df_tt = df_tt0.groupby('country')
for cc, df_tt1 in df_tt:
    df_tt2 = df_tt1.reset_index()
    ax.plot(df_tt2['date'], df_tt2['total_confirmed'], label=str(cc))
ax.set_xticks(np.arange(0, df_tt1['date'].size,5))
fig.autofmt_xdate()
plt.xlabel('date')
plt.ylabel('total_confirmed')
plt.legend(title='country')
```



(b) The user creates another plot by adapting the existing one.

Figure 4.2: The screenshots of NL2Viz in Jupyter Notebook while working with the motivating example.

Alice first wants to see the trend of confirmed cases for all countries in Europe. Alice understands that she needs to restrict the `continent` column to "Europe". Because there are multiple countries in Europe, Alice also needs to group data by country before she can iterate and plot a line chart for each country. In the plot, the x-axis is the date sorted chronologically and the y-axis is the confirmed cases for that date. 4.2a shows the desired plot and code.

The plotting code is non-trivial. Alice not only has to pick the right functions in `matplotlib` (e.g., `plot`), but also needs to process the data and construct the desired arguments for these functions. In our user study (Section 4.4.4), data scientists usually could not remember the usage of plotting functions and have to look up their documentation or search for similar code

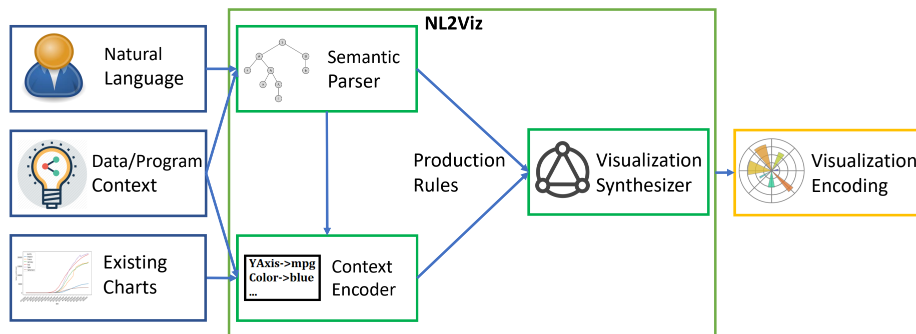


Figure 4.3: The workflow of NL2Viz.

in help forums. This context switching breaks the data scientists' workflow and negatively affects their productivity.

In contrast, Alice can perform the same task in NL2Viz by typing *"%plot line showing total confirmed cases for countries in Europe"* (*%plot* is our magic command to invoke NL2Viz in Jupyter Notebook). Because our target audience is data scientists who have sufficient coding skills, NL2Viz shows both the plot and the code to produce it (4.2a). Having access to the code allows Alice to tune the plot if she wants. For instance, she can modify the code to change the x-axis tick labels from every 5 days to a different number. Alice can also reuse the code. For example, Alice can easily wrap the synthesized code inside a function that plots the total number of confirmed cases in any given continent, and then loops over the function to create plots for all continents.

Alice also has an option to interactively change the existing plots using NL. For instance, she may type *%plot change y label to "Total confirmed cases"* to update the y-axis label, or *"%plot change to Asia"* to change the plot to countries in Asia (4.2b). Doing so is feasible because NL2Viz also uses knowledge of existing plots when synthesizing plots from text.

Our approach. We next explain how our NL2Viz approach synthesizes the desired visualization program from the following three modalities of specifications for the example from 4.2:

- The data/program context, which includes the "COVID-19" dataset, as shown in 4.1.
- The visualization context, which includes existing plots and their NL instructions.
- An NL instruction that describes the desired visualization task, e.g., *"%plot line showing total confirmed cases for countries in Europe"* provided by Alice as the instruction.

Given these inputs, NL2Viz synthesizes the desired program in two steps, as shown schematically in 4.3. In the first phase (*semantic parsing*), NL2Viz parses the three inputs into

symbolic constraints. Then, in the second phase (*program synthesis*), NL2VIZ synthesizes a complete program that satisfies these constraints.

More specifically, given an NL instruction and the current data context and program context, NL2VIZ uses semantic parsing techniques [91] to generate a *ranked list* L of tuples, each of which T consists of two sets of constraints: a set $\mathcal{R}_{must}$ of *hard constraints* and a set $\mathcal{R}_{may}$ of *soft constraints* (Figure 4.4 shows a simplified version of our grammar for the semantic parser). The set $\mathcal{R}_{must}$ includes hard constraints that *must* be satisfied by the desired program P, whereas constraints in set $\mathcal{R}_{may}$ are soft, indicating that they *may* be satisfied by P. In NL2VIZ, the constraints take the form of (a subset of) rules (of the grammar) used to generate plotting programs. For instance, for the example from 4.2, our semantic parser generates the following tuple $(\mathcal{R}_{must}, \mathcal{R}_{may})$ (among possibly others)[2]:

$$\begin{aligned}
\mathcal{R}_{must} \;=\; \{ \quad & \texttt{PlotType} \rightarrow \text{“LinePlot”}, \\
& \texttt{YAxis} \rightarrow \text{“total_confirmed”}, \\
& \texttt{FilterColumn} \rightarrow \text{“continent”}, \\
& \texttt{GroupColumn} \rightarrow \text{“country”}, \\
& \texttt{FilterValue} \rightarrow \text{“Europe”} \quad \} \\
\mathcal{R}_{may} \;=\; \{ \quad & \texttt{XAxis} \rightarrow \text{“date”}, \\
& \texttt{DataFrame} \rightarrow \text{“df”} \quad \}
\end{aligned} \tag{4.1}$$

Here, the first rule `PlotType` → “LinePlot” in $\mathcal{R}_{must}$ is a hard constraint: when we synthesize the target plotting program using the visualization domain-specific language's (DSL's) context-free grammar (CFG), the derivation in the synthesize process should use the rule `PlotType` → “LinePlot”. The hard constraints in $\mathcal{R}_{must}$ are extracted from the English instruction that directly corresponds to the user's intent. In contrast, the first constraint `XAxis` → “date” in $\mathcal{R}_{may}$ is soft, indicating that the program *may* use the “date” column as the x-axis. These constraints in $\mathcal{R}_{may}$ are generated from analyzing the data/program context and existing plots. Since these inputs provide only contextual hints that may be useful for deriving the complete program, we treat $\mathcal{R}_{may}$ as soft constraints.

Once NL2VIZ finishes generating hard and soft constraints from specifications, our second phase (program synthesis) synthesizes a complete visualization program from these constraints. NL2VIZ synthesizes a program from the visualization CFG that uses all rules in $\mathcal{R}_{must}$ and as many rules in $\mathcal{R}_{may}$ as possible. In the final step, NL2VIZ translates the program in the DSL to the target language (Python). For instance, given the preceding tuple $(\mathcal{R}_{must}, \mathcal{R}_{may})$, our synthesizer is able to generate the desired program P in 4.2a. Our

[2]Note that although the NL instruction does not mention `continent`, the parser is able to include that column because the parser could derive a relationship between “Europe" and `continent` from the data context.

```
      Root ::= PlotElems
 PlotElems ::= PlotElem PlotElems?
  PlotElem ::= HistoElems| ··· | FilterElem| GroupElem
HistoElems ::= HistoElem HistoElems?
 HistoElem ::= HistoType| Column| Bins| Stack| Log| Density
       ...
 GroupElem ::= GroupType| GroupColumns| GroupOperator
```

Figure 4.4: Simplified version of the NL grammar.

synthesizer generates one program P_i for each tuple T_i in L and finally returns a program P that has the smallest cost among all P_i's. (The cost of each derivation is defined later in Section 4.3.4.)

Given the NL instruction in 4.2b, the semantic parser returns the hard constraint set $\mathcal{R}_{must}$ = {FilterValue → "Asia", FilterColumn → "continent"}. The soft constraint set $\mathcal{R}_{may}$ now also includes the constraints of the previous plot. Given these constraints, NL2VIZ is able to adapt the plot in 4.2a to the plot in 4.2b with minimal human guidance.

We next discuss the design and the implementation of NL2VIZ.

4.3 APPROACH

Figure 4.3 depicts our overall workflow for converting the given NL instruction to visualization code. First, we use a semantic parser to extract $\mathcal{R}_{must}$, the set of constraints that must be used, from the user-provided NL instruction. We then analyze the data/program context to extract a set of constraints that may be used (i.e., $\mathcal{R}_{may}$). Finally, our synthesis algorithm synthesizes the program in our visualization domain-specific language from the extracted constraints, and translates the program into Python.

4.3.1 Parsing NL Instruction to Constraints

Figure 4.4 shows a partial simplified version of our attribute grammar (NL grammar) used to parse an NL instruction to constraints. We design this grammar by analyzing online tutorials, visualization courses, and Jupyter Notebooks with high upvotes in the Kaggle competitions [93]. We attach semantic rules in the form of S-attributes to the grammar. Each nonterminal in the NL grammar is associated with a list of attribute-value pairs, which

$$
\begin{aligned}
\texttt{Plot} &::= \mathit{plt}(\texttt{Data}, \texttt{Mappings}, \texttt{PlotType}, \texttt{Legends}) \\
\texttt{Data} &::= \texttt{DataSet} \mid \mathit{process}(\texttt{Data}, \texttt{Processor}) \\
\texttt{Processor} &::= \mathit{filter}(...) \mid \mathit{groupBy}(...) \mid \mathit{orderBy}(...) \\
\texttt{Mappings} &::= \mathit{nil} \mid \mathit{list}(\texttt{Mapping}, \texttt{Mappings}) \\
\texttt{Mapping} &::= \mathit{xaxis}(\texttt{XAxis}, \texttt{Scaling}?, \texttt{Options}?) \mid \mathit{yaxis}(...) \mid ... \\
\texttt{Scaling} &::= \mathit{range}(\texttt{Float}, \texttt{Float}) \mid \mathit{log}(\texttt{Scaling}) \mid \\
&\qquad \mathit{step}(\texttt{Scaling}, \texttt{Float}) \mid ... \\
\texttt{Options} &::= \mathit{stacking}(\texttt{Options}) \mid \mathit{transparent}(\texttt{Options}) \mid ...) \\
\texttt{PlotType} &::= \text{“}\texttt{Histogram}\text{”} \mid \text{“}\texttt{Scatter}\text{”} \mid \text{“}\texttt{Lineplot}\text{”} \mid ... \\
\texttt{Legends} &::= \mathit{legend}(\texttt{Title} : \texttt{String}, \texttt{Labels}, ...) \\
\texttt{Labels} &::= \mathit{label}(\texttt{XAxisLabel} : \texttt{String}, \texttt{YAxisLabel} : \texttt{String}, ...)
\end{aligned}
$$

Figure 4.5: Simplified visualization DSL.

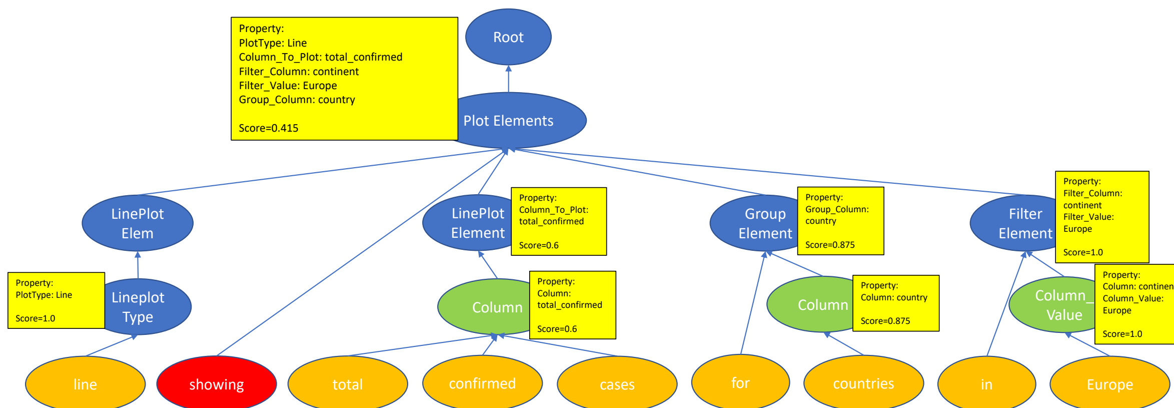


Figure 4.6: The parse structure for the motivating example.

corresponds to the semantics of this nonterminal. Given an NL instruction, we use a semantic parser [91] (which uses an enhanced CYK algorithm [94]) to parse the instruction into a list of attribute-value pairs. These pairs form our $\mathcal{R}_{must}$ set.

Figure 4.6 shows a simplified parse structure for the example in Section 4.2. We obtain the parse structure by making the following enhancements to the CYK algorithm.

Setting attribute values for terminals. We use *annotators* to initialize the attribute values. For example, for the nonterminal `Column`, we have an annotator *ColumnValue* that parses the token "europe" into a `ColumnValue` nonterminal symbol. The *ColumnValue* annotator maps one or multiple consecutive tokens to a value in some column in the given dataset. By using annotators, the semantic parser is able to parse tokens in a data-context-sensitive manner (e.g., parsing "Europe" as a `ColumnValue` and inferring "continent" as its

`ColumnName`) and program-context-sensitive manner (e.g., parsing "foo" as a function name if a function of that name appears in the Jupyter Notebook code cell).

Setting attribute values for rule $N ::= N_1 N_2$***.*** The attributes are propagated from children to parent following the semantic rules. Most semantic rules just propagate the lists of attributes without change. However, in some cases, the semantic rules can change the attribute name in children's list. For example, in the `FilterElem` symbol, the attribute `ColumnValue` changes its attribute name (e.g., `ColumnValue` becomes `FilterValue`).

Fitness score. Each nonterminal also has a fitness score in the range of $[0, 1]$ to represent the probability of producing that parse structure. For instance, the token "countries" does not exactly match the column name "country" in the dataset; hence, its symbol "Column" is given a score of 0.87 based on the edit distance of the two strings. The score of a parent symbol is simply the product of the scores of its children.

Note that our NL grammar is inherently ambiguous to capture different interpretations of an NL instruction. From an NL instruction, our semantic parser produces multiple parse structures, each of which has a $\mathcal{R}_{must}$ set and a fitness score.

4.3.2 Using Data/Program Context to Construct May-use Constraints

Because the instruction usually does not contain all information necessary to synthesize the visualization (i.e., $\mathcal{R}_{must}$ is not complete), NL2Viz uses multiple heuristics to infer the potential omitted information (i.e., $\mathcal{R}_{may}$) from the data/program context. For example, when plotting a scatter plot, if the mapping of data columns to axes is not evident from the user's NL instruction, our approach prefers a categorical column to be on the x-axis.

Our heuristics in NL2Viz also capture popular *data preprocessing patterns*. For example, if the user wants to plot a line plot, but we find that there are multiple points on the same x-axis coordinate in the dataset, then it is likely that there is an inherent grouping step by the column on the x-axis before plotting. NL2Viz analyzes each column to determine (a) the *type* of values in that column, (b) whether the column is categorical, and (c) all the *distinct values* in that column. We use this information to create $\mathcal{R}_{may}$. For example, even if the instruction in Figure 4.2 does not mention "continent" in the text, NL2Viz infers that `AuxColumn` $\rightarrow$ "continent" in the FilterElem rule based on data insights, and adds it to $\mathcal{R}_{may}$.

4.3.3 Designing Visualization Domain-Specific Language

Given the sets of $\mathcal{R}_{must}$ and $\mathcal{R}_{may}$, our synthesis algorithm synthesizes a visualization program in a domain-specific language (DSL). Figure 4.5 shows a simplified version of this

visualization DSL (nonterminals start with uppercase letters, function symbols start with lowercase letters, and terminals are within quotes). Programs in this DSL are then translated to a target visualization library (such as `matplotlib` or `seaborn`) in the final translation step.

To design this DSL, we first perform a preliminary study on the Jupyter Notebook dataset released by Felipe et al. [95]. Based on the stats of different plots used in the dataset and the documentation of popular visualization libraries, such as `mathplotlib`, `seaborn`, and `ggplot2`, we include the frequently used plot types and parameters including the column to be plotted and the size/color/style of the visualization element. Additional grammar rules in $\mathcal{R}$ link these parameters with the corresponding plot type.

Based on the analysis of the plotting code fragments collected from the Jupyter Notebook dataset, we also include rules (in $\mathcal{R}$) that perform *data preprocessing operations*. For example, we observe that three commonly used typical patterns of data preprocessing are filtering, grouping, and ordering; hence, we extend the grammar rules $\mathcal{R}$ to include rules that perform these steps.

4.3.4 Constrained Syntax-Guided Synthesis

Having defined the NL grammar and the visualization DSL, we next illustrate our main synthesis algorithm (Algorithm 4.1). The algorithm takes a grammar $\mathcal{G}$ = (terminal set $\mathcal{T}$, nonterminal set $\mathcal{N}$, production rules $\mathcal{R}$, start symbol `Plot`), and a pair $(\mathcal{R}_{must}, \mathcal{R}_{may})$ of must-use and may-use constraints (used for constraining derivations) as input; informally a derivation is application of a grammar rule toward generating the target program. The algorithm returns a program (generated by $\mathcal{G}$), generating which undergoes a derivation that satisfies $\mathcal{R}_{must}$ and minimizes the cost function.

In the visualization DSL, we begin with the start symbol `Plot` and perform a series of derivations to further extend to a complete program. More formally, a derivation is a sequence of terms that start with the start symbol `Plot`, and each subsequent term is obtained from the previous term by applying a production rule in $\mathcal{G}$. If we use $P_1 \rightarrow_{r_1} P_2$ to denote that P_2 is derived from P_1 by applying r_1, then a derivation of P, denoted by d, can be written as

$$\texttt{Plot} \rightarrow_{r_0} P_1 \rightarrow_{r_1} P_2 \rightarrow_{r_2} \cdots P_k \rightarrow_{r_k} P \tag{4.2}$$

A program is *incomplete* if it contains a non-terminal symbol. A *complete* program contains only functions and terminal symbols. As an example, the derivation for the first program shown in Section 4.2 is shown in Figure 4.7 with some simplification.

```
Plot → plt(Data, Mappings, PlotType, Legends)
Data →³ process(process(DataSet, Processor), Processor)
DataSet → "df"
Processor → filter(FilterColumn, FilterValue)
      →² [filter("continent", "Europe"), group("country")]
Processor → group(GroupColumn) → group("country")
Mappings →² list(Mapping, list(Mapping, nil))
Mapping → xaxis(XAxis) → x("date")
Mapping → yxis(YAxis) → y("total_confirmed")
PlotType → "Lineplot"
Legends → Labels → labels(XAxisLabel, YAxisLabel)
      →² labels("date", "total_confirmed")
```

Figure 4.7: The derivation of the program in Section 4.2.

