Functional Synthesis via Formal Methods and Machine Learning

A *thesis* submitted
in partial fulfillment of the requirements
for the degree of

Doctor of Philosophy
by

Priyanka Golia
17111270

to the

Department of Computer Science & Engineering
Indian Institute of Technology, Kanpur
and
School of Computing
National University of Singapore

DECLARATION

This is to certify that the thesis titled **Functional Synthesis via Formal Methods and Machine Learning** has been authored by me. It presents the research conducted by me under the supervision of Prof. Subhajit Roy (IITK) and Prof. Kuldeep S. Meel (NUS). To the best of my knowledge, it is an original work, both in terms of research content and narrative, and has not been submitted elsewhere, in part or in full, for a degree. Further, due credit has been attributed to the relevant state-of-the-art and collaborations (if any) with appropriate citations and acknowledgements, in line with established norms and practices.

Priyanka.

Priyanka Golia
Program: Doctor of Philosophy
Department of Computer Science & Engineering,
Indian Institute of Technology Kanpur, India
and
School of Computing,
National University of Singapore, Singapore

August 16, 2023

CERTIFICATE

It is certified that the work contained in the thesis titled **Functional Synthesis via Formal Methods and Machine Learning**, by **Priyanka Golia**, has been carried out under our supervision and that this work has not been submitted elsewhere for a degree.

Subhajit Roy

Prof. Subhajit Roy
Department of Computer Science & Engineering
Indian Institute of Technology Kanpur, India

[illegible]

Prof. Kuldeep S. Meel
School of Computing
National University of Singapore, Singapore

February 8, 2023

SYNOPSIS

Name of student: **Priyanka Golia** Roll no: **17111270**

Degree for which submitted: **Doctor of Philosophy**

Department: **Computer Science & Engineering**

Thesis title:

Functional Synthesis via Formal Methods and Machine Learning

Name of Thesis Supervisors:

Prof. Subhajit Roy	Prof. Kuldeep S. Meel
Department of Computer Science & Engineering	School of Computing
IITK, India	NUS, Singapore

Month and year of thesis submission: **August 16, 2023**

Functional synthesis, a fundamental task in computer science, involves automatically generating functions that meet specific user requirements. Its practical applications span a wide range, from automatically repairing programs to cryptanalysis. While theoretical investigations have shown that certain instances of functional synthesis can be exceptionally time-consuming, the need for practical usability has spurred the development of algorithms that showcase remarkable scalability. Despite these significant strides, practical challenges persist, and there are still real-world situations where current methods encounter limitations. This highlights the need for continued research and innovation in intriguing field of functional synthesis. This thesis takes a comprehensive perspective on functional synthesis, examining it through the lens of recent advancements in machine learning and formal methods. By leveraging the power of these cutting-edge technologies, we aim to enhance the process of automatically generating functions that meet the given formal requirements.

The central focus of the thesis lies in presenting innovative approaches to enhance scalability in functional synthesis and its more complex variants. Inspired by recent progress in machine learning and formal methods, the thesis reimagines functional synthesis as a classification problem. It leverages cutting-edge techniques such as constrained sampling for data generation and automated reasoning for proof-guided repair. Therefore, the key components or ingredients that constitute these advancements are constrained sampling for data generation, machine learning for learning candidate functions, and formal methods for repairing and verifying these candidates. By combining these ingredients, the thesis aims to achieve remarkable progress in the field of functional synthesis and tackle its scalability challenges.

The thesis presents Manthan, a data-driven approach to functional synthesis that capitalizes on the aforementioned key ingredients. By introducing Manthan, the scalability of functional synthesis has witnessed a significant boost, surpassing the capabilities of previous state-of-the-art methods. Notably, Manthan demonstrates the successful handling of 40% more instances, signifying a remarkable advancement in the field. One of the pivotal contributions of the thesis is the exploration of variants of functional synthesis that are known to be more harder from the theoretical complexity view, particularly those involving explicit dependencies. These explicit dependencies impart a higher degree of descriptive power and exhibit applications in diverse domains, such as circuit repair, controller synthesis, and equivalence checking. The designed approach has pushed the boundaries of synthesis with explicit dependencies, effectively addressing instances that had previously posed challenges for state-of-the-art tools.

Our proposed approaches showed impressive scalability, which inspired us to explore their applications in various fields. One area we focused on in this thesis is program synthesis, a fundamental problem in computer science. In program synthesis, the goal is to automatically synthesize a program that fulfills given user requirements. In recent developments, these requirements include both syntactic (grammar-related) and semantic aspects. In the thesis, we specifically looked at the role of syntactic requirements (grammar) in program synthesis, and examined the feasibility of synthesis techniques that do not rely on grammatical restrictions, which we termed Theory-constrained synthesis, or $\mathbb{T}$-constrained synthesis. The thesis proposed a reduction of $\mathbb{T}$-constrained synthesis to functional synthesis with explicit dependencies when there are no syntactic requirements. This means we can do program synthesis without the need for grammar via functional synthesis. As a result of this reduction, our proposed method, Manthan, has become the state-of-the-art approach for synthesizing programs when dealing with bit vectors as the underlying theory. The proposed advancement opens up new possibilities for program synthesis and expands the scope of functional synthesis to different application areas.

Our proposed approaches, which combines formal methods and machine learning, offers scalability and flexibility, making it applicable to various domains and applications. Another area where our approach shows great promise is in real-world scenarios that involve both hard constraints (which must be satisfied) and soft constraints (which should be satisfied to the best extent possible). The thesis presents a framework for synthesizing systems that not only guarantee the fulfillment of hard constraints but also achieve a predefined level of goodness for soft constraints,which makes a significant advancement in making functional synthesis more practical and valuable in real-world applications.

List of Publications

This thesis is based on the following publications.

1. **Manthan: A data-driven approach for Boolean function synthesis**

   Priyanka Golia, Subhajit Roy, and Kuldeep S. Meel.

   In Proceedings of International Conference on Computer-Aided Verification (CAV), 2020.

2. **Engineering an efficient Boolean functional synthesis engine.**

   Priyanka Golia, Friedrich Slivovsky, Subhajit Roy, and Kuldeep S. Meel.

   In Proceedings of International Conference On Computer Aided Design (ICCAD), 2021.

   Best Paper Award Nomination.

3. **Program synthesis as dependency quantified formula modulo theory.**

   Priyanka Golia, Subhajit Roy, and Kuldeep S. Meel.

   In Proceedings of International Joint Conference on Artificial Intelligence (IJCAI), 2021.

4. **Synthesis with explicit dependencies.**

   Priyanka Golia, Subhajit Roy, and Kuldeep S. Meel.

   In Proceedings of Design, Automation and Test in Europe (DATE), 2023.

   Best Paper Award Nomination.

5. **Good enough synthesis with hard constraints.**

   Priyanka Golia, Subhajit Roy, and Kuldeep S. Meel.

   Under Review.

To my grandparents, parents

and

members of my family

Acknowledgements

I am deeply grateful to my advisors, Prof. Kuldeep S. Meel and Prof. Subhajit Roy, for their invaluable guidance and support throughout my Ph.D. journey. Their unwavering dedication, support, and belief in hard work and incremental progress have been invaluable to my personal and professional growth. Their determination and perseverance in problem-solving has been a valuable lesson in research for me.

Thank you Subhajit for his patience and willingness to listen to my half-baked ideas. His guidance and support have been instrumental in helping me grow.

I am grateful to Kuldeep for his kind and supportive guidance that has helped me to grow not only as a researcher, but also as a person. Our engaging discussions on a variety of topics of life in general have played a crucial role in my transformation from an under-confident and shy individual to a more confident person. I am thankful to Kuldeep for his mentorship and encouragement during my academic job search.

Kuldeep and Subhajit, thank you for everything; your guidance and understanding have helped me to navigate through difficult moments and provided me with the motivation and encouragement I needed to keep going. I am truly thankful to have had such a supportive and understanding mentors/advisors by my side.

I am thankful to Akshay, Supratik, Shetal for their guidance, support and discussions about functional synthesis. I am grateful to Diptarka for being a friend and mentor at times and for listening to all of my complaints about life and work. I have been fortunate to collaborate with Sourav Chakraborty, Brendan Juba, Mate Soos, and Friedrich Slivovsky.

I would also like to express my gratitude to my friends at IITK and NUS. Amit,

Ashish, Avideep, Bhargav, Chetan, Mahesh, Muzafar, Pranjal, Pranav, Prateek, Rahul, Sumanta and Vinay, for being there throughout my Ph.D. journey at IITK. Yash, Teo, Aashish, Soundarya, and Umair, thank you for the weekend parties and impromptu plans that helped me survive my Ph.D. Thank you Awanish, Prantik, Pankaj, Nitesh, Sujit, Sumit for helping me improve by providing detailed feedback on my research. At NUS, I met my new family, the "MeelGroup" ; Thank you to Yash, Teo, Jiong, Arijit, Bishwa, Tim, Shubham, Mahi, Anna, Gunjan, and Pang for listening to my talk preparations and giving me constructive feedback on every aspect of my Ph.D. research. From my undergraduate days until now, I am incredibly grateful for my pillars of support: Anurag, Bidesh, Mahantesh, and Mayank. Your presence in my life has been a constant source of strength and encouragement.

I lost my grandparents (*Nana ji R.L. Inaniya and Dada ji M.L. Golia*) during my Ph.D. journey, but I know how proud they must be of me today. *Nana ji, Dada ji, thank you for fighting the odds and allowing me to chase my dreams.* I want to thank all my family members: B. R. Ranwa and Para Devi (grandparents-in-law), Sugni Devi and Tulsi Devi (grandparents) for their support throughout. I am eternally grateful to my parents, Preshita and Parsa Ram Golia, and my parents-in-law, Indra and Ramkishan Ranwa, for their unwavering support and encouragement throughout this journey. Their belief in me and their constant encouragement have meant the world to me and have been a major driving force behind my success on completing this thesis. *Thank you Maa, Papa!* I also want to thank my brother, Vibhu, his presence in my life has brought me so much joy and comfort. Thank you Amit Bhaiya, Praveen, Anshu, Tanu, Shubhu, Adi, Piu, Mohi and Urvi for their constant love and support.

Last but not the least, I want to thank my partner, Rajendra, for his patience, understanding, and support throughout this process. I am so grateful to have him by my side through ups and downs. *Thank you Raj for tolerating me!*

Papa, Mummy, Raj, we did it! Thank you all for being there for me every step of the way and for being such an important part of my life.

Contents

Acknowledgements ix

List of Tables xiv

List of Figures xv

I Prologue 1

1 Introduction 2

1.1 Functional Synthesis . . . 3

1.2 Contributions . . . 7

1.2.1 A Data-Driven Approach . . . 7

1.2.2 Synthesizing Functions with Explicit Dependencies . . . 9

1.2.3 Program Synthesis via Functional Synthesis . . . 10

1.2.4 Beyond All-or-Nothing Approach . . . 11

1.3 Tools . . . 12

1.4 Outline . . . 13

2 Preliminaries and Related Work 14

2.1 Boolean Formulas and Beyond . . . 14

2.2 Skolem Synthesis . . . 16

2.3 Henkin Synthesis . . . 18

2.4 Program Synthesis . . . 20

2.5 Satisficing Synthesis . . . 22

II Ingredients for Functional Synthesis 24

3 Preprocessing 26

3.1 Finding Unates 26

3.2 Extracting Unique Functions 28

4 Data Generation 32

4.1 What kind of a Sampling Strategy? 33

4.2 Adaptative Weighted Sampling Strategy 34

5 Candidate Learning 37

5.1 Learning via Binary Classification 38

5.2 Learning via Multi Classification 41

6 Verification and Repairing of Candidates 48

6.1 Verification 48

6.2 Repair of Candidates 49

6.2.1 Fault Localization 50

6.2.2 Repair Synthesis 52

III Recipe and Results for Functional Synthesis 58

7 Skolem Synthesis 60

7.1 Approach 61

7.2 Experimental Results 65

8 Henkin Synthesis 83

8.1 Approach 84

8.2 Experimental Results 92

IV Generalization from Functional Synthesis 97

9 Program Synthesis as Dependency Quantified Formulas 99

9.1 Reduction of $\mathbb{T}$-Constrained Synthesis to DQF($\mathbb{T}$) 101

9.1.1 When $\mathbb{T}$ is Bitvector (BV): 104

9.2 Experimental Results 105

10 Satisficing Synthesis **110**

10.1 Overview 112

10.2 Approach 115

10.3 Experimental Results 122

V Conclusion **128**

11 Concluding Remarks **130**

12 Future Directions **132**

References **135**

List of Tables

4.1 Data Generation: An example of uniform sampling 33

7.1 Skolem Synthesis: No. of benchmarks solved by different tools. 68

7.3 Skolem Synthesis: Manthan with different samplers. 70

7.5 Skolem synthesis: Pairwise comparison of Manthan2 with Manthan . . 77

7.6 Skolem Synthesis: Performance Summary over 609 benchmarks . . . 77

7.8 Skolem Synthesis: # of benchmarks with different size unique functions. 79

8.1 Henkin synthesis: Manthan3 vs other state-of-the-art tools. 95

9.1 Program synthesis as DQF($\mathbb{T}$): Tools used in evaluation. 106

9.2 Program synthesis as DQF($\mathbb{T}$): Benchmarks solved. 107

9.3 Program synthesis as DQF($\mathbb{T}$): Top tools in different categories. . . . 108

9.4 Program synthesis as DQF($\mathbb{T}$): QBF instances solved. 109

10.1 Satisificing synthesis: $\varphi_H(X, Y)$ as formula and $\varphi_S(X, Y)$ as data. . . 124

10.2 Satisificing synthesis: $\varphi_H(X, Y)$ and $\varphi_S(X, Y)$ as formulas. 126

10.3 Satisificing synthesis: $\varphi_H(X, Y)$ and $\varphi_S(X, Y)$ as data. 127

List of Figures

1.1 Application of functional synthesis: Circuit repair. 5

1.2 Overview of data-driven approach for functional synthesis. 8

4.1 Data Generation: An adaptive weighted sampling strategy. 35

5.1 An Example for learning candidates via binary classification. 39

5.2 An Example for learning candidates via multi classification. 46

6.1 Repair of candidates: Moving from decision trees to decision list. . . 54

7.1 Manthan for Skolem synthesis. 60

7.2 Skolem synthesis by Manthan example: Data generation 64

7.3 Skolem synthesis by Manthan example: Learning candidates 64

7.4 Skolem Synthesis: Manthan versus competing tools. 68

7.5 Skolem Synthesis: Heatmap for Manthan with # of instances solved. 71

7.6 Skolem synthesis: Time for different phases of Manthan. 72

7.7 Skolem synthesis: Impact of self-substitution on *pdtpmsmiim*. 74

7.8 Skolem synthesis: Impact of self-substitution on *pdtpmsmiim-all-bit*. . 74

7.9 Skolem synthesis: Manthan2 vis-a-vis state-of-the-art synthesis tools. 77

7.10 Skolem Synthesis: # of benchmarks by % ratio of unique variables. . 79

7.11 Skolem synthesis: Multi-classification and LexMaxSAT turned off. . . 81

7.12 Skolem synthesis: Multi-classification and LexMaxSAT turned on. . . 82

8.1 Henkin synthesis by Manthan3 example: Data generations. 91

8.2 Henkin synthesis by Manthan3 example: Learning candidate for y_1. . 91

8.3 Henkin synthesis by Manthan3 example: Learning candidate for y_2. . 91

8.4 Henkin synthesis by Manthan3 example: Learning candidate for y_3. . 91

8.5 Henkin synthesis: VBS with/without Manthan3. 94

8.6 Henkin synthesis: Manthan3 vs. VBS(HQS2+Pedant). 95

8.7 Henkin synthesis: Manthan3 vs.HQS2. 96

8.8 Henkin synthesis: Manthan3 vs. Pedant. 96

8.9 Henkin synthesis: Pedant vs.HQS2. 96

10.1 HSsynth for satisficing synthesis: 113

10.2 Satisificing synthesis: Satisficing measure of synthesized $F(X)$. 127

Part I

Prologue

Chapter 1

Introduction

Imagine a world where complex surgeries are performed with flawless precision by robotic surgeons, self-driving cars effortlessly navigate through busy streets, and smart homes anticipate our needs, creating a seamless and convenient living environment. This futuristic vision is gradually becoming a reality as sophisticated technologies continue to integrate into our daily lives. In this rapidly evolving world, the reliability and trustworthiness of automated systems play a vital role. We rely on these systems to make critical decisions, handle sensitive tasks, and enhance our overall quality of life. Therefore, the question arises: How can we ensure that these automated systems are trustworthy? How can we trust on the automated system that they will work as intended in all possible scenarios they might face?

This is where the quest for dependable systems begins. Trust is sought through two prominent approaches: (i) certification, which involves subjecting these systems to extensive testing and formal verification. Certification ensures that the automated systems fulfil their promises by subjecting them to rigorous tests and verification processes, guaranteeing that the system meets strict standards of accuracy and reliability, (ii) the second approach takes a different path. Instead of certifying the system's performance, the approach focuses on constructing systems that are inherently reliable and accurate, that is building the automated system in such a way that their correct behavior is guaranteed by design.

This thesis embarks on a journey to explore the latter approach, which we refer to as "correct by construction." In particular, the thesis is about automated synthesis of a correct-by-construction system from the given specifications. Automated synthesis is a technique that utilizes formal specifications to automatically generate systems, such as functions, programs, or circuits, that can be proven to satisfy the given requirements. While automated synthesis is a well-studied area in computer science, recent years have seen significant advancements in artificial intelligence, formal methods, and automated reasoning. These advancements develop and deploy mathematically-rigorous and algorithmically-efficient solutions to verify and design the correct behavior of systems.

This thesis takes a comprehensive perspective on functional synthesis which involves synthesizing functions that satisfy given specification. The thesis views functional synthesis through the lens of recent advancements in machine learning and formal methods. By harnessing the power of these cutting-edge technologies, our objective is to propose scalable techniques to automatically generate functions that meet the specified formal requirements.

1.1 Functional Synthesis

Functional synthesis is the process of finding a function that provably meets requirements of the specification. The objective of functional synthesis is to synthesize functions that correctly map each possible input assignments to an appropriate output, ensuring that the combined inputs and outputs satisfy the given specification.

Functional synthesis is a fundamental problem of computer science and has a long history dating back to the work of Boole [Boo47] and Lowenheim [L10], they studied variants of this problem in the context of finding most general unifiers. It was later pursued rigorously by Skolem and Herbrand [L10]. Skolem focused on the existence of *Skolem-normal form*, which refers to a form in which existentially quantified variables in a given specification could be eliminated. In regards to that, Functional synthesis is also referred as **Skolem synthesis**. Specifically, the given a specification, $\forall X \exists Y \varphi(X, Y)$, where X and Y are sets of inputs and outputs and φ describes the underlying given relation specification between inputs and outputs, the task of functional synthesis is to synthesize outputs Y in terms of inputs X (i.e. $Y = F(X)$) such that the specification is met. The functions corresponding to output Y is called Skolem functions.

The problem has a wide range of applications, including certified QBF solving [RT15, RTRS18], circuit synthesis and repair [KS00], program synthesis [SGF13], automated program repair [JMF14], cryptanalysis [MM00], logic minimization [Bra89, BS89]. Few of the applications are discussed in detail:

Cryptanalysis In cryptography, large prime numbers are used to create strong encryption keys. If someone were able to efficiently compute the factors of a large composite number, they could potentially break the encryption key and access the information it was protecting. Let $X = \{x_1, \ldots, x_{2n}\}$, $Y^1 = \{y^1_1, \ldots, y^1_n\}$, $Y^2 = \{y^2_1, \ldots, y^2_n\}$, and specification $\varphi(X, Y^1, Y^2)$ be $X = (Y^1 \times_n Y^2) \wedge \neg(Y^1 = 1_{[n]}) \wedge \neg(Y^2 = 1_{[n]})$, where 1_n represent the bit vector 1 of size n. The specification states that Y^1 and Y^2 are non-trivial factors of X. The Skolem function corresponding to Y^1 and Y^2 can be used to compute the factors of X for the formula $\forall X \exists Y^1, Y^2 \varphi(X, Y^1, Y^2)$, and efficiently computing functions to compute these factors could potentially be used to break cryptographic systems.

Circuit Repair In circuit repair, given an incomplete implementation and specification, the task is to complete the implementation s.t. it is functionally equivalent to specification [GRS+13]. Skolem synthesis could be used to regenerate faulty circuits. Let us consider specific example[1] shown in Figure 1.1. In the Figure 1.1, BB_1 and BB_2 are faculty part of the circuit and we need to synthesize y_1 and y_2 in terms of inputs x_1, x_2 to fill the black-box BB_1 and BB_2. For the considered example, $\forall x_1, x_2, \exists y_1, y_2 \neg(((y_1 \vee y_2) \vee (x_1 \wedge \neg x_2)) \oplus (x_1 \oplus x_2))$

[1]Image is taken(modified) from Equivalence Checking of Partial Designs Using Dependency Quantified Boolean Formulae, Gitina et al '13 [GRS+13].

would represent the underlying specification, and Skolem functions corresponding to y_1 and y_2 would be the required repair.

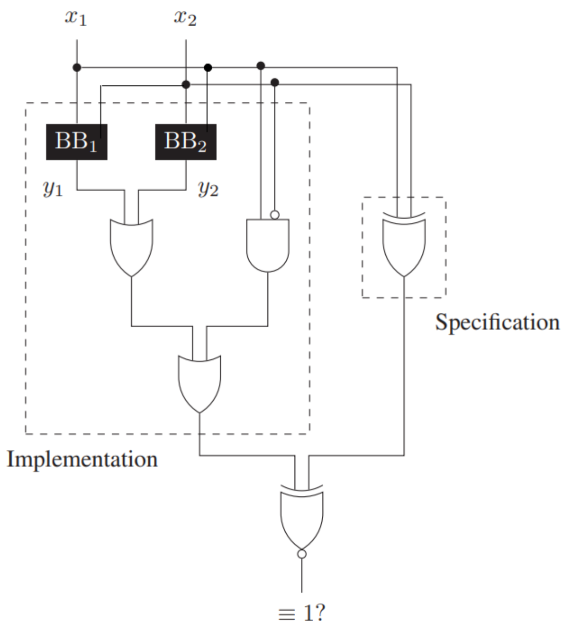


Figure 1.1: Application of functional synthesis: Circuit repair.

Certified QBF Solving The QBF-SAT problem is the problem of determining whether a given Quantified Boolean Formula (QBF) is semantically equivalent to True. This problem is known to be in the complexity class PSPACE-complete [Gar79] and is thought to be significantly harder to solve than propositional satisfiability. Despite this, there have been significant advances in practical QBF-SAT solving in recent years [Jan18b, Jan18a, LB10, LE17, Rab19, RT15, RTRS18]. This has led researchers in various fields, such as equivalence checking of partial functions [GRS+13], finding strategies for incomplete games [PRA01], controller synthesis [BKS14], to encode their problems as QBF-SAT instances and use modern QBF-SAT solvers to solve them. In many of these cases, however, it is not sufficient to simply determine whether a QBF is True; it is also necessary to extract a plan, winning strategy, or similar artifact from the QBF-SAT reasoning in order to apply the solution to the specific problem at hand. Certified QBF solving is a method for addressing this need. Certified QBF solving is a method that involves producing a certificate, a proof of the truth or falsity of a QBF, that can be used to extract additional information beyond a simple yes/no answer. When a QBF is True, Skolem functions can be used to construct a certificate of satisfiability. If the specification in which the outputs are replaced with the synthesized Skolem function is tautology, then QBF is True. If a QBF is False, a certificate of unsatisfiability can be obtained by applying the same process to a QBF in the opposite quantifier form.

The theoretical investigations have shown that certain instances of functional synthesis can be exceptionally time-consuming. Specifically, complexity theory studies have demonstrated that there are instances where Boolean functional synthesis requires super-polynomial time. Furthermore, it has been shown that there are instances where a polynomial-sized Skolem function vector is insufficient unless the Polynomial Hierarchy (PH) collapses [ACG+18]. Despite the inherent theoretical difficulty, the practical relevance of functional synthesis has prompted the development of algorithms that exhibit scalability. Over the past two decades, there has been a growing interest in functional synthesis, leading to the emergence of effective approaches for synthesizing functions effectively [AAC+19, ACG+18, Rab19]. Nevertheless, practical challenges persist, and there are still scenarios where state of the arts methods face limitations.

This thesis takes a comprehensive perspective on functional synthesis, examining it from the standpoint of recent advancements in machine learning and formal methods. Notably, formal methods have experienced remarkable advancements, particularly in the area of Satisfiability (SAT) problem solving. The SAT problem involves finding a satisfying assignment, which is a truth assignment to variables in a specification under which it evaluates to true. It is a widely known NP-complete problem [Coo23]. However, significant progress has been made in SAT solvers over the past decade, leading to what is often referred to as the “NP revolution” [MSLM09]. According to Knuth, who discussed these advancements in SAT solvers, “these so-called SAT solvers can now routinely find solutions to practical problems that involve millions of variables and were previously considered extremely challenging" [Knu15]. Furthermore, recent research in this field has extended beyond finding a single satisfying assignment, surpassing the limitations of NP. Efficient methods have been developed for counting the number of satisfying assignments [SGM20, SRSM19], as well as constrained samplers for generating satisfying assignments [CFM$^+$15, CMV13, SGRM18], algorithms for maximum satisfiability [MML14], and more. These advancements have significantly broadened the scope and capabilities of formal methods tools.

Inspired by recent progress in machine learning and formal methods, the thesis proposes efficient and scalable techniques for functional synthesis.

1.2 Contributions

This thesis presents significant contributions aimed at advancing the field of functional synthesis. The specific contributions are discussed in detail below.

1.2.1 A Data-Driven Approach

We proposed a data-driven approach, called Manthan [GRM20, GSRM21], to synthesize functions efficiently. Motivated by progress in machine learning, Manthan views functional synthesis as a classification problem, relying on advances in constrained sampling for data generation, and advances in formal methods for a novel proof-guided repair and provable verification.

Given a relation specification $\varphi(X, Y)$ over input X and output Y. The task for Manthan is to synthesize output Y in terms of X, that is, $Y := F(X)$ such that given specification is met. This process involves three essential components, as illustrated in Figure 1.2: (i) constrained sampling for data generation, (ii) machine learning to learn candidate functions using a dependency-driven classifier, and (iii) verification and repair of candidates using formal methods. Each components of Manthan is crucial and employs different techniques to ensure scalability.

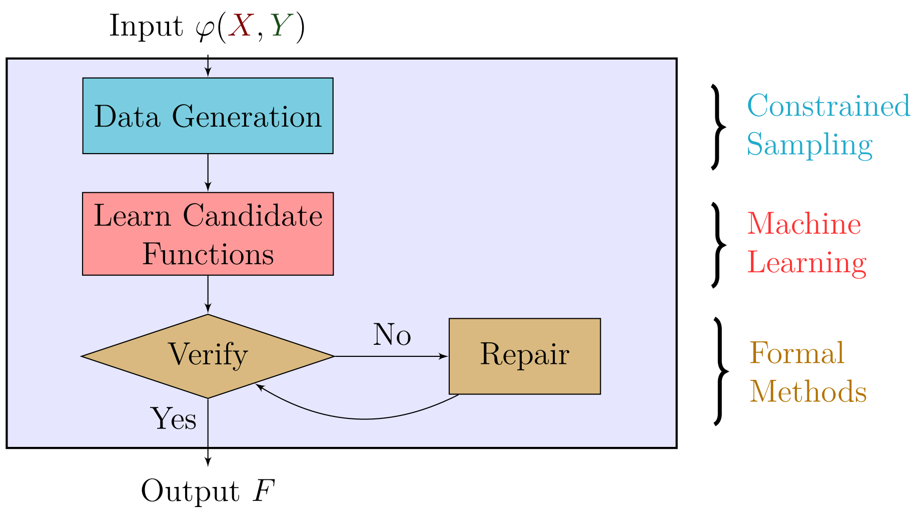


Figure 1.2: Overview of data-driven approach for functional synthesis.

Data Generations: Current machine learning techniques typically rely on training data composed of feature valuations and their corresponding labels. In our case, we consider X as the feature set and Y as the labels. However, unlike traditional machine learning setups where each assignment to X has a unique label (Y), our scenario involves a relation between X and Y, which may not be strictly a function. To address this, we have developed an *adaptive weighted sampling strategy* that enables the generation of a representative dataset suitable for training a classifier. We rely on progress in constrained sampling to generate data efficiently.

Learn Candidates Functions: When considering training data consisting of feature valuations (X) and their associated labels (Y), a common machine learning approach is to employ multi-class classification to learn a symbolic representation, Y = h(X), where h represents the learned classifier. However, this approach does not guarantee that h can be represented as a vector of Boolean functions. To address this limitation, the thesis introduces an innovative *dependency-aware classifier*. This classifier constructs a vector of decision trees, each corresponding to an output, with each decision tree expressed as a Boolean function. This approach ensures that the learned classifier can be represented as a collection of Boolean functions.

Verify and Repair: As machine learning techniques often yield good but approximate solutions, we enhance Manthan with automated reasoning methods to verify the accuracy of decision tree-based candidates and to get a counterexample for which candidates fail to meet the specification. In order to fix the counterexample and to leverage the high test accuracy achieved by machine learning models, we propose a *proof-guided repair* technique. This iterative approach aims to identify and apply minor repairs to the candidate functions until we converge to a provably correct function vector. We utilize a **MaxSAT solver** to identify potential repair candidates and extract **unsatisfiability cores** from the infeasibility proofs. These cores capture the reasons why the candidate functions fail to meet the specification and assist in constructing effective repairs.

Impact. We have developed a tightly integrated framework called Manthan that exhibits remarkable scalability. Notably, Manthan has successfully synthesized functions for 509 out of a total of 609 instances from the standard suite. To provide context, previous state-of-the-art tools [ACG+18, AAC+19, RTRS18] were only able to solve a maximum of 280 instances. This means that Manthan has surpassed the state-of-the-art by solving an additional 40% of instances, demonstrating its significant improvement in performance.

1.2.2 Synthesizing Functions with Explicit Dependencies

The achievements in functional synthesis paved the way for studying different variants of it, which are more harder from the complexity-theoretic perspective and more useful in the real world. We turned our attention towards functional synthesis with explicit dependencies, in which instead of depending on all inputs, the output Y is only allowed to depend on a subset of inputs X. In particular, given a relation specification $\varphi(X, Y)$ over input X and output Y, the task is to synthesize each output y_i in terms of H_i, that is, $Y := f(H_i)$, where H_i is a subset of input X such that the given specification is met. Identifying whether such $f(H_i)$ exists for each y_i is NEXPTIME-complete [PRA01]. The explicit dependencies provides more succinct descriptive power and have wide-ranging applications in computer design and in formal methods, such as engineering change of order [JKL20], topologically constrained synthesis [BCJ14], equivalence checking of partial functions [GRS+13], finding strategies for incomplete games [PRA01], controller synthesis [BKS14], circuit realizability [BCJ14], program synthesis [SGF13], and synthesis of fragments of linear-time temporal logic [CHOP13].

Towards this, we proposed Manthan3 [GRM23] to synthesize functions with explicit dependencies. Manthan3 generates data, learns and repairs the candidates in accordance with additional constraints imposed by these dependencies.

Impact. We showed that different approaches are suited for different classes of instances, and Manthan3 pushed the envelope in synthesis with explicit dependencies by handling the instances for which none of the state-of-the-art tools could not synthesize functions. In particular, all state-of-the-tools were able to synthesize functions for 204 instances out of 563 standard instances in a virtual best portfolio setting, and Manthan3 could achieve the smallest synthesizing time on 42 benchmarks out of 204, including 26 instances for which none of the other competing tools could synthesize functions.

1.2.3 Program Synthesis via Functional Synthesis

Motivated by its impressive scalability, our focus shifted towards program synthesis as a means to explore potential applications for functional synthesis. Program synthesis involves automatically generating a program that meets specific user requirements. In recent developments, these requirements encompass both syntactic (grammar-related) and semantic aspects [ABJ+13a]. In the thesis, we specifically investigated the significance of syntactic requirements (grammar) in program synthesis, and examined the feasibility of synthesis techniques that do not rely on grammatical restrictions, which we termed Theory-constrained synthesis, or $\mathbb{T}$-constrained synthesis. We proposed a reduction of $\mathbb{T}$-constrained synthesis to functional synthesis with explicit dependencies when there are no syntactic requirements.

Impact. Our reduction allows us to transform Manthan as a state of the art approach for program synthesis tasks over bit-vector theory [GRM21]. Manthan was able to synthesize functions for 592 instance over a total of 609, whereas, the state-of-the-art SyGuS solver could synthesize functions for *only* 488 instances — Manthan was able to handle more Sinstances than the state-of-the-art program synthesis solver. Our reduction of program synthesis to functional synthesis motivates need of domain-agnostic approaches.

1.2.4 Beyond All-or-Nothing Approach

Traditional approaches to designing and verifying systems often adhere to an “all-or-nothing” principle, where they either provide rigorous theoretical guarantees or no guarantees at all. Methods that offer rigorous guarantees tend to sacrifice scalability, while scalable techniques often lack guarantees. This ”all-or-nothing” approach poses a significant bottleneck to the widespread adoption of synthesis and verification techniques in real-world settings.

In reality, not all constraints have equal priorities. Certain constraints may have higher priority and require stricter guarantees than others. For instance, when synthesizing a controller for an autonomous vehicle, a high-priority constraint would be to ensure that the vehicle does not harm bystanders. Conversely, while it is desirable for the vehicle to exhibit smooth speed changes and lane transitions, these constraints may not carry the same level of importance. Acknowledging the varying priority levels of constraints in real-world scenarios can lead to more practical and adaptable approaches to synthesis and verification, as it allows for a more nuanced treatment of constraints based on their significance.

Such scenarios also arise in other contexts: consider a financial institution that needs to decide whom to give a loan. They must follow mandatory regulations such as non-discrimination based on gender, race and more protected attributes.

Furthermore, the prior dataset available in this regard could be of great use to inform the likelihood of the loan being defaulted. Therefore, an ideal system should agree as much as possible with the dataset while following the mandatory regulations.

The aforementioned cases highlight the need for a general framework where end users can define different classes of constraints: *hard constraints* that must never be violated, and *soft constraints* that should be satisfied to the greatest extent possible. In such cases, it is typically desirable to have a quantifiable measure that captures the degree to which the system satisfies the soft constraints. To address this, a synthesis engine is necessary to ensure that the system can meet the specified threshold for satisfying these soft constraints.

Towards this, we proposed a general-purpose framework, called HSsynth [GRM] that relies on advances in automated reasoning and formal methods to provably satisfy hard constraints and achieve satisficing threshold on soft constraints.

Impact. The prototype implementation of HSsynth demonstrates its versatility in handling various scenarios involving different combinations of hard and soft constraints, such as formulas, data, and their combinations, to synthesize the required system. When treating soft constraints as data, HSsynth achieved efficient system repair to satisfy hard constraint, taking less than 20 seconds. Moreover, the satisficing measure of the synthesized system exceeded 80% for nearly all considered datasets and hard constraints.

In cases where both soft and hard constraints were expressed as formulas, HSsynth exhibited impressive performance, completing synthesis in approximately half the time of the complete synthesis approach. This time-saving was observed for approximately 25% to 60% for range of hard constraints examined.

1.3 Tools

The following open-source tools have been developed as part of this thesis:

Manthan: `https://github.com/meelgroup/manthan`

DeQuS: `https://github.com/meelgroup/dequs`

1.4 Outline

This thesis is divided into five parts. The next chapter 2 introduces the necessary notations and background.

We then move to Part II discuss the all necessary ingredients required to do functional synthesis via formal methods and machine learning. In this part, we first discuss the different preprocessing techniques in Chapter 3. In Chapter 4, we discussed how to generate data given a relation specification. Moving on, we discussed machine learning techniques to learn candidate functions in Chapter 5, and formal methods technique to verify and repair candidates in Chapter 6.

We then move to Part III that discusses the recipe and results for functional synthesis with or without explicit dependencies. Chapter 7 presents the algorithm and detailed experimental results for Skolem synthesis (functional synthesis without explicit dependencies). Chapter 8 presents the algorithm and detailed experimental results for Henkin synthesis (functional synthesis with explicit dependencies).

We then move to Part IV that discusses generalization of functional synthesis. Chapter 9 establishes the link between functional synthesis and program synthesis by showing a reduction of program synthesis to dependency quantified formulas, and Chapter 10 presents a complete approach to uplift the machine learning and formal method based techniques to handle real-world scenarios in which all given constraints might not have the same priority. Finally, Part V summarizes the thesis.

Chapter 2

Preliminaries and Related Work

2.1 Boolean Formulas and Beyond

We use a lower case letter to represent a propositional variable and an upper case letter to represent a set of variables. A literal is either a variable or its negation, and a clause is considered as a disjunction of literals. A formula φ represented as conjunction of clauses is considered in Conjunctive Normal Form (CNF). $\mathit{Vars}(\varphi)$ represents the set of variables appearing in φ. A satisfying assignment(σ) of the formula φ maps $\mathit{Vars}(\varphi)$ to $\{0, 1\}$ such that φ evaluates to True under σ. We use $\sigma \models \varphi$ to represent σ as a satisfying assignment of φ. For a set of variables V, we used $\sigma[V]$ to denote the restriction of σ to V. If φ evaluates to True for all possible valuation of $\mathit{Vars}(\varphi)$, φ is considered as tautology.

We use $\varphi(V)|_{v_i=b}$ to denote *substitutions*: a formula obtained after substituting every occurrence of v_i in $\varphi(V)$ by b, where b can be a constant (0 or 1) or a formula, and V is $\mathit{Vars}(\varphi)$. Let $P \subseteq Vars(\varphi)$ is called *projection set* of formula φ such that an assignment $\sigma_{\downarrow}P : P \mapsto \{0, 1\}$ can be extended to an assignment σ such that $\sigma \models \varphi$ and $\sigma_{\downarrow}P = \sigma[P]$. The satisfiable region of a formula φ is a set of all satisfying assignments of φ.

Unsatisfiable core. An *unsatisfiable core* of a formula in CNF is a subset of clauses for which there is no satisfying assignment. We use UnsatCore to denote an unsatisfiable core when the formula is understood from the context.

MaxSAT. For a given CNF formula in which some clauses are declared as *hard constraints* and the rest are declared as *soft constraints*, the problem of (partial) MaxSAT is to find an assignment of the given formula that satisfies all hard constraints and maximizes the number of satisfied soft constraints.

Model counting. Given a formula φ as input, compute the number of satisfying assignments of formula, that is, compute $|sol(\varphi)|$. The problem of projected model counting is given a formula and a projection set P, compute $|sol(\varphi)_{\downarrow P}|$.

Constrained sampling. A constrained sampler takes a formula φ and a number of required satisfying assignments N, and returns satisfying assignments $\sigma_1, \ldots, \sigma_N$ in accordance to the distribution induced by the sampler.

Given a propositional formula $\varphi(X)$ and a weight function $W(\cdot)$ assigning non-negative weights to every literal of the formula, we refer to the *weight* of a satisfying assignment σ, denoted as $W(\sigma)$, as the product of weights of all the literals appearing in σ, i.e., $W(\sigma) = \prod_{l \in \sigma} W(l)$. A *sampler* $\mathcal{A}(\cdot, \cdot)$ is a probabilistic generator that guarantees $\forall \sigma \in R_{\varphi}$, $\mathsf{Pr}\left[\mathcal{A}(\varphi, \mathsf{Bias}) = \sigma\right] \propto W(\sigma)$, where Bias is weight vector for literals of variables of φ.

Uniform sampling is to generate satisfying assignment of given specification uniformly at random, that is, weight of every literals is considered to be 0.5. A uniform sampler samples the required number of satisfying assignments uniformly at random from the solution space of the formula.

Let us take a particular example to understand the weighted sampling. Consider weights of literals is given to us as: $\langle W(x_1) = 0.5,\ W(x_2) = 0.5,\ W(y_1) = 0.9,\ W(y_2) = 0.1 \rangle$. The weight of the satisfying assignment $\sigma : \langle x_1 \leftrightarrow 1,\ x_1 \leftrightarrow 0,\ y_1 \leftrightarrow 0,\ y_2 \leftrightarrow 1 \rangle$ is product of weights of literals of σ, that is, $0.5 \times (1 - 0.5) \times$

$(1-0.9) \times 0.1$ is 0.0025. Therefore, the probability of sampling σ from solution space of specification is proportional to 0.0025.

This thesis explores the problem of functional synthesis, commonly referred to as Skolem synthesis in the literature, as well as synthesis with explicit dependencies, known as Henkin synthesis. Furthermore, the thesis also introduces Satisficing synthesis, which goes beyond the traditional all-or-nothing approach. In the following sections, we will provide a detailed formal introduction to each of these problems.

2.2 Skolem Synthesis

A Quantified Boolean Formula (QBF) ϕ is $Q_1X_1, Q_2X_2, \ldots, Q_nX_n\varphi(X_1, \ldots, X_n)$, where each Q_i belongs to the set $\{\forall, \exists\}$ quantifiers, X_i represents a set of variables, and φ is a CNF formula over X_1 to X_n. In the context of 2-QBF formulas, only two levels of quantification are permitted. Thus, a 2-QBF formula ϕ can be represented as $Q_1X_1, Q_2X_2\varphi(X_1, X_2)$. This thesis focuses exclusively on 2-QBF formulas.

For a formula ϕ of the form $\forall X\exists\varphi(X, Y)$, where $X = \{x_1, \ldots, x_n\}$ and $Y = \{y_1, \ldots, y_m\}$, it is considered False if there exists an assignment of X to $\{0, 1\}$ for which there is no assignment of Y to $\{0, 1\}$ such that $\varphi(X, Y)$ is satisfiable. Conversely, ϕ is considered True. The problem of Skolem synthesis deals with synthesizing functions for each y_i in terms of X such that φ is satisfiable.

Problem Statement

Skolem Synthesis:

Given a Boolean specification $\varphi(X, Y)$ between set of inputs $X = \{x_1, \cdots, x_n\}$ and vector of outputs $Y = \langle y_1, \cdots, y_m\rangle$, the problem of **Skolem synthesis** is to synthesize a function vector $\boldsymbol{f} = \langle f_1(X), \cdots, f_m(X)\rangle$ such that $y_i \leftrightarrow f_i(X)$ and $\forall X(\exists Y\varphi(X, Y) \leftrightarrow \varphi(X, \boldsymbol{f}))$. We refer to $\boldsymbol{f}$ as the Skolem function vector and f_i as the Skolem function for y_i.

Related work. The origins of Boolean functional synthesis can be traced back to Boole's seminal work [Boo47], which was further developed in terms of decidability by Lowenheim and Skolem [L10]. From a complexity theory perspective, solving Boolean functional synthesis is classified as PSPACE-complete [Gar79], and specifically, determining True and False 2-QBF formula falls under the complexity class Σ_2^p. Complexity studies have shown that there exist instances where Boolean functional synthesis requires super-polynomial time, and it has also been demonstrated that polynomial-sized Skolem function vectors are insufficient unless the Polynomial Hierarchy (PH) collapses [ACG+18].

Motivated by the success of the CEGAR (Counter-Example Guided Abstraction Refinement) approach in model checking, CEGAR-based techniques have been explored in the context of synthesis. The key idea is to use Conflict-Driven Clause Learning (CDCL) SAT solvers to verify and refine candidate Skolem functions [AAC+19, ACG+18, ACJS17, JSC+15]. Another line of research focuses on representing the specification, $F(X, Y)$, in forms that are amenable to efficient synthesis for a class of functions. Early approaches employed Reduced Ordered Binary Decision Diagrams (ROBDDs), building on the functional composition approach proposed by Balabanov and Jiang [BJ11]. Chakraborty et al. extended the ROBDD-based approach to factored specifications, leveraging the work of Tabajara and Vardi [CFTV18, TV17]. Factored specifications had previously been explored in the context of CEGAR-based approaches. Drawing inspiration from the success of knowledge compilation in probabilistic reasoning, Akshay et al. made significant progress in proposing a new negation normal form called SynNNF [AAC+19]. SynNNF provides a generalized and efficient representation for functional specifications, facilitating efficient functional synthesis.