The algorithm works by maintaining a worklist that consists of tuples (P, d, c), where P is a (potentially incomplete) program generated by derivation d whose cost is c. In each iteration, the algorithm works by picking an element (P, d, c) from the worklist. If the program P cannot be completed to a program that satisfies $\mathcal{R}_{must}$ (determined using a subroutine `feas`) or the current cost c is already more than the best cost found so far, we just prune this search branch and continue with the next iteration (Line 7). If not, then we further process this tuple (P, d, c). We first check whether P is already a complete program (Line 9), and if so, we update the best solution found so far and continue to the next iteration (Lines 10-12). If P is not complete, we apply all possible single-step rewrites to P and add new items to our worklist (Lines 15-17).

We next describe the subroutine $\texttt{feas}(d, \mathcal{R}_{must})$ that checks whether derivation d satisfies the constraint $\mathcal{R}_{must}$. If d is a complete derivation (generating a complete program), then $\texttt{feas}(d, \mathcal{R}_{must})$ returns "true" iff all rules in $\mathcal{R}_{must}$ are included in derivation d.

Since we aim to satisfy all constraints in $\mathcal{R}_{must}$ and as many constraints in $\mathcal{R}_{may}$ as possible, for each derivation we define its cost to be equal to the number of the production rules (used in this derivation) that do not belong to the set $\mathcal{R}_{may}$. In the following definition of the cost function, we use the notation $d|_i$ to denote the subderivation $(S, r_0, P_1, \cdot\cdot, r_{i-1}, P_i)$ of the derivation d consisting of the first i rule applications. If d has k rule applications, $d|_k = d$. Given the may-use constraint $\mathcal{R}_{may}$, the cost of a derivation d is defined as follows:

$$\texttt{cost}(d, \mathcal{R}_{may}) \quad = \quad \sum_{i=1}^{k} \texttt{cost}_e(r_i \mid d|_i, \mathcal{R}_{may}) \tag{4.3}$$

Algorithm 4.1: Branch-and-bound for cSyGuS.

Inputs : A CFG $\mathcal{G} := (\mathcal{N}, \mathcal{T}, \mathcal{R}, S)$, a pair $(\mathcal{R}_{must}, \mathcal{R}_{may})$ of constraints on $\mathcal{G}$

Output: A program P whose derivation d in $\mathcal{G}$ satisfies $\mathcal{R}_{must}$ and has minimum cost under $\texttt{cost}(d, \mathcal{R}_{may})$

```
1  P* ← Null ;                                        // best program found so far
2  c* ← ∞ ;                                           // cost of the best program found so far
3  Q ← {(S, ⟨S⟩, 0)} ;                                 // worklist queue
4  while Q ≠ {} do
5  |  (P, d, c) ← Remove an element from Q;
6  |  if feas(d, R_must) is false or c > c* then
7  |  |  continue
8  |  end
9  |  if P has no nonterminals then
10 |  |  P* ← P;
11 |  |  c* ← c;
12 |  |  continue
13 |  end
14 |  R ← all rules in R applicable on P;
15 |  foreach r ∈ R do
16 |  |  add (P', ⟨d, r, P'⟩, c') to Q where P →_r P' and c' = c + cost_e(r|d, R_may)
17 |  end
18 end
19 return P*
```

where the elementary cost function $\texttt{cost}_e$ is defined as

$$\texttt{cost}_e(r \mid d, \mathcal{R}_{may}) \quad = \quad \begin{cases} 0 & \text{if } r \in \mathcal{R}_{may} \\ 1 & \text{otherwise} \end{cases} \tag{4.4}$$

The cost of a derivation is simply the number of rules (in the derivation) that are not included in $\mathcal{R}_{may}$. Note that
$\texttt{cost}(d, \mathcal{R}_{may}) = 0$ iff every production used in d lies in $\mathcal{R}_{may}$.

4.3.5 Extension to An Interactive System

We have implemented our approach with a supporting interactive tool. After NL2Viz synthesizes the first Python program and shows the generated plot, if the user is not satisfied with it, then the user can give another NL instruction to refine the plot. In the subsequent re-synthesis runs, NL2Viz uses additional information from the *program context* – the production rules used to generate the previous programs are included in the set $\mathcal{R}_{may}$ (as may-use production rules) to help the synthesizer prefer programs that are similar to the

mpg	cylinders	displacement	horsepower	weight	acceleration	model_year	origin	name
12.0	8	400.0	167	4906	12.5	73	usa	ford country
23.8	4	151.0	85	2855	17.6	78	usa	oldsmobile starfire sx
26.0	4	79.0	67	1963	15.5	74	europe	volkswagen dasher
23.5	6	173.0	110	2725	12.6	81	usa	chevrolet citation
32.2	4	108.0	75	2265	15.2	80	japan	toyota corolla
26.4	4	140.0	88	2870	18.1	80	usa	ford fairmont
19.2	8	267.0	125	3605	15.0	79	usa	chevrolet malibu classic (sw)
17.5	6	250.0	110	3520	16.4	77	usa	chevrolet concours
31.0	4	76.0	52	1649	16.5	74	japan	toyota corona
15.0	6	250.0	72	3158	19.5	75	usa	ford maverick

Figure 4.8: The "AUTO-MPG" dataset (sampled 10 rows).

previously generated programs.

4.4 EVALUATION

We implement our approach as a Python package that registers a *magic ipython command* [96] `plot` in the popular Jupyter Notebook environment [89]. Hence, a user can input "`plot a histogram of cylinders`" to obtain an appropriate plot (see 4.2). The Jupyter interface for NL2VIZ also supports rudimentary auto-complete, suggesting column names, keywords, and pre-processing function names. The data for plotting is assumed to be in the form of a dataframe object from the widely used Pandas library [97]. We choose Matplotlib and Seaborn as the target plotting libraries for the generated code.

Our evaluation aims to answer four specific research questions:

- **RQ4.1: One-shot accuracy.** How accurately can NL2VIZ produce the target plot from a single NL instruction? How effectively can the data/program context help NL2VIZ resolve ambiguities in the user's NL instruction?
- **RQ4.2: Plot-and-change accuracy.** How accurately can NL2VIZ create a new chart from an NL change instruction? How much does the user benefit from NL2VIZ in this scenario in terms of the instruction length reduction?
- **RQ4.3: Comparison with the state of the art.** How does NL2VIZ compare with other related state-of-the-art tools for visualization synthesis?
- **RQ4.4: Usability.** How usable and accurate is NL2VIZ in a real setting?

Benchmark We collect 303 NL instructions for 54 plots from two sources.

Auto-MPG plot descriptions. In this source, there are 267 manually written NL instructions for 18 plots selected from online tutorials that use the "Auto-MPG" dataset[3], which contains technical specifications of 398 cars as shown in Figure 4.8. These NL instructions are provided by 15 professionals with experience in data science.

Homework and COVID-19 assignments. We also collect 36 scenarios from homework assignments and Jupyter Notebooks that use COVID-19 datasets in GitHub. In these scenarios, we use the problem statements as the NL instructions and the plots as the expected results. We exclude 5 instructions from these scenarios in which the plot types are not supported by NL2Viz.

NL2VIS dataset. Luo et al. [92] publish an NL2Visualization dataset named NL2VIS. The NL2VIS dataset is generated by applying a neural-network-based NL2SQL-to-NL2VIS model on a popular NL2SQL dataset named Spider [98]. Although the NL2VIS dataset has an impressive number of 25,750 (NL, VIS) pairs, we find that the dataset mainly focuses on the data preprocessing steps as most visualizations in the dataset are a direct presentation of the output by a SQL query from the Spider dataset without the plot options such as formats and legends. So we evaluate NL2Viz on the NL2VIS dataset in only RQ4.3 to compare with the results reported in Luo et al.'s paper [92].

4.4.1 RQ4.1 Results: One-Shot Accuracy

Correctness We classify the output plots of NL2Viz into four separate types based on how well they match the ground truth:

- **Exact Match.** In this case, the output of NL2Viz exactly matches the ground truth.
- **Functionally Equivalent.** In this case, the output plot is functionally equivalent to the ground truth plot, but differs in a minor, often visual, detail. For example, when the instruction does not specify that a histogram should have normalized frequency on the y-axis, the ground truth does use frequencies, while NL2Viz produces a histogram using counts instead of frequencies. However, the two plots are functionally equivalent for most purposes. Therefore, we also consider this type to be correct.
- **Functionally Different.** In this case, the output plot differs from the ground truth in significant details. For example, the output plot has an axis plotted using a linear scale, while the ground truth uses a logarithmic scale.

[3]https://www.kaggle.com/uciml/autompg-dataset

Table 4.1: Accuracy of NL2VIZ. The columns em, fe, fd, and nm denote exact match, functionally equivalent, functionally different, and no match, respectively.

Category	#	Correct			Incorrect		
		em	fe	Tot.	fd	nm	Tot.
Total	295	173	47	220	37	38	75
Homework	31	19	2	21	5	5	10
AUTO-MPG Total	264	154	45	199	32	33	65
AUTO-MPG - Easy	119	86	16	102	15	2	17
AUTO-MPG - Hard	145	68	29	97	17	31	48

- **No match.** Here, the produced plot is completely different from the ground truth. We differentiate "functionally different" from "no match" because in a "no match" case, often the time the semantic parser is not able to capture the intention from the NL instruction, while in a "functionally different" case, usually the synthesizer is not able to synthesize a reasonable program indicating that we should extract more may-use constraints with higher accuracy.

When measuring the accuracy, we consider the exact match and functionally equivalent types to be correct, and functionally different and no match to be incorrect.

Results The results of the evaluation are summarized in Table 4.1. Overall, NL2VIZ is able to produce the correct output in 74.5% of the cases (with 58.6% being an exact match, and 15.9% being functionally equivalent). Of the remaining, 12.5% of the cases are functionally different.

To further analyze the results, we separate the AUTO-MPG instances into two categories, *hard* and *easy*, based on whether the plot requires or not additional data preprocessing steps and additional parameters not in the input or dataframe. The idea to differentiate hard and easy plots is due to the observation that in an easy scenario, the synthesizer should be able to generate a correct program using only or mostly must-use constraints; while in a hard scenario, some information of the plot is often missing or vague in the NL instruction such that the synthesizer requires enough may-use constraints to generate a correct program. Among the 18 plots, 8 are categorized as easy and 10 as hard. As shown in Table 4.1, the system achieves an accuracy of 85.7% and 70% in the easy and hard cases, respectively. We discuss the failure cases below.

Analysis of Failure Cases For the Homework category, the main reason for failures is missing context – the homework assignments often contain relevant data in previous

Table 4.2: Accuracy of the baseline approach (NL2VIZ without context and data understanding).

Category	#	Correct			Incorrect		
		em	fe	Tot.	fd	nm	Tot.
Total	295	84	21	105	50	140	190
Homework	31	7	2	9	12	10	22
AUTO-MPG Total	264	77	19	96	38	130	168
AUTO-MPG - Easy	119	77	16	93	21	5	26
AUTO-MPG - Hard	145	0	3	3	17	125	142

discussions or problems. In the two "no match" cases in the "AUTO-MPG - Easy" category, the semantic parser interprets the parameter representing point sizes as a group by column. For example, for the input *"scatter plot of mpg and acceleration with point size by cylinder"*, the semantic parser interprets cylinder as a group by column instead of a parameter for the point size. Most "functionally different" cases in both the "AUTO-MPG - Easy" and "AUTO-MPG - Hard" categories involve bin sizes in histograms. For example, for the input *"plot a bar graph that shows me the number of rows with MPG in range 5 to 10, 10 to 15, and so on"*, NL2VIZ is unable to identify the bin sizes due to the limitations of its DSL grammar.

For the "AUTO-MPG - Hard" category, the limitation of NL2VIZ is in the parsing of the filtering or group by clauses. For example, the input fragment *yearly* or *annual* represents the "group by model_year" clause in the "Auto-MPG" dataset. However, NL2VIZ is unable to do these translations in a fraction of the cases. As mentioned in Section 4.3.1, we choose to use a context free grammar to represent the NL instruction because the grammar would cover most cases. However, for words such as *yearly* or *annual* that are functionally equivalent to the column name *model_year*, we are unable to enumerate and cover all equivalent words in the grammar. Recent developments in the field of detecting equivalence via extrapolation [99] present a potential solution, and we believe that these scenarios can be correctly handled with better NL understanding.

Effect of Context and Data Understanding We also run a baseline synthesizer whose results are shown in Table 4.2. The baseline uses only NL information (extracted using the semantic parser described in Section 4.3.1), and does not use the data/program context. As expected, the code generated by the baseline approach in certain cases is incomplete. For example, in the instruction *"plot scatter average mpg by cylinder"*, there is an inherent group operator since it is required to calculate the aggregation function average for the "mpg" column. However, the "cylinder" column cannot function as both the column for the x-axis

Table 4.3: Results of change experiments. The columns Avg. InitLen., Avg. ChgLen., Avg. FinLen., and Avg. Red. stand for Average Initial Instruction Length, Average Change Instruction Length, Average Final Instruction Length, and Average Instruction Length Reduction in Correct cases, respectively. Instruction length is in the number of characters.

Plot Type	Change Category	Total	Correct	Avg. InitLen.	Avg. ChgLen.	Avg. FinLen.	Avg. Red.
Easy	Add	5	5	37.2	13.6	46.4	68%
	Replace	5	3	36.4	16.4	38.2	57%
Hard	Add	10	8	55.6	22.5	73.0	64%
	Replace	10	5	54.3	29.1	61.7	54%
	Add+Replace	10	4	57.2	37.0	69.9	49%

and group column, resulting in a missing group operation. Tables 4.1 and 4.2 show that using the data/program context substantially improves the results, especially in the "AUTO-MPG - Hard" category. In particular, no cases of the "AUTO-MPG - Hard" category can get exact match results because all the plots in this category require information from the data context.

Performance We measure the performance of NL2VIZ by the execution time. It turns out that NL2VIZ is quite efficient, and we set a timeout bound to be 30 seconds. The average execution time for an instruction is around 3 seconds, and 95% of the instructions finish within 5 seconds. There are 3 instructions causing timeout, all of which are in the "AUTO-MPG - Hard" category. They all have a length of more than 150 characters with the longest one being 340 characters, resulting in timeout in the phase of semantic parsing.

4.4.2 RQ4.2 Results: Plot-and-Change Accuracy

This section shows the evaluation results of NL2VIZ with respect to RQ4.2, i.e., in the setting where NL2VIZ is provided with an already existing plot and a change instruction. For example, the initial plot could have been generated given the instruction *"scatter plot of mpg and acceleration grouped by cylinders"* and the change instruction can be *"change to average mpg and accleration"*. For the initial instruction, an initial plot is generated such that each car is a single point colored based on the number of cylinders. For the change instruction, a plot should be generated such that each point corresponds to a group of cars with the same number of cylinders, with the coordinates given by average mpg and average acceleration.

Table 4.3 shows the information about the 40 tests. We separate the plots into the "Easy" and "Hard" types based on whether contextual information is needed to generate a plot. Further, we group the change instructions into the *Replace* and *Add* categories: *Replace*

change instructions replace the value of some plot aspect with another, while *Add* change instructions add a new aspect to a plot. We do not consider the *Delete* category: in most practical scenarios, users start with a simple plot and add more complexity over time.

Accuracy Results Table 4.3 shows the accuracy results of NL2VIZ on the change experiments. We find that NL2VIZ performs well in general, achieving 67.5% accuracy (25 correct out of 40 total). NL2VIZ is more effective in processing *Add* instructions than *Replace* instructions. *Replace* instructions change an existing, correct aspect in the plot. Hence, NL2VIZ needs to both locate the aspect to be replaced and parse the new aspect correctly. While in an *Add* instruction, because the new aspect usually does not interfere with existing rules, NL2VIZ just needs to add the new aspect.

Instruction Length Results We also evaluate how much instruction length is saved by the change instruction. For each initial instruction and change instruction pair, we also generate a *combined* instruction from which NL2VIZ can produce the intended plot in one attempt. In terms of absolute numbers, we can see that the average length of a *Replace* instruction is higher than that of an *Add* instruction, due to the need of specifying both the component to replace and the replacement. On the other hand, the average length of a *combined* instruction is higher than that of an *Add* instruction, due to the additional information added by the *combined* instruction.

We also measure the average instruction-length reduction, given by $1 - \frac{Length(change\ instruction)}{Length(combined\ instruction)}$. We evaluate the setting where the user has already used the initial instruction to create a plot, and later wants to update it, either by using the change instruction or directly using the *combined* instruction. We find that in general we achieve 58% instruction-length reduction via the interaction enabled by the change instructions. We achieve higher reduction in the *Add* change category since the combined instruction text has to specify both the old and new aspects, while the change instruction has to specify only the new aspects.

Analysis of Failure Cases We note that the *Add* change category has a higher accuracy than the *Replace* change category because the aspect that the user wants to replace is often implicit in the change instruction. In the example in Section 4.2, the change instruction is *"change to Asia"*. In this instruction, by the help of data context, it is easy to infer that the aspect needs to be replaced is *"Europe"*. However, in some cases, it is unclear whether the instruction is to change an existing aspect or to add a new aspect. For example, when the initial instruction is *"plot scatter of mpg versus model year"*, which would cause to produce

Table 4.4: Comparison with the state of the art.

	DeepEye	NL4DV	SEQ2VIS	NL2VIZ
Easy	9.5%	11.5%	67.4%	83.9%
Medium	15.4%	22.5%	69.6%	74.2%
Hard	1.4%	7.6%	60.5%	41.5%
Extra Hard	6.1%	4.1%	61.8%	13.3%
Overall	9.1%	13.7%	65.7%	58.8%

a scatter plot with each point representing a car and its mpg (y-axis) versus model year (x-axis). The change instruction is *"change to average mpg for all cylinders"*. The idea of this change is to produce a final instruction that is *"plot scatter of average mpg for all cylinders"*. The final plot is a scatter plot where x-axis represents cylinder number and y-axis represents average mpg. In this case, the *"for all cylinders"* in the instruction text represents a *Replace* change. However, it can also be the case that the final instruction is *"plot scatter of average mpg versus model year group by cylinders"*. In this case, the final plot is a scatter plot where x-axis represents model year and y-axis represents average mpg with different cylinders, having points with different colors on the plot.

4.4.3 RQ4.3 Results: Comparison with the State of the Art

Luo et al. [92] report the accuracy of their neural-translation-model-based tool SEQ2VIS along with two other rule-based and semantic-parser-based tools DeepEye [100] and NL4DV [101] on a test set containing 3990 (NL, VIS) pairs from their NL2VIS dataset.