Incremental determinization has been another avenue of research for constructing Skolem functions incrementally [HSB14, JBS+07, NPL+12, Rab19, RTRS18]. Several approaches have been proposed for the specific case when the specification $\exists Y \varphi(X, Y)$ is valid, i.e., $\forall X \exists Y \varphi(X, Y)$ is true. Chakraborty et al. recently introduced an approach based on the idea of sequential relational decomposition, where each CNF clause of the specification is treated as a combination of input and output clauses, and a cooperation-based strategy is employed [CFTV18]. The advancement of modern CDCL solvers has led to the exploration of heuristics for problems beyond NP, including the extraction of Skolem functions from proofs constructed for formulas expressed as $\forall X \exists Y \varphi(X, Y)$ [BJ11, BJ12].

2.3 Henkin Synthesis

A formula ϕ is Dependency Quantified Boolean Formula (DQBF) if it can be represented as $\phi : \forall x_1 \dots x_n \; \exists^{H_1} y_1 \dots \exists^{H_m} y_m \varphi(X, Y)$ where $X = \{x_1, \dots, x_n\}$, $Y = \{y_1, \dots, y_m\}$ and $H_i \subseteq X$ represents the dependency set of y_i, that is, variable y_i can only depend on H_i. Each H_i is called Henkin dependency and each quantifier $\exists^{H_i}$ is called *Henkin* quantifier [Hen61].

A DQBF ϕ is considered to be True, if there exists a function $f_i : \{0,1\}^{|H_i|} \mapsto \{0,1\}$ for each existentially quantified variable y_i, such that $\varphi(X, f_1(H_1), \dots, f_m(H_m))$, obtained by substitution of each y_i by its corresponding function f_i, is a tautology. Given a DQBF ϕ, the problem of DQBF satisfiability, is to determine whether a given DQBF is True or False. In several cases, the decision problem is not enough, and we are interested in a functional formulation, called Henkin synthesis, wherein the task is to synthesize functions for each of the existentially quantified y_i variables.

Problem Statement

Henkin Synthesis:

Given a True DQBF formula, $\forall x_1 \dots x_n \; \exists^{H_1} y_1 \dots \exists^{H_m} y_m \; \varphi(x_1, \dots, x_n, y_1, \dots, y_m)$ where $x_1, \dots, x_n \in X$, $y_1, \dots, y_m \in Y$, $H_i \subseteq X$, the problem of **Henkin Synthesis** is to synthesize a function vector $\boldsymbol{f} : \langle f_1, \dots, f_m \rangle$ such that $\varphi(X, f_1(H_1), \dots, f_m(H_m))$ is a tautology. $\boldsymbol{f}$ is called Henkin function vector and each f_i is a Henkin function.

From now onwards, we used $\forall X \exists^{H_1} y_1 \ldots \exists^{H_m} y_m \; \varphi(x_1, \ldots, x_n, \; y_1, \ldots, y_m)$ and $\forall X \exists^{H} Y \varphi(X, Y)$ interchangeably.

Henkin synthesis vs Skolem synthesis. Henkin synthesis generalize the Skolem synthesis. The problem of Skolem synthesis is defined in the context of a special case of DQBF, called 2-Quantified Boolean Formula (2-QBF), for which $H_1 = H_2 = \ldots = H_m = X$. In such a case, one omits the usage of H_i and simply represents ϕ as $\forall X \exists Y \varphi(X, Y)$. $\phi : \forall X \exists Y \varphi(X, Y)$ is a 2-QBF formula, where variables X is universally quantified and Y is existentially quantified variables and $\varphi(X, Y)$ is an arbitrary Boolean formula over X and Y.

Considering an example to highlight the difference between Henkin and Skolem functions. Let $\phi : \forall x_1, x_2 \; \exists y_1 \; \varphi(x_1, x_2, y_1)$, where $\varphi(x_1, x_2, y_1)$ is $(y_1 \leftrightarrow (x_1 \vee x_2))$ — ϕ is a 2-QBF formula. There exists a Skolem function $f_1(x_1, x_2) := (x_1 \vee x_2)$ such that $\forall x_1, x_2 (\exists y_1 \varphi(x_1, x_2, y_1) \leftrightarrow \varphi(x_1, x_2, f_1(x_1, x_2)))$. Now, let us consider the case where we have restricted the dependencies for y_1, $\phi : \forall x_1, x_2 \exists^{H_1} y_1 \; (y_1 \leftrightarrow (x_1 \vee x_2))$, where $H_1 = \{x_1\}$. ϕ is a DQBF formula, and y_1 can only depend on x_1, that is, in some sense it can only *see* the valuation of x_1. The only possible functions over x_1 are: $\{0, 1, x_1, \neg x_1\}$ and by simple enumeration over all choices, we can conclude that there does not exists a function, $f_1(x_1)$ such that $\varphi(x_1, x_2, f_1(x_1))$ is a tautology; that is, there does not exists a Henkin functions and corresponding DQBF is false.

Related work. DQBF has been classified as NEXPTIME-complete in terms of complexity [PRA01]. However, due to its wide range of applications, there has been significant interest in solving DQBF problems in recent years [FKBV14, GRS+13, GWR+15, Sìč20, RSS21, TR19, WWSB16].

The first DPLL-based approach to solve DQBF satisfiability was introduced by Frohlich et al. [FKB12]. Building on this direction, Tentrup and Rabe proposed the idea of using clausal abstraction for DQBF solving [TR19]. Gitina et al. presented a basic variable elimination strategy for solving DQBF instances, which involves transforming a DQBF instance into a QBF instance by eliminating variables that introduce non-linear dependencies. This strategy was further enhanced through optimizations in subsequent work [GRS$^+$13, GWR$^+$15]. Frohlich et al. also proposed a similar approach, where a DQBF instance is transformed into a SAT instance through local universal expansion on each clause [FKBV14].

Wimmer et al. devised a method to obtain Henkin functions from DQBF solvers that are based on variable elimination techniques [WWSB16]. Elimination-based DQBF solvers perform a sequence of quantifier elimination transformations on a DQBF instance to obtain an equisatisfiable sequence of formulas $\varphi_1, \varphi_2, \ldots, \varphi_k$. They demonstrated that for a True DQBF instance, the Henkin function for φ_{i-1} can be obtained from φ_i.

Reichl, Slivovsky, and Szeider proposed a different approach using interpolation-based definition extraction [RSS21]. They introduced "arbiter variable" that represent the value of an existential variable for assignments of its dependency sets, in cases where the variable is not uniquely defined. The method extracts the definitions for existential variables in terms of their dependency sets and the arbiter variables. The approach is certifying by design and yields Henkin functions.

2.4 Program Synthesis

Program synthesis is synthesizing program from the given set of requirements or constraints. In our formulation, the constraints over the functions to be synthesized are specified in the vocabulary of a given background theory $\mathbb{T}$ along with the function symbols. Notice that the background theory specifies the domain of values for each variable type along with the interpretation for the function(s) and predicate symbols in the vocabulary.

Problem Statement

$\mathbb{T}$-constrained synthesis:

Given a background theory $\mathbb{T}$, a set of typed function symbols $\{f_1, f_2, \ldots f_k\}$, a specification φ over the vocabulary of $\mathbb{T} \cup \{f_1, f_2, \ldots f_k\}$, the problem of $\mathbb{T}$-constrained synthesis is to find the set of expressions $\{e_1, e_2, \ldots e_k\}$ defined over vocabulary of $\mathbb{T}$ such that $\varphi[f_1/e_1, f_2/e_2, \ldots f_k/e_k]$ is valid modulo $\mathbb{T}$.

Related work. The roots of program synthesis can be traced back to the 1930s. The idea of constructing interpretable solutions with proofs through the composition of solutions to smaller sub-problems [Kol32] build the foundation for program synthesis. This was followed by deductive synthesis strategies in [Gre81, MW71].

Alur et al. introduced the concept of using grammars for syntax-guided synthesis [ABJ+13b]. They demonstrated that incorporating grammars that is syntactic restriction can lead to efficient program synthesis and results in more interpretable programs [ABJ+13b, ARU17, URD+13].

Another approach is building synthesizers on top of SAT/SMT solvers [RDK+15]. The CVC4 synthesis engine [RDK+15] was the first implementation of a synthesis engine within an SMT solver. It extracts desired functions from the unsatisfiability proofs of negated synthesis conjectures. Reynolds et al. proposed two techniques in [RDK+15]: counterexample-guided quantifier instantiation and enumerative syntax-guided synthesis. The former is fast but can produce verbose solutions, while the latter is slower but yields concise solutions.

The recent advancements in machine learning have prompted several attempts to incorporate machine learning techniques in various synthesis domains, such as program synthesis [ABJ+13b], invariant generation [FG19], decision-tree synthesis for functions in Linear Integer Arithmetic theory using pre-specified examples [FG19], and strategy synthesis for QBF [Jan18b].

Data-driven approaches for invariant synthesis have been explored in the ICE learning framework [END+18, GLMN14, GNMR16], which leverages program behavior data from test executions to propose invariants by learning from the data. It checks for inductiveness and, in case of failure, expands the data using the generated counterexamples. The use of proof artifacts, such as unsat cores, has been investigated in verification [GLST05] and program repair [VR17], while MaxSAT has been employed in program debugging [BPR16, JMS11].

Recent research has also focused on understanding the impact of the provided grammar on the performance of existing synthesis tools. Kim et al. introduced semantic-guided synthesis [KHDR21], and Padhi et al. empirically demonstrated that increasing the expressiveness of the grammar leads to a significant deterioration in tool performance [PMNS19].

2.5 Satisficing Synthesis

Let us consider two types of constraints, (i) a set of hard constraints $\varphi_H(X,Y)$ and (ii) a set of soft constraints $\varphi_S(X,Y)$. Our goal is to synthesize a system $F(X)$ that satisfies all the hard constraints and achieves a satisficing measure on the soft constraints above predefined thresholds. The satisficing measure $SM()$ essentially captures the number of input valuations for which the soft constraints $\varphi_S(X,Y)$ are satisfied by $F(X)$, which could be computed as per Equation

$$SM(\psi(X,Y),G(X)) := \frac{ModelCount((\psi(X,Y) \wedge (Y \leftrightarrow G(X)))_{\downarrow X})}{ModelCount(\psi(X,Y)_{\downarrow X})} \tag{2.1}$$

formally, we define propose the notion of satisficing synthesis as follows:

Problem Statement

Satisficing Synthesis:

Given (i) hard constraints $\varphi_H(X, Y)$, (ii) soft constraints $\varphi_S(X, Y)$, and (iii) a satisficing threshold ε, where X is a set of inputs and Y is a set of outputs, the objective is to synthesize a system $F(X)$ such that following holds:

- $\forall X(\exists Y \varphi_H(X, Y) \leftrightarrow \varphi_H(X, F(X)))$,
- $SM(\varphi_S(X, Y), F(X)) \geq \varepsilon$.

Related work. Our setting of hard and soft constraints can be applied to various synthesis settings. Here, we will provide a brief overview of some of these domains.

In the domain of approximate synthesis, the objective is to synthesize a circuit C' from a given circuit C such that the size of C' is smaller than C, while maintaining an accuracy threshold. This problem involves optimizing circuit size while preserving functionality accuracy [HLR12, NLBR14, SAC+20, XMK15].

Best-effort synthesis techniques tackle scenarios where synthesizing a program that satisfies all constraints is too challenging. Instead, they focus on synthesizing as many specified constraints as possible within the available resources. These techniques employ bottom-up enumeration to generate candidate solutions that satisfy m constraints, then attempt to satisfy $m + 1$ constraints. Finally, a ranking-based method is used to select the candidate that satisfies the maximum number of constraints while reducing its size [PP20].

Programming-by-examples approaches have been explored in the presence of size constraints for synthesized functions [BGHZ15, PSY18, ZS13]. Techniques that incorporate user intent, processed using natural language processing techniques as soft constraints, along with input-output examples as hard constraints, developed to synthesize systems [CMF19]. Recently, Kalita et al. introduced the problem of automatically creating abstract transformers as an example-based synthesis, where positive examples are considered hard constraints and negative examples are treated as soft constraints [KMD+22].

Part II

Ingredients for Functional Synthesis

In this part of this, we are going to propose different components or ingredients for functional synthesis.

- Preprocessing of given specification (Chapter 3).
- Generating data from the specification (Chapter 4)
- Data-driven learning of candidates (Chapter 5).
- Verifying and repairing candidates using formal methods (Chapter 6).

Chapter 3

Preprocessing

In functional synthesis, several preprocessing techniques are employed to identify functions that are computationally simple. These functions typically fall into two categories: constant functions, which have fixed output values of either 0 or 1, and unique functions, which can be uniquely determined based on their inputs. For a unique function, once the input value X is established, the corresponding output value y_i is uniquely defined.

In this chapter, we will explore various syntactic and semantic techniques used to ascertain constant and unique functions.

3.1 Finding Unates

Pivotal Insights

We employ SAT solvers to identify a specific subset of variables that exhibit constant (unate) functions. By doing so, we extract the constant functions and effectively reduce the overall number of functions that require learning.

First, let's focus on identifying constant functions, which are referred to as unates. Put simply, a variable y_i is considered a unate if it consistently evaluates to 1 (or 0) across all possible input valuations that result in the specification being true. We can categorize y_i as a positive unate if the function f_i associated with it is always 1, and as a negative unate if f_i is always 0.

There are both syntactic and semantic approaches to identify such constant functions. For example, if an output variable y_i consists solely of positive literals (e.g., y_i itself) or negative literals (e.g., $\neg y_i$) in the specification φ, then the corresponding function f_i is a constant 1 or 0, respectively. However, it's important to note that a variable being a positive unate or a negative unate doesn't necessarily mean it will be represented solely by positive or negative literals. Therefore, in general cases, we require a semantic verification to efficiently identify these unates.

A variable y_i is a positive unate if and only if $\varphi(X,Y)_{\downarrow y_i=0} \rightarrow \varphi(X,Y)|_{\downarrow y_i=1}$ — that is, whenever y_i takes value 0 to satisfy the specification, it could also take value 1. Similarly, if y_i is considered to be negative unate, $\varphi(X,Y)_{\downarrow y_i=1} \rightarrow \varphi(X,Y)|_{\downarrow y_i=0}$ [ACG+18].

If Formula 3.1 turns out to be UNSAT, then y_i is positive unate and corresponding function f_i is constant function 1.

$$\varphi(X,Y)|_{y_i=0} \wedge \neg\varphi(X,Y)|_{y_i=1} \tag{3.1}$$

Similarly, if Formula 3.2 turns out to be UNSAT, then y_i is negative unate and corresponding function f_i is constant function 0.

$$\varphi(X,Y)|_{y_i=1} \wedge \neg\varphi(X,Y)|_{y_i=0} \tag{3.2}$$

We could use SAT solver to check the satisfiability of Formula 3.1 and 3.2, if Formula 3.1 (resp. Formula 3.2) turns out to be UNSAT, then we can consider the corresponding y_i to be positive (resp. negative) unate — one SAT solver call corresponding to an output variable is sufficient to detect if it is unate or not.

In [ACG+18], authors have proposed an algorithm to detect unates efficiently. Preprocess performs SAT queries on the formulas constructed on aforementioned formulas 3.1 and 3.2.

Algorithm to Extract Unate Functions. As described in Algorithm 1, Preprocess sequentially detects unate for every y_j of Y variable. Preprocess first checks for the satisfiability of positive unate formula (line 3) for y_j. If y_j is a positive unate, then f_j is 1 (line 7), and y_j is added to set U. If y_j is not a positive unate, then Preprocess checks for the satisfiability of negative unate formula (lines 9). If y_j is a negative unate, then f_j is 0 (line 13), and y_j is added to set U. List U represents the list of unates among Y variables.

Algorithm 1 Preprocess($\varphi(X, Y)$)

```
1:  $U \leftarrow \emptyset$
2:  for each $y_j \in Y$ do
3:      $ret_{pos}, \rho_{pos} \leftarrow$ CheckSat($\varphi(X,Y)|_{y_j=0} \wedge \neg\varphi(X,Y)|_{y_j=1}$)
4:      if $ret_{pos}$ = UNSAT then
5:          $U \leftarrow U \cup y_j$
6:          $\varphi(X,Y) \leftarrow \varphi(X,Y)|_{y_j=1}$
7:          $f_j \leftarrow 1$                                  ▷ $y_j$ is positive unate
8:      else
9:          $ret_{neg}, \rho_{neg} \leftarrow$ CheckSat($\varphi(X,Y)|_{y_j=1} \wedge \neg\varphi(X,Y)|_{y_j=0}$)
10:         if $ret_{neg}$ = UNSAT then
11:             $U \leftarrow U \cup y_j$
12:             $\varphi(X,Y) \leftarrow \varphi(X,Y)|_{y_j=0}$
13:             $f_j \leftarrow 0$                              ▷ $y_j$ is negative unate
14: return $\boldsymbol{f}, U$
```

3.2 Extracting Unique Functions

Pivotal Insights

To minimize the number of functions that need to be learned, we utilize an interpolation-based technique to identify a specific subset of variables with unique functions.

To identify uniquely defined functions, we employed interpolation-based techniques. In the literature, a variable is considered uniquely defined when its value can be uniquely determined based on the values of other variables. This relationship is often referred to as the "definition" of a variable in terms of others. To clarify further, let us begin by providing a formal definition for this:

Definition 3.1 ([LM08]). *Let $\varphi(W)$ be a formula, $w \in W$, $S \subseteq W \setminus w$. $\varphi(W)$* defines *w in terms of S if and only if there exists a formula $\psi(S)$ such that $\varphi(W) \models w \leftrightarrow \psi(S)$. In such a case, $\psi(S)$ is called a* definition *of w on S in $\varphi(W)$.*

Now that, we have established that $\psi(S)$ is a function or "definition" corresponding to w in $\varphi(W)$. Next, we need to identify if such definition exists corresponding to output variable. To this end, given $\varphi(W)$ defined on $W = \{w_1, w_2, \ldots w_n\}$. We create another set of *fresh* variables $Z = \{z_1, z_2, \ldots z_n\}$. Let $\varphi(W \mapsto Z)$ represent the formula where every $w_i \in W$ in φ is replaced by $z_i \in Z$.

Lemma 3.2 (Padoa's Theorem [Bet56]).

$$Let,\ I(W, Z, S, i) = \varphi(W) \wedge \varphi(W \mapsto Z) \wedge \left(\bigwedge_{w_j \in \mathcal{S}; j \neq i} (w_j \leftrightarrow z_j) \right)$$
$$\wedge w_i \wedge \neg z_i$$

φ defines $w_i \in W$ in terms of S if and only if $I(W, Z, S, i)$ is UNSAT.

In the preprocessing stage, our objective is to identify a subset $Z \subseteq Y$ and its corresponding function vector $\boldsymbol{\varphi_Z}$ in such a way that $\boldsymbol{\varphi_Z}$ can be extended to form a valid function vector $\boldsymbol{\varphi}$. It is worth noting that the set of unate variables naturally forms a determined set Z.

To expand the set Z further, we utilize the concept of definability. Through an iterative process, we identify variables $y_i \in Y$ that can be defined in terms of the remaining variables, ensuring that their definitions f_i adhere to the dependency constraints imposed by the definitions of variables in Z. To extract these corresponding definitions, we employ Padoa's theorem (Lemma 3.2), which allows us to check whether y_i can be defined in terms of the other variables.

While the Padoa's theorem offers a decision procedure to ascertain whether a variable y_i can be defined in terms of the remaining variables, an alternative approach was proposed in the work by the authors of [Sli20]. In their study, they introduced a method that utilizes interpolation-based techniques to extract the corresponding definition of y_i. Here are the following key observations [Sli20]:

- If $I(W, Z, S, i)$ is unsatisfiable, then there exists an interpolant ψ such that
  - The support of φ is a subset of W
  - $\varphi \wedge y_i \models \psi$
  - $\psi \models \neg\varphi[W \mapsto Z; y_i \mapsto z_i] \vee \neg z_i$, or equivalently, $\varphi[W \mapsto Z; y_i \mapsto z_i] \wedge \neg z_i \models \psi$

In the work by Slivovsky [Sli20], a technique was proposed to leverage off-the-shelf interpolant extraction solvers for the extraction ψ as unique functions. The utilization of this unique function extraction technique offers significant advantages by reducing the number of variables that need to undergo learning and repair. Unique functions, by their nature, do not require repair. This reduction in the reliance on learning is beneficial. Moreover, the interpolation-based extraction method enables the computation of complex functions with large sizes. These functions would typically necessitate an impractical number of samples to learn, surpassing the feasibility of learning-based techniques.

We would like to emphasize the importance of allowing variable y_i to depend, within the constraints of dependencies, on other variables in set Y. Let's consider the example where $X = x_1$ and $Y = y_1, y_2$. Suppose we have a specification $\varphi(X, Y) := (y_1 \vee y_2) \wedge (\neg y_1 \vee \neg y_2)$. Neither y_1 nor y_2 is defined solely by x_1. However, y_2 can be defined in terms of y_1 (and thus, also x_1, y_1) with its corresponding function $f_2(x_1, y_1) := \neg y_1$.

It is important to note that if we did not allow y_1 to depend on y_2, y_1 would not be uniquely defined. By permitting such dependencies, we ensure the uniqueness of the definitions and the ability to capture complex relationships between variables.

Algorithm to Extract Unique Functions. Algorithm 2 presents the subroutine UniDef. UniDef assume access to subroutine FindUniqueDef, which takes a formula $\varphi(X, Y)$, a variable y_i, and a defining set $X, y_1, \ldots, y_{i-1}$ as input, and determines

Algorithm 2 UniDef($\varphi(X, Y)$,$\boldsymbol{f}$,dependson)

```
1: univar ← $\emptyset$
2: for $y_i \in Y \setminus$ unates do
3:     definingvar ← $X \cup \{y_1, \ldots, y_{i-1}\}$
4:     ret, def ← FindUniqueDef($\varphi(X, Y)$,$y_i$,definingvar)
5:     if ret = true then
6:         univar ← univar $\cup\ y_i$
7:         $f_i$ ← def
8:         for $y_j \in f_i$ do
9:             dependson[$y_i$] ← dependson[$y_i$] $\cup\ y_j$
10: return unates $\cup$ univar, $\boldsymbol{f}$, dependson
```

whether the given variable y_i is defined with respect to the defining set or not. If the variable y_i is defined, FindUniqueDef returns true, along with the extracted definition f_i. Otherwise, it returns false (and an empty definition).

UniDef calls subroutine FindUniqueDef with defining set $\{X, y_1, \ldots, y_{i-1}\}$ for each existentially quantified variable y_i which is not unate at line 4. If FindUniqueDef returns true, UniDef adds y_i to the set *univar* at line 6. UniDef adds variables occurring in f_i to the list dependson[y_i] at line 9.

Chapter 4

Data Generation

Pivotal Insights

We utilize a constrained sampler to generate satisfying assignments, adhering to a specified weight function, from the solution space of the given specification. These generated samples are then treated as data and fed into the learning algorithm.

In order to utilize machine learning-based approaches for synthesis, it is necessary to generate data from a given relation specification. Although we can consider using all satisfying assignments of the specification as our data, it is important to note that the number of satisfying assignments can exponentially increase with the number of inputs. Therefore, we need to generate some satisfying assignments from the solution space of the specification, and constrained samplers can assist us in the task of generating samples. As discussed in Chapter 2, a constrained sampler takes a formula φ and a number of required satisfying assignments N, and returns satisfying assignments $\sigma_1, \ldots, \sigma_N$ in accordance to the distribution induced by the sampler. There can be different weighted distributions from which we can generate samples. In recent years, various types of samplers have been designed, including uniform samplers [CMV13, CMV14, CFM+15, DLBS18, EGSS12, SGM20, SGRM18], and weighted samplers [GSRM19], and choosing the appropriate sampling strategy to generate training data samples is a crucial decision.

4.1 What kind of a Sampling Strategy?

One obvious option is to employ uniform sampling over the inputs X and output Y. However, the relational nature of the specification φ that connects X and Y presents intriguing challenges and possibilities. To illustrate this, let's examine a specific example where the relational specification is given as $\varphi : (x_1 \vee x_2 \vee y_1)$. Table 4.1 displays all the possible satisfying assignments for φ, and one potential satisfying assignment that could be generated using a uniform sampler.

Table 4.1: Data Generation: An example of uniform sampling

x_1	x_2	y_1
0	0	1
0	1	0/1
1	0	0/1
1	1	0/1

$$\xrightarrow{\text{Uniform Sampler}}$$

x_1	x_2	y_1
0	0	1
0	1	0
1	0	1
1	1	0

Our aim is to design sampling subroutines that facilitate the discovery of Skolem functions with compact descriptions, considering the relationship between description and sample complexity. When we feed the data generated by a uniform sampler to a machine learning classifier for learning approximately correct candidate functions, it can be challenging for the classifier to learn the simpler functions. This challenge is evident even in the provided example with only two inputs and outputs, whereas in general, we deal with variables of much higher orders, possibly in the thousands. Therefore, it is crucial for us to generate high-quality data that can effectively train the classifier.

For instance, referring to the example presented in Table 4.1, where $X = x_1, x_2$ and $Y = y_1$, let us define $\varphi := (x_1 \vee x_2 \vee y_1)$. Note that φ has 7 solutions over $X \cup Y$, with $y_1 = 0$ appearing in 3 solutions and $y_1 = 1$ appearing in 4 solutions. Moreover, there exist multiple possible Skolem functions, such as $y_1 = \neg(x_1 \wedge x_2)$. Now, if we were to uniformly sample solutions of φ over the set of variables x_1, x_2, y_1, we would observe an almost equal number of samples with $y_1 = 0$ and $y_1 = 1$.

However, upon closer examination of φ, we realize that it is possible to construct a function for y_1 as constant 1 and for y_2 as constant 0. This condition arises from the fact that when both x_1 and x_2 are assigned the value 0, y_1 must be assigned the value 1 to satisfy φ. However, for all other input valuations, y_1 can take on either 0 or 1 to make φ true. Similarly, we want y_2 to take on the value 0 for all possible valuations of x_1 and x_2.

In the aforementioned example, our objective is to generate valuations where y_1 (resp. y_2) is consistently assigned the value 1 (resp. 0) for all possible combinations of inputs x_1 and x_2. Unfortunately, uniform sampling alone would not be capable of generating the required samples. To generate data that can facilitate the learning of simpler functions by the classifier, we need to rely on weighted sampling methods.

4.2 Adaptative Weighted Sampling Strategy

In order to incorporate the discussed intuition, we propose a novel approach of collecting samples using weighted sampling. Our approach involves utilizing a function called Bias, which takes a mapping from a sequence of variables to the desired weights of their positive literals. This function assigns appropriate weights to each of the positive literals based on the given mapping. To simplify notation, we represent the assignment of weights as `Bias(a,b)`, where positive literals corresponding to universal variables are assigned a weight of a, and positive literals corresponding to existential variables are assigned a weight of b. For instance, Bias(0.5, 0.9) assigns a weight of 0.5 to the positive literals of universally quantified variables and 0.9 to the positive literals of existentially quantified variables.

We have introduced an innovative technique called **adaptive weighted sampling** for generating data. This technique involves a multi-step process. Initially, we generate a set of samples using Bias(0.5, 0.9) and another set of samples using Bias(0.5, 0.1). These biased samplings are carried out to emphasize the generation of samples with outputs skewed towards 1 and 0, respectively. In both cases, we maintain a bias of 0.5 for the inputs, aiming to generate input valuations uniformly from the solution space.

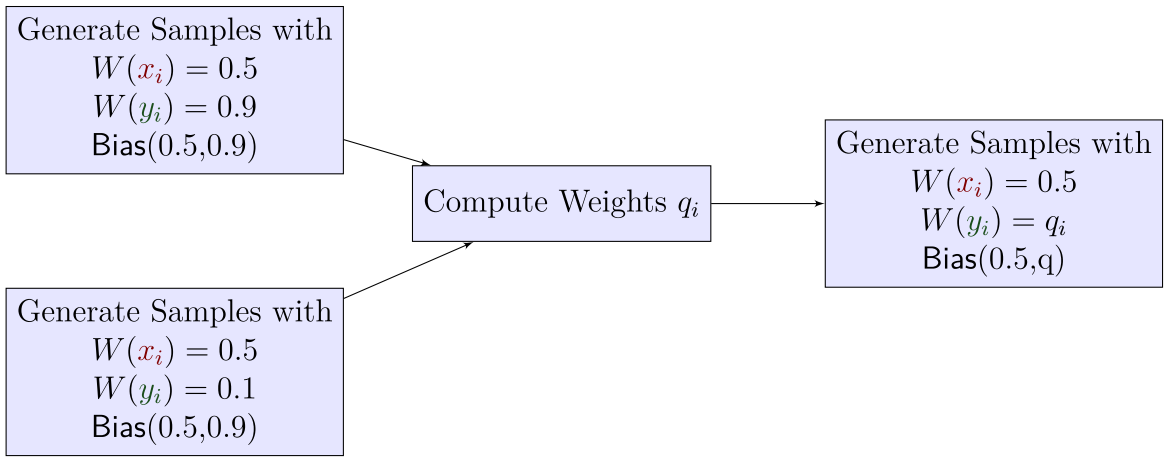

Generate Samples with $W(x_i) = 0.5$ $W(y_i) = 0.9$ Bias(0.5,0.9)
Generate Samples with $W(x_i) = 0.5$ $W(y_i) = 0.1$ Bias(0.5,0.9)
Compute Weights q_i
Generate Samples with $W(x_i) = 0.5$ $W(y_i) = q_i$ Bias(0.5,q)

Figure 4.1: Data Generation: An adaptive weighted sampling strategy.

Subsequently, we utilize the generated samples to predict the bias q for the outputs. This bias value q determines the distribution of the output values in the generated data that will be used to train the learning algorithm. By analyzing this initial set of samples, we determine the optimal value of q. Once the optimal q is determined, we generate the remaining samples with Bias(0.5, q), where q is fixed based on the analysis conducted on the initial set of samples.

Algorithm 3 GetSamples($\varphi(X, Y), N$)

```
1: $\Sigma_1 \leftarrow$ wCMSGen$(\varphi(X,Y), 500, 0.5, 0.9)$
2: $\Sigma_2 \leftarrow$ wCMSGen$(\varphi(X,Y), 500, 0.5, 0.1)$
3: for $y_j \in Y$ do
4:     $m_j \leftarrow \text{Count}(\Sigma_1 \cap (y_j = 1))/500$
5:     $n_j \leftarrow \text{Count}(\Sigma_1 \cap (y_j = 0))/500$
6:     if $(0.35 < m_j < 0.65) \wedge (0.35 < n_j < 0.65)$ then
7:         $q_j \leftarrow m_j$
8:     else
9:         $q_j \leftarrow 0.9$
10: $\Sigma \leftarrow$ wCMSGen$(\varphi(X,Y), N, 0.5, q)$
11: return $\Sigma$
```

Algorithm to Generate Data GetSamples presented in Algorithm 3 is used to generate the data which is fed to machine learning algorithm to learn approximately correct candidate functions. GetSamples takes $\varphi(X, Y)$ and number of required samples as input and returns, Σ, a subset of satisfying assignments of $\varphi(X, Y)$. GetSamples first generates 500 samples each with Bias$(0.5, 0.9)$(line 1), and with

$\mathsf{Bias}(0.5, 0.1)$(line 2). The constrained sampler CMSGen [GSCM21] is used to generate the samples. Then, GetSamples in line 4, calculates m_j for all y_j, m_j is a ratio of number of samples with y_j being 1 to the total number of samples, i.e. 500. Similarity, in line 5, it calculates n_j for all y_j, n_j is a ratio of number of samples with y_j being 0 to the total number of samples. Finally, GetSamples generates required number of samples with $\mathsf{Bias}(0.5, q)$; for a y_j, q is m_j if both m_j and n_j are in range 0.35 to 0.65, else q is 0.9. Finally, again GetSamples generates the required number of samples with $\mathsf{Bias}(0.5, q)$.

Chapter 5

Candidate Learning

Pivotal Insights

We propose the design of a dependency-aware classifier that constructs a vector of decision trees, each corresponding to a specific variable y_i. Each decision tree represents an approximately correct function associated with the corresponding variable.

We approach the problem of functional synthesis from a machine learning perspective, where the learned machine learning model for the classification of a variable y_i serves as a candidate function for y_i. To gather training data regarding the function's behavior, we leverage advancements in constrained sampling techniques to sample solutions of $\varphi(X, Y)$. It is important to note that $\varphi(X, Y)$ defines a relation, not necessarily a function, between X and Y. However, machine learning techniques typically assume the presence of a function between features and labels, requiring the use of sophisticated sampling strategies as discussed in Chapter 4.

When considering features and labels, our objective is to learn the relationship between Y and X. Thus, we treat X as a set of features, and the assignments to Y as a set of class labels.

Off-the-shelf classification techniques typically demand a training dataset that is several times larger than the number of potential class labels. However, this requirement becomes highly impractical for problems involving more than a thousand variables. To address the challenge of limited training data, we take into account two well-known observations from the functional synthesis literature:

1. a function f_i for variable y_i typically doesn't depend on all the variables in X,
2. a function vector $\boldsymbol{f}$ where f_i depends on variable y_j is a valid vector if the function f_j is not dependent on y_i (i.e., acyclic dependency), i.e., there exists a partial order $\prec_d$ over $\{y_1, \dots y_m\}$.

We relies on the aforementioned observations and propose a dependency-aware classifier based learning for functions.

5.1 Learning via Binary Classification

Based on the aforementioned observations, we propose an algorithmic procedure for learning candidate functions as decision trees in an iterative manner, focusing on binary classification for each variable y_i. The approach addresses the challenges posed by limited training data. In the iterative process, we update the set of potential features for a given variable y_i based on the candidate functions generated thus far.

To ensure proper feature selection, we consider the dependencies between variables. Specifically, if variable y_j uses y_i as a feature in its candidate function and y_i appears as a decision node in the learned classifier's decision tree, then when learning the candidate function for y_i, we exclude y_j as a feature. In other words, we say that y_j is dependent on y_i, denoted as $y_j \prec_d y_i$. Conversely, if y_i does not appear in the candidate function f_j for y_j, y_j is allowed to be a feature for the candidate function f_i of y_i.

Once we have obtained the candidate function for each output variable, we can derive a valid linear extension, denoted as *TotalOrder*, of the partial order $\prec_d$ based on the candidate function vector $\boldsymbol{f}$. Finally, the candidate function for y_i is computed by considering the disjunction of labels along the edges of all paths from the root to leaf nodes with a label of 1 in the decision tree.

Figure 5.1 illustrates an example of a learned classifier for the output variable y_1, given the valuations of x_1 and x_2. The classifier is encoded as a function, specifically

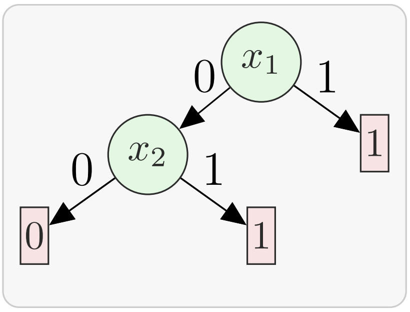


Figure 5.1: Binary Classification: Learned decision tree with label y_1 and features x_1, x_2.

the disjunction of all paths that lead to leaf nodes labeled as 1. In this case, the candidate function f_1 is represented as $p_1 \vee p_2$, where $p_1 = \neg x_1 \wedge x_2$ and $p_2 = x_1$.

We can make two key observations about the learned candidate functions:

- When learning a candidate function, we essentially learn one level of a decision list. In this level, if p_1 holds true, the output is 1; if p_2 holds true, the output is 1; otherwise, the output is 0. It is important to note that the decision nodes within a level of the decision list can be interchanged (i.e., p_1 and p_2 can be swapped), but decision nodes across different levels cannot be interchanged.
- Through the use of machine learning, we learn approximate correct candidate functions instead of exact function abstractions. In approximation learning, candidate functions may have double-sided errors, meaning that a candidate function f_i can evaluate to 0 (or 1) for a particular input valuation, while the corresponding correct valuation for y_i is 1 (or 0). On the other hand, in abstraction learning, candidate functions have single-sided errors.

Algorithm to Learn Candidates via Binary Classification. Algorithm 4 presents CandidateSkF to learn approximately correct candidate functions. Algorithm 4 assumes access to following three subroutines:

CreateDecisionTree takes the feature and label sets as input (training data) and returns a decision tree t. We use the ID3 algorithm [Qui86] to construct a decision tree t where the internal node of t represents a feature on which a decision is made, the branches represent partitioning of the training data on the decision, and the leaf nodes represent the classification outcomes (i.e class labels). The ID3 algorithm iterates over the training data, and in each iteration, it selects a new attribute to extend the tree by a new decision node: the selected attribute is one that causes the maximum drop in the impurity of the resulting classes; we use Gini Index [Qui86] as the measure of impurity. The algorithm, then, extends the tree by the selected decision and continues extending building the tree. The algorithm terminates on a path if either it exhausts all attributes for decisions, or the impurity of the resulting classes drop below a (user-specified) impurity decrease parameter.

Label takes a leaf node of the decision tree as input and returns the class label corresponding to the node.

Path takes a tree t and two nodes of t (node a and node b) as input and outputs a conjunction of literals in the path from node a to node b in t.

Algorithm 4 CandidateSkF$(\Sigma, \varphi(X, Y), y_j, D)$

```
1: $featset \leftarrow X$
2: for each $y_k \in Y \setminus y_j$ do
3:     if $y_j \notin d_k$ then
4:         $featset \leftarrow featset \cup y_k$                ▷ if $y_k$ is not dependent on $y_j$
5: $feat, lbl \leftarrow \Sigma_{\downarrow featset}, \Sigma_{\downarrow y_j}$
6: $t \leftarrow$ CreateDecisionTree$(feat, lbl)$
7: for each n ∈ LeafNodes(t) do
8:     if Label(n) = 1 then
9:         $\pi \leftarrow$ Path$(t, root, n)$
10:        $f_j \leftarrow f_j \vee \pi$
11: for each $y_k \in f_j$ do
12:     $d_j \leftarrow d_j \cup y_k \cup d_k$
13: return $f_j, D$
```

As we seek to learn functions, we employ binary classifiers with class labels `0` and `1`. CandidateSkF shows our algorithm for extracting a Boolean function from the decision trees: lines 2-4 find a feature set (*featset*) to predict y_j. The feature set includes all X variables and the subset of Y variables that are not dependent on y_j. CandidateSkF creates decision tree t using samples Σ over the feature set.

Lines 7-10 generate candidate Skolem function f_j by iterating over all the leaf nodes of t. In particular, if a leaf node is labeled with 1, the candidate function is updated by disjoining with the formula returned by subroutine Path.

CandidateSkF also updates d_j in D, d_j is set of all Y variables on which, y_j depends. If y_j depends on y_k, then by transitivity y_j also depends on d_k; in line 12, CandidateSkF updates d_j accordingly.

5.2 Learning via Multi Classification

We have identified that the process of learning candidate functions for each output variable y_i individually becomes computationally expensive when dealing with larger instances involving thousands of variables. To address this scalability challenge, the thesis proposes an alternative approach that involves learning a group of candidate functions simultaneously through multi classification.

Pivotal Insights

Instead of relying solely on binary classification, we introduce a clustering-based approach that takes advantage of multi-classification to learn candidate functions for a set of variables together at a time.

There are mainly two key observations that are essential for designing multi-classification approach:

Variable Retention: While variable elimination is commonly employed in formal methods to simplify problem instances, it poses a challenge when it comes to the learning phase and the identification of ”determined features” (as discussed in Chapter 3). Determined features D refer to variables for which we have already determined their unique function. Variable elimination has been widely recognized as an effective preprocessing strategy in formal methods, with numerous studies exploring its benefits and applications [AAC+19, ACG+18, BLS11, GRM20]. While eliminating determined features by substitution does not impact the existence of functions for other variables $y_i \in Y \setminus D$, it significantly affects the size and complexity of these functions, as they are no longer allowed to depend on the eliminated variables in determined features D.

Therefore, while variable elimination can simplify the problem representation, it comes at the cost of losing valuable information and potentially increasing the complexity of the remaining functions. It is important to carefully consider the trade-off between problem simplification and the preservation of determined features when applying variable elimination in the synthesis process.

For example, consider the following scenario: let $X = \{x_1, x_2\}$, $Y = \{y_1, y_2\}$ and $\varphi(X, Y) = (y_1 \vee y_2) \wedge (\neg y_1 \vee \neg y_2) \wedge (y_1 \leftrightarrow (x_1 \oplus x_2))$. Observe that the function for y_2 in terms of X in the transformed formula $\varphi(x_1, x_2, y_2)$ will have to be learned as $\neg(x_1 \oplus x_2)$. However, when allowing learning over y_1, then the desired function for y_2 can simply be learned as $\neg y_1$.

A key observation that has significantly influenced performance is the decision to retain variables with determined functions instead of substituting and eliminating them from the specification. When it is determined that a candidate function for a variable accurately represents the required function, we refrain from substituting it in the specification and instead keep it as a potential feature during the learning and repair process. In the example mentioned earlier, even though we have identified the function corresponding to y_1, we don't substitute it in the specification, allowing it to be used in learning the function for y_2.

This approach bears resemblance to the concept of ”latent features” in machine learning, where certain features are not directly observable but are derived from the observable features. By retaining variables with unique functions, we effectively transform observable features into latent features that can be recovered and utilized by the learning algorithm.

We conclude that, contrary to conventional wisdom, variables in the determined set D should not be eliminated and instead should be retained as features for the learning and repairing the approximately correct functions.

Partition the set of Y variables into disjoint subsets:

An important question that still requires an answer is how to determine the optimal variable partitioning. Our approach is guided by the intuition that variables with low cohesion within a partition impose fewer constraints, resulting in larger decision trees and an increased number of classes. Therefore, our aim is to learn and group variables that are related to each other.

To achieve this, we utilize a structure known as the "primal graph" introduced in the work by Shtrichman and Strichman [SS10]. In the primal graph, each node represents a variable that appears in the specification, and an edge exists between two nodes if and only if the corresponding variables share a clause. By leveraging the information provided by the primal graph, we can determine the distance between variables and use it to cluster the variables in the set Y into disjoint subsets.

By employing the distance metric in the primal graph, we can identify variables that share common clauses or have a higher degree of interconnectedness. These variables are more likely to be related and exhibit higher cohesion. Clustering them together allows us to capture their inherent dependencies and to learn candidate functions more effectively.

Algorithm to cluster output variables. ClusterY Algorithm 5 presents the subroutine ClusterY, it takes formula the $\varphi(X, Y)$, k : an edge distance parameter, s : maximum allowed size of a cluster of Y variables, and U : list of unate and uniquely defined Y variables, and it returns a list of all subsets of Y that would be learned together. ClusterY assumes access following subroutine:

1. kHopNeighbor, which takes a graph, variable y, and an integer k as input, and returns all variables within distance k of y in the graph.

2. RemoveNode, which takes a graph and a vertex y as input, and removes the vertex y along with its incident edges from the graph.