Due to the different target libraries of NL2Viz and the NL2VIS dataset (Matplotlib vs. Vega-Lite), although it is possible to translate one to another, we find it not reasonable to directly compare the visualization programs as different programs may lead to the same, or the essentially same plots. We also notice that in Luo et al.'s paper [92], they measure the "tree matching accuracy" to compare their SEQ2VIS tool with other tools. The "tree matching accuracy" measures whether the flow of data transformation for each data column and its corresponding data shown on the axis are correct. Therefore, for each test case, we need to manually examine whether our synthesized visualization program is equivalent to their program, which is the labeled output. We are able to manually verify the 500 (NL, VIS) pairs sampled from their 3990-pairs test set. We treat our output to be correct if it is an Exact Match or Functionally Equivalent as defined in Section 4.4.1.

Table 4.4 shows NL2Viz's accuracy against other state-of-the-art tools on the NL2VIS dataset. Out of the 500 sampled pairs, there are 32 visualizations that are currently not

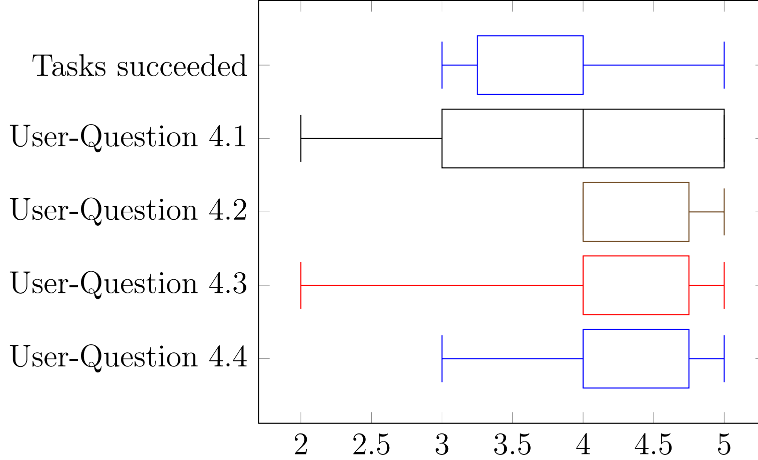


Figure 4.9: The distribution of succeeded tasks and scores among 6 participants.

supported by NL2Viz. First, we can see that NL2Viz outperforms the other two rule-based tools by a large margin in all difficulty groups. The reason is that our tool supports the synthesis of data preprocessing steps while the other two tools do not have support or have only very limited support. When compared to SEQ2VIS (a neural-translation-model-based approach), we find that in the "Easy" and "Medium" categories, our NL2Viz tool outperforms SEQ2VIS. A possible reason is that the "Easy" cases here are similar to the "Easy" cases in Section 4.4.1, and the "Medium" cases here are similar to the "Hard" cases, and for both categories we see similar accuracy as that in Section 4.4.1, being 83.9% vs. 85.1% and 74.2% vs. 70%. In these two categories, our NL grammar and Visualization DSL can cover the target visualization program. Therefore, we are able to achieve higher accuracy than a learning-based approach by using constrained syntax-guided synthesis. However, in the "Hard" and "Extra Hard" cases, the accuracy of NL2Viz declines drastically. The reason is that the NL instructions in these two categories are usually generated from nested SQL queries, especially for the "Extra Hard" category. Our grammar or DSL does not cover such data processing steps; therefore, it is impossible for NL2Viz to produce correct visualization.

Overall, NL2Viz achieves lower but comparable accuracy compared with the SEQ2VIS tool, while being able to achieve higher accuracy in "Easy" to "Medium" cases without the need for training data. NL2Viz also outperforms existing rule-based approaches by a large margin by leveraging the data/program context.

4.4.4 RQ4.4 Results: Usability and Interaction

To evaluate the usability of NL2Viz, we ask volunteers who have an average of 7.4 years of experience in data science to complete 5 plotting tasks using NL2Viz. Among the 5 plots in

the tasks, there are 2 histogram/bar charts, 1 scatter plot, and 2 line plots. Each participant is first asked to answer multiple background questions, and is then given an explanation of the "Auto-MPG" dataset. Next, we ask the participants to complete the 5 tasks in order until they are satisfied or are not willing to try NL2Viz any more. Each participant is given 20 minutes to finish all 5 tasks. Then, the participants are asked to rate NL2Viz on a scale of 1 (least positive) to 5 (most positive) on the following aspects:

- **User-Question 4.1.** Do you find NL2Viz easy to use?
- **User-Question 4.2.** Do you find it easy to interpret the code generated by NL2Viz and change for future usage?
- **User-Question 4.3.** Do you want to use NL2Viz before you do visualization in future?
- **User-Question 4.4.** Is NL2Viz able to understand your input?

On average, the participants successfully complete 4.13 out of 5 tasks. Figure 4.9 shows that the participants are generally in favor of NL2Viz, with an average of 4 out of 5 for all questions. The high variance for question 3 is due to that participants with high expertise who are very familiar with writing visualization code strongly do not prefer to use NL2Viz.

Suggestions from the participants highlight two major issues. Three participants (S2, S4, and S6) suggest that NL2Viz should have a built-in feature of "highlighting" to show which part of their natural language corresponds to which part of the generated code. This feature would help users change their input when NL2Viz misinterprets their intention. Another suggestion is to modify NL2Viz to produce multiple candidate plots for the same input (S1, S2, and S3). It is fairly straightforward to modify our semantic parser and synthesizer to produce multiple candidates, and we intend to make this modification. Other suggestions include allowing for click selecting columns as an input modality (S3), displaying a confidence score for the generated plot (S6), and a separate cleaning step before visualization (S5).

4.5 SUMMARY

In this chapter, we have presented a novel NL2Visualization approach and its supporting tool named NL2Viz for automatically synthesizing visualization programs from a user's NL instruction. The key idea underlying our approach is to leverage not only the NL instruction, but also the other contextual information (namely data context and program context) and then convert the different kinds of specifications provided by the user

into *symbolic constraints*, which can be used to generate the desired visualization programs using syntax-guided program synthesis. Moreover, NL2Viz includes an interactive interface for allowing the user to further refine and reuse the resulting visualization. We evaluate NL2Viz on a real-world visualization benchmark and a public dataset to show the effectiveness of NL2Viz. We also perform a user study involving 6 data scientist professionals to demonstrate the usability of NL2Viz, the readability of the generated code, and NL2Viz's effectiveness in helping users generate desired visualizations effectively and efficiently.

CHAPTER 5: NRRANKER: TRANSLATING NATURAL LANGUAGE TO REGULAR EXPRESSION VIA INTEGRATING PRE-TRAINED MODEL AND TEST GENERATION

5.1 OVERVIEW

Translating natural language (NL) descriptions into code (NL2Code) [102, 103, 104, 105, 106, 107] has become an important task for both natural language processing (NLP) and program synthesis [107, 108, 109, 110, 111, 112, 113, 114]. End users may not have the expertise to understand and write a complete program or a code snippet. Even for experienced developers, tasks such as writing a regular expression (regex in short) or a SQL query could be time-consuming and error-prone. Therefore, synthesizing programs from NL descriptions is useful for both end users and developers.

To help address the ambiguity of NL description, recent research [115, 116, 117] has proposed various approaches to leverage additional specifications, especially pairs of a test input and its expected test output, named as test cases, to help with the synthesis process. For example, in the task of NL to regular expression (NL2Regex), test cases are often defined as example strings (test inputs) and the output label (expected output) of each string to be either positive or negative, indicating whether the resulting regex should match the string. For example, Zhong et al. [118] and Ye et al. [116] have proposed approaches that require such positive/negative examples to filter the candidate regexes being synthesized by checking whether each candidate regex matches the positive/negative examples.

Although existing approaches show the advantage of leveraging test cases in NL2Code tasks [115, 116, 117], these approaches still face two major challenges. First, unlike the traditional program-by-example (PbE) scenario [3], in NL2Code tasks, test cases are often missing or of low quality. Since end users of NL2Code systems may not have domain expertise, end users may not be able to provide examples, or the examples provided by end users may not be helpful in identifying plausible candidates. Second, even when an NL2Code system has access to example test cases, these test cases often remain a fixed test set. The fixed test set allows the system to eliminate synthesized candidate programs that cause at least one example test case from the test set to fail, but the system is unable to effectively rank plausible candidate programs that do not cause any example test case to fail. For example, in the HumanEval evaluation set proposed by Chen et al. [119], each task is equipped with example test cases. The CodeX [119] model is able to achieve the accuracy of 77.4% when the result is considered correct if one or more candidate programs out of the top-100 synthesized candidates are correct (noted as pass@100) determined by whether they can pass

a comprehensive set of evaluation test cases. All these top-100 candidates are able to pass the example test cases. But when we consider only whether the top-1 synthesized candidate program is correct (noted as pass@1), the accuracy drastically decreases to only 33.5%.

To address the aforementioned two challenges, in this work, based on two major insights, we propose a new approach named NRRanker that generates example test inputs for differentiating behaviors of the candidate programs being synthesized, and further integrates an existing NL2Code system with the preceding test generation using a pre-trained model. **Insight 5.1.** To filter/rank the candidate programs being synthesized does not require the generated example test cases to be equipped with correct expected test outputs (i.e., not requiring generated test cases to all pass on a correct program). The authors of the AlphaCode approach [120] train a separate model (from the model used to synthesize programs) to generate test cases. Although the generated test cases may contain mistakes (e.g., incorrect expected test outputs), these test cases still help improve the pass@10 accuracy by nearly 6%. **Insight 5.2.** A pre-trained code model could help predict the expected test output of the given generated test input. Recent development [119, 120] in pre-trained large language models (LLMs) has helped improve the accuracy in NL2Code tasks with the help of pre-trained models' flexibility in understanding various NL descriptions. With the idea that predicting expected test outputs is an easier task than generating a complete program, we find that using a pre-trained model could predict the expected test output and achieve considerable accuracy, thus helping filter/rank the candidate programs.

We implement NRRanker for the task of NL2Regex. First, we obtain a list of candidate regexes being synthesized using an existing approach [116, 121]. Second, our approach generates test input strings to differentiate the candidate regexes. For each candidate regex a, we create a set D_a of example test cases being the union of differentiating example input strings (for the pair of a and each remaining candidate regex) that can match a but cannot match the remaining candidate regex, and these example input strings' output labels (positive or negative) predicted via a pre-trained code model, indicating whether such example input string is consistent with the NL description. Finally, for each candidate regex a, we compute the ratio of the input strings with a positive label among D_a (indicating how high percentage of example string inputs in D_a whose matching results with a are likely to be consistent with the NL description determined via a pre-trained code model) and rank the candidate regexes based on the ratio.

We evaluate NRRanker on two datasets: KB13 [122] and Turk [121]. Our evaluations show that NRRanker is able to achieve the top-1 accuracy of 89.8% and 90.1% on the two datasets, respectively, outperforming state-of-the-art approaches by 11.1% on average.

In summary, this chapter makes three major contributions:

Table 5.1: Regex: Syntax → Semantics

<table>
<tr><th colspan="3">Non-Terminals</th></tr>
<tr><td>x & y → x and y</td><td>x.* → starts with x</td><td>x & y & z → x and y and z</td></tr>
<tr><td>x | y → x or y</td><td>.*x.* → contains x</td><td>x{N,} → x: ≥ N times</td></tr>
<tr><td>~(x) → not x</td><td>x | y | z → x or y or z</td><td>x{1, N} → x: ≤ N times</td></tr>
<tr><td>x → only x</td><td>(x)+ → x: ≥ 1 time</td><td>\b x\b → words with x</td></tr>
<tr><td>.*x → ends with x</td><td>(x)* → x: ≥ 0 time</td><td>.*x.*y → x followed by y</td></tr>
<tr><th colspan="3">Terminals</th></tr>
<tr><td>[a-z] → a lowercase letter</td><td>word → the string ‘word’</td><td>[0-9] → a number</td></tr>
<tr><td>[A-Z] → an uppercase letter</td><td>[AEIOUaeiou] → a vowel</td><td>. → a character</td></tr>
</table>

- We propose a new approach named NRRanker that integrates an existing NL2Code system with novel test generation using a pre-trained model.
- We implement NRRanker for the task of the NL2Regex problem.
- Our evaluations on the KB13 and Turk datasets show that NRRanker achieves 89.8% and 90.1% top-1 accuracy, respectively, outperforming state-of-the-art approaches by 11.1% on average.

5.2 BACKGROUND

In this section, we introduce the definition of regular expression used in this chapter and two existing approaches, DeepRegex [121] and DeepSketch [116].

5.2.1 Regular Expression

Regular expressions (regexes) often come in different formats, such as the syntax in the POSIX standard, the Perl syntax, or other user-defined forms. In this chapter, we follow the syntax specified in the dataset collected by Locascio et al. [121] since we evaluate our approach on the dataset created in their work.

Table 5.1 shows the grammar of regex used by Locascio et al. For each rule listed in the table, the left-hand side of each rule is the syntax of the notation, and the right-hand side is the corresponding semantic. For example, the rule "(x)+ → x:≥ 1 time " illustrates that "(x)+" means that the element x appears at least once.

5.2.2 DeepRegex

To leverage learning approaches in the task of NL2Regex, Locascio et al. [121] use an LSTM-based seq-to-seq model to directly transform the NL description into the correspond-

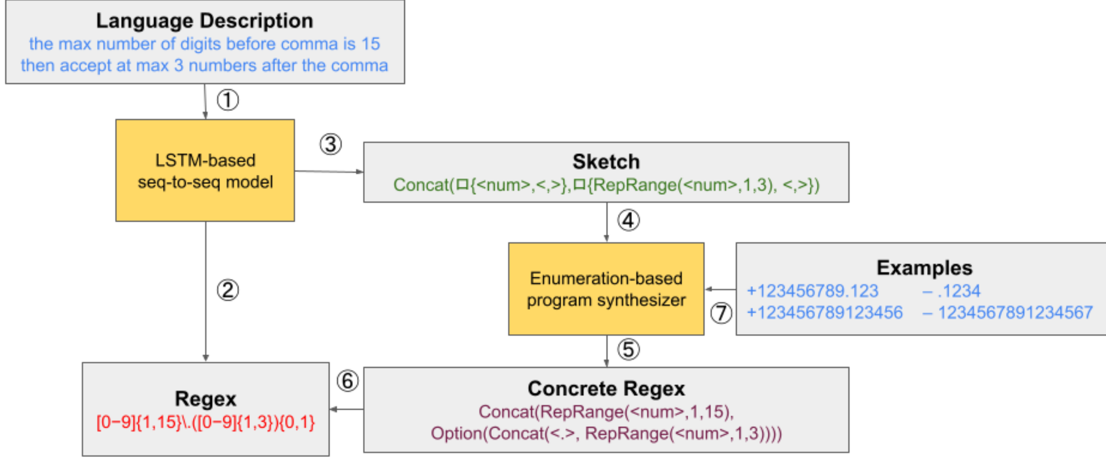


Figure 5.1: DeepRegex and sketch-based NL2SQL example

ing regex, without utilizing any other syntax or semantic information.

Specifically, let $L = l_1, l_2, ..., l_m$ be the input NL description, where each l_i is a word in the vocabulary. Locascio et al. use an RNN model with LSTM cells to generate the regex (another sequence of tokens) $R = s = r_1, r_2, ..., r_n$ as the output. For example, **1** and **2** in Figure 5.1 correspond to the two steps in DeepRegex. The input is the NL description "*the max number of digits before comma is 15 then accept at max 3 numbers after the comma*". Feed the input into the LSTM-based model, and we can get the following output regex:

$$[0-9]\{1,15\}\backslash,([0-9]\{1,3\})\{0,1\} \tag{5.1}$$

5.2.3 DeepSketch

To leverage the multi-modality of inputs (both NL descriptions and example test cases), Ye et al. [116] propose a sketch-based approach. They assume that NL could first be used to build a partial regex (a sketch or a regex with holes), representing the high-level structure. Then they use example test cases to help fill in the details. In our work, we adapt the instantiation of Ye's regex generation tool namely DeepSketch to generate the initial list of candidate regexes.

In Ye's work, to maintain a clearer structure of regex, they introduce the intermediate representation (IR) of regex used in DeepSketch (see Equation (5.2)); the IR's semantic is different but self-explanatory. This IR can be easily transformed to the output regex that

conforms to the syntax in Table 5.1.

$$\begin{aligned} S := C &\mid StartsWith(S) \mid EndsWith(S) \\ &\mid Contains(S) \mid Optional(S) \\ &\mid Repeat(S,k) \mid KleeneStar(S) \\ &\mid RepAtLeast(S,k) \\ &\mid RepRange(S,k_1,k_2) \\ &\mid Concat(S,S) \mid And(S,S) \mid Or(S,S) \\ &\mid \square\{S,...,S\} \end{aligned} \tag{5.2}$$

In DeepSketch, end users provide an NL description and a set of example test cases with positive/negative labels. Then they get a regex that conforms to the syntax in Table 5.1 as the output.

First, DeepSketch feeds the NL description $L = l_1, l_2, ..., l_m$ into a seq-to-seq model that is similar to DeepRegex to generate the sketch (another sequence of tokens) $S = s_1, s_2, ..., s_n$. For example, **1** and **3** in Figure 5.1 correspond to the first step of DeepSketch. The language description "*the max number of digits before comma is 15 then accept at max 3 numbers after the comma*" is fed into the seq-to-seq model, and we get the following sketch:

$$Concat(\square\{<num>,<,>\},\square\{RepRange(<num>,1,3),<,>\}) \tag{5.3}$$

Second, DeepSketch uses a search-based program synthesis technique to fill in the holes (denoted as the square in Equation (5.3), namely synth-operator). Specifically, DeepSketch expands each synth-operator with the options in its following brace recursively until a specified maximum depth d. If the expanded regex contains no holes, then this concrete regex is checked with positive and negative examples. The concrete regex is returned to users if it matches the expected test outputs on the given test inputs, or is discarded otherwise. For example, **4**, **5**, and **7** in Figure 5.1 correspond to the second step of DeepSketch. One way to expand the sketch in Equation (5.3) is shown below:

$$\begin{aligned} &Concat(RepRange(<num>,1,15),\ Option(\\ &\quad Concat(<,>,RepRange(<num>,1,3)))) \end{aligned} \tag{5.4}$$

Finally, DeepSketch transforms the regex in Equation (5.4) into the form in Equation (5.1), as **6** shown in Figure 5.1.