Algorithm 5 ClusterY($\varphi(X,Y)$,k,s,U)

```
1:  graph = $\emptyset$
2:  for each $clause$ of $\varphi(X,Y)$ do
3:      if $\langle y_i, y_j \rangle$ pair in $clause$ then
4:          if $y_i \notin U$ and $y_j \notin U$ then
5:              AddEdge(graph,$y_i$,$y_j$)
6:  subsetY = $\emptyset$
7:  for $y_i \in Y$ do
8:      while $k \geq 0$ do
9:          chunk $\leftarrow$ kHopNeighbor(graph,$y_i$,k)
10:         if size(chunk) $\leq s$ then
11:             break
12:         $k \leftarrow k - 1$
13:     subsetY $\leftarrow$ subsetY.add(chunk)
14:     for $y_j \in$ chunk do
15:         RemoveNode(graph,$y_j$)
16: return subsetY
```

We first construct a primal graph G with existentially quantified Y variables as nodes, and if y_i and y_j share a clause, then there is an edge between them. For a variable y_i in given sequence $\{y_1, \ldots, y_{|Y|}\}$, we cluster all the variables that are at $\leq k$ distance from y_i in the graph, where k is a edge distance threshold. Once, such a subset of variables Y is found, remove them as nodes along with their edges in order to find disjoint subsets.

ClusterY first creates a graph *graph* with $Y \setminus U$ as vertex set and edges between variables y_i and y_j that share a clause in $\varphi(X,Y)$. ClusterY then calls subroutine kHopNeighbor for each variable y_i. The set of variables returned by kHopNeighbor is stored as chunk. If the size of chunk is greater than s, ClusterY reduces the value of k by one at line 12, and calls kHopNeighbor again with the updated value of k. Otherwise, ClusterY adds chunk to subsetY at line 13. Finally at line 15, ClusterY removes the nodes corresponding to each variable of chunk from *graph*.

Now, let us discuss the multi-classification approach for learning approximately correct candidates. We could use the following strategy:

- Identify determined features and partition the set of remaining Y variables into disjoint subsets,

- Use a multi classifier to learn candidate function for each partition.

In the multi-classification approach, the candidate function for a variable y_i within a selected subset is obtained by taking the disjunction of all paths from the root to a leaf node with a label of y_i equal to 1. Additionally, we update the partial dependency relation by setting $y_i \prec_d y_j$ for all variables y_j that occur in the candidate function f_i.

The feature set for learning a specific subset includes a variable y_j only if $y_j \not\prec_d y_i$ for every variable y_i in the subset. This ensures that variables depending on other variables within the same subset are excluded from the feature set during the learning process, promoting independence among the variables in the subset. Let's consider a scenario with two different subsets: y_1, y_2 and y_3, y_4. Suppose we have the partial dependency $y_1 \prec_d y_3$. In this case, when learning the candidate functions for subset y_3, y_4, the feature set would include the variables X, y_2.

To illustrate the multi-classification based approach, let's consider an example. We have a scenario with two input variables $X = x_1, x_2$ and two output variables $Y = y_1, y_2$ in the formula $\exists Y \varphi(X, Y)$. Figure 5.2 depicts the learned decision tree, with labels y_1, y_2 and features x_1, x_2.

Assuming that neither y_1 nor y_2 are determined features, we would expect to have $2^2 = 4$ classes to learn candidates for the two output variables together. However, as shown in Figure 5.2, the decision tree classifies the labels into only 3 classes, denoted as $\langle 01, 10, 11 \rangle$.

In this example, the candidate function f_1 corresponds to y_1 and is obtained by taking the disjunction of paths from the root to leaf nodes with a label of y_1 equal to 1. In this case, the paths corresponding to classes 10 and 11 are selected. Hence, the candidate function for y_1 is $\psi_1 = (\neg x_1 \wedge x_2) \vee (x_1)$.

Similarly, the candidate function f_2 for y_2 is derived by considering the paths from the root to leaf nodes with a label of y_2 equal to 1. In this example, the paths corresponding to class 01 and 11 are chosen, resulting in the candidate function $f_2 = (\neg x_1 \wedge \neg x_2) \vee (x_1)$.

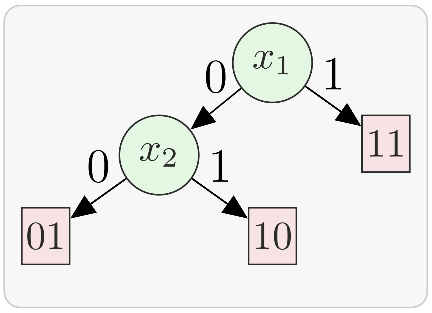


Figure 5.2: Multi Classification: Learned decision tree with label y_1, y_2 and features x_1, x_2.

Algorithm 6 CandidateSkF(Σ,$\varphi(X,Y)$,$\boldsymbol{f}$,chunk,dependson)

```
1: featset ← X
2: D ← ∅
3: for each $y_j \in$ Y do
4:     for each $y_i \in$ chunk do
5:         if $y_i \in$ dependson[$y_j$] then
6:             D ← D ∪ $y_j$
7: for each $y_j \in$ Y\chunk do
8:     if $y_j \notin$ D then
9:         featset ← featset ∪ $y_j$
10: feat, lbl ← $\Sigma_{\downarrow featset}$, $\Sigma_{\downarrow \mathsf{chunk}}$
11: dt ← CreateDecisionTree(feat,lbl)
12: for each $y_i \in$ chunk do
13:     for each l ∈ LeafNodes(dt) do
14:         if Label($y_i$,l) =1 then
15:             $\pi$ ← Path(dt,root,l)
16:             $f_i \leftarrow f_i \vee \pi$
17:     for each $y_j \in f_i$ do
18:         dependson[$y_i$] ← dependson[$y_i$] ∪$y_j$
19: return $\boldsymbol{f}$, dependson
```

Algorithm to Learn Candidates via Multi Classification. Algorithm 6 presents the multi-classification based learning approach called CandidateSkF. It takes a set Σ of samples, $F(X, Y)$, Ψ: a candidate function vector, chunk: the set of variables to learn candidates, and dependson: a partial dependency vector as input, and finds the candidates corresponding to each of the variables y_i in chunk. CandidateSkF assumes access to subroutines CreateDecisionTree and Path, which are same as in Algorithm 4. The following are the additional subroutines used by CandidateSkF:

1. LeafNodes, which takes a decision tree dt as an input and returns a list of leaf nodes of dt.

2. Label(y_i, l), which takes a variable y_i and a leaf node l as input, and returns 1 if the class label corresponding to the node l has value 1 at the i^{th} index.

CandidateSkF starts off by initializing the set *featset* of features with the set X of input variables. It then attempts to find a list D of variables y_j such that $y_j \prec_d y_i$ where y_i belongs to chunk. Next, CandidateSkF adds $Y \setminus D$ to *featset*, and creates a decision tree dt using samples from Σ over *featset* to learn the chunk variables. For a leaf node l of dt, if Label(y_i, l) returns 1, then f_i is updated with the disjunction of the formula returned by subroutine Path. Finally, CandidateSkF iterates over all y_j occurring in f_i to add them to the list dependson$[y_i]$.

Chapter 6

Verification and Repairing of Candidates

6.1 Verification

In our machine learning-based approach, we aim to learn an approximately correct function vector. While machine learning techniques are highly efficient, the synthesized functions are only approximations and may not guarantee the correctness of the underlying specification. Hence, it becomes essential to perform a verification check to ensure the accuracy of the candidate function vector.

The important question then arises: how do we verify the correctness of the candidate function vector?

Pivotal Insights We perform a simple SAT call to verify the correctness of synthesized function vector.

The goal is to determine if there exists an input valuation for X such that, when using the corresponding function values for each $y_i \in Y$, the formula $\varphi(X, Y)$ evaluates to false, even though there exists a different valuation of Y that satisfies $\varphi(X, Y)$. In other words, we want to identify if there are any cases where the function vector produces incorrect values for Y, leading to a false evaluation of $\varphi(X, Y)$.

The verification query constructs an *error formula* $E(X, Y, Y')$ as follows [JSC+15]:

$$E(X, Y, Y') := \varphi(X, Y) \land \neg\varphi(X, Y') \land (Y' \leftrightarrow F(X)) \tag{6.1}$$

In error formula 6.1, we introduce a new set of variables Y' such that $|Y'| = |Y|$. In the second and third term of error formula 6.1, we have replaced Y by Y' in the specification and set the Y' are set as candidate functions. The term $(Y' \leftrightarrow F(X))$ reperesnts that $(y'_1 \leftrightarrow f_1), \ldots, (y'_{|Y|} \leftrightarrow f_{|Y|})$. Note that input X is same in first and second term in the formula.

If the error formula $E(X, Y, Y')$ is unsatisfiable, it indicates that the candidate function vector is indeed the required function vector. On the other hand, if $E(X, Y, Y')$ is satisfiable, the solution of $E(X, Y, Y')$ can be used to identify and refine the erroneous functions within the candidate function vector.

6.2 Repair of Candidates

The repair strategy is guided by the understanding that the candidate function vector obtained during the candidate learning phase is an approximation and may contain errors. Candidates undergo repair only when verification check is failed, that is, when error formula (Formula 6.1) is satisfiable. Let us assume that σ is a satisfying assignment of $E(X, Y, Y')$ and referred to as counterexample for the current candidate function vector $\boldsymbol{f}$. When repairing the candidate function vector, there are two crucial questions to address:

- Among the candidate functions corresponding to each output variable y_i, which one(s) need to be repaired in order to resolve the counterexample?
- Once we have identified the set of candidate functions that require repair, how can we effectively repair them while considering their interdependencies?

In order to identify and implement a series of minor repairs to correct the erring functions, we rely on two key techniques: fault localization and repair synthesis.

6.2.1 Fault Localization

Pivotal Insights

We introduce the use of MaxSAT solvers to identify a minimal set Y variables whose corresponding candidate functions need to be repaired in order to resolve the counterexample.

Consider the following example to illustrate the need for an intelligent technique to identify candidates for repair. Let's take $\sigma := \langle x_1 \leftrightarrow 1, x_2 \leftrightarrow 1, y_1 \leftrightarrow 1, y_2 \leftrightarrow 1, y_1' \leftrightarrow 0, y_2' \leftrightarrow 0\rangle$ as an instance. Here, y_i' represents the output values produced by the candidate functions with the given input valuation, while y_i represents one of the valid output valuations for the inputs, where the specification φ is to True.

In this case, we observe that the candidate function vector is incorrect, as the expected output values y_1 and y_2 should be 1, but the candidate functions produce output values y_1' and y_2' of 0. Therefore, it is necessary to identify the candidate functions that need to be repaired to correct this inconsistency.

At the first glance, we might consider repairing the candidates for which y_i is not the same as y_i' in σ. In the previous example, this would lead us to repair both f_1 and f_2 in order to fix σ. However, this approach may result in unnecessary repairs and potentially steer us away from the correct candidate function vector.

To understand it better, let's consider the fact that the underlying specification is a relational specification that is for a given input valuation, there can be multiple valid output valuations for which the specification evaluates to True. In our example, for the input valuation $x_1 \leftrightarrow 1, x_2 \leftrightarrow 1$, one possible output valuation could be $y_1 \leftrightarrow 0, y_2 \leftrightarrow 1$. In this case, we would not want to repair f_1 since it correctly produces the desired output. But, we don't know in advance whether such counterexamples like $\hat{\sigma} := \langle x_1 \leftrightarrow 1, x_2 \leftrightarrow 1, y_1 \leftrightarrow 0, y_2 \leftrightarrow 1, y_1' \leftrightarrow 0, y_2' \leftrightarrow 0\rangle$ exist or not.

To identify the initial candidates for repair, we aim to find a counterexample σ and analyze its vicinity. Our goal is to identify a small number of functions (corresponding to Y variables) that require changes in their outputs for the formula to behave correctly on σ. We strive to preserve the current outputs of most functions (corresponding to Y variables) on σ while ensuring the satisfaction of the formula.

To tackle this problem, we formulate it as a partial MaxSAT query. In a given CNF formula, we classify certain clauses as *hard constraints*, which must be satisfied, and the remaining clauses as *soft constraints*. The objective of partial MaxSAT is to find an assignment that satisfies all hard constraints while maximizing the number of satisfied soft constraints. We construct the query as follows:

- Hard constraints: $\varphi(X, Y) \wedge (X \leftrightarrow \sigma[X])$
- Soft constraints: $(Y \leftrightarrow \sigma[Y'])$

All Y variables whose valuation constraint $(Y \leftrightarrow \sigma[Y'])$ does not hold in the MaxSAT solution are identified as erring functions that may need to be repaired. The Y variables corresponding to which the MaxSAT solver had to drop the soft constraint to come up with a solution that satisfies all hard constraints, the functions corresponding to such Y variables needed to be repaired.

However, using partial MaxSAT might lead to increase in number of repair iteration due to lack of incorporating interdependencies among candidates.

Let us illustrate the need for a dependency-aware approach by considering an example. Suppose we have $X = x_1, x_2$ and $Y = y_1, y_2$ with the specification $\varphi(X, Y) = (y_1 \vee y_2) \wedge (\neg y_1 \vee \neg y_2)$ in $\exists Y \varphi(X, Y)$. Let the candidate functions be $f_1 = 1$ and $f_2 = 1$, and the total order be y_1, y_2.

To identify a candidate for repair, we invoke MaxSAT with the hard constraints $\varphi(X, Y) \wedge (X \leftrightarrow \sigma[X])$ and the soft constraints $(y_1 \leftrightarrow 1) \wedge (y_2 \leftrightarrow 1)$, where σ is a satisfying assignment of the error formula. Assuming that either y_1 or y_2 can be flipped to fix the counterexample σ, let us assume that MaxSAT does not satisfy the soft constraint $(y_2 \leftrightarrow 1)$, selecting f_2 for repair.

However, we encounter a problem. Since y_2 is not allowed to depend on y_1, we cannot fix y_2 without repairing y_1. Thus, to fix the counterexample σ, we cannot repair f_2 in this repair iteration, requiring an additional iteration. This situation could have been avoided if f_1 had been selected for repair before f_2 because y_1 is allowed to depend on y_2.

The use of MaxSAT queries in determining repair candidates fails to consider these dependencies, resulting in a larger number of unnecessary repairs. To address this issue, we need a **dependency-aware** identification of repair candidates.

Lexicographic MaxSAT for Dependency-Aware Fault Localization To address the problem of identifying repair candidates with consideration for dependencies, we can use a *lexicographic* partial MaxSAT, denoted as LexMaxSAT. LexMaxSAT is a variant of partial MaxSAT that incorporates a preference order for satisfying soft constraints [IMM18]. By using LexMaxSAT, we can significantly reduce the number of iterations required to fix a counterexample.

Pivotal Insights

We propose the use of lexicographic MaxSAT to identify erring candidates while considering dependencies among candidate functions. Lexicographic MaxSAT is a variant of MaxSAT that prioritizes soft constraints in a lexicographic order.

In LexMaxSAT, we encode the problem as a query with two components: (i) hard constraints, which include $\varphi(X, Y) \wedge (X \leftrightarrow \sigma[X])$, and (ii) soft constraints, which include $(Y \leftrightarrow \sigma[Y'])$. The soft constraints are ordered based on the *TotalOrder* of the variables. By utilizing LexMaxSAT, we can achieve a more efficient identification of repair candidates, leading to a reduced number of iterations required to fix counterexamples.

6.2.2 Repair Synthesis

Pivotal Insights

We utilize unsatisfiability cores obtained from the infeasibility proofs, which capture the reasons why the current candidate functions fail to satisfy the specification, to construct a good repair.

Let's consider a counterexample: $\sigma := \langle x_1 \leftrightarrow 1, x_2 \leftrightarrow 1, y_1 \leftrightarrow 0, y_2 \leftrightarrow 1, y_1' \leftrightarrow 0, y_2' \leftrightarrow 0\rangle$. We assume that we have identified function f_2 as needing repair. One potential repair could be the following: if x_1 and x_2 are 1, and y_1 is 0, then y_2 needs to be 0. In other words, if $x_1 \wedge x_2 \wedge \neg y_1$, then y_2 should be 0. However, this repair would only address the specific counterexample σ and would not generalize to a set of potential counterexamples.

Let's denote the literals that appear in the "if" condition as β. In this example, β is $\langle x_1, x_2, \neg y_1 \rangle$. Generally, we aim to keep β as small as possible to generalize over counterexamples. We rely on unsatisfiable cores to identify the reasons why a particular function needs repair, and we use extracted reasons to construct β formulas for function repair.

Let y_k be the variable corresponding to the erring function, f_k, identified to repair with respect to the counterexample σ. To synthesize a repair for the function, we apply a proof-guided strategy and construct a formula $G_k(X, Y)$:

$$\begin{aligned} G_k(X,Y) &= (y_k \leftrightarrow \sigma[y_k']) \wedge \varphi(X,Y) \wedge (X \leftrightarrow \sigma[X]) \wedge (\hat{Y} \leftrightarrow \sigma[\hat{Y}]) \\ &\text{where } \hat{Y} \subset Y \text{ and } \hat{Y} = \{\mathit{TotalOrder}[\mathit{index}(y_k)+1], \cdots, \mathit{TotalOrder}[|Y|]\} \end{aligned} \tag{6.2}$$

Intuitively, formula G_k attempts to find the reason behind, *why with the current valuations of input variables, f_k should not return the current output in order to meet the specification?*. Now, there can be cases regarding satisfiability of G_k.

- If $G(X, Y)$ is unsatisfiable, it implies that there exists a reason within $\hat{Y}$ that can explain the discrepancy between the specification and the current function. In this case, we construct a *repair formula*, β, as a conjunction over literals in the unsatisfiable core from the proof of infeasibility. If the erring value of the function is *true*, we strengthen the function f_k by conjoining it with the repair formula ($f_k \leftarrow f_k \wedge \neg\beta$); otherwise, we weaken the function f_k by disjoining it with the negation of the repair formula ($f_k \leftarrow f_k \vee \beta$).

- If $G(X, Y)$ is satisfiable, it implies that the current function f_k of y_k is consistent with valuation of X and $\hat{Y}$, and the reason of the discrepancy lies outside

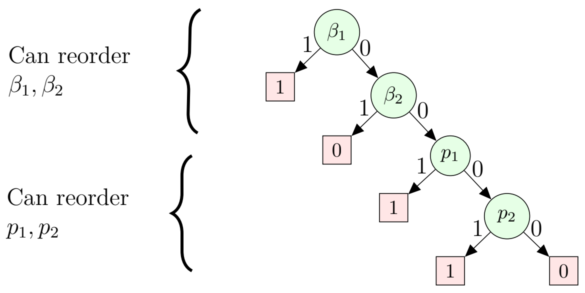


Figure 6.1: Repair of candidates: Moving from decision tree to two level decision list.

$\hat{Y}$ (or the specification is already met due to corrections to other functions in the previous loop iterations). In such a case, we attempt to find $y_t \notin \hat{Y}$, whose function f_t is not consistent with current f_k, and it adds f_t to the list of candidate functions that need to be repair.

The ordering of the variable is important; $G(X, Y)$ does not constrain Y variables that depend on y_k to avoids cyclic dependencies between variables.

Let's delve deeper into the repair procedure and understand it from the perspective of machine learning. Recall that the learned candidates are one-level decision lists, such as "if p_1 then 1, else if p_2 then 1, else 0," where p_1 and p_2 can be interchanged. Now, the repair iterations essentially add another level to the decision list, transforming it from a one-level to a two-level decision list as illustrated in Figure 6.1.

In each repair iteration, a β decision node is added to the classifier. These β repair nodes and the path nodes p can be interchanged. However, we cannot reorder them among themselves as we performed repairs on top of the candidate functions. Eventually, we obtain synthesized functions represented as a two-level decision list.

Falling back on Self Substitution Some functions are difficult to learn through data and thereby requiring a long sequence of incremental repairs for convergence. To handle such scenarios, we make the following observation: though synthesizing functions via self-substitution [FTV16]. Self-substitution is defined as follows:

Theorem 6.1 ([FTV16]). *Let $\phi = \exists y \varphi(X, y)$ be a 2-QBF formula. Then, $\exists y \varphi(X, y)$ is logically equivalent to $\varphi(X, \varphi(X, 1))$, and is also logically equivalent to $\varphi(X, \neg\varphi(X, 0))$.*

Note that synthesizing functions via self-substitution can lead to an exponential blowup in the worst case, it is inexpensive if the number of variables synthesized via this technique is small. We use this observation to quickly synthesize a function for an erring variable if we detect its candidate function is poor (detected by comparing the number of times it enters into repair iterations against an empirically determined threshold). Of course, this heuristic does not scale well if the number of such variables is large.

Algorithm to Repair Candidates. Algorithm 7 presents RepairSkF algorithm. RepairSkF is invoked with a counterexample σ.

Algorithm 7 RepairSkF($\varphi(X, Y), \boldsymbol{F}, \sigma, \mathit{TotalOrder}$)

```
 1: $H \leftarrow \varphi(X,Y) \wedge (X \leftrightarrow \sigma[X])$; $S \leftarrow (Y \leftrightarrow \sigma[Y'])$
 2: $Ind \leftarrow$ MaxSATList$(H, S)$
 3: for $y_k \in Ind$ do
 4:     $\hat{Y} \leftarrow \{\mathit{TotalOrder}[index(y_k)+1], \cdots, \mathit{TotalOrder}[|Y|]\}$
 5:     if CheckSubstitute$(y_k)$ then
 6:         $f_k \leftarrow$ DoSelfSubstitution$(\varphi(X,Y), y_k, Y \setminus \hat{Y})$
 7:     else
 8:         $G_k \leftarrow (y_k \leftrightarrow \sigma[y'_k]) \wedge \varphi(X,Y) \wedge (X \leftrightarrow \sigma[X]) \wedge (\hat{Y} \leftrightarrow \sigma[\hat{Y}])$
 9:         $ret, \rho \leftarrow$ CheckSat$(G_k)$
10:         if $ret = UNSAT$ then
11:             $C \leftarrow$ FindCore$(G_k)$
12:             $\beta \leftarrow \bigwedge_{l \in C} ite((\sigma[l] = 1), l, \neg l)$
13:             $f_k \leftarrow ite((\sigma[y'_k] = 1), f_k \wedge \neg\beta, f_k \vee \beta)$
14:         else
15:             for $y_t \in Y \setminus \hat{Y}$ do
16:                 if $\rho[y_t] \neq \sigma[y'_t]$ then
17:                     $Ind \leftarrow Ind.Append(y_t)$
18:             $\sigma[y_k] \leftarrow \sigma[y'_k]$
19: return $\boldsymbol{F}$
```

RepairSkF is invoked with a counterexample σ. RepairSkF first performs *fault localization* to find the initial set of erring candidate functions; to this end, it calls the MaxSATList subroutine (line 2) with $\varphi(X, Y) \wedge (X \leftrightarrow \sigma[X])$ as hard-constraints and $(Y \leftrightarrow \sigma[Y])$ as soft-constraints. MaxSATList employs a MaxSAT solver to find the solution that satisfies all the hard constraints and maximizes the number of satisfied soft constraints, and then returns a list (*Ind*) of Y variables such that for each of the variables appearing in (Ind) the corresponding soft-constraint was not satisfied by the optimal solution returned by MaxSAT solver.

Since candidate function corresponding to the variables in Ind needs to refine, RepairSkF now attempts to synthesize a repair for each of these candidate functions. Repair synthesis loop (lines 3–19) starts off by collecting the set of Y variables, $\hat{Y}$, on which y_k of Ind can depend on as per the ordering constraints (line 4). Next, it invokes the subroutine CheckSubstitute, which returns True if the candidate function corresponding to y_k has been refined more than a chosen threshold times (fixed to 10 in our implementation), and the corresponding decision tree constructed during execution CandidateSkF has exactly one node.

If CheckSubstitute returns true, RepairSkF calls DoSelfSubstitution to perform self-substitution. DoSelfSubstitution takes a formula $\varphi(X, Y)$, an existentially quantified variable y_k and a list of variables which depends on y_k and performs self substitution of y_k with constant 1 in the formula $\varphi(X, Y)$[JSC+15].

If CheckSubstitute returns false, RepairSkF attempts a proof-guided repair for y_k. RepairSkF calls CheckSat in line 9 on G_k, which corresponds to formula 6.2: if G_k is SAT, then CheckSat returns a satisfying assignment(ρ) of G_k in σ, else CheckSat returns unsatisfiable in the result, ret.

1. If ret is UNSAT, we proceed to refine f_k such that for $f_k(X \leftrightarrow \sigma[X], \hat{Y} \leftrightarrow \sigma[\hat{Y}]) = \sigma[y_k]$. Ideally, we would like to apply a refinement that generalizes to potentially other counter-examples, i.e. solutions of $E(X, Y, Y')$. To this end, RepairSkF calls FindCore with G_k; FindCore returns the list of variables (C) that occur in the clauses of UnsatCore of G_k. Accordingly, the algorithm constructs a *repair formula* β as a conjunction of literals in σ corresponding to variables in C (line 12). If $\sigma[y'_k]$ is 1, then f_k is f_k with conjunction of negation of β and if $\sigma[y'_k]$ is 0, then f_k is f_k with disjunction of β.

2. If ret is SAT and ρ is a satisfying assignment of G_k, then there exists a function vector such that the value of f_k agrees with $\sigma[y_k]$ for the valuation of X and $\hat{Y}$ set to $\sigma[X]$ and $\sigma[\hat{Y}]$. However, for any $y_t \in Y \setminus \hat{Y}$ if $\sigma[y'_t] \neq \rho[y'_t]$, then for such

a y_t, the function corresponding to y_t may need to refine . Therefore, RepairSkF adds y_t to list of candidates to refine, Ind. Note that since $\sigma \models E(X, Y, Y')$, there exists at least one iteration of the loop (lines 3– 18) where ret is UNSAT.

Part III

Recipe and Results for Functional Synthesis

This part of thesis focuses on the recipe of functional synthesis, both with and without explicit dependencies by using ingredients discussed in Part II of In Chapter 7, we present the algorithm along with comprehensive experimental results for Skolem synthesis, which involves functional synthesis without explicit dependencies. Furthermore, Chapter 8 introduces the algorithm and provides detailed experimental results for Henkin synthesis, which involves functional synthesis with explicit dependencies.

Chapter 7

Skolem Synthesis

In the previous chapter, as shown in Figure 7.1, we have discussed different ingredients of the functional synthesis including preprocessing, generating data, learning candidate functions, and verifying and repairing candidates. This chapter of thesis put all these ingredients together to come-up with a data-driven approach called, Manthan [GRM20, GSRM21] for Skolem synthesis.

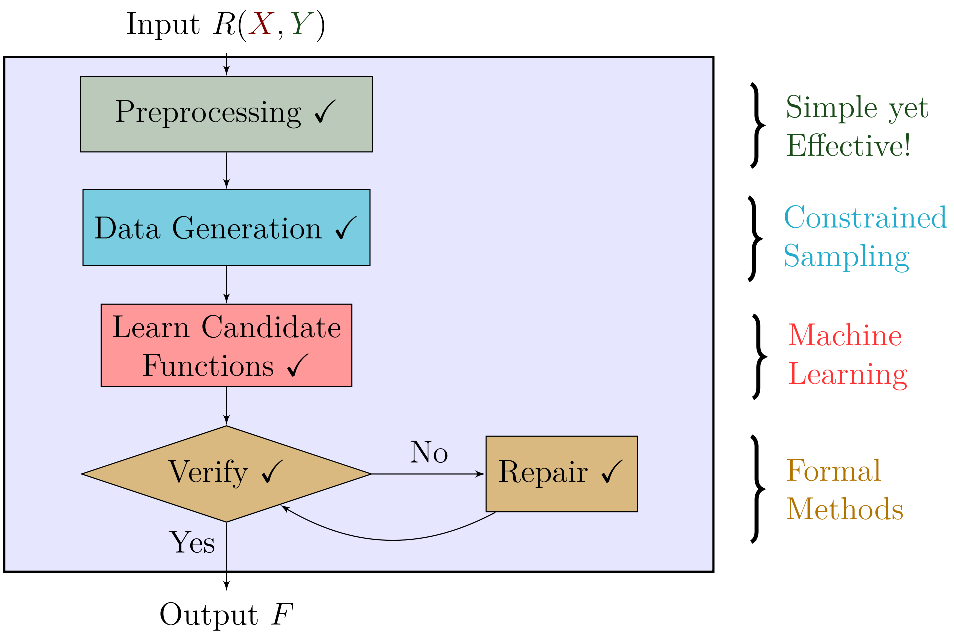


Figure 7.1: Manthan for Skolem synthesis.

Manthan begins by considering the specification $\varphi(X, Y)$ and performs preprocessing (Chapter 3) to identify unates and uniquely determined functions. Subsequently, Manthan utilizes constrained sampling to generate data (Chapter 4), which is then used as input for a machine-learning algorithm to learn initial candidate functions (Chapter 5). Additionally, Manthan capitalizes on advancements in automated reasoning and formal methods, relying on SAT and MaxSAT techniques to identify faulty candidates and perform necessary repairs (Chapter 6).

7.1 Approach

Algorithm 8 showcases Manthan algorithm. Given a formula $\varphi(X, Y)$ as input, Manthan generates a Skolem function vector $\boldsymbol{f}$ as its output.

Manthan does multiclassification to learn candidate functions for a set of Y variables together 5. Hence, during the execution, Manthan considers predetermined values for k and s. Here, k represents the maximum edge distance used to cluster the Y variables, while s denotes the maximum number of Y variables that can be jointly learned.

Algorithm 8 Manthan($\varphi(X, Y)$)

```
1: $\boldsymbol{f} \leftarrow \{f_1 = \emptyset, \ldots, f_{|Y|} = \emptyset\}$
2: dependson $\leftarrow \{\}$
3: U, $\boldsymbol{f}$, dependson $\leftarrow$ UniDef($\varphi(X, Y)$,$\boldsymbol{f}$,dependson)
4: $\Sigma \leftarrow$ GetSamples($\varphi(X, Y)$)
5: subsetY $\leftarrow$ ClusterY($\varphi(X, Y)$,k,s,U)
6: for each chunk $\in$ subsetY do
7:     $\boldsymbol{f}$, dependson $\leftarrow$ CandidateSkF($\Sigma$,$\varphi(X, Y)$, $\boldsymbol{f}$, chunk, dependson)
8: $TotalOrder \leftarrow$ FindOrder(dependson)
9: repeat
10:     $E(X, Y, Y') \leftarrow \varphi(X, Y) \wedge \neg\varphi(X, Y') \wedge (Y' \leftrightarrow \boldsymbol{f})$
11:     ret, $\sigma \leftarrow$ CheckSat(E(X,Y,Y'))
12:     if ret = SAT then
13:         ind $\leftarrow$ FindRepairCandidates($\varphi, \sigma, TotalOrder$)
14:         for $y_k \in$ ind do
15:             $\boldsymbol{f} \leftarrow$ RepairSkF($\varphi, \sigma$,$\boldsymbol{f}$, $TotalOrder$)
16: until ret = UNSAT
17: return $\boldsymbol{f}$
```

Algorithm 8 outlines Manthan. At line 3, the algorithm begins by extracting Skolem functions for the unates and uniquely defined variables in the formula $\varphi(X, Y)$. The resulting set U comprises Y variables that either possess unate characteristics or have unique Skolem functions. Note that if $U = Y$, that is, if all Y variables are either unate or uniquely defined, then Manthan terminates after UniDef.

To generate the required samples, Manthan proceeds to line 4. Subsequently, at line 5, Manthan calls the subroutine ClusterY to cluster the Y variables not present in U. The subroutine returns a list denoted as subsetY, representing various subsets of Y variables for which candidate functions will be learned jointly.

The learning process for each subset is executed by calling CandidateSkF at line 7. Additionally, CandidateSkF updates the dependencies among the Y variables based on the learned candidate functions. To establish a total order of the Y variables according to their dependencies, Manthan determines the order *TotalOrder* at line 8.

Next, Manthan verifies the satisfiability of the error formula $E(X, Y, Y')$. If $E(X, Y, Y')$ is found to be satisfiable, the algorithm proceeds to line 13 and calls the subroutine FindRepairCandidates to identify the list of candidates that require repair. Subsequently, at line 15, the algorithm invokes the subroutine RepairSkF to repair the candidates. This process continues until the error formula $E(X, Y, Y')$ becomes unsatisfiable. Finally, Manthan returns a Skolem function vector.

We now illustrate Manthan through an example.

Example 7.1. *Let* $X = \{x_1, x_2\}$, $Y = \{y_1, y_2, y_3, y_4\}$ *in* $\exists Y \varphi(X, Y)$ *where* $\varphi(X, Y)$ *is* $(x_1 \vee x_2 \vee y_1) \wedge (x_2 \vee \neg y_1 \vee y_2) \wedge (y_3 \vee y_4) \wedge (\neg y_3 \vee \neg y_4)$.

Finding Unate and Unique Functions FindUniqueDef finds that y_4 is defined by $\{x_1, x_2, y_1, y_2, y_3\}$ and returns the Skolem function $f_4 = \neg y_3$. We get $Z = \{y_4\}$ as a determined set.

Learning Candidate Functions Afterwards, Manthan proceeds to generate training data by sampling (Figure 7.2). The goal is to cluster the variables $Y \setminus Z = y_1, y_2, y_3$ into different groups to be learned jointly. In this case, as y_1 and y_2 share a clause, the clustering subroutine, ClusterY, returns the clusters y_1, y_2, y_3.

Manthan then attempts to learn candidate Skolem functions f_1 and f_2 together by constructing a decision tree (Figure 7.3). The construction of the decision tree uses the samples from x_1, x_2, y_3 as features and the samples from y_1, y_2 as labels. To synthesize the candidate function f_1, a disjunction is taken over all paths that end in leaf nodes with label 1 at index 1 in the learned decision tree. In Figure 7.3, f_1 is synthesized as $(x_1 \vee (\neg x_1 \wedge \neg x_2))$. Similarly, considering paths to leaf nodes with label 1 at index 2, we obtain $f_2 = (\neg x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_2)$, which simplifies to $\neg x_1$.

To predict y_3, samples from x_1, x_2, y_1, y_2 are used. By considering the path to the leaf node of the learned decision tree with label 1, we obtain $f_3 = x_2$.

At the end of CandidateSkF, we have $f_1 := (x_1 \vee (\neg x_1 \wedge \neg x_2)),\ f_2 := \neg x_1,\ f_3 := x_2$, and $f_4 := \neg y_3$. Let us assume the total order returned by FindOrder is $\mathit{TotalOrder} = \{y_4, y_3, y_2, y_1\}$.

Verifiying and Repairing Candidates We construct the error formula, $E(X, Y, Y') = \varphi(X, Y) \wedge \neg\varphi(X, Y') \wedge (Y' \leftrightarrow \boldsymbol{f})$, which is found to be satisfiable (SAT) with the counterexample $\sigma = \langle x_1 \leftrightarrow 1,\ x_2 \leftrightarrow 0,\ y_1 \leftrightarrow 0,\ y_2 \leftrightarrow 1,\ y_3 \leftrightarrow 0,\ y_4 \leftrightarrow 1,\ y'_1 \leftrightarrow 1,\ y'_2 \leftrightarrow 0,\ y'_3 \leftrightarrow 0,\ y'_4 \leftrightarrow 1\rangle$.

The subroutine FindRepairCandidates is called, and it invokes LexMaxSAT with $\varphi(X, Y) \wedge (x_1 \leftrightarrow \sigma[x_1]) \wedge (x_2 \leftrightarrow \sigma[x_2])$ as hard constraints and $((y_1 \leftrightarrow \sigma[y'_1]), 4) \wedge ((y_2 \leftrightarrow \sigma[y'_2]), 3) \wedge ((y_3 \leftrightarrow \sigma[y'_3]), 2) \wedge ((y_4 \leftrightarrow \sigma[y'_4]), 1)$ as soft constraints. The preference order of the soft constraints is determined by their weights. FindRepairCandidates returns $\mathit{ind} = y_2$.

Repair synthesis begins for f_2 by checking the satisfiability of $G_2 = \varphi(X, Y) \wedge (x_1 \leftrightarrow \sigma[x_1]) \wedge (x_2 \leftrightarrow \sigma[x_2]) \wedge (y_1 \leftrightarrow \sigma[y'_1]) \wedge (y_2 \leftrightarrow \sigma[y'_2])$. The formula G_2 is found to be unsatisfiable, and Manthan proceeds to call FindCore, which identifies variable y_1 as the core. This indicates that the constraints $(y_1 \leftrightarrow \sigma[y'_1])$ and $(y_2 \leftrightarrow \sigma[y'_2])$ are not jointly satisfiable in G_2.

As the output f_2 for the assignment σ needs to change from 0 to 1, f_2 is repaired by disjoining it with y_1, resulting in $f_2 := \neg x_1 \vee y_1$ as the updated

x_1	x_2	y_1	y_2	y_3
0	0	1	1	0
0	1	0	1	1
1	1	1	0	1

Figure 7.2: Skolem synthesis by Manthan example: Data generation for φ.

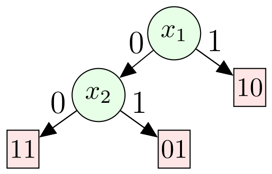


Figure 7.3: Learning candidates for y_1, y_2 with feature set $\{x_1, x_2, y_3\}$.

candidate.

For the revised candidate vector $\boldsymbol{f}$, the error formula becomes unsatisfiable (UNSAT), indicating a successful repair. Thus, $\boldsymbol{f}$ is returned as a Skolem function vector.

Remark 7.2. *If the subroutine* FindRepairCandidates *had employed regular (unweighted) MaxSAT instead of lexicographic MaxSAT, it might have returned* $Ind = y_1$. *However, considering the total order TotalOrder, where y_1 occurs after y_2, the formula G_1 would not be allowed to impose constraints on y_2. Therefore,* $G_1 = \varphi(X, Y) \wedge (x_1 \leftrightarrow \sigma[x_1]) \wedge (x_2 \leftrightarrow \sigma[x_2]) \wedge (y_1 \leftrightarrow \sigma[y_1'])$.

With the counterexample σ, formula G_1 is found to be SAT. As a result, Manthan *would need to find another repair candidate (in this case y_2), thus failing to resolve the counterexample σ in a single repair iteration. Consequently,* Manthan *mitigates the number of repair iterations by employing lexicographic MaxSAT.*

It is important to note that each phase of Manthan incorporates several optimizations, as outlined below:

1. Preprocessing: In this phase (discussed in Chapter 3), Manthan identifies uniquely determined variables and their corresponding functions. Rather than eliminating these variables, Manthan retains them, leading to optimization in subsequent steps.

2. Candidate learning: During candidate learning (discussed in Chapter 5), Manthan employs multi-classification techniques to learn the candidate functions. This optimization enhances the efficiency and accuracy of the learning process.

3. Use of lexicographic MaxSAT: Manthan utilizes lexicographic MaxSAT (discussed in Chapter 6) for the identification of erring candidates. This approach ensures that repair candidates are prioritized based on a preference order, reducing the number of repair iterations required.

Henceforth, we will distinguish between two different variants or versions of Manthan [GRM20]: one without the optimizations and another with the optimizations. The latter version will be referred to as Manthan2 [GSRM21].

7.2 Experimental Results

We conducted experiments using the prototype implementation of Manthan[1], which demonstrated its scalability in addressing the challenge at hand.

We will initially present the results obtained using Manthan and subsequently compare the performance of Manthan2 against Manthan, highlighting the impact and significance of each optimization. We now discuss the experimental set.

Benchmarks: We performed experiments on the union of all the benchmarks employed in the most recent works [AAC$^+$19, ACG$^+$18],which includes 609 benchmarks from different sources: Prenex-2QBF track of QBFEval-17[qbfa], QBFEval-18[qbfb], disjunctive[ACJS17], arithmetic[TV17] and factorization[ACJS17]. QBF competition benchmarks are consists of controller synthesis, reactive synthesis, game strategy synthesis. Arithmetic benchmarks are of simple arithmetic functions such as ceil, floor, max, min, and in factorization benchmarks, a system needs to compute the factors of the given input.

Setup: We used Open-WBO [MML14] for our MaxSAT queries and PicoSAT [Bie08] to compute UNSAT cores. We used PicoSAT for its ease of usage and we expect further performance improvements by upgrading to one of the state of the art SAT solvers. We have used the Sklearn[skl] Python library to create decision trees while

[1]Manthan is available open-sourced at `https://github.com/meelgroup/manthan`

learning candidates. We have also used ABC [LG] to represent and manipulate Boolean functions. To allow for the input formats supported by the different tools, we use the utility scripts available with the BFSS distribution [ACG+18] to convert each of the instances to both QDIMACS and Verilog formats.

All experiments were conducted on a high-performance computer cluster with each node consisting of a E5-2690 v3 CPU with 24 cores and 96GB of RAM, with a memory limit set to 4GB per core. All tools were run in a single-threaded mode on a single core with a timeout of 7200 seconds.

We used the PAR-2 score to compare different techniques, which corresponds to the Penalized Average Runtime, where for every unsolved instance there is a penalty of $2 \times$ timeout. timeout was 7200 seconds. We generally use cactus plot to showcase instances solved in which the number of instances are shown on the x-axis and the time taken on the y-axis; a point (x, y) implies that a solver took less than or equal to y seconds to find Skolem function of x instances on a total of 609 instances.

Tools: Manthan is compared with the state of the art synthesis tools, BFSS [ACG+18], C2Syn [AAC+19], BaFSyn [CFTV18] and the current state of the art 2-QBF solvers CADET [RTRS18],CAQE [RT15] and DepQBF [LE17]. The certifying 2-QBF solver produces QBF certificates, that can be used to extract Skolem functions [BJ11]. Developers of BaFSyn and DepQBF confirmed that the tools produce Skolem function for only valid instances, i.e. when $\forall X \exists Y \varphi(X, Y)$ is valid.

Experimental Evaluation: Manthan

The objective of our experimental evaluation was two-fold: to understand the impact of various design choices on the runtime performance of Manthan and to perform an extensive comparison of runtime performance vis-a-vis state of the art synthesis tools. In particular, we sought to answer the following questions:

1. How does the performance of Manthan compare with state of the functional synthesis engines?

2. How do the usage of different sampling schemes and the quality of samplers impact the performance of Manthan?

3. What is the impact of candidate learning phase on the performance of Manthan?

4. What is the distribution of the time spent in each component of Manthan: (i) preprocessing, (ii) data generation, (iii) candidate learning, (iv) verify and repair?

5. How does employing self-substitution for some Skolem functions impact Manthan?

Our observations indicate that Manthan exhibits substantial improvements compared to state-of-the-art tools. It successfully solves 356 benchmarks, while the best-performing existing tool can only solve 280. Remarkably, Manthan resolves an additional 60 benchmarks that none of the state-of-the-art tools could solve. To provide a broader perspective on the runtime performance, techniques developed over the past five years have been able to solve a range of 206 to 280 benchmarks, representing a difference of 74. Consequently, Manthan contributes to a significant increase of 76 benchmarks solved (from 280 to 356).

We also noted that the performance of Manthan is influenced by the choice of sampling schemes and the underlying samplers. In our experiments, we found that adaptive weighted sampling produces superior results compared to uniform sampling. Moreover, we observed interesting trade-offs between the number of samples and the minimum impurity decrease during candidate learning.

The diversity of our extensive benchmark suite provides a nuanced understanding of the time distribution across different phases of Manthan, highlighting the critical nature of each phase in determining its overall performance. Additionally, we observed that Manthan demonstrates notable performance improvements with self-substitution, reducing the number of repairs required.

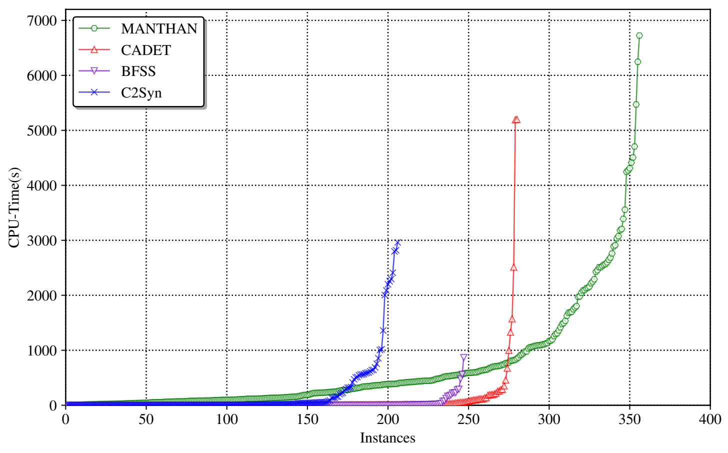


Figure 7.4: Skolem Synthesis: Manthan versus competing tools.