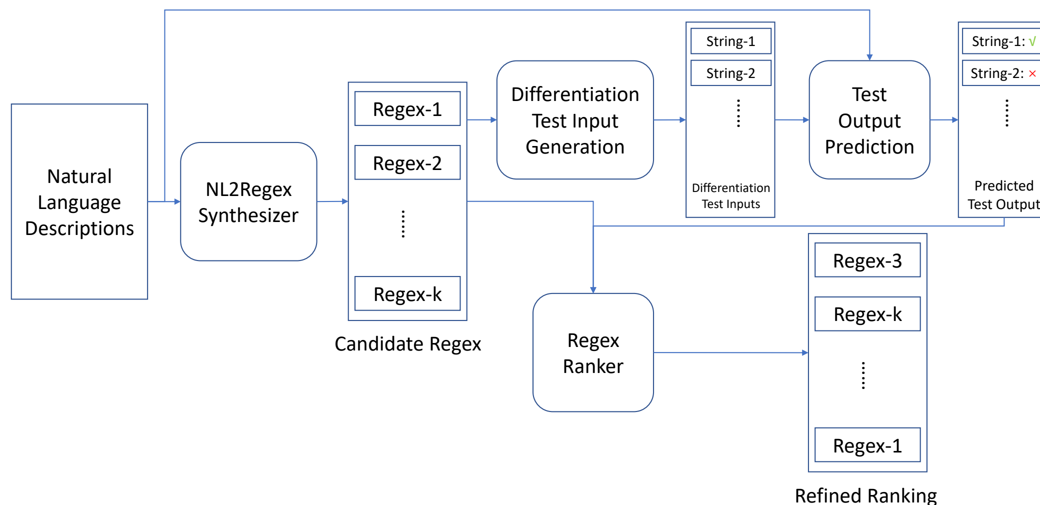


Figure 5.2: The workflow of NRRanker

5.2.4 Pre-trained Code Model in the NL2Code Task

As formally defined in CodeXGLUE [123], a Text2Code or NL2Code task aims to generate code from an NL description. Various approaches have been proposed to address this task [102, 124], which incorporates techniques such as semantic parsing, deep learning, and obtains promising results.

In addition, an NL2Code task can also be directly tackled with large language model (LLM), or more specifically pre-trained code model, which has recently shown great capabilities in code understanding and generation [103, 104, 105, 125, 126, 127]. People pre-train those models on datasets [128, 129] that contain billions of code and NL, and later fine-tune the models for the downstream tasks.

To interact with a pre-trained model, usually a "prompt" is designed as the input to the model [130]. In an NL2Code task, the "prompt" is often designed to obtain the NL description of the task, the additional constraints that the target program should meet, such as input/output examples, and a partial program to complete in the code completion task.

5.3 APPROACH

The overall workflow of NRRanker is shown in Figure 5.2. First, we apply existing NL2Regex systems to get a list of candidate regular expressions (regexes) that are being synthesized. In this work, we use DeepRegex [121] or DeepSketch [116] to obtain the list of candidate regexes discussed in Section 5.2. Then we generate test input strings to differenti-

ate the candidate regexes. For each candidate regex a, we generate strings (for the pair of a and each remaining candidate regex) that can match a but cannot match the remaining candidate regex. These strings are named differentiation test input strings matched by a. Next, we use a pre-trained code model to predict the expected test output of the differentiation strings matched by a. The expected test output indicates whether the "ground-truth" regex would match the test input string, i.e., the string is consistent with the NL description. If the string matched by a is consistent with the NL description, we label it as positive (noted as "$\surd$" in Figure 5.2), and otherwise negative (noted as "$\times$" in Figure 5.2). Based on the ratio of expected output labels being positive among the differentiation strings matched by each candidate regex, we produce a refined ranking of the candidate regexes, with the first regex being the one that is most likely to be correct.

5.3.1 Generation of Differentiation Test Inputs

We first obtain the list of candidate regexes noted as $R = \{r_1, r_2, ..., r_k\}$ using an existing NL2Regex system. We would like to generate test input strings so that we can determine the correctness of each candidate regex by checking whether the matching result with the candidate regex on each string is consistent with the NL description. The simplest way to generate strings would be simply sampling multiple strings from each r_i. Since these sample strings would always match each r_i, we can just check whether these strings are consistent with the NL description. The issue is that, based on our observation, the candidate regexes are semantically similar to the correct answer and also similar to each other; therefore, randomly sampled strings would not be effective in identifying wrong regexes.

Algorithm 5.1 shows the algorithm to generate the strings. First, we notice that both DeepRegex and DeepSketch could generate semantically equivalent regexes in the list of candidates. To reduce redundancy in the calculation, we first check the equivalence of different regex in the list by comparing whether their corresponding minimal Deterministic Finite Automaton (DFA) is equivalent, which is noted as the subroutine `DFAEquivalent`() in the algorithm. After we get the "unique" regex set U, for each pair of regex (r_i, r_j) in the set U, we sample strings from both $r_i\& \sim (r_j)$ and $R_j\& \sim (r_i)$ and insert the sampled strings to the corresponding set $\texttt{DStr}_i$ and $\texttt{DStr}_j$ so that all strings in $\texttt{DStr}_i$ would match r_i and similarly for $\texttt{DStr}_j$. Ideally, if r_i is a correct regex, then all test input strings in $\texttt{DStr}_i$ should have the expected output label of "positive", and for every other R_j, some strings in $\texttt{DStr}_j$ (strings sampled from $r_j\& \sim (r_i)$) should have the expected output label of "negative". So theoretically, if all output labels can be predicted perfectly, NRRanker would always put the correct regex on the top-1 position in the refined ranking.

Algorithm 5.1: Differentiation Test Input String Generation Algorithm.

Input : R: A list of candidate regular expressions $\{r_1, ..., r_k\}$

Outputs: PStr: A set of test input strings that should have the expected output label to be positive.

NStr: A set of test input strings that should have the expected output label to be negative.

U: A list of index $\{p_1, ..., p_l\}$. r_{p_i} is the unique regex under DFA-equivalence. Formally, p_i meets the condition that $\forall j < p_i, \neg\texttt{DFAEquivalent}(r_j, r_{p_i})$.

DStr: A list of l sets of test input strings. For each r_{p_i}, $\texttt{DStr}_i$ is a set of strings whose expected output labels are to be determined.

```
1  PStr ← GetSamplesIntersect(r_1, ..., r_k) ;
2  NStr ← GetSamplesIntersect(~ r_1, ..., ~ r_k) ;
3  U ← ∅ ;
4  for i ← 1 to k do
5      Flag ← True;
6      for j ← 1 to i − 1 do
7          if j ∈ U and DFAEquivalent(r_j, r_i) then
8              Flag ← False;
9              break;
10         end
11     end
12     if Flag = True then U.pushback(i);
13 end
14 l ← U.length();
15 DStr ← {∅, ...(l times)};
16 for i ← 1 to l do
17     for j ← 1 to i − 1 do
18         DStr_i.insert(GetSamplesIntersect(r_{U_i}, ~ r_{U_j})) ;
19         DStr_j.insert(GetSamplesIntersect(r_{U_j}, ~ r_{U_i})) ;
20     end
21 end
22 return PStr, NStr, U, DStr
```

Note that since we do not require users to provide example test cases, to give the pre-trained model necessary contextual information, we sample strings from the intersection of

```
1  def check(input):
2      """
3      Decide whether the input is a string that the
4      max number of digits before comma is 15 then
5      accept at max 3 numbers after the comma
6      """
7
8  def test_check():
9      Assert(check('12345678901234.123'), True)
10     Assert(check('99997773.5'), True)
11     Assert(check('46288923.598'), True)
12     Assert(check('139.931'), True)
13     Assert(check('012345678901234.0'), True)
14     Assert(check('12345678901234.1235'), False)
15     Assert(check('9999a7773.5'), False)
16     Assert(check('46288923.5983'), False)
17     Assert(check('139.0931'), False)
18     Assert(check('5012345678901234.012'), False)
19     Assert(check('012345678901234.012'),
```

Figure 5.3: An example prompt

all r_i (`GetSamplesIntersect`$(r_1, ..., r_k)$ in the algorithm) as the test input strings that are expected to have positive test output label and strings from the intersection of all $\sim (r_i)$ (`GetSamplesIntersect`$(\sim r_1, ..., \sim r_k)$ in the algorithm) as the test input strings that are expected to have negative test output label. Such expected positive/negative test inputs could be wrong if any r_i is not a correct regex. However, in such a case, any ranking would not result in a correct regex output. Therefore we simply treat strings from $r_1 \& r_2 \& ... \& r_k$ and $\sim r_1 \& \sim r_2 \& ... \& \sim r_k$ as positive/negative test inputs.

In the algorithm, we use the `GetSamplesIntersect` function to represent the process of sampling strings from the intersection of different regexes. If we calculate the intersection DFA of all DFAs transformed from each regex and then sample the strings from the intersection DFA, both time and space complexity could be exponential. Therefore, when sampling the strings from the intersection, we randomly sample strings from the first regex and then check whether the other regex accepts the sampled string. We find that this technique is much more efficient in practice.

5.3.2 Test Output Prediction

After the differentiation test input strings are generated, we now need to determine whether the "ground-truth" regex could match the strings. Since we do not have access to the "ground-truth" regex in a practical setting, we could check only whether the string is consistent with the NL description instead. We find this scenario similar to the few-shot learning

```
response = openai.Completion.create(
model="code-davinci-002",
prompt="def␣check(input):\n␣␣␣␣\"\"\"\n␣␣␣␣......", #Same as
    ↪ the prompt text in Fig.5.3
temperature=0,
max_tokens=256,
top_p=1,
frequency_penalty=0,
presence_penalty=0,
stop=[")"]
)
```

Figure 5.4: Sample Python request of the prompt

scenario in NLP, in which LLM such as GPT-3 has shown impressive performance [131]. Therefore, we turn to the CodeX pre-trained code model [119] to predict the expected test output label of the differentiation strings.

As discussed in Section 5.2.4, to interact with the pre-trained model, we need to design a prompt with the necessary input. Figure 5.3 shows the example prompt that we construct with the NL description "*the max number of digits before comma is 15 then accept at max 3 numbers after the comma*". Lines 1-6 are the pseudo-function representing the "ground-truth" regex, with the comments in Lines 3-5 containing the corresponding NL description. Lines 9-13 are the strings in `PStr` as the positive test cases provided to the pre-trained model, while Lines 14-18 are the strings in `NStr` as the negative test cases provided.

This prompt transforms the prediction problem of test output labels into a code completion task. The model reads the prompt as input and is expected to produce either "True)" or "False)" as the predicted output label. The exact Python request to interact with CodeX is shown in Figure 5.4.

5.3.3 Regex Ranker

After obtaining the expected test output labels from the pre-trained model, we rank all candidate regexes in descending order by the ratio of expected test output labels to be positive among $\texttt{DStr}_i$. The ratio indicates the percentage of test input strings in $\texttt{DStr}_i$ (they all match r_i) that are likely to be consistent with the NL description determined by the pre-trained model. If there is a tie, we keep the original order in the initial candidate list.

5.4 EVALUATION

To evaluate the performance of NRRanker, we implement it on the task of natural language to regular expression (NL2Regex). NRRanker takes in the given NL description as the input and applies an existing tool, namely DeepRegex [121] or DeepSketch [116] first to obtain the initial list of candidate regexes being synthesized. In this process, as discussed in Section 5.2, DeepSketch takes in a set of test cases in the training or synthesis process. We do allow DeepSketch to take the same set of test cases to conduct a fair comparison. However, NRRanker itself does not take into account any additional test cases other than the NL description. After applying the three steps explained in Section 5.3 to the initial candidate list, NRRanker would produce a ranked list of the regexes as an output sorted with the first regex in the list to be the most likely correct regex.

Our evaluation aims to answer the following three research questions:

- **RQ5.1.** How accurate is the top regex produced by NRRanker compared to the baselines? In this research question, we calculate the accuracy of the top-1 regex produced by each tool to compare the results.

- **RQ5.2.** How does NRRanker improve the ranking of the candidate regexes? In this research question, we evaluate the accuracy of the entire list of candidate regex produced by each tool in terms of the top-1, 3, 5, 10, 20 accuracy.

- **RQ5.3.** How is the accuracy on the label prediction of expected test output related to the top-1 accuracy of the resulting regex? What factors affect the accuracy on the label prediction of expected test output? In this research question, we evaluate the accuracy of the output labels predicted by the pre-trained model against the ground-truth and compare the accuracy with different test generation strategies.

Datasets. We evaluate our approach on the KB13 and NL-RX datasets in this chapter.

- **KB13.** KB13 [122] contains 824 pairs of NL descriptions and regexes. These pairs are collected in two steps. First, Amazon Mechanical Turk workers are asked to generate NL descriptions of certain lines of files. Then, programmers from oDesk are asked to generate corresponding regexes for each NL description generated by the workers.

- **NL-RX-Synth.** NL-RX-Synth [121] contains 10000 pairs of NL descriptions and regexes collected by Locascio et al.; NL-RX-Synth is is much larger than the size of KB13. To save human effort, Locascio et al. construct a small grammar to randomly generate regexes and their corresponding NL descriptions.

- **NL-RX-Turk.** NL-RX-Turk [121] contains the same 10000 pairs of NL descriptions and regexes and was generated by post-processing the NL-RX-Synth dataset. To generate more "natural" language descriptions, the authors generate the NL-Regex pairs in two steps. First, the authors use their hand-crafted grammar to randomly generate regexes as in NL-RX-Synth. Then, the author ask some programmers to interpret the generated regexes and get their corresponding NL descriptions. In this chapter, we evaluate our approach on only **NL-RX-Turk** (noted as **Turk**) due to the limited budget on using the pre-trained model.

Baseline. We use the DeepRegex or DeepSketch tool to generate initial candidates to NRRanker, and also be the baseline to compare with. We directly adapt the implementation released by Ye et al. in their DeepSketch paper [116]. (They have done a re-implementation of DeepRegex as well.) We note the original DeepRegex model as **DeepRegex** in the evaluation. We use DeepRegex trained with maximum marginal likelihood (MML) on a test set as the baseline for DeepRegex with examples, noted as **DeepRegex-MML** in the evaluation. We use DeepkSketch trained with MML as the baseline for DeepSketch, noted as **DeepSketch** in the evaluation. DeepSketch also allows for the input of additional supervised heuristic sketches. Therefore, we also include this option as a baseline, noted as **DeepSketch-Gold** (DeepSketch with pseudo-gold sketches as prior knowledge) in the evaluation. We make these models to produce their list of k-best regexes. In the evaluation, the k is set to 20.

Training. For training, we directly apply the trained model released by Ye et al. in `https://github.com/xiye17/SketchRegex/tree/master/DeepSketch`. Therefore the training settings and hyperparameters remain the same. For the test output prediction discussed in Section 5.3.2, we use the "code-davinci-002" model provided by OpenAI as the pre-trained code model to predict the output labels.

5.4.1 RQ5.1. Regex Accuracy

In RQ5.1, we aim to evaluate how much NRRanker improves the top-1 accuracy. So for the four initial models (DeepRegex, DeepRegex-MML, DeepSketch, and DeepSketch-Gold) upon which we are applying NRRanker, we calculate the accuracy of the top-1 regex produced by each model and the accuracy of the top-1 regex after NRRanker produces the refined ranked list.

Note that two regexes could be different in syntax but semantically equivalent, indicating that they would match the same set of strings. Here, we denote the top-1 accuracy as

Table 5.2: Top-1 accuracy without/with NRRanker

Naive : Top-1 accuracy of the model.
NRRanker : Top-1 accuracy after applying NRRanker on the model
DeepRegex, **DeepRegex-MM**, **DeepSketch**, **DeepSketch-Gold** : different models explained in **baseline**

	Naive		NRRanker	
Dataset	KB13	Turk	KB13	Turk
DeepRegex	65.6%	58.2%	80.1%	84.0%
DeepRegex-MML	68.2%	62.4%	82.0%	85.7%
DeepSketch	82.5%	84.3%	84.0%	86.8%
DeepSketch-Gold	86.4%	86.2%	89.8%	90.1%

the percentage of the top-1 regex (in the candidate list) that is semantically equivalent to the ground-truth regex. The equivalence is checked by the same `DFAEquivalent` process mentioned in Section 5.3.

Table 5.2 shows the results of the top-1 accuracy with and without applying NRRanker. NRRanker is able to achieve 89.8% and 90.1% top-1 accuracy on each dataset, respectively, when taking the best model's synthesized regexes as the initial candidates. Overall, we can see that NRRanker improves the top-1 accuracy by 11.1% on average. On the KB13 dataset, NRRanker improves the top-1 accuracy by 8.3% on average, and on the Turk dataset, NRRanker improves the top-1 accuracy by 13.9% on average.

When looking into the specific models, NRRanker has much more improvement over the candidate list produced by DeepRegex than DeepSketch, with an improvement of 19.4% on DeepRegex and DeepRegex-MML compared to 2.8% on DeepSketch and DeepSketch-Gold on average. The reason is that DeepRegex does not take an example test set as part of input when performing the NL to regex translation. Therefore, applying NRRanker on DeepRegex's result would improve the top-1 accuracy by a larger margin.

We also notice that despite the results of DeepSketch-Gold having a higher top-1 accuracy than the results of DeepSketch, NRRanker's improvement on DeepSketch-Gold is higher (4.2% compared to 2.0% on average). The reason is that DeepSketch-Gold would use a pseudo-gold sketch that is generated from heuristics and the ground-truth regex as an additional input. Therefore, the regexes synthesized by DeepSketch-Gold would actually be similar in the sketch structure to the ground-truth regex. Meanwhile, the regexes synthesized by DeepSketch would include some candidate regexes that are unlikely to be the correct answer. Therefore, a refined ranking of the DeepSketch-Gold results would be more helpful.

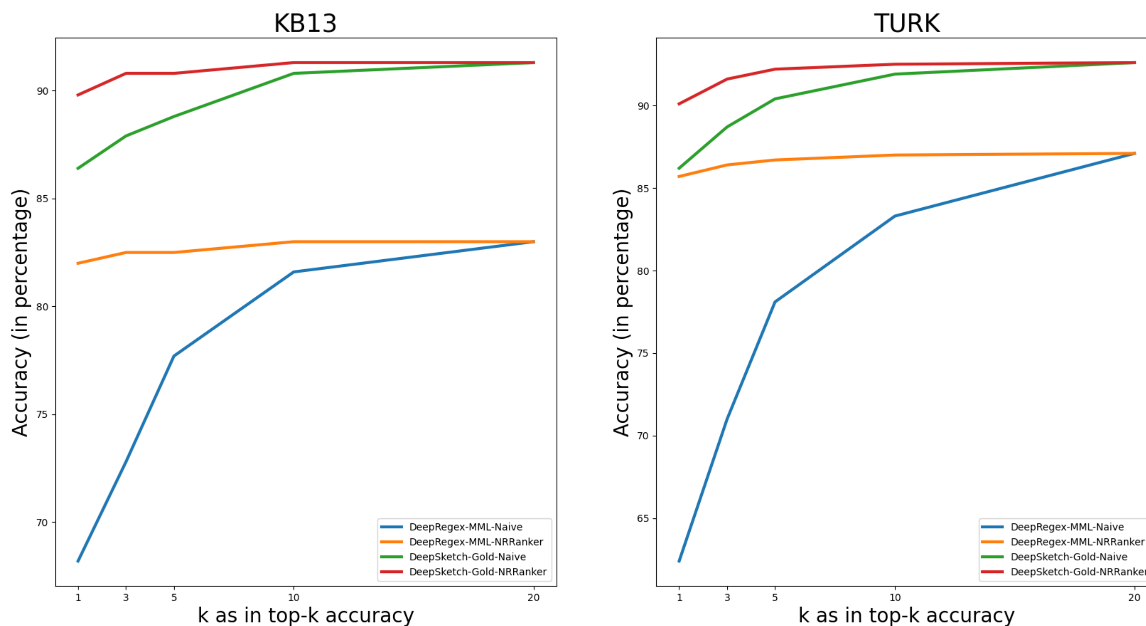


Figure 5.5: Top-k accuracy comparison on KB13 and TURK

5.4.2 RQ5.2. Ranking Efficiency

In RQ5.2, we aim to evaluate how NRRanker improves the ranking of the candidate regex list. So for the four models, we calculate the accuracy of the top-1,3,5,10,20 regexes in the candidate list produced by each model with or without applying NRRanker. Top-k accuracy is defined as the percentage of candidate lists in which any of the top-k regexes is semantically equivalent to the ground-truth regex.