Table 7.1: Skolem Synthesis: No. of benchmarks solved by different tools.

Total	BaFSyn	CAQE	DepQBF	C2Syn	BFSS	CADET	Manthan	All Tools
609	13	54	59	206	247	280	**356**	476

Comparison with other tools

We now present performance comparison of Manthan with the current state of the art synthesis tools, BFSS [ACG$^+$18], C2Syn [AAC$^+$19], BaFSyn [CFTV18] and the current state of the art 2-QBF solvers CADET [RTRS18],CAQE [RT15] and DepQBF [LE17]. The certifying 2-QBF solver produces QBF certificates, that can be used to extract Skolem functions [BJ11]. Developers of BaFSyn and DepQBF confirmed that the tools produce Skolem function for only valid instances, i.e. when $\forall X \exists Y \varphi(X, Y)$ is valid. Note that the current version of CAQE does not support certification and we have used CAQE version 2 for the experiments after consultation with the developers of CAQE.

We present the number of instances solved Table 7.1. Out of 609 benchmarks, the most number of instances solved by any of the remaining techniques is 280 while Manthan is able to solve 356 instances – a significant improvement over state of the art. We will focus on top 4 synthesis tools from Table 7.1 for further analysis.

For a deeper analysis of runtime behavior, we present the cactus plot. An interesting behavior predicted by cactus plot and verified upon closer analysis is that

Table 7.2: Skolem Synthesis: Manthan vs other state-of-the-art tools.

		C2Syn	BFSS	CADET	All Tools
Manthan	Less	13	85	111	122
	More	**163**	**194**	**187**	**60**

for instances that can be solved by most of the tools, the initial overhead due to a multi-phase approach may lead to relatively larger runtime for Manthan. However, with the rise in empirically observed hardness of instances, one can observe the strengths of the multi-phase approach. Overall, Manthan solves 76 more instances than the rest of the remaining techniques.

We present a pairwise comparison of Manthan with other techniques in Table 7.2. The second row of the table indicates the number of instances solved by the respective technique in the corresponding column but not by Manthan, while the third row represents the number of instances solved by Manthan but not by the corresponding technique. Notably, Manthan solves 194, 163, and 187 instances that are not solved by BFSS, C2Syn, and CADET, respectively.

Although BFSS and CADET solve more than 80 instances that Manthan fails to solve, they do not provide complementary solutions. Specifically, there are only 121 instances that can be solved by either BFSS or CADET, but Manthan cannot solve. Further analysis of Manthan's performance on these instances revealed that the decision trees generated by CandidateSkF were shallow, indicating significant under-fitting. On the other hand, there are 130 instances that Manthan solves, but neither CADET nor BFSS can solve. These instances exhibit high dependencies between variables that Manthan can infer from the samples, enabling it to predict effective candidate Skolem functions.

Akshay et al. [AAC+19] suggest that C2Syn is an orthogonal approach to BFSS. Interestingly, Manthan solves 81 instances that cannot be solved by either C2Syn or BFSS. Furthermore, when considering the instances that C2Syn and BFSS solve together, there are 86 instances where Manthan fails to solve.

Overall, Manthan successfully solves **60** instances that none of the above-mentioned state-of-the-art tools can handle, highlighting its superior performance and efficacy.

Impact of the sampling scheme

To evaluate the impact of adaptive sampling and the quality of distributions generated by underlying samplers, we extended Manthan with samples drawn from different samplers for both adaptive and non-adaptive sampling. Specifically, we utilized QuickSampler [DLBS18], KUS [SGRM18], UniGen2 [CMV14], and CMSGen[GSCM21]. However, KUS and UniGen2 could only produce samples for a limited number of benchmarks (14 and 49 respectively) within a timeout of 3600 seconds. Therefore, we have excluded KUS and UniGen2 from further analysis. We also experimented with a naive enumeration of solutions using the off-the-shelf SAT solver, CryptoMiniSat[Soo19]. It is worth noting that QuickSampler performed worse than CMSGen for uniformity testing using Barbarik [CM19]. In our implementation, we had to disable the validation phase of QuickSampler to generate a sufficient number of samples within a reasonable timeframe. To statistically validate the intuition described in Chapter 4, we performed adaptive sampling using CMSGen.

Table 7.3 presents the performance of Manthan with different samplers listed in Column 1. Columns 2, 3, and 4 represent the number of instances solved during the execution of respective phases: finding unates, candidates learning, and Repair. Finally, Column 5 lists the total number of instances solved.

Table 7.3: Skolem Synthesis: Manthan with different samplers.

Sampler	No. of instances solved			#Solved
	finding unates	candidates learning	Repair	
CryptoMiniSat	66	14	191	271
QuickSampler	66	28	181	275
CMSGen	66	51	228	345
wCMSGen	66	66	224	**356**

Two important findings emerge from Table 7.3: Firstly, as the quality of samplers improves, the performance of Manthan also improves. Specifically, we observe that with the enhancement in the quality of samples, Manthan solves more instances in candidates learning. Secondly, we observe a significant increase in the number of instances solved in candidates learning when using samples from wCMSGen. It is important to note that the adaptive sampling scheme proposed in Chapter 4 should be viewed as a proof of concept, and our results will encourage the development of more sophisticated schemes.

Impact of candidate learning

To analyze the impact of different design choices in candidates learning, we analyzed the performance of Manthan for different samples (1000, 5000 and 10000) generated by GetSamples and for different choices of minimum impurity decrease (0.001, 0.005, 0.0005) to prune the decision tree while learning the candidate functions. Figure 7.5 shows a heatmap on the number of instances solved on each combination of the hyper parameters; the closer the color of a cell is to the red end of the spectrum, the better the performance of Manthan.

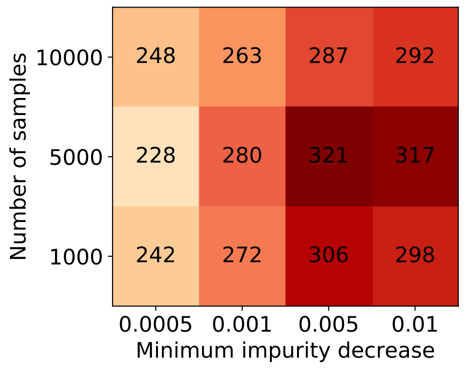


Figure 7.5: Skolem Synthesis: Heatmap for Manthan with # of instances solved. Skolem Synthesis: Heatmap for Manthan with # of samples vs decision tree pruning. [Best viewed in colour].

At the first look, Figure 7.5 presents a puzzling picture: It seems that increasing the number of samples does not improve the performance of Manthan. On a closer analysis, we found that the increase in the number of samples leads to an increase in the runtime of CandidateSkF but without significantly increasing the number of instances solved during candidates learning. The runtime of CandidateSkF is depen-

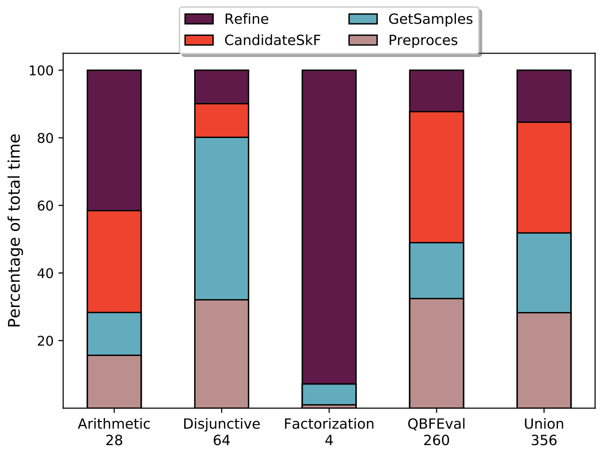


Figure 7.6: Skolem synthesis: Fraction of time spent in different phases of Manthan over different classes of benchmarks. [Best viewed in colour.].

dent on the number of samples and $|Y|$. On the other hand, we see an interesting trend with respect to minimum impurity decrease where the performance first improves and then degrades.

A plausible explanation for such a behavior is that with an increase in *minimum impurity decrease*, the generated decision trees tend to underfit while significantly low values of *minimum impurity decrease* lead to overfitting. Based on the above observations, we set the value of minimum impurity decrease to 0.005 and set the number of samples to (1) 10000 for $|Y| < 1200$, (2) 5000 for $1200 < |Y| \leq 4000$, and (3) 1000 for $|Y| > 4000$.

Division of time taken across different phases

To analyze the time taken by different phases of Manthan across different categories of the benchmarks, we normalize the time taken for each of the four core subroutines, Preprocess, GetSamples, CandidateSkF, and RepairSkF, for every benchmark that was solved by Manthan such that the sum of time taken for each benchmark is 1. We then compute the mean of the normalized times across different categories instances. Figure 7.6 shows the distribution of mean normalized times for different categories: Arithmetic, Disjunction, Factorization, QBFEval, and all the instances.

The diversity of our benchmark suite shows a nuanced picture and shows that the time taken by different phases strongly depends on the family of instances. For example, the disjunctive instances are particularly hard to sample and an improvement in the sampling techniques would lead to significant performance gains. On the other hand, a significant fraction of runtime is spent in the CandidateSkF subroutine indicating the potential gains due to improvement in decision tree generation routines. In all, Figure 7.6 identifies the categories of instances that would benefit from algorithmic and engineering improvements in Manthan's different subroutines.

Impact of self-substitution

To understand the impact of self-substitution, we profile the behavior of candidate Skolem functions with respect to number of repairs for two of our benchmarks; *pdtpmsmiim-all-bit* and *pdtpmsmiim*. In Figure 7.8 and 7.7, we use histograms with the number of candidate Skolem functions on y-axis and required number of repairs on x-axis. A bar of height a i.e $y = a$ at b i.e $x = b$ in Figure 7.8 and 7.7 represents that a candidate Skolem functions converged in b repairs. The histograms show that only a few Skolem functions require a large number of repairs: the tiny bar towards the right end in Figure 7.8 represents that for the benchmark *pdtpmsmiim-all-bit* only 1 candidate Skolem function required more than 60 repairs whereas all other candidate Skolem functions needed less than 15 repairs. Similarly, for the benchmark *pdtpmsmiim* Figure 7.7 shows that only 1 candidate Skolem function was refined more than 15 times, whereas all other Skolem functions required less than 5 repairs. We found similar behaviors in many of our other benchmarks.

Based on the above trend and an examination of the decision trees corresponding to these instances, we hypothesize that some Skolem functions are hard to learn through data. For such functions, the candidate Skolem function generated from the data-driven phase in Manthan tends to be poor, and hence Manthan requires a long series of repairs for convergence. Since our repair algorithm is designed for small, efficient corrections, we handle such hard to learn Skolem functions by synthesizing via self-substitution. Manthan detects such functions via a threshold on the number of repairs, which is empirically determined as 10, to identify hard to

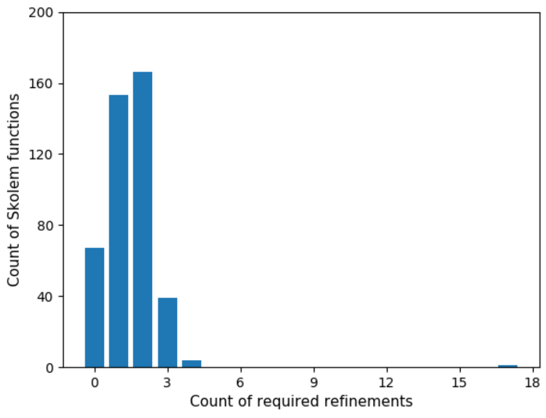


Figure 7.7: Skolem synthesis: # of Skolem functions vs # of repairs for benchmark *pdtpmsmiim*.

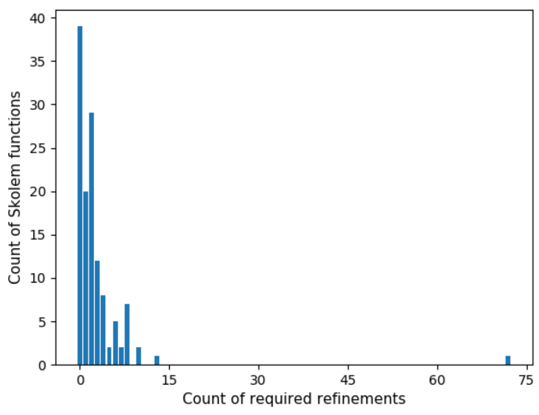


Figure 7.8: Skolem synthesis: # of Skolem functions vs # of repairs for benchmark *pdtpmsmiim-all-bit*.

learn instances and sets them up for self-substitution.

In our experiments, we found 75 instances out of 356 solved instances required self-substitution, and for 51 of these 75 instances, only one variable undergoes self-substitution. Table 7.4 shows the impact of self-substitution for five of our benchmarks: Manthan has significant performance improvement with self-substitution in terms of the required number of repairs, which in turns affects the overall time. Taking the case of the last benchmark, all the other Skolem functions for it were synthesized earlier than 40 repair cycles, and the last 16 iterations were only needed for 2 of the poor candidate functions to hit our threshold for self-substitution. Note that self-substitution can lead to an exponential blowup in the size of the formula,

Table 7.4: Manthan : Impact of self substitution

Benchmarks $\exists Y F(X,Y)$	$\|X\|$	$\|Y\|$	No. of Refinements		Time(s)	
			Self-Substitution		Self-Substitution	
			Without	With	Without	With
kenflashpo2-all-bit	71	32	319	10	35.88	19.22
eijkbs1512	316	29	264	10	42.88	32.35
pdtpmsmiim-all-bit	429	30	313	10	72.75	36.08
pdtpmssfeistel	1510	68	741	10	184.11	115.07
pdtpmsmiim	418	337	127	56	1049.29	711.48

but it works quite well in our design as most Skolem functions are learnt quite well in the candidates learning phase.

Experimental Evaluation: Manthan2

Manthan with different algorithmic improvements resulted in Manthan2, which led to dramatic scalability. Let us discuss the empirical experimental in detail, in particular, our empirical evaluation sought answers to the following questions:

1. What is the improvement of Manthan2 over Manthan in terms additional instances solved?

2. What is the impact on the performance of Manthan2 of each of the proposed modifications? Specifically:

   (a) Does Manthan2 benefit from extracting the unique Skolem functions via unique function determination?

   (b) Does retaining variables help Manthan2 to learn and repair candidates efficiently?

   (c) How does the candidate learning time improve with multi-classification?

   (d) What is the impact of using LexMaxSAT to find candidates to repair?

Summary of Results Manthan2 [GSRM21] outperforms all the state-of-the-art tools by solving 509 benchmarks, while Manthan [GRM20] solves 356 benchmarks—an increase of **153** benchmarks over Manthan. It is worth emphasizing that the increment of 153 is more than twice the improvement shown by Manthan over CADET [Rab19], which could solve 280 benchmarks. There was an increase of 76 benchmarks in the number of solved benchmarks over the state-of-the-art due to Manthan. With Manthan2, we see an increment of 153 benchmarks over Manthan, which is more than the twice of 76, i.e we see $\sim 200\%$ of performance improvement.

Moreover, we found that extracting unique functions is useful. There are 246 benchmarks out of 609 for which the ratio of Y variables being uniquely defined to the total number of Y is greater than 95%, that is, Manthan2 could extract Skolem functions for that many variables via unique function extraction. There is an increase of 25 benchmarks in the number of solved instances by retaining variables in the determined set to learn and repair candidates. Further, learning candidate functions for a subset of variables together with the help of multi-classification reduces the PAR-2 score from 3227.11 to 2974.91. Finally, we see a reduction of 100 seconds in the PAR-2 score by LexMaxSAT.

Manthan2 **vis-a-vis** Manthan

Table 7.5 presents a pairwise comparison of Manthan2 with Manthan. The first column (PreRepair) presents the number of benchmarks that needed no repair iteration to synthesize a Skolem function vector. The second column (Repair) represents the number of benchmarks that underwent repair iterations. The third column (Self-Sub) presents the number of benchmarks for which at least one variable had to undergo self-substitution.

We investigate the reason for the increase in the number of benchmarks solved in PreRepair, and observed that Manthan2 could extract Skolem functions via unique function extraction for 90% of the variables for 274 out of these 385 benchmarks.

We also observed a significant decrease in the number of benchmarks that needed repair iterations. Out of 124 benchmarks that underwent repair to synthesize a Skolem function vector, only 33 benchmarks needed self-substitution with Manthan2, whereas there are 75 out of 224 benchmarks that needed self-substitution with

Table 7.5: Skolem synthesis: Pairwise comparison of Manthan2 with Manthan

	PreRepair	Repair	Self-Sub
Manthan	132	224	75
Manthan2	385	124	33

Table 7.6: Skolem Synthesis: Performance Summary over 609 benchmarks

	C2Syn	BFSS	CADET	Manthan	Manthan2
Solved	206	247	280	356	509
PAR-2	9594.83	8566.87	7817.58	6374.39	2858.61

Manthan. The fact that fewer benchmarks required self-substitution to synthesize a Skolem function vector shows that Manthan2 could find some hard-to-learn Skolem functions.

Manthan2 vs. other competing tools. We compared Manthan2 with state-of-the-art tools: C2Syn [AAC+19], BFSS [ACG+18], CADET [Rab19]. Figure 7.9 shows a cactus plot to compare the run-time performance of different tools.

As shown in Figure 7.9, Manthan2 significantly improves on the state of the art techniques, both in terms of the number of instances solved and runtime performance. In particular, Manthan2 is able to solve 509 instances while Manthan can solve only 356 instances, thereby achieving an improvement of 153 instances in the number of instances solved. To measure the runtime performance in more detail,

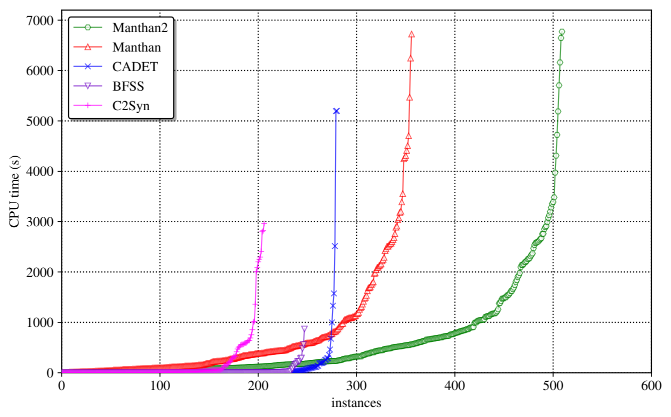


Figure 7.9: Skolem synthesis: Manthan2 vis-a-vis state-of-the-art synthesis tools.

Table 7.7: Skolem synthesis: Manthan2 vs. other state-of-the-art tools.

		C2Syn	BFSS	CADET	Manthan	All
Manthan2	Less	17	18	21	24	40
	More	320	280	250	177	71

we computed PAR-2 scores for all the techniques. The PAR-2 scores for Manthan2 and Manthan are 2858.61 and 6374.39, which is an improvement of 3521.78 seconds. Finally, we sought to understand if Manthan2 performs better than the union of all the other tools. Here, we observe that Manthan2 solves 71 instances that the other tools could not solve, whereas there are only 40 instances not solved by Manthan2 that were solved by one of the other tools.

Performance gain with each technical contribution

We now provide a detailed discussion on the impact and significance of each optimization incorporated in Manthan2 compared to Manthan.

Impact of unique function extraction

We now present the impact of extracting Skolem function for uniquely defined variables. Figure 7.10 shows the percentage of uniquely determined functions on the x-axis, and number of benchmarks on y-axis. A bar at x shows that y many benchmarks had $x\%$ of Y variables that are uniquely defined. As shown in Figure 7.10, there are 246 benchmarks out of 609 with more than 95% uniquely defined variables; therefore, Manthan2 could extract Skolem functions corresponding to these variables via unique function extraction. There are only 5 benchmarks where all the Y variables are defined. Our analysis shows that extracting unique functions significantly reduces the number of Y variables that needed to be learned and repaired in the subsequent phases of Manthan2.

We also analyzed the performance of Manthan2 with respect to unique function size. Note that we measure size in terms of number of clauses, as the extracted functions are in CNF. A benchmark is considered to have size S if the maximum

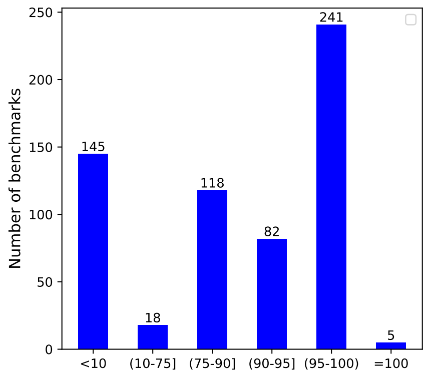


Figure 7.10: Skolem Synthesis: # benchmarks by % ratio of uniquely defined output variables.

Table 7.8: Skolem Synthesis: # benchmarks with different maximum function size for uniquely defined variables. Function size is measured in terms of number of clauses in the synthesized functions.

	[1-10]	(10-100]	(100-1000]	(> 1000)
#-benchmarks	209	203	61	136

size among all its unique functions is S.

Table 7.8 shows the number of benchmarks with different maximum unique function sizes. There are 136 benchmarks for which at least one uniquely defined variable has function size greater than 1000 clauses. In general, larger size functions require more data to learn. Table 7.8 shows that Manthan2 was able to extract some hard-to-learn Skolem functions.

An interesting observation is that there were 54 benchmarks that required self-substitution for just one variable with Manthan. However, Manthan2 was able to identify that particular variable as uniquely defined and the corresponding function size was more than 3000 clauses. This observation emphasizes that it is important to extract the functions for uniquely defined variables with large function size in order to efficiently synthesize a Skolem function vector. Therefore, even if there is only one variable with large function size, it is important to extract the corresponding function—the primary reason for considering maximum size instead of mean or median size in Table 7.8.