Table 5.3 and Figure 5.5 show the top-k accuracy without/with NRRanker on the KN13 and Turk datasets. Note that the top-20 accuracy remains the same after NRRanker is applied since NRRanker does not synthesize any new candidate regex. Overall, we can see that NRRanker does a great job in terms of ranking a correct regex higher. If we define the relative accuracy to be the ratio of top-1 accuracy divided by top-20 accuracy, indicating that the conditional top-1 accuracy if at least one semantically correct regex is among the list of top-20 candidates. (If any regex in the top-20 is not correct, then no ranking of the candidates would be helpful). On average, NRRanker can achieve a relative accuracy of 98.2%, compared to a relative accuracy of 76.2% without NRRanker.

We also find that compared to DeepRegex, DeepSketch is able to produce a candidate list with a better ranking, with a relative accuracy of 94.7% compared to 75.6% with DeepRegex. The most possible cause is still the help of the test cases that DeepSketch is leveraging.

Table 5.3: Top-k accuracy without/with NRRanker

(a) KB13

	Naive					NRRanker				
	Top-1	Top-3	Top-5	Top-10	Top-20	Top-1	Top-3	Top-5	Top-10	Top-20
DeepRegex	65.6%	71.8%	76.9%	80.1%	81.1%	80.1%	80.6%	80.6%	80.6%	81.1%
DeepRegex-MML	68.2%	72.8%	78.3%	82.0%	83.0%	82.0%	82.5%	82.5%	83.0%	83.0%
DeepSketch	82.5%	83.5%	84.4%	85.0%	85.4%	84.0%	84.4%	85.0%	85.0%	85.4%
DeepSketch-Gold	86.4%	87.9%	89.3%	90.8%	91.3%	89.8%	90.8%	90.8%	91.3%	91.3%

(b) Turk

	Naive					NRRanker				
	Top-1	Top-3	Top-5	Top-10	Top-20	Top-1	Top-3	Top-5	Top-10	Top-20
DeepRegex	58.2%	70.2%	77.2%	82.4%	86.0%	84.0%	85.0%	85.6%	85.9%	86.0%
DeepRegex-MML	62.4%	71.8%	78.9%	83.3%	87.1%	85.7%	86.4%	86.7%	87.0%	87.1%
DeepSketch	84.3%	86.2%	87.2%	88.4%	89.1%	86.8%	87.4%	88.2%	88.8%	89.1%
DeepSketch-Gold	86.2%	88.7%	90.4%	91.9%	92.6%	90.1%	91.6%	92.2%	92.5%	92.6%

5.4.3 RQ5.3. The Prediction Accuracy of Expected Test Output Labels

In RQ5.3, we aim to evaluate the accuracy of predicting the expected test output labels. The accuracy is defined as the percentage of the output labels predicted by the pre-trained model being correct. We also would like to understand the connection between the output label prediction accuracy and the top-1 accuracy of the resulting candidate regexes. To calculate the output label prediction accuracy, for each differentiate test input string generated, we check whether the ground-truth regex would match the string and compare with the predicted label. We compare NRRanker with three other strategies in the task of output label prediction. The first baseline is noted as "Random-String", in which instead of generating the differentiation test input strings discussed in Section 5.3.1, we simply sample strings from each candidate regex and let the pre-trained model determine the expected test output of the test input string. The second baseline is noted as "No-PStr-NStr". The prompts that we generate in this baseline would not include the two sets of strings "`PStr`" and "`NStr`" in Algorithm 5.1 that can be seen in Lines 9-18 in the example prompt shown in Figure 5.3. Instead, the prompt in this baseline would include only the string whose expected test output label is to be predicted. The third baseline is "Majority-Vote", in which instead of using the pre-trained model to predict the expected output label, we match the test input string with each candidate regex and use the label of the majority result to be the expected output label.

Table 5.4 and Figure 5.6 show the evaluation results. The first finding is that NRRanker actually does not achieve a very high accuracy in terms of the output label accuracy, with an overall accuracy of 80.8%. This finding gives us the insight that although the expected

Table 5.4: Expected Test Output label accuracy and Top-1 accuracy with different strategies

Random-String : Sample random strings instead of differentiation strings for each regex.
No-PStr-NStr : Not providing the expected-positive and expected-negative examples to the pre-trained model
Majority-Vote : Use the majority vote as the expected label of each string.

(a) KB13

	NRRanker		Random-String		No-PStr-NStr		Majority-Vote	
Accuracy	Label	Top-1	Label	Top-1	Label	Top-1	Label	Top-1
DeepRegex	85.0%	84.0%	89.8%	80.3%	67.6%	60.1%	51.5%	52.9%
DeepRegex-MML	83.7%	85.7%	90.4%	82.8%	67.2%	61.7%	52.1%	53.2%
DeepSketch	81.1%	86.8%	89.2%	84.1%	60.2%	58.2%	54.4%	58.7%
DeepSketch-Gold	79.9%	90.1%	88.1%	85.8%	59.5%	63.5%	56.3%	60.2%

(b) Turk

	NRRanker		Random-String		No-PStr-NStr		Majority-Vote	
Accuracy	Label	Top-1	Label	Top-1	Label	Top-1	Label	Top-1
DeepRegex	85.0%	84.0%	89.8%	80.3%	67.6%	60.1%	51.0%	47.3%
DeepRegex-MML	83.7%	85.7%	89.6%	82.8%	67.2%	61.7%	51.6%	48.8%
DeepSketch	81.1%	86.8%	89.2%	84.1%	60.2%	58.2%	55.2%	52.3%
DeepSketch-Gold	79.9%	90.1%	88.1%	85.8%	59.5%	63.5%	58.2%	54.9%

test output prediction in this chapter does not achieve very high accuracy, it can still help produce a substantially better ranking of the candidates, thus improving the top-1 accuracy.

Another interesting finding is that the "Random-String" strategy actually achieves higher output label accuracy than the differentiation string strategy used by NRRanker, with the overall accuracy being 88.9% on average. The possible explanation is that the strings randomly sampled from each regex would be less complicated than the strings that need to differentiate two regexes. Somehow counter-intuitively, higher output label accuracy does not lead to higher top-1 accuracy when we actually compare the top-1 accuracy of the "Random-String" strategy against NRRanker. This result is potentially due to the fact that randomly sampled strings are less effective in eliminating the plausible candidate regexes since most strings matched by the plausible candidate regexes are highly likely to be also matched by the ground-truth regex. We can see that the top-1 accuracy of the "Random-String" strategy on the DeepSketch model is similar to the top-1 accuracy of the DeepSketch model without applying NRRanker (column "Naive" in Table 5.2). The reason is that the randomly sampled test input strings serve similar roles to the positive/negative test input strings that are already leveraged in the DeepSketch model. Therefore using the "Random-String" strategy does not improve the top-1 accuracy in such cases. However, since

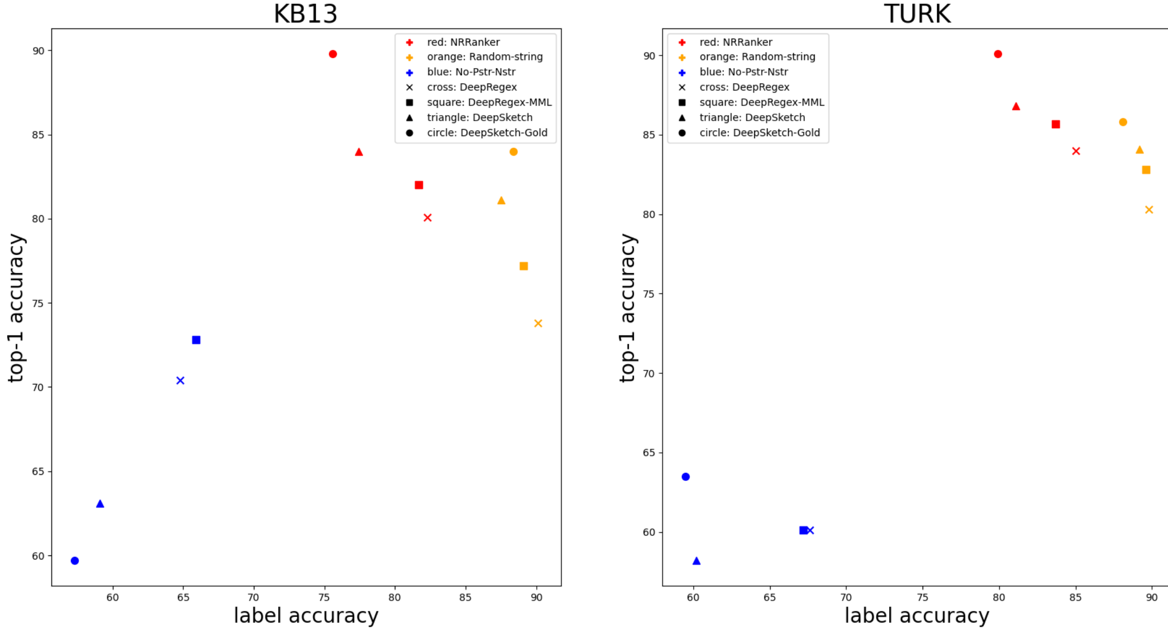


Figure 5.6: Top-1 Accuracy versus Expected Test Label Accuracy on KB13 and Turk

the DeepRegex model does not take advantage of example test cases, the "Random-String" strategy could still improve the top-1 accuracy substantially.

On the other hand, the "No-PStr-NStr" strategy has much worse accuracy in terms of both label prediction accuracy and top-1 accuracy. The label prediction accuracy is only 62.7%. (Note that randomly guessing the output label would have an expected accuracy of 50%) This result gives us the insight that the few-shots learning scenario does help the pre-trained code model a lot in understanding the task that it is given; such insight is already shown by NLP researchers [131]. Such a low accuracy in label prediction has made the refined ranking even worse than original list of candidates.

For the "Majority-Vote" strategy, we find that both the label prediction accuracy and the top-1 accuracy are really poor. The reason is that usually the correct regex would differ with other candidate regex when matching with the differentiation strings. So, using the majority vote as the estimate for the output label often the time would be incorrect.

We also notice that the output label prediction accuracy of NRRanker is lower in the DeepSketch model, with the average accuracy in DeepRegex and DeepSketch being 83.2% and 78.5%, respectively. The reason is that the DeepSketch model generates regexes that are semantically more similar to each other. Therefore, it would be harder to predict the expected test output in the input strings of the differentiation test for more similar regexes. The randomly sampled strings in the "Random-String" strategy do not have such a difficulty gap, so there is not much difference in label accuracy in DeepRegex versus DeepSketch

(89.5% versus 88.3%) in the "Random-String" strategy.

5.5 DISCUSSION

In our evaluation, one observation is that despite NRRanker achieving only 80.8% accuracy on the expected test output labels predicted, it still improves the ranking of the candidates by a large margin. This result leads to the insight that even getting access to a non-perfect oracle that is able to predict test output labels for new test inputs would be helpful in eliminating plausible candidate programs being synthesized in NL2Code tasks. Therefore, Chen et al. [132] and Inala et al. [133] have both proposed new approaches to simulate an oracle as discussed in Section 6.3.2. Compared to their approaches, our proposed approach in this chapter does not require training any additional model while leveraging a pre-trained code model to predict the expected test output label. While directly letting the pre-trained model produce a complete program still remains a major challenge [104, 105, 125], tasks such as generating additional test inputs or predicting test output labels could be simpler for the pre-trained model and thus more helpful in the synthesis process.

The second observation is that the test inputs on which the candidate programs would potentially have different test outputs could be more helpful than the test inputs on which the expected test outputs are more accurate, as shown in Section 5.4.3. This observation leads to the insight that we should try to generate test cases that can differentiate behaviors of the candidate programs other than simply generating the test cases based on the NL description. In this chapter, our proposed NRRanker approach functions more like the post-processing of the results produced by other tools. A potential generalization of NRRanker would be deeper integration into the synthesis process. Current program synthesis approaches that leverage multi-modality synthesis often work on a fixed set of test cases [115, 117]. If we are able to dynamically verify the current candidates during the synthesis process, we believe that it could further improve the top-1 accuracy in the result. However, one major challenge is that example-based synthesis (program-by-example, PbE) often relies on the correctness of the test cases [3]. It would be challenging to adapt such approaches to a set of non-perfect test cases.

5.6 SUMMARY

In this chapter, we have presented a novel NL2Regex approach named NRRanker to synthesize a regular expression from a NL description given by users. The key insights

underlying our approach are to generate test inputs that can differentiate behaviors of the candidate regular expressions being synthesized and generate their expected test outputs from the NL description. NRRanker further integrates an existing NL2Code system with the preceding test generation using pre-trained models. We have evaluated our approach on two most commonly used datasets to show the effectiveness of NRRanker. NRRanker achieves the top-1 accuracy of 89.8% and 90.1% in the two datasets, respectively, outperforming the state-of-the-art approaches by 11.1% on average.

CHAPTER 6: RELATED WORK

6.1 INPUT GRAMMAR INFERENCE

Our approach draws on both existing work in grammar inference and techniques used in machine learning and automated test generation.

Inferring Input Grammars. Höschele et al. [27, 134] propose the Autogram tool based on dynamic taint tracing to extract syntactic entities of a given seed input. By tracing the data flow of particular characters of the seed input in the parsing method of the target program, Autogram decomposes formats into meaningful fields. However, since Autogram traces only paths taken by a given seed input, Autogram requires that the given set of seed inputs capture all meaningful features of the input grammar.

Coverage-Guided Fuzzing. In recent years, coverage-guided fuzzing has achieved substantial success with tools such as American Fuzzy Lop (AFL)[4] and Fairfuzz [40]. However, many fuzzers fail to explore paths reaching which requires a difficult check such as string-equality comparisons. Recent advances in grey-box fuzzing have utilized lightweight program analysis in order to mitigate this problem. Steelix [135] tracks progress in string-comparison checks in order to incrementally discover inputs that can bypass these checks. Angora [136] solves path constraints via a searching algorithm based on gradient descent. Other tools, such as Driller [137], use more heavyweight program analysis in order to bypass these checks by symbolically executing an intermediate representation and solving constraints in order to bypass these checks. REINAM is able to discover constants that are capable of bypassing these checks as a part of the grammar-inference process as shown in REINAM's inference of the url protocol, which necessarily requires discovery of protocol names such as "mailto" and "https", which would be unlikely to be discovered by grey-box approaches.

Machine Learning for Fuzzing. Godefroid et al. [32] use a recurrent neural network to learn an input model and generate inputs for fuzzing with the Learn & Fuzz algorithm. They have recently formalized fuzzing as a reinforcement learning problem [138]. Mingyuan et al. [139] propose an approach to treat the mutation generation problem as a multi-armed bandit problem to allow dynamically mutation strategy adjustment. There approach could further extend REINAM's mutation process.

PCFG Inference from Examples. The problem of inferring a probabilistic context free grammar (PCFG) from a set of examples has been studied extensively for the purpose of natural language processing. Belz [140] extends standard split/merge grammar inference

[4]http://lcamtuf.coredump.cx/afl/

techniques in order to optimize grammars from examples. However, it requires a large corpus of annotated examples, which is not viable for inferring program-input grammars. Scicluna and Higuera [141] propose an unsupervised approach to grammar inference, which naturally does not need annotated examples. While they show that this approach works for small samples with respect to NLP standards (the polynomial number of examples with respect to the number of productions in ideal grammars), this approach is still not viable for inferring program-input grammars. REINAM addresses these issues by expanding its corpus by analyzing the target program before synthesizing its grammar.

6.2 ROOT CAUSE ANALYSIS

Anomaly Detection. Anomaly detection aims to detect potential issues in the system. Anomaly detection approaches using time series data can generally be categorized into three types: (1) batch-processing and historical analysis such as Surus [142]; (2) machine-learning-based, such as Donut [60]; (3) use of adaptive concept drift, such as StepWise [81].

GROOT currently uses a combination of manually written thresholds, statistical models, and machine learning (ML) algorithms to detect anomalies. Since our approach is event-driven, as long as fairly accurate alerts are generated, GROOT is able to incorporate them.

Root Cause Analysis. Traditional RCA approaches (e.g., Adtributor [143] and HotSpot [144]) find the multi-dimensional combination of attribute values that would lead to certain quality of service (QoS) anomalies. These approaches are effective at discrete static data. Once continuous data are introduced by time-series information, these approaches would be much less effective.

To address these difficulties, there are two categories of approaches based on ML and graph, respectively.

ML-based RCA. Some ML-based approaches use features such as time series information [70, 77] and features extracted using textual and temporal information [51]. Some other approaches [60] conduct deep learning by first constructing the dependency graph of the system and then representing the graph in a neural network. However, these ML-based approaches face the challenge of lacking training data. Gan et al. [58] proposed Seer to make use of historical tracking data. Although Seer also focuses on the microservice scenario, it is designed to detect QoS violations while lacking support for other kinds of errors. There is also an effort to use unsupervised learning such as GAN [60], but it is generally difficult to simulate large, complicated distributed systems to give meaningful data.

Graph-based RCA. A recent survey [50] on RCA approaches categorizes more than 20 RCA algorithms by more than 10 theoretical models to represent the relationships between

Table 6.1: The scale of experiments in existing RCA approaches' evaluations (QPS: Queries per second)

Approach	Year	Scale	Validated on Real Incidents?
FChain [68]	2013	<= 10 VMs	No
CauseInfer [69]	2014	20 services on 5 servers	No
MicroScope [80]	2018	36 services, ∼5000 QPS	No
APG [77]	2018	<=20 services on 5 VMs	No
Seer [58]	2019	<=50 services on 20 servers	Partially
MicroRCA [63]	2020	13 services, ∼600 QPS	No
RCA Graph [72]	2020	<=70 services on 8 VMs	No
Causality RCA [78]	2020	<=20 services	No

components in a microservice system. Nguyen et al. [68] proposed FChain, which introduces time series information into the graph. But they still use server/VM as nodes in the graph. Chen et al. [69] proposed CauseInfer, which constructs a two-layered hierarchical causality graph. It applies metrics as nodes that indicate service-level dependency. Schoenfisch et al. [71] proposed to use Markov Logic Network to express conditional dependencies in the first-order logic, but still build dependency on the service level. Lin et al. [80] proposed Microscope, which targets the microservice scenario. It builds the graph only on service-level metrics, so it cannot get full use of other information and lacks customization. Brandon et al. [72] proposed to build the system graph using metrics, logs, and anomalies, and then use pattern matching against a library to identify the root cause. However, it is difficult to update the system to facilitate the changing requirements. Wu et al. [63] proposed MicroRCA, which models both services and machines in the graph and tracks the propagation among them. It would be hard to extend the graph from machines to the concept of other resources, such as databases for Groot.

As mentioned in Section 3.1, by using the event graph, Groot mainly overcomes the limitations of existing graph-based approaches in two aspects: (1) build a more accurate and precise causality graph use the event-graph-based model; (2) allow adaptive customization of link construction rules to incorporate domain knowledge in order to facilitate the rapid requirement changes in the microservice scenario.

Our Groot approach uses a customized page rank algorithm in the event ranking, and can also be seen as an unsupervised ML approach. Therefore, Groot is complementary to other ML approaches, as long as they can accept our event causality graph as a feature.

Settings and Scale. The challenges of operational, scale, and monitoring complexities are observed, especially being substantial in the industrial settings. Hence, we believe that the target RCA approach should be validated at the enterprise scale and against actual incidents for effectiveness.

Table 6.1 lists the experimental settings and scale in existing RCA approaches' evaluations. All the listed existing approaches are evaluated in a relatively small scenario. In contrast, our experiments are performed upon a system containing 5,000 production services on hundreds of thousands of VMs. On average, the sub-dependency graph (constructed in Section 3.3.1) of our service-based data set is already 77.5 services, more than the total number in any of the listed evaluations. Moreover, 7 out of the 8 listed approaches are evaluated under simulative fault injection on top of existing benchmarks such as RUBiS, which cannot represent real-world incidents; Seer [58] collects only the real-world results with no validations. Our data set contains 952 actual incidents collected from real-world settings.

6.3 NL2CODE

Natural Language to Code Natural language has been an important source of input specification for the program synthesis problem due to its flexibility and ease of use for users of different backgrounds [14].

In the task of NL descriptions to general-purpose programs, pre-trained models have recently shown their strength in code understanding and generation. For example, AlphaCode [104], Codex [105] and PANGU-CODER [125] are three of the most powerful and popular closed-source tools to transform natural language specifications to programs. There are also many open-source big models trying to address the NL2Code problem such as CODEGEN [103], INCODER [126], and PolyCoder [127].

To focus more on specific program synthesis tasks, there are tasks such as NL2SQL and NL2Regex, which aim to transform natural language into domain-specific languages instead of general-purpose languages. In these specified tasks, existing approaches have shown better accuracy.

For example, in the NL2SQL task, Iyer et al. [145] propose to directly use neural sequence models to transform natural language into the corresponding SQL. Later, Iyer et al. [124] propose an idiom-based semantic parsing approach to transform NL into SQL. Moreover, Finegan-Dollak et al. [146] provide new datasets and new ways to more properly evaluate these text-to-SQL systems.

In the NL2Regex task, Kushman et al. [122] use a probabilistic combinatory categorical grammar model to transform NL to a regex. And Locascio et al. [121] use a neural translation model to directly transform a sequence of NL tokens to a sequence of regex tokens without utilizing any domain-specific crafting. Later Zhong et al. [147] propose a semantics-based approach to optimize the semantics-based objective for generating regular expressions from NL descriptions.

Test Generation in NL2Code As discussed in Section 5.1, a set of test cases could substantially improve the accuracy of the NL2Code task. Test cases such as examples have been an important part of program synthesis. Therefore, researchers have proposed different approaches to help with test case generation in NL2Code tasks.

CODET [132] can generate code solutions and corresponding test cases with different prompts. And the best solution is selected based on a compound metric of how many test cases each solution can pass and how many solutions can pass each test case (called dual execution agreement in Chapter 5).

Inala et al. [133] propose a fault-aware ranker that takes into account a context and a program generated from this context and predicts the different types of execution information without actually executing the program. Inala et al. show that by trying to predict the execution information (such as the compile/runtime error type), the ranker can better rank the sampled programs from a code generation model.

Syntax Guided Synthesis. The constrained syntax-guided synthesis problem is an extension of the syntax-guided synthesis (SyGuS) problem first introduced by Alur el al. [148]. Our constrained SyGuS problem asks for a program that is not only generated by the given grammar, but also uses specific rules and non-terminals of the grammar. Successful solution strategies for SyGuS are based on bottom-up enumeration [149, 150, 151], model-based quantifier instantiation [152], and top-down search over the grammar [153]. Hu et al. [154] consider QSyGuS, a variant of the SyGuS problem where a cost model given by a weighted tree automaton assigns costs to programs, and the task is to generate the minimal cost program that satisfies the semantic constraint. However, the solution used by Hu et al. is infeasible in our setting because of the presence of derivation constraints.

6.3.1 Natural Language to Visualization

Natural Language to Visualization. The idea of using natural language (NL) as a query interface for visualization is getting popular with the development in NL2Code.

Tong et al. [155] present DataTone, a mixed-initiative approach to address the ambiguity problem in NL interfaces for visualization. Unfortunately, because DataTone is not publicly available, we could not perform a direct comparison between DataTone and NL2Viz. Zhang et al. [156] propose Text-to-Viz, whose usage scenario is quite different from ours. Text-to-Viz is a visualization recommendation tool that focuses on data exploration. It does not support precise NL instructions to a specific visualization. Instead, the user's input works as a guide to explore charting options on certain columns or combination of columns. We find it not fair to compare Text-to-Viz's accuracy on our dataset as it is

not designed to produce visualization with the NL instruction provided. Similarly, Sun et al. [157] propose ARTICULATE, a two-step process to generate visualization from the given NL instruction. First, it parses the NL instruction into commands using supervised learning. It then generates visualizations for the commands using heuristics. ARTICULATE also focuses on data exploration instead of synthesizing precise visualization according to the given NL instruction.

Narechania et al. [101] propose NL4DV, which has similar functionalities as NL2VIZ. It is also integrated into the Jupyter Notebook environment while producing the results in the Vega-Lite format [158]. However, NL4DV relies on only the NL instruction to generate the visualization. It checks the data only to identify the database entities in the NL instruction without leveraging other contextual information from the data/program context. Similarly, it also lacks the ability to create the necessary data preprocessing steps. Luo et al. [92] publish a public dataset named NL2VIS consisting of 25,750 (NL, VIS) pairs. They propose a NL2SQL-to-NL2VIS model to translate the (NL, SQL) pairs in the popular Spider [98] dataset to the (NL, VIS) pairs. Luo et al. [92] also propose a learning-based approach named SEQ2VIS based on the SEQ2SEQ model [159] used in NL2SQL tasks. They evaluate their approach on the dataset by comparing it with two other approaches namely NL4DV [101] and DEEPEYE [100], which is a keyword-based approach previously proposed by them too. They find their approach largely outperforms the other two approaches. However, since the Spider dataset is an NL2SQL dataset. Their approach focuses only on how the output is calculated using the data transformations defined in the SQL query. The NL instructions in the NL2VIS dataset completely ignore important options of visualizations such as formats and legends. Despite this dataset's limitations, we also evaluate NL2VIZ on this dataset in Section 4.4.3.

It is worth noting that unlike NL2VIZ, the preceding related approaches do not support further refinement on the result, limiting the ability of users to further modify or reuse the results later in other tasks.

Rong et al. [160] propose CODEMEND, which uses neural network to infer the correspondence between the given NL query and functions/parameters in the target visualization program. Similarly, Setlur et al. [161] propose EVIZAS, which allows users to refine existing visualizations by asking questions or direct manipulation. However, both approaches lack the ability to generate complete visualization code and also cannot generate the necessary data preprocessing steps. These approaches can be seen as complementary with NL2VIZ: these approaches can be combined with the interactive technique in NL2VIZ to provide better user experience after the first-shot query.

Visualization Recommendation. Visualization recommendation focuses on producing

the recommended visualization encoding based on design domain knowledge [162].

Dominik et al. [163] present DRACO, which represents a visualization as a set of logical facts and thus converts visualization design patterns into a set of constraints. It then uses constraint solving to recommend the best visualization scheme based on the collection of domain knowledge. Ding et al. [164] present QUICKINSIGHTS to discover interesting patterns from multi-dimensional datasets by formalizing the notion of interesting patterns (insights) and present them as visualizations. Siddiqui et al. [165, 166] propose an interactive visual analytic platform named ZENVISAGE to find desired visual patterns from large datasets. It extends VISPEDIA proposed by Chan et al. [167], which performs only a keyword query of collected graphs.

While the output of NL2VIZ is also a visualization, the focus is different. Visualization recommendation tries to follow visualization design patterns. NL2VIZ focuses on eliminating the ambiguity in NL instructions by bringing insights from data. NL2VIZ is also extensible and can be integrated with existing visualization recommendation approaches.

6.3.2 Natural Language to Regex

Regular expressions (regexes) are very useful for pattern matching. However, end users might find it challenging to write a proper regex for their tasks. To reduce this difficulty, various approaches have been proposed.

For example, Kushman et al. [122], Locascio et al. [121], Zhong et al. [147] and Park et al. [168] propose different approaches to transform the NL description to the corresponding regex. In addition, many existing methods use test cases (namely example strings) to infer the correct regex for users [169, 170, 171, 172, 173, 174, 175, 176]. Moreover, there also exists a lot of work combining the power of both NL descriptions and test cases to generate the desired regexes [115, 116, 117].

More specifically, Locascio et al. [121] use a neural translation model to directly transform a sequence of NL tokens to a sequence of regex tokens without utilizing any domain-specific crafting. Later Zhong et al. [147] propose an approach to optimize the semantics-based objective for generating regexes from NL specifications. Then Chen et al. [117] and Ye et al. [116] both propose an approach to first parse the NL description into a sketch (partial regex with holes) and then to concretize the sketch with the help of examples. Most recently, Li et al. [115] propose TRANSREGEX, which is the first to treat the NLP-and-example-based regex synthesis problem as the problem of NLP-based synthesis with regex repair.

CHAPTER 7: CONCLUSIONS AND FUTURE WORK

Constraints are key components in code analysis and generation tasks. Probabilistic constraints, with the ability to describe the confidence level of each constraint, can help improve the performance of code analysis and generation systems by incorporating knowledge with uncertainty. However, it is challenging to extract and apply the probabilistic constraints in code analysis and generation tasks.

In this dissertation, we investigate different approaches to leverage the probabilistic constraints in code analysis (Chapters 2 and 3) and generation (Chapters 4 and 5). First, this dissertation investigates the problem of grammar-guide fuzz testing in code analysis (Chapter 2). Second, this dissertation investigates the problem of root cause analysis in code analysis of microservice infrastructure (Chapter 3). Third, this dissertation investigates the problem of natural language to visualization in code generation, more specifically NL2Code (Chapter 4). Lastly, this dissertation investigates the problem of natural language to regular expression in NL2Code (Chapter 5).

In the problem of grammar-guided fuzz testing, this dissertation has presented REINAM, an input-grammar inference framework. In REINAM, we learn the probabilistic constraints of the program input in the form of a PCFG. We use reinforcement learning to leverage the feedback from the program under test to adjust the probability distribution for the PCFG. We evaluate REINAM on 11 real world benchmarks with manually written grammars used in real world scenarios. Our evaluation results show that REINAM outperforms Glade (a state-of-the-art tool) in terms of precision, recall, and fuzz testing coverage for most of the benchmarks. On average, REINAM improves code coverage by 18% compared to the baseline fuzzer and by 5% compared to Glade's grammar-based fuzzer.

In the problem of root cause analysis, this dissertation has presented Groot, a graph-based approach of root cause analysis (RCA) for microservice architecture. In Groot, the task is to find the root cause of a reported anomaly in a microservice system. We use a set of rule constraints to define the events and causalities between the events in the system and then construct an event-based causality graph to perform RCA. We implement and evaluate Groot on eBay's production system that serves more than 159 million users and features more than 5,000 services. Our experimental results show that Groot is able to achieve 95% top-3 accuracy and 78% top-1 accuracy for 952 real production incidents collected over 15 months with the help of event-building rule constraints.

In the problem of natural language to visualization, this dissertation has presented NL2Viz. In this work, we specifically focus on extracting constraints from natural language input as

the hard constraints and complement the hard constraints using constraints extracted from other sources of input such as code or data context as the soft constraints. We implement NL2Viz as a plug-in tool in the popular Jupyter Notebook environment. Our evaluation results show that NL2Viz achieves the accuracy comparable to state-of-the-art tools, and data scientists find the tool easy to use in real world scenarios.

In the problem of natural language to regular expressions (NL2Regex), this dissertation has presented an NL2Regex tool namely NRRanker. In NRRanker, we generate probabilistic constraints, which are additional test input strings that can differentiate behaviors of the candidate regular expressions being synthesized and corresponding expected test outputs generated from the NL description. NRRanker integrates an existing NL2Regex system with the preceding test generation using a pre-trained model. We have evaluated our approach on two most commonly used datasets to show the effectiveness of NRRanker. NRRanker achieves the top-1 accuracy of 89.8% and 90.1% on the two datasets, respectively, outperforming state-of-the-art approaches by 11.1% on average.

Based on the empirical evaluations of our proposed approaches in the four tasks, we find that the guidance of probabilistic constraints substantially improves the effectiveness of constraint-based approaches on the tasks of code analysis and generation.

7.1 FUTURE WORK

In this section, we discuss future work that can further advance the work presented in this dissertation.

Probability Quantification for Probabilistic Constraints. As mentioned in Section 1.1, although the subject investigated in this dissertation is probabilistic constraints, due to certain limitations, we are not able to explicitly assign accurate probability distributions to the constraints. For example, we are not able to assign the probability for the additional test input-output pairs described in Chapter 5. Existing approaches [132, 133] have been proposed to train additional models for test generation in NL2Code. If we can also train additional models to measure the accuracy of the additional tests generated, it would be more helpful in filtering and ranking the candidate programs when we can estimate the confidence level of each additional test being generated.

Automated Probabilistic Constraints Generation. In the tasks of code analysis and generation discussed in this dissertation, we treat constraints as an intermediate task for the final tasks such as fuzz testing or visualization generation. To design the syntax and

semantics of the constraints, we need to carefully consider the domain knowledge and constraints defined by existing systems. Such process requires a lot of manual effort. Existing approaches [4, 177] have been proposed under a specific set of domain-specific languages (DSLs). If we can use a DSL given by users to specify the syntax of the constraints, it would be helpful to have a model that can automatically infer the specific constraints and corresponding semantics from the examples provided by the users.

Constraint-based Language Model Training. With the recent development in large language models [178], large language models trained with a code corpora [103, 104, 105, 125, 126, 127] have been shown to be useful in various programming tasks including code generation. However, existing code-based large models treat the code as simple natural language. They just replace the natural language tokens with code-based tokens (identifiers, keywords and so on). This treatment results in that these code-based large models consider only the syntactic features. It would be helpful to also have the constraints in the training corpora and design a specific mechanism for models to incorporate the constraints so that semantic features can also be considered by the models.

REFERENCES

[1] D. E. Knuth, "Semantics of context-free languages," *Mathematical systems theory*, vol. 2, no. 2, pp. 127–145, 1968.

[2] D. Barstow, "A knowledge base organization for rules about programming," *ACM SIGART Bulletin*, no. 63, pp. 18–22, 1977.

[3] S. Gulwani, O. Polozov, R. Singh et al., "Program synthesis," *Foundations and Trends® in Programming Languages*, vol. 4, no. 1-2, pp. 1–119, 2017.

[4] O. Polozov and S. Gulwani, "Flashmeta: A framework for inductive program synthesis," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2015, pp. 107–126.

[5] S. Gulwani, "Dimensions in program synthesis," in *Proceedings of the 12th international ACM SIGPLAN symposium on Principles and practice of declarative programming*, 2010, pp. 13–24.

[6] D. Yang, Y. Zhang, and Q. Liu, "Blendfuzz: A model-based framework for fuzz testing programs with grammatical inputs," in *2012 IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications*. IEEE, 2012, pp. 1070–1076.

[7] T. Guo, P. Zhang, X. Wang, and Q. Wei, "Gramfuzz: Fuzzing testing of web browsers based on grammar analysis and structural mutation," in *2013 Second International Conference on Informatics & Applications (ICIA)*. IEEE, 2013, pp. 212–215.

[8] J. Wang, B. Chen, L. Wei, and Y. Liu, "Superion: Grammar-aware greybox fuzzing," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 724–735.

[9] H. Wang, P. Nguyen, J. Li, S. Kopru, G. Zhang, S. Katariya, and S. Ben-Romdhane, "Grano: Interactive graph-based root cause analysis for cloud-native distributed data platform," *Proceedings of the Very Large Data Base Endowment*, vol. 12, no. 12, pp. 1942–1945, 2019.

[10] Y. Feng, R. Martins, O. Bastani, and I. Dillig, "Program synthesis using conflict-driven learning," *ACM SIGPLAN Notices*, vol. 53, no. 4, pp. 420–435, 2018.

[11] R. Bunel, M. Hausknecht, J. Devlin, R. Singh, and P. Kohli, "Leveraging grammar and reinforcement learning for neural program synthesis," *arXiv preprint arXiv:1805.04276*, 2018.

[12] Y. Diao, H. Jamjoom, and D. Loewenstern, "Rule-based problem classification in it service management," in *2009 IEEE International Conference on Cloud Computing*. IEEE, 2009, pp. 221–228.

[13] O. Bastani, R. Sharma, A. Aiken, and P. Liang, "Synthesizing program input grammars," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2017, p. 95 110.

[14] A. Desai, S. Gulwani, V. Hingorani, N. Jain, A. Karkare, M. Marron, and S. Roy, "Program synthesis using natural language," in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 345–356.

[15] "Gnu grep," 2018. [Online]. Available: https://www.gnu.org/software/grep/

[16] R. Majumdar and R.-G. Xu, "Directed test generation using symbolic grammars," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. ACM, 2007, pp. 134–143.

[17] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," in *ACM Sigplan Notices*, vol. 43, no. 6. ACM, 2008, pp. 206–215.

[18] J. Caballero, H. Yin, Z. Liang, and D. Song, "Polyglot: Automatic extraction of protocol message format using dynamic binary analysis," in *Proceedings of the 14th ACM conference on Computer and communications security*. ACM, 2007, pp. 317–329.

[19] G. Misherghi and Z. Su, "Hdd: hierarchical delta debugging," in *Proceedings of the 28th international conference on Software engineering*. ACM, 2006, pp. 142–151.

[20] H. J. Wang, C. Guo, D. R. Simon, and A. Zugenmaier, "Shield: Vulnerability-driven network filters for preventing known vulnerability exploits," in *ACM SIGCOMM Computer Communication Review*, vol. 34, no. 4. ACM, 2004, pp. 193–204.

[21] M. C. Rinard, "Living in the comfort zone," *ACM SIGPLAN Notices*, vol. 42, no. 10, pp. 611–622, 2007.

[22] M. Rinard, "Acceptability-oriented computing," in *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. ACM, 2003, pp. 221–239.

[23] "Pdf reference," 2009. [Online]. Available: https://www.adobe.com/devnet/pdf/pdf_reference.html

[24] M. Höschele and A. Zeller, "Mining input grammars from dynamic taints," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2016, pp. 720–725.

[25] D. Babić, D. Reynaud, and D. Song, "Malware Analysis with Tree Automata Inference," in *CAV'11: Proceedings of the 23rd Int. Conference on Computer Aided Verification*, ser. Lecture Notes in Computer Science, vol. 6806. Springer, 2011, pp. 116–131.

[26] D. Babić, M. Botinčan, and D. Song, "Symbolic grey-box learning of input-output relations," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2012-59, May 2012.

[27] M. Höschele, A. Kampmann, and A. Zeller, "Active learning of input grammars," *arXiv preprint arXiv:1708.08731*, 2017.

[28] H. Ishizaka, "Polynomial time learnability of simple deterministic languages," *Machine Learning*, vol. 5, no. 2, pp. 151–164, 1990.

[29] D. Angluin, "Learning regular sets from queries and counterexamples," *Information and computation*, vol. 75, no. 2, pp. 87–106, 1987.

[30] M. Botinčan and D. Babić, "Sigma*: symbolic learning of input-output specifications," in *ACM SIGPLAN Notices*, vol. 48, no. 1. ACM, 2013, pp. 443–456.

[31] J. Oncina and P. Garcia, "Identifying regular languages in polynomial time," in *Advances in Structural and Syntactic Pattern Recognition*. World Scientific, 1992, pp. 99–108.

[32] P. Godefroid, H. Peleg, and R. Singh, "Learn&fuzz: Machine learning for input fuzzing," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 2017, pp. 50–59.

[33] M. Pradel and K. Sen, "Deep learning to find bugs," 2017.

[34] K. Böttinger, P. Godefroid, and R. Singh, "Deep reinforcement fuzzing," *arXiv preprint arXiv:1801.04589*, 2018.

[35] F. Doshi-Velez and B. Kim, "A roadmap for a rigorous science of interpretability," *arXiv preprint arXiv:1702.08608*, 2017.

[36] C. Rudin, "Algorithms for interpretable machine learning," in *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2014, pp. 1519–1519.

[37] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[38] N. Tillmann and J. De Halleux, "Pex–white box test generation for. net," in *International conference on tests and proofs*. Springer, 2008, pp. 134–153.

[39] T. Xie, N. Tillmann, J. de Halleux, and W. Schulte, "Fitness-guided path exploration in dynamic symbolic execution," in *Dependable Systems & Networks, 2009. DSN'09. IEEE/IFIP International Conference on.* IEEE, 2009, pp. 359–368.

[40] C. Lemieux and K. Sen, "Fairfuzz: Targeting rare branches to rapidly increase greybox fuzz testing coverage," *arXiv preprint arXiv:1709.07101*, 2017.

[41] R. S. Sutton, D. A. McAllester, S. P. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," in *Advances in neural information processing systems*, 2000, pp. 1057–1063.

[42] "What is a good regular expression to match a url?" 2010. [Online]. Available: https://stackoverflow.com/questions/3809401/what-is-a-good-regular-expression-to-match-a-url

[43] "Norvig lisp," 2010. [Online]. Available: http://norvig.com/lispy.html

[44] "Xml standard," 2015. [Online]. Available: https://www.w3.org/standards/xml/core

[45] "Grammar of css," 2015. [Online]. Available: https://www.w3.org/TR/CSS21/grammar.html

[46] "Antlr," 2018. [Online]. Available: https://www.antlr3.org/

[47] "Csv grammar," 2019. [Online]. Available: http://www.harward.us/~nharward/antlr/csv.g

[48] "First order logic grammar," 2019. [Online]. Available: https://www.antlr3.org/grammar/1336156363937/FOL.g

[49] A. Balalaie, A. Heydarnoori, and P. Jamshidi, "Microservices architecture enables DevOps: Migration to a cloud-native architecture," *IEEE Software*, vol. 33, no. 3, pp. 42–52, 2016.

[50] M. Solé, V. Muntés-Mulero, A. I. Rana, and G. Estrada, "Survey on models and techniques for root-cause analysis," *arXiv preprint arXiv:1701.08546*, 2017.

[51] N. Zhao, P. Jin, L. Wang, X. Yang, R. Liu, W. Zhang, K. Sui, and D. Pei, "Automatically and adaptively identifying severe alerts for online service systems," in *Proceedings of 2020 IEEE Conference on Computer Communications.* IEEE, 2020, pp. 2420–2429.

[52] J. Xu, Y. Wang, P. Chen, and P. Wang, "Lightweight and adaptive service api performance monitoring in highly dynamic cloud environment," in *Proceedings of 2017 IEEE International Conference on Services Computing.* IEEE, 2017, pp. 35–43.

[53] L. Tang, T. Li, F. Pinel, L. Shwartz, and G. Grabarnik, "Optimizing system monitoring configurations for non-actionable alerts," in *Proceedings of 2012 IEEE Network Operations and Management Symposium.* IEEE, 2012, pp. 34–42.

[54] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen, "Performance debugging for distributed systems of black boxes," *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5, pp. 74–89, 2003.

[55] H. Zawawy, K. Kontogiannis, and J. Mylopoulos, "Log filtering and interpretation for root cause analysis," in *Proceedings of 2010 IEEE International Conference on Software Maintenance*. IEEE, 2010, pp. 1–5.

[56] V. Nair, A. Raul, S. Khanduja, V. Bahirwani, Q. Shao, S. Sellamanickam, S. Keerthi, S. Herbert, and S. Dhulipalla, "Learning a hierarchical monitoring system for detecting and diagnosing service issues," in *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2015, pp. 2029–2038.

[57] S. Lu, B. Rao, X. Wei, B. Tak, L. Wang, and L. Wang, "Log-based abnormal task detection and root cause analysis for Spark," in *Proceedings of 2017 IEEE International Conference on Web Services*. IEEE, 2017, pp. 389–396.

[58] Y. Gan, Y. Zhang, K. Hu, D. Cheng, Y. He, M. Pancholi, and C. Delimitrou, "Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices," in *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2019, pp. 19–33.

[59] J. Mace, R. Roelke, and R. Fonseca, "Pivot tracing: Dynamic causal monitoring for distributed systems," in *Proceedings of the 25th ACM Symposium on Operating Systems Principles*. ACM, 2015, pp. 378–393.

[60] H. Xu, W. Chen, N. Zhao, Z. Li, J. Bu, Z. Li, Y. Liu, Y. Zhao, D. Pei, Y. Feng et al., "Unsupervised anomaly detection via variational auto-encoder for seasonal KPIs in web applications," in *Proceedings of the 2018 World Wide Web Conference*. ACM, 2018, pp. 187–196.

[61] M. Ma, W. Lin, D. Pan, and P. Wang, "Ms-rank: Multi-metric and self-adaptive root cause diagnosis for microservice applications," in *Proceedings of 2019 IEEE International Conference on Web Services*. IEEE, 2019, pp. 60–67.

[62] Y. Meng, S. Zhang, Y. Sun, R. Zhang, Z. Hu, Y. Zhang, C. Jia, Z. Wang, and D. Pei, "Localizing failure root causes in a microservice through causality inference," in *Proceedings of 2020 IEEE/ACM 28th International Symposium on Quality of Service*. IEEE, 2020, pp. 1–10.

[63] L. Wu, J. Tordsson, E. Elmroth, and O. Kao, "Microrca: Root cause localization of performance issues in microservices," in *Proceedings of 2020 IEEE/IFIP Network Operations and Management Symposium*. IEEE, 2020, pp. 1–9.

[64] M. Kim, R. Sumbaly, and S. Shah, "Root cause detection in a service-oriented architecture," *ACM SIGMETRICS Performance Evaluation Review*, vol. 41, no. 1, pp. 93–104, 2013.

[65] H. Baek, A. Srivastava, and J. Van der Merwe, "Cloudsight: A tenant-oriented transparency framework for cross-layer cloud troubleshooting," in *Proceedings of 2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE, 2017, pp. 268–273.

[66] G. Da Cunha Rodrigues, R. N. Calheiros, V. T. Guimaraes, G. L. d. Santos, M. B. De Carvalho, L. Z. Granville, L. M. R. Tarouco, and R. Buyya, "Monitoring of cloud computing environments: Concepts, solutions, trends, and future directions," in *Proceedings of the 31st Annual ACM Symposium on Applied Computing*. ACM, 2016, pp. 378–383.

[67] *How SRE teams are organized, and how to get started*, Accessed: 2020-12-10, https://cloud.google.com/blog/products/devops-sre/how-sre-teams-are-organized-and-how-to-get-started/.

[68] H. Nguyen, Z. Shen, Y. Tan, and X. Gu, "Fchain: Toward black-box online fault localization for cloud systems," in *Proceedings of 2013 IEEE 33rd International Conference on Distributed Computing Systems*. IEEE, 2013, pp. 21–30.

[69] P. Chen, Y. Qi, P. Zheng, and D. Hou, "Causeinfer: Automatic and distributed performance diagnosis with hierarchical causality graph in large distributed systems," in *Proceedings of 2014 IEEE Conference on Computer Communications*. IEEE, 2014, pp. 1887–1895.

[70] M. Ma, Z. Yin, S. Zhang, S. Wang, C. Zheng, X. Jiang, H. Hu, C. Luo, Y. Li, N. Qiu et al., "Diagnosing root causes of intermittent slow queries in cloud databases," *Proceedings of the Very Large Data Base Endowment*, vol. 13, no. 8, pp. 1176–1189, 2020.

[71] J. Schoenfisch, C. Meilicke, J. von Stülpnagel, J. Ortmann, and H. Stuckenschmidt, "Root cause analysis in it infrastructures using ontologies and abduction in markov logic networks," *Information Systems*, vol. 74, pp. 103–116, 2018.

[72] Á. Brandón, M. Solé, A. Huélamo, D. Solans, M. S. Pérez, and V. Muntés-Mulero, "Graph-based root cause analysis for service-oriented and microservice architectures," *Journal of Systems and Software*, vol. 159, p. 110432, 2020.

[73] D. Y. Yoon, N. Niu, and B. Mozafari, "Dbsherlock: A performance diagnostic tool for transactional databases," in *Proceedings of the 2016 International Conference on Management of Data*. ACM, 2016, pp. 1599–1614.

[74] V. Jeyakumar, O. Madani, A. Parandeh, A. Kulshreshtha, W. Zeng, and N. Yadav, "Explainit!–a declarative root-cause analysis engine for time series data," in *Proceedings of the 2019 International Conference on Management of Data*. ACM, 2019, pp. 333–348.

[75] H. Jayathilaka, C. Krintz, and R. Wolski, "Performance monitoring and root cause analysis for cloud-hosted web applications," in *Proceedings of the 26th International Conference on World Wide Web*. ACM, 2017, pp. 469–478.

[76] M. A. Marvasti, A. V. Poghosyan, A. N. Harutyunyan, and N. M. Grigoryan, "An anomaly event correlation engine: Identifying root causes, bottlenecks, and black swans in IT environments," *VMware Technical Journal*, vol. 2, no. 1, pp. 35–45, 2013.

[77] J. Weng, J. H. Wang, J. Yang, and Y. Yang, "Root cause analysis of anomalies of multitier services in public clouds," *IEEE/ACM Transactions on Networking*, vol. 26, no. 4, pp. 1646–1659, 2018.

[78] J. Qiu, Q. Du, K. Yin, S.-L. Zhang, and C. Qian, "A causality mining and knowledge graph based method of root cause diagnosis for performance anomaly in cloud applications," *Applied Sciences*, vol. 10, no. 6, p. 2166, 2020.

[79] C. Manning, P. Raghavan, and H. Schütze, "Introduction to information retrieval," *Natural Language Engineering*, vol. 16, no. 1, pp. 100–103, 2010.

[80] J. Lin, P. Chen, and Z. Zheng, "Microscope: Pinpoint performance issues with causal graphs in micro-service environments," in *Proceedings of International Conference on Service-Oriented Computing*. Springer, 2018, pp. 3–20.

[81] M. Ma, S. Zhang, D. Pei, X. Huang, and H. Dai, "Robust and rapid adaption for concept drift in software system anomaly detection," in *Proceedings of 2018 IEEE 29th International Symposium on Software Reliability Engineering*. IEEE, 2018, pp. 13–24.

[82] Z. Wu, E. Johnson, W. Yang, O. Bastani, D. Song, J. Peng, and T. Xie, "Reinam: reinforcement learning for input-grammar inference," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 488–498.

[83] F. F. Xu, Z. Jiang, P. Yin, B. Vasilescu, and G. Neubig, "Incorporating external knowledge through pre-training for natural language to code generation," in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, ser. ACL '20. ACL, 2020, pp. 6045–6052.

[84] V. Zhong, C. Xiong, and R. Socher, "Seq2sql: Generating structured queries from natural language using reinforcement learning," *arXiv preprint arXiv:1709.00103*, 2017.

[85] N. Yaghmazadeh, Y. Wang, I. Dillig, and T. Dillig, "Sqlizer: Query synthesis from natural language," *Proceedings of the ACM Programming Language*, vol. 1, no. OOPSLA, oct 2017. [Online]. Available: https://doi.org/10.1145/3133887

[86] H. Kim, B.-H. So, W.-S. Han, and H. Lee, "Natural language to sql: Where are we today?" *Proceedings of the Very Large Data Base Endowment.*, vol. 13, no. 10, p. 1737–1750, jun 2020. [Online]. Available: https://doi.org/10.14778/3401960.3401970

[87] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. D. Lago et al., "Competition-level code generation with alphacode," *arXiv preprint arXiv:2203.07814*, 2022.

[88] L. Grammel, M. Tory, and M.-A. Storey, "How information visualization novices construct visualizations," *IEEE Transactions on Visualization and Computer Graphics*, vol. 16, no. 6, pp. 943–952, 2010.

[89] Jupyter, "Project Jupyter," https://jupyter.org/, 2020, accessed: 2020-05-15.

[90] Y. Artzi, "Cornell SPF: Cornell semantic parsing framework," 2016.

[91] J. Berant, A. Chou, R. Frostig, and P. Liang, "Semantic parsing on Freebase from question-answer pairs," in *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, ser. EMNLP '13. ACL, Oct. 2013. [Online]. Available: https://aclanthology.org/D13-1160 pp. 1533–1544.

[92] Y. Luo, N. Tang, G. Li, C. Chai, W. Li, and X. Qin, "Synthesizing natural language to visualization (nl2vis) benchmarks from nl2sql benchmarks," in *Proceedings of the 2021 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '21. ACM, 2021. [Online]. Available: https://doi.org/10.1145/3448016.3457261 p. 1235–1247.

[93] Kaggle, "Kaggle competitions," https://www.kaggle.com/competitions, 2020, accessed: 2020-05-15.

[94] T. Kasami, "An efficient recognition and syntax-analysis algorithm for context-free languages," *Coordinated Science Laboratory Report no. R-257*, 1966.

[95] J. F. Pimentel, L. Murta, V. Braganholo, and J. Freire, "A large-scale study about quality and reproducibility of jupyter notebooks," in *Proceedings of the 16th International Conference on Mining Software Repositories*, ser. MSR '19. IEEE Press, 2019, pp. 507–517.

[96] IPython, "IPython magic commands," https://ipython.readthedocs.io/en/stable/interactive/magics.html, 2020, accessed: 2020-05-15.

[97] Pandas, "Pandas: Python data analysis library," https://pandas.pydata.org/, 2019, accessed: 2019-11-20.

[98] T. Yu, R. Zhang, K. Yang, M. Yasunaga, D. Wang, Z. Li, J. Ma, I. Li, Q. Yao, S. Roman, Z. Zhang, and D. Radev, "Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task," in *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, ser. EMNLP '18. ACL, 2018, pp. 3911–3921.

[99] J. Mitchell, P. Stenetorp, P. Minervini, and S. Riedel, "Extrapolation in NLP," in *Proceedings of the Workshop on Generalization in the Age of Deep Learning*. ACL, June 2018. [Online]. Available: https://aclanthology.org/W18-1005 pp. 28–33.

[100] Y. Luo, X. Qin, N. Tang, G. Li, and X. Wang, "Deepeye: Creating good data visualizations by keyword search," in *Proceedings of the 2018 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '18. ACM, 2018. [Online]. Available: https://doi.org/10.1145/3183713.3193545 p. 1733–1736.

[101] A. Narechania, A. Srinivasan, and J. Stasko, "Nl4dv: A toolkit for generating analytic specifications for data visualization from natural language queries," *IEEE Transactions on Visualization and Computer Graphics*, vol. 27, no. 2, pp. 369–379, 2020.

[102] D. Guo, D. Tang, N. Duan, M. Zhou, and J. Yin, "Coupling retrieval and meta-learning for context-dependent semantic parsing," in *Proceedings of the 57th Conference of the Association for Computational Linguistics, ACL 2019, Florence, Italy, July 28- August 2, 2019, Volume 1: Long Papers*, A. Korhonen, D. R. Traum, and L. Màrquez, Eds. Association for Computational Linguistics, 2019, pp. 855–866.

[103] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "A conversational paradigm for program synthesis," *CoRR*, vol. abs/2203.13474, 2022.

[104] Y. Li, D. H. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. D. Lago, T. Hubert, P. Choy, C. de Masson d'Autume, I. Babuschkin, X. Chen, P. Huang, J. Welbl, S. Gowal, A. Cherepanov, J. Molloy, D. J. Mankowitz, E. S. Robson, P. Kohli, N. de Freitas, K. Kavukcuoglu, and O. Vinyals, "Competition-level code generation with alphacode," *CoRR*, vol. abs/2203.07814, 2022.

[105] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating large language models trained on code," *CoRR*, vol. abs/2107.03374, 2021.

[106] Y. Wang, W. Wang, S. R. Joty, and S. C. H. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*, M. Moens, X. Huang, L. Specia, and S. W. Yih, Eds. Association for Computational Linguistics, 2021, pp. 8696–8708.

[107] J. Austin, A. Odena, M. I. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. J. Cai, M. Terry, Q. V. Le, and C. Sutton, "Program synthesis with large language models," *CoRR*, vol. abs/2108.07732, 2021.

[108] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," in *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds., 2020.

[109] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language understanding," *CoRR*, vol. abs/1810.04805, 2018.

[110] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, I. Guyon, U. von Luxburg, S. Bengio, H. M. Wallach, R. Fergus, S. V. N. Vishwanathan, and R. Garnett, Eds., 2017, pp. 5998–6008.

[111] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari, "Oracle-guided component-based program synthesis," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, J. Kramer, J. Bishop, P. T. Devanbu, and S. Uchitel, Eds. ACM, 2010, pp. 215–224.

[112] A. Solar-Lezama, L. Tancau, R. Bodík, S. A. Seshia, and V. A. Saraswat, "Combinatorial sketching for finite programs," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006*, J. P. Shen and M. Martonosi, Eds. ACM, 2006, pp. 404–415.

[113] R. Alur, R. Bodík, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa, "Syntax-guided synthesis," in *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*. IEEE, 2013, pp. 1–8.

[114] N. Jain, S. Vaidyanath, A. S. Iyer, N. Natarajan, S. Parthasarathy, S. K. Rajamani, and R. Sharma, "Jigsaw: Large language models meet program synthesis," in *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 2022, pp. 1219–1231.

[115] Y. Li, S. Li, Z. Xu, J. Cao, Z. Chen, Y. Hu, H. Chen, and S. Cheung, "TRANSREGEX: multi-modal regular expression synthesis by generate-and-repair," in *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 2021, pp. 1210–1222.

[116] X. Ye, Q. Chen, X. Wang, I. Dillig, and G. Durrett, "Sketch-driven regular expression generation from natural language and examples," *Trans. Assoc. Comput. Linguistics*, vol. 8, pp. 679–694, 2020.

[117] Q. Chen, X. Wang, X. Ye, G. Durrett, and I. Dillig, "Multi-modal synthesis of regular expressions," in *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, A. F. Donaldson and E. Torlak, Eds. ACM, 2020, pp. 487–502.

[118] Z. Zhong, J. Guo, W. Yang, T. Xie, J.-G. Lou, T. Liu, and D. Zhang, "Generating regular expressions from natural language specifications: Are we there yet?" in *Workshops at the Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[119] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman et al., "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[120] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. D. Lago et al., "Competition-level code generation with alphacode," *arXiv preprint arXiv:2203.07814*, 2022.

[121] N. Locascio, K. Narasimhan, E. DeLeon, N. Kushman, and R. Barzilay, "Neural generation of regular expressions from natural language with minimal domain knowledge," in *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing, EMNLP 2016, Austin, Texas, USA, November 1-4, 2016*, J. Su, X. Carreras, and K. Duh, Eds. The Association for Computational Linguistics, 2016, pp. 1918–1923.

[122] N. Kushman and R. Barzilay, "Using semantic unification to generate regular expressions from natural language," in *Human Language Technologies: Conference of the North American Chapter of the Association of Computational Linguistics, Proceedings, June 9-14, 2013, Westin Peachtree Plaza Hotel, Atlanta, Georgia, USA*, L. Vanderwende, H. D. III, and K. Kirchhoff, Eds. The Association for Computational Linguistics, 2013, pp. 826–836.

[123] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. B. Clement, D. Drain, D. Jiang, D. Tang, G. Li, L. Zhou, L. Shou, L. Zhou, M. Tufano, M. Gong, M. Zhou, N. Duan, N. Sundaresan, S. K. Deng, S. Fu, and S. Liu, "Codexglue: A machine learning benchmark dataset for code understanding and generation," in *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual*, J. Vanschoren and S. Yeung, Eds., 2021.

[124] S. Iyer, A. Cheung, and L. Zettlemoyer, "Learning programmatic idioms for scalable semantic parsing," in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing, EMNLP-IJCNLP 2019, Hong Kong, China, November 3-7, 2019*, K. Inui, J. Jiang, V. Ng, and X. Wan, Eds. Association for Computational Linguistics, 2019. [Online]. Available: https://doi.org/10.18653/v1/D19-1545 pp. 5425–5434.

[125] F. Christopoulou, G. Lampouras, M. Gritta, G. Zhang, Y. Guo, Z. Li, Q. Zhang, M. Xiao, B. Shen, L. Li, H. Yu, L. Yan, P. Zhou, X. Wang, Y. Ma, I. Iacobacci, Y. Wang, G. Liang, J. Wei, X. Jiang, Q. Wang, and Q. Liu, "Pangu-coder: Program synthesis with function-level language modeling," *CoRR*, vol. abs/2207.11280, 2022. [Online]. Available: https://doi.org/10.48550/arXiv.2207.11280

[126] D. Fried, A. Aghajanyan, J. Lin, S. Wang, E. Wallace, F. Shi, R. Zhong, W. Yih, L. Zettlemoyer, and M. Lewis, "Incoder: A generative model for code infilling and synthesis," *CoRR*, vol. abs/2204.05999, 2022.

[127] F. F. Xu, U. Alon, G. Neubig, and V. J. Hellendoorn, "A systematic evaluation of large language models of code," in *MAPS@PLDI 2022: 6th ACM SIGPLAN International Symposium on Machine Programming, San Diego, CA, USA, 13 June 2022*, S. Chaudhuri and C. Sutton, Eds. ACM, 2022, pp. 1–10.

[128] L. Gao, S. Biderman, S. Black, L. Golding, T. Hoppe, C. Foster, J. Phang, H. He, A. Thite, N. Nabeshima, S. Presser, and C. Leahy, "The pile: An 800gb dataset of diverse text for language modeling," *CoRR*, vol. abs/2101.00027, 2021.

[129] H. Husain, H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "Codesearchnet challenge: Evaluating the state of semantic code search," *CoRR*, vol. abs/1909.09436, 2019.

[130] P. Liu, W. Yuan, J. Fu, Z. Jiang, H. Hayashi, and G. Neubig, "Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing," *arXiv preprint arXiv:2107.13586*, 2021.

[131] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell et al., "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[132] B. Chen, F. Zhang, A. Nguyen, D. Zan, Z. Lin, J. Lou, and W. Chen, "Codet: Code generation with generated tests," *CoRR*, vol. abs/2207.10397, 2022. [Online]. Available: https://doi.org/10.48550/arXiv.2207.10397

[133] J. P. Inala, C. Wang, M. Yang, A. Codas, M. Encarnación, S. K. Lahiri, M. Musuvathi, and J. Gao, "Fault-aware neural code rankers," *CoRR*, vol. abs/2206.03865, 2022. [Online]. Available: https://doi.org/10.48550/arXiv.2206.03865

[134] M. Hoschele and A. Zeller, "Mining input grammars with autogram," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, May 2017, pp. 31–34.

[135] Y. Li, B. Chen, M. Chandramohan, S.-W. Lin, Y. Liu, and A. Tiu, "Steelix: Program-state based binary fuzzing," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017. New York, NY, USA: ACM, 2017. [Online]. Available: http://doi.acm.org/10.1145/3106237.3106295 pp. 627–637.

[136] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *39th IEEE Symposium on Security and Privacy*, 2018.

[137] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Krügel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution," in *NDSS*, 2016.

[138] K. Böttinger, P. Godefroid, and R. Singh, "Deep reinforcement fuzzing," *CoRR*, vol. abs/1801.04589, 2018. [Online]. Available: http://arxiv.org/abs/1801.04589

[139] M. Wu, L. Jiang, J. Xiang, Y. Huang, H. Cui, L. Zhang, and Y. Zhang, "One fuzzing strategy to rule them all," in *Proceedings of the International Conference on Software Engineering*, 2022.

[140] A. Belz and A. Belz, "Pcfg learning by nonterminal partition search," in *In Proceedings of ICGI 2002*. Springer, 2002, pp. 14–27.

[141] J. Scicluna and C. D. L. Higuera, "Pcfg induction for unsupervised parsing and language modelling."

[142] *Surus*, Accessed: 2020-08-15, https://github.com/Netflix/Surus.

[143] R. Bhagwan, R. Kumar, R. Ramjee, G. Varghese, S. Mohapatra, H. Manoharan, and P. Shah, "Adtributor: Revenue debugging in advertising systems," in *Proceedings of 11th USENIX Symposium on Networked Systems Design and Implementation*. USENIX, 2014, pp. 43–55.

[144] Y. Sun, Y. Zhao, Y. Su, D. Liu, X. Nie, Y. Meng, S. Cheng, D. Pei, S. Zhang, X. Qu et al., "Hotspot: Anomaly localization for additive KPIs with multi-dimensional attributes," *IEEE Access*, vol. 6, pp. 10 909–10 923, 2018.

[145] S. Iyer, I. Konstas, A. Cheung, J. Krishnamurthy, and L. Zettlemoyer, "Learning a neural semantic parser from user feedback," in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 1: Long Papers*, R. Barzilay and M. Kan, Eds. Association for Computational Linguistics, 2017, pp. 963–973.

[146] C. Finegan-Dollak, J. K. Kummerfeld, L. Zhang, K. Ramanathan, S. Sadasivam, R. Zhang, and D. R. Radev, “Improving text-to-sql evaluation methodology,” in *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics, ACL 2018, Melbourne, Australia, July 15-20, 2018, Volume 1: Long Papers*, I. Gurevych and Y. Miyao, Eds. Association for Computational Linguistics, 2018, pp. 351–360.

[147] Z. Zhong, J. Guo, W. Yang, J. Peng, T. Xie, J. Lou, T. Liu, and D. Zhang, “Semregex: A semantics-based approach for generating regular expressions from natural language specifications,” in *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, Brussels, Belgium, October 31 - November 4, 2018*, E. Riloff, D. Chiang, J. Hockenmaier, and J. Tsujii, Eds. Association for Computational Linguistics, 2018, pp. 1608–1618.

[148] 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*, 2015. [Online]. Available: https://doi.org/10.3233/978-1-61499-495-4-1

[149] A. Udupa, A. Raghavan, J. V. Deshmukh, S. Mador-Haim, M. M. Martin, and R. Alur, “Transit: Specifying protocols with concolic snippets,” in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’13. ACM, 2013. [Online]. Available: https://doi.org/10.1145/2491956.2462174 p. 287–296.

[150] R. Alur, P. Cerný, and A. Radhakrishna, “Synthesis through unification,” in *Proceedings the 27th International Conference on Computer Aided Verification*, ser. CAV ’15. Springer, 2015. [Online]. Available: https://doi.org/10.1007/978-3-319-21668-3\%5F10 pp. 163–179.

[151] R. Alur, A. Radhakrishna, and A. Udupa, “Scaling enumerative program synthesis via divide and conquer,” in *Proceedings of the 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS ’17, 2017. [Online]. Available: https://doi.org/10.1007/978-3-662-54577-5\%5F18 pp. 319–336.

[152] A. Reynolds, M. Deters, V. Kuncak, C. Tinelli, and C. W. Barrett, “Counterexample-guided quantifier instantiation for synthesis in SMT,” in *Proceedings the 27th International Conference on Computer Aided Verification*, ser. CAV ’15. Springer, 2015. [Online]. Available: https://doi.org/10.1007/978-3-319-21668-3\%5F12 pp. 198–216.

[153] W. Lee, K. Heo, R. Alur, and M. Naik, “Accelerating search-based program synthesis using learned probabilistic models,” in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’18. ACM, 2018. [Online]. Available: https://doi.org/10.1145/3192366.3192410 p. 436–449.

[154] Q. Hu and L. D'Antoni, "Syntax-guided synthesis with quantitative syntactic objectives," in *Proceedings the 30th International Conference on Computer Aided Verification*, ser. CAV '15. Springer, 2018. [Online]. Available: https://doi.org/10.1007/978-3-319-96145-3\%5F21 pp. 386–403.

[155] T. Gao, M. Dontcheva, E. Adar, Z. Liu, and K. G. Karahalios, "Datatone: Managing ambiguity in natural language interfaces for data visualization," in *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*, ser. UIST '15. ACM, 2015. [Online]. Available: https://doi.org/10.1145/2807442.2807478 p. 489–500.

[156] W. Cui, X. Zhang, Y. Wang, H. Huang, B. Chen, L. Fang, H. Zhang, J.-G. Lou, and D. Zhang, "Text-to-viz: Automatic generation of infographics from proportion-related natural language statements," *IEEE Transactions on Visualization and Computer Graphics*, vol. 26, no. 1, pp. 906–916, 2019.

[157] Y. Sun, J. Leigh, A. Johnson, and S. Lee, "Articulate: A semi-automated model for translating natural language queries into meaningful visualizations," in *Proceedings of the 10th International Conference on Smart Graphics*, ser. ICSG '10. Springer-Verlag, 2010, pp. 184–195.

[158] A. Satyanarayan, D. Moritz, K. Wongsuphasawat, and J. Heer, "Vega-lite: A grammar of interactive graphics," *IEEE Transactions on Visualization and Computer Graphics*, vol. 23, no. 1, pp. 341–350, 2016.

[159] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, ser. NIPS '14. MIT Press, 2014, p. 3104–3112.

[160] X. Rong, S. Yan, S. Oney, M. Dontcheva, and E. Adar, "Codemend: Assisting interactive programming with bimodal embedding," in *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*, ser. UIST '16. ACM, 2016. [Online]. Available: https://doi.org/10.1145/2984511.2984544 p. 247–258.

[161] V. Setlur, S. E. Battersby, M. Tory, R. Gossweiler, and A. X. Chang, "Eviza: A natural language interface for visual analysis," in *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*, ser. UIST '16. ACM, 2016. [Online]. Available: https://doi.org/10.1145/2984511.2984588 p. 365–377.

[162] K. Wongsuphasawat, D. Moritz, A. Anand, J. Mackinlay, B. Howe, and J. Heer, "Towards a general-purpose query language for visualization recommendation," in *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*, ser. HILDA '16. ACM, 2016. [Online]. Available: https://doi.org/10.1145/2939502.2939506

[163] D. Moritz, C. Wang, G. L. Nelson, H. Lin, A. M. Smith, B. Howe, and J. Heer, "Formalizing visualization design knowledge as constraints: Actionable and extensible models in draco," *IEEE Transactions on Visualization and Computer Graphics*, vol. 25, no. 1, pp. 438–448, 2018.

[164] R. Ding, S. Han, Y. Xu, H. Zhang, and D. Zhang, “Quickinsights: Quick and automatic discovery of insights from multi-dimensional data,” in *Proceedings of the 2019 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD ’19. ACM, 2019. [Online]. Available: http://doi.acm.org/10.1145/3299869.3314037 pp. 317–332.

[165] T. Siddiqui, A. Kim, J. Lee, K. Karahalios, and A. Parameswaran, “Effortless data exploration with zenvisage: An expressive and interactive visual analytics system,” *Proceedings of the Very Large Data Base Endowment.*, vol. 10, no. 4, p. 457–468, nov 2016. [Online]. Available: https://doi.org/10.14778/3025111.3025126

[166] T. Siddiqui, J. Lee, A. Kim, E. Xue, X. Yu, S. Zou, L. Guo, C. Liu, C. Wang, K. Karahalios, and A. G. Parameswaran, “Fast-forwarding to desired visualizations with zenvisage,” in *Online Proceedings of the 8th Biennial Conference on Innovative Data Systems Research, 2017*, ser. CIDR ’17. www.cidrdb.org, 2017. [Online]. Available: http://cidrdb.org/cidr2017/papers/p43-siddiqui-cidr17.pdf

[167] B. Chan, J. Talbot, L. Wu, N. Sakunkoo, M. Cammarano, and P. Hanrahan, “Vispedia: On-demand data integration for interactive visualization and exploration,” in *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD ’09. ACM, 2009. [Online]. Available: https://doi.org/10.1145/1559845.1560003 p. 1139–1142.

[168] J. Park, S. Ko, M. Cognetta, and Y. Han, “Softregex: Generating regex from natural language descriptions using softened regex equivalence,” in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing, EMNLP-IJCNLP 2019, Hong Kong, China, November 3-7, 2019*, K. Inui, J. Jiang, V. Ng, and X. Wan, Eds. Association for Computational Linguistics, 2019, pp. 6424–6430.

[169] A. Bartoli, A. D. Lorenzo, E. Medvet, and F. Tarlao, “Inference of regular expressions for text extraction from examples,” *IEEE Trans. Knowl. Data Eng.*, vol. 28, no. 5, pp. 1217–1230, 2016.

[170] G. J. Bex, F. Neven, T. Schwentick, and S. Vansummeren, “Inference of concise regular expressions and dtds,” *ACM Trans. Database Syst.*, vol. 35, no. 2, pp. 11:1–11:47, 2010.

[171] G. J. Bex, W. Gelade, F. Neven, and S. Vansummeren, “Learning deterministic regular expressions for the inference of schemas from XML data,” *ACM Trans. Web*, vol. 4, no. 4, pp. 14:1–14:32, 2010.

[172] D. D. Freydenberger and T. Kötzing, “Fast learning of restricted regular expressions and dtds,” *Theory Comput. Syst.*, vol. 57, no. 4, pp. 1114–1158, 2015.

[173] M. Lee, S. So, and H. Oh, "Synthesizing regular expressions from examples for introductory automata assignments," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2016, Amsterdam, The Netherlands, October 31 - November 1, 2016*, B. Fischer and I. Schaefer, Eds. ACM, 2016, pp. 70–80.

[174] Y. Li, Z. Xu, J. Cao, H. Chen, T. Ge, S. Cheung, and H. Zhao, "Flashregex: Deducing anti-redos regexes from examples," in *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*. IEEE, 2020, pp. 659–671.

[175] A. Bartoli, G. Davanzo, A. D. Lorenzo, E. Medvet, and E. Sorio, "Automatic synthesis of regular expressions from examples," *Computer*, vol. 47, no. 12, pp. 72–80, 2014.

[176] Y. Li, X. Zhang, J. Cao, H. Chen, and C. Gao, "Learning k-occurrence regular expressions with interleaving," in *Database Systems for Advanced Applications - 24th International Conference, DASFAA 2019, Chiang Mai, Thailand, April 22-25, 2019, Proceedings, Part II*, ser. Lecture Notes in Computer Science, G. Li, J. Yang, J. Gama, J. Natwichai, and Y. Tong, Eds., vol. 11447. Springer, 2019, pp. 70–85.

[177] Y. Pu, Z. Miranda, A. Solar-Lezama, and L. Kaelbling, "Selecting representative examples for program synthesis," in *International Conference on Machine Learning*. PMLR, 2018, pp. 4161–4170.

[178] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.