Impact of learning and repairing over determined features

As discussed in Chapter 3, either we can substitute the definitions of uniquely defined variables in formula and reduce the Y variable size, or we can allow non-uniquely defined variables to learn and repair candidates in terms of uniquely defined variable. To this end, we did an experiment considering both the variant.

We now present the impact of variable retention. Manthan2 could solved 502 instances with a PAR-2 score of 3227.11 by retaining variables in the *determined set* to use them further as features in learning and repairing the other candidates, whereas, if we eliminate them, it could solve only 477 instances with a PAR-2 score of 3523.28—a difference of 25 benchmarks.

It is worth mentioning that there are 370 instances that needed no repair iterations (solved in PreRepair) to synthesize a Skolem function vector when learned with determined features, whereas, if Manthan2 does not consider determined features, we see a reduction of 6 benchmark in the number of instances solved in PreRepair.

Interestingly, despite having fewer determined features, it is crucial to utilize them for learning and repairing candidates. For instance, let's consider the benchmark *query64_01*, which has a total of 597 variables. Out of these, only five variables can be identified as determined features. If we were to remove these five variables, Manthan2 would be unable to synthesize a Skolem function vector even with more than 150 repair iterations within the 7200-second timeout. However, by retaining these determined features, Manthan2 can successfully synthesize a Skolem function vector within just nine repair iterations, taking less than 400 seconds.

Efficacy of multi-classification and impact of LexMaxSAT

As discussed in Chapter 5, two essential questions arise when using multi-classification to learn candidates for a subset of Y together: 1) how to divide the Y variables into different subsets, and 2) how many variables should be learned together?

We experimented with following techniques to divide Y variables into subsets of sizes 5 and 8, i.e, s = 5 or 8:

1. Randomly dividing Y variables into different disjoint subsets.
2. Clustering Y variables in accordance to the edge distance (parameter k) in the primal graph: (i) using $k = 2$ (ii) using $k = 3$

Figure 7.11 and 7.12 shows a heatmap of PAR-2 scores for different configurations of Manthan2. A lower PAR-2 score, i.e., a tilt towards the red end of the spectrum in Figure 7.11 and 7.12, indicates a favourable configuration. The columns of Figure 7.11 correspond to different ways of dividing Y variables into different subsets: (i) *Random*, (ii) $k = 2$, and (iii) $k = 3$. The rows of Figure 7.11 show results for different maximum sizes of such subsets, i.e., s = 5, 8. The number of instances solved in each configuration is also shown in brackets. For comparison, the PAR-2 score of Manthan2 with binary classification is 3227.11s and it solved 502 benchmarks.

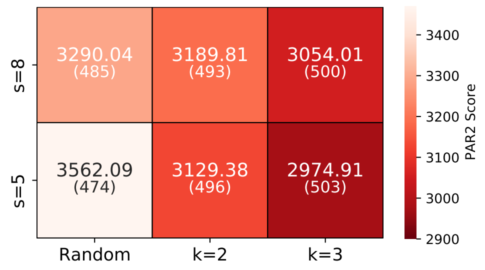


Figure 7.11: Skolem synthesis: Manthan2 with multi-classification and LexMaxSAT turned off.

Let us first discuss Figure 7.11, i.e, the results without LexMaxSAT. Manthan2 shows a performance improvement with the proposed clustering-based approach in comparison to randomly dividing Y variables into subsets. As shown in Figure 7.11, we observed a drop in PAR-2 score when moving from random to cluster-based partitioning of Y variables.

We see a better PAR-2 score with graph-based multi-classification compared to binary classification, though the number of instances solved (except with k=3, s=5) is lower than the number of instances solved with binary classification. This shows that dividing Y variables using a cluster-based approach is effective in reducing the candidate learning time. Manthan2 performs best with $k = 3$ and $s = 5$, where it could solve 503 benchmarks (1 more instance than with binary classification) with a PAR-2 score of 2974.9s, which amounts to a reduction of 252 seconds over the PAR-2 score with binary classification. We observe a similar trend with LexMaxSAT turned on (as shown in Figure 7.12).

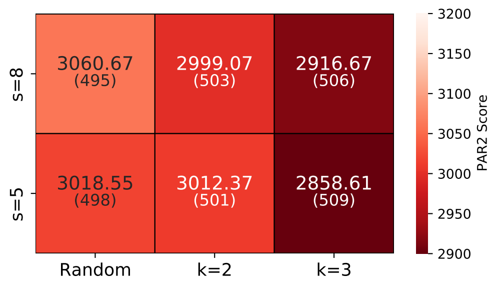


Figure 7.12: Skolem synthesis: Manthan2 with multi-classification and LexMaxSAT turned on.

Finally, let us move our attention towards the impact of LexMaxSAT, shown in Figure 7.12. Manthan2 uses LexMaxSAT only if the number of candidates to repair exceeds 50 times the number of candidates chosen by MaxSAT. A comparison of Figure 7.11 and Figure 7.12 shows that with LexMaxSAT, Manthan2 solves at least 3 more benchmarks for all the configurations.

Manthan2 performs best when we turn on LexMaxSAT and set $k = 3$ as well as $s = 5$. The results discussed in Chapter 7.2 were achieved with this configuration.

Chapter 8

Henkin Synthesis

Let's transition from general-purpose monolithic functional synthesis to modular design, where different components of synthesized systems depend on distinct sets of inputs. In this context, synthesis with explicit dependencies involves handling Dependency Quantified Formulas (DQBF). As mentioned in Chapter 2, a formula ϕ is considered a DQBF if it can be represented as $\phi : \forall x_1 \dots x_n \exists^{H_1} y_1 \dots \exists^{H_m} y_m \varphi(X, Y)$, where $X = x_1, \dots, x_n$, $Y = y_1, \dots, y_m$, and $H_i \subseteq X$ represents the dependency set of y_i. This means that variable y_i can only depend on the variables in H_i. Each H_i is referred to as a Henkin dependency, and each quantifier $\exists^{H_i}$ is known as a *Henkin* quantifier [Hen61].

A DQBF ϕ is considered to be True, if there exists a function $f_i : \{0,1\}^{|H_i|} \mapsto \{0,1\}$ for each existentially quantified variable y_i, such that $\varphi(X, f_1(H_1), \dots, f_m(H_m))$, obtained by substitution of each y_i by its corresponding function f_i, is a tautology. Given a DQBF ϕ, the problem of DQBF satisfiability, is to determine whether a given DQBF is True or False. The DQBF satisfiability is a decision problem that looks for an answer to the question: *Does there exist a function corresponding to each existentially quantified variable, in terms of its Henkin dependencies, such that the formula substituted with the function in places of existentially quantified variables is a tautology?* Owing to wide variety of applications that can be represented as DQBF, recent years have seen a surge of interest in DQBF solving [GRS$^+$13, FKBV14, RSS21, TR19, WWSB16].

In many cases, a mere True/False answer is not sufficient as one is often interested in determining the definitions corresponding to those functions. For instance, in the context of engineering change of order (ECO), in addition to just knowing whether the given circuit could be rectified to meet the *golden* specification, one would also be interested in deriving corresponding patch functions [JKL20]. Owing to the naming of dependencies, we call such patch functions to be Henkin functions (formally discussed in Chapter 2).

We have made significant progress in advancing Henkin synthesis by introducing the Manthan framework, which enables the synthesis of Henkin functions [GRM23]. In our research, we proposed a novel approach called Manthan3 for Henkin function synthesis. Unlike existing techniques, Manthan3 combines the power of machine learning with automated reasoning to synthesize Henkin functions.

8.1 Approach

We now discuss each of the component discussed in Part II for Henkin synthesis.

Candidate Learning: Manthan3 learns a binary decision tree classifier for each existentially quantified variable y_i to learn the candidate function f_i corresponding to it. The valuations of y_i in the generated samples are considered labels, and the valuations of corresponding Henkin dependencies H_i are considered the feature set to learn a decision tree. A Henkin function f_i corresponding to y_i is computed as a disjunction of labels along all the paths from the root node to leaf nodes with label **1** in the learned decision tree.

Due to the Henkin dependencies, the feature set for y_i must be restricted only to H_i. However, in order to learn a good decision tree, we can include all the y_j in the set of features for which $H_j \subset H_i$. The function f_j can be simply expanded within f_i so that f_i is only expressed in terms of H_i. For the cases when $H_j = H_i$, such use of the Y variables is allowed as long it does not cause the cyclic dependencies; that is, if y_j appears in the learned candidate f_i, then y_i is not allowed as a feature to learn candidate f_j. If y_j appears in f_i, then we say y_i depends on y_j, denoted as $y_i \prec_d y_j$. Manthan3 discovers requisite variable ordering constraints among such Y variables on the fly as the candidate functions are learned, just like, Manthan.

Verification: The learned candidate vector may not always be a valid Henkin vector. Therefore, the candidate functions must be verified. $\boldsymbol{f}$ is a Henkin function vector only if $\varphi(X, f_1(H_1), \ldots, f_m(H_m))$ is a tautology. Manthan3 first, make a SAT solver call on the formula $E(X, Y') = \neg\varphi(X, Y') \wedge (Y' \leftrightarrow \boldsymbol{f})$.

If formula $E(X, Y')$ is UNSAT, Manthan3 returns the function vector $\boldsymbol{f}$ as a Henkin function vector. If formula $E(X, Y')$ is SAT and δ is a satisfying assignment of $E(X, Y')$, Manthan3 needs to find out whether $\varphi(X, Y)$ has a propositional model extending assignment of X. Therefore, Manthan3 performs another satisfiability check on formula $\varphi(X, Y) \wedge (X \leftrightarrow \delta[X])$. If satisfiability checks return UNSAT, the corresponding $DQBF$ formula is False, and there does not exist a Henkin function vector; therefore, Manthan3 terminates. Furthermore, if $\varphi(X, Y) \wedge (X \leftrightarrow \delta[X])$ is SAT, and π is a satisfying assignment and we need to repair the candidate function vector. Note that $\pi[X]$ is same as $\delta[X]$, and $\pi[Y]$ is a possible extending assignment of X, and $\delta[Y']$ presents the output of candidate function vector with $\delta[X]$. Now, we have a counterexample σ as $\pi[X] + \pi[Y] + \delta[Y']$.

Candidate Repair: We apply a counterexample driven repair approach for candidate functions. As Manthan3 attempts to fix the counterexample σ, it first needs to find which candidates to repair out of f_1 to f_m candidates. Manthan3 takes help of MaxSAT solver to find out the repair candidates, and makes a MaxSAT query with $\varphi(X, Y) \wedge (X \leftrightarrow \sigma[X])$ as hard constraints and $(Y \leftrightarrow \sigma[Y'])$ as soft constraints. It selects a function f_i for repair if the corresponding soft constraint $y_i \leftrightarrow \sigma[y_i']$ is falsified in the solution returned by the MaxSAT solver. Once, we have candidate functions to repair, Manthan3 employs unsatisfiability cores obtained from the infeasibility proofs capturing the reason for candidate functions to not meet the specification to construct a repair.

Let us now assume that Manthan3 selects f_i corresponding to variable y_i as a potential candidate. Manthan3 constructs another formula $G_i(X, Y)$ (Formula 8.1) to find the repair, which is a modification of Formula 6.2 discussed in Chapter 6:

$$\begin{aligned} G_i(X,Y) : \varphi(X,Y) \wedge (H_i \cup \hat{Y} \leftrightarrow \sigma[H_i \cup \hat{Y}]) \wedge (y_i \leftrightarrow \sigma[y_i']) \\ \text{where } \hat{Y} \subseteq Y \text{ such that } \forall y_j \in \hat{Y} : H_j \subseteq H_i \\ \text{and } \{\mathit{TotalOrder}[\mathit{index}(y_j)] > \mathit{TotalOrder}[\mathit{index}(y_i)]\} \end{aligned} \tag{8.1}$$

Informally, in order to determine whether f_i needs to be repaired, we conjunct the specification $\varphi(X, Y)$ with the conjunction of unit clauses that set the valuation of y_i to the current output of f_i and the valuation of all the dependencies as per the counter-example. We describe the intuition behind construction of $G_i(X, Y)$. The formula $G_i(X, Y)$ is constructed to answer: *Whether is it possible for y_i to be set to the output of f_i given the valuation of its Henkin dependencies?*.

The answer to the above question depends on whether $G_i(X, Y)$ is UNSAT or SAT. $G_i(X, Y)$ being UNSAT indicates that it is not possible for y_i to be set to the output of f_i and the UnsatCore of $G_i(X, Y)$ captures the reason. Accordingly, Manthan3 uses the UnsatCore of $G_i(X, Y)$ to repair the candidate function f_i. In particular, Manthan3 uses all the variables corresponding to unit clauses in Unsat-Core of $G_i(X, Y)$ to construct a repair formula β, and depending on the valuation of y_i' in the counter example σ, β is used to strengthen or weaken the candidate f_i to satisfy the counterexample.

On the other hand, if $G_i(X, Y)$ is SAT, Manthan3 attempts to find alternative candidate functions to repair. $G_i(X, Y)$ being SAT indicates that with the current valuation to Henkin dependencies, y_i could take a value as per the output of candidate f_i; however, to fix the counterexample σ, we need to repair another candidate function. To this end, let ρ be a satisfying assignment of $G_i(X, Y)$, then all y_j variables for which $\rho[y_j]$ is not the same as $\sigma[y_j']$ are added to the queue of potential candidates to repair.

In Formula 8.1, we add a constraint $\hat{Y} \leftrightarrow \sigma[\hat{Y}]$ in $G_i(X, Y)$ where $\hat{Y}$ is a set of Y variables such that for all y_j of $\hat{Y}$, $H_j \subseteq H_i$. Fixing valuations for such y_j variables helps Manthan3 to synthesize a better repair for candidate f_i. Consider the following example. Let $\forall X \exists^{H_1} \exists^{H_2} \varphi(X, Y)$, where $\varphi(X, Y) : (y_1 \leftrightarrow x_1 \oplus y_2)$, $H_1 = \{x_1\}$ and $H_2 = \{x_1\}$. Let us assume that we need to repair the candidate f_1, and $G_1(X, Y) = (y_1 \leftrightarrow \sigma[y_1']) \wedge \varphi(X, Y) \wedge (x_1 \leftrightarrow \sigma[x_1])$. As $G_1(X, Y)$ does not include the current value of y_2 that led to the counterexample, it misses out on driving f_1 in a direction that would ensure $y_1 \leftrightarrow x_1 \oplus y_2$. In fact, in this case repair formula β would be empty, thereby failing to repair.

The repair loop continues until either $E(X, Y')$ is UNSAT or $\varphi(X, Y) \wedge (X \leftrightarrow \delta[X])$ is UNSAT, where δ is a satisfying assignment of $E(X, Y')$. If $E(X, Y')$ is UNSAT, we have a Henkin function vector $\boldsymbol{f}$, and if $\varphi(X, Y) \wedge (X \leftrightarrow \delta[X])$ is UNSAT, then the given DQBF instance is False and there does not exist a Henkin function vector.

Algorithm for Manthan3. Manthan3 (Algorithm 9) takes a DQBF instance $\forall X \exists^{H_1} y_1 \ldots \exists^{H_m} y_m \varphi(X, Y)$ as input and outputs a Henkin function vector $\boldsymbol{f} := \langle f_1, \ldots, f_m \rangle$.

Algorithm 9 assumes access to the following subroutines:

1. GetSamples: It takes a specification as input and calls a constrained sampler to produce samples Σ of specifications. Each sample in Σ is a satisfying assignment of specifications.

2. CandidateSkF: This subroutine generates the candidate function corresponding to an existential variable. It takes a specification φ, generated samples Σ, existential variable y_i corresponding to which we want to learn a candidate function and a vector D that keeps track of dependencies among Y variables as input. CandidateSkF returns a candidate function f_i corresponding to y_i, and updates the dependencies in D for y_i. We discussed CandidateSkF routine in detail in Algorithm 4.

3. FindOrder: It takes a set D collection of d_i, where each d_i is the list of Y

Algorithm 9 Manthan3($\forall X \exists^H Y.\varphi(X,Y)$)

```
1: $\Sigma \leftarrow$ GetSamples($\varphi(X,Y)$)
2: $D \leftarrow \{d_1 = \emptyset \ldots, d_{|Y|} = \emptyset\}$
3: for $\langle H_i, H_j \rangle$ do
4:     if $H_j \subset H_i$ then
5:         $d_j \leftarrow d_j \cup y_i$
6: for $y_i \in Y$ do
7:     $f_i, D \leftarrow$ CandidateSkF($\Sigma, \varphi(X,Y), y_i, D$)
8: $TotalOrder \leftarrow$ FindOrder($D$)
9: repeat
10:    $E(X,Y') \leftarrow \neg\varphi(X,Y') \wedge (Y' \leftrightarrow \boldsymbol{f})$
11:    $ret, \delta \leftarrow$ CheckSat($E(X,Y')$)
12:    if ret = SAT then
13:        $res, \pi \leftarrow$ CheckSat($\varphi(X,Y) \wedge (X \leftrightarrow \delta[X])$)
14:        if res = UNSAT then
15:            return $\forall X \exists^H Y.\varphi(X,Y)$ is False.
16:        $\sigma \leftarrow \pi[X] + \pi[Y] + \delta[Y']$        ▷ //$\sigma$ is a counterexample
17:        $\boldsymbol{f} \leftarrow$ RepairSkF($\varphi(X,Y), \boldsymbol{f}, \sigma, TotalOrder$)
18: until ret = UNSAT
19: $\boldsymbol{f} \leftarrow$ Substitute($\varphi(X,Y), \boldsymbol{f}, TotalOrder$)
20: return $\boldsymbol{f}$
```

variables, which can depend on y_i. FindOrder obtains a valid linear extension, *TotalOrder*, from the partial dependencies in D.

4. CheckSat: It takes a specification as input and makes a SAT call to do a satisfiability check on the specification. It returns the outcome of satisfiability check as SAT or UNSAT. In the case of SAT, it also returns a satisfiable assignment of the specification.

5. RepairSkF: This subroutine repairs the current candidate function vector to fix the counterexample. It takes the specification, candidate function vector, a counterexample, and *TotalOrder*, a linear extension of dependencies among Y variables as input, and returns a repaired candidate function vector. Algorithm 7 discusses RepairSkF subroutine in detail.

Algorithm 9 starts with generating samples Σ by calling GetSamples subroutine at line 1. Next, Algorithm 9 initializes the set D (line 2), which is a collection of d_i, where d_i represents the set of Y variables that depends on y_i. Lines 3-5 introduce variable ordering constraints based on the subset relations in each $\langle H_i, H_j \rangle$ pair, that is, if $H_j \subset H_i$, then y_i can depend on y_j. Line 7 calls the subroutine CandidateSkF for every y_i variable to learn the candidate function f_i. Next, at line 8, Manthan3 calls FindOrder to compute *TotalOrder*, a topological ordering among the Y variables that satisfy all the ordering constraints in D.

In line 11, CheckSat checks the satisfiability of the formula $E(X, Y')$ described at line 10. If $E(X, Y')$ is SAT, then Manthan3 at line 13 performs another satisfiability check to ensure that propositional model to X can be extended to Y. If CheckSat at line 13 is UNSAT, then Algorithm 9 terminates at line 15 as there does not exists a Henkin function vector, otherwise Manthan3 has a counterexample σ to fix. The candidate vector $\boldsymbol{f}$ goes into a repair iteration (line 17) based on the counterexample σ, that is, the subroutine RepairSkF repairs the current function vector $\boldsymbol{f}$ such that σ now gets fixed. Manthan3 returns a function vector $\boldsymbol{f}$ only if $E(X, Y')$ is UNSAT.

Implementation Optimization: Notice that the satisfiability checks at line 11 and 13 of Algorithm 9 can be combined by checking satisfiability of $E(X, Y, Y') : \varphi(X, Y) \wedge \neg\varphi(X, Y') \wedge (Y' \leftrightarrow \boldsymbol{f})$. The formula $E(X, Y, Y')$ (Formula 6.1) to verify the Skolem function vector [JSC+15]. If $E(X, Y, Y')$ is SAT, and σ is a satisfying assignment of $E(X, Y, Y')$, Manthan3 needs to repair the candidate function vector to fix counterexample σ. Otherwise, if $E(X, Y, Y')$ is UNSAT, Manthan3 needs to perform another satisfiability check to ensure that the given DQBF instance is True, that is, a final satisfiability check on $E(X, Y') : \neg\varphi(X, Y') \wedge (Y' \leftrightarrow \boldsymbol{f})$. If $E(X, Y')$ is UNSAT, Manthan3 can return the function vector $\boldsymbol{f}$ as a Henkin function vector, else, the given DQBF instance is False and there does not exists a Henkin function vector.

By definition of Henkin functions, we know that the following lemma holds:

Lemma 8.1. *$\boldsymbol{f}$ is a Henkin function vector if and only if $\neg\varphi(X, Y) \wedge (Y \leftrightarrow \boldsymbol{f})$ is UNSAT.*

Manthan3 returns a function vector only when $E(X, Y') : \neg\varphi(X, Y') \wedge (Y' \leftrightarrow$

$\boldsymbol{f}$) is UNSAT, and each function f_i follows Henkin dependencies by construction. Therefore Manthan3 is sound, and returns a Henkin function vector.

Limitations: There are instances for which Manthan3 might not be able to repair a candidate vector, and consequently is not complete. The limitation is that the formula $G(X, Y)$ (Formula 8.1) is not aware of Henkin dependencies.

Let us consider an example, $\phi : \forall X \exists^{H_1} y_1 \, \exists^{H_2} y_2 \; \varphi(X, Y)$ where $X = \{x_1, x_2, x_3\}$, $Y = \{y_1, y_2\}$, $\varphi(X, Y) := \neg(y_1 \oplus y_2)$, $H_1 = \{x_1, x_2\}$, and $H_2 = \{x_2, x_3\}$. Note that ϕ is True and Henkin functions are $\boldsymbol{f} := \langle f_1(x_1, x_2) : x_2, f_2(x_2, x_3) : x_2\rangle$. Let us assume the candidates learned by Manthan3 is $\boldsymbol{f} := \langle f_1(x_1, x_2) : x_2,\; f_2(x_2, x_3) : \neg x_2\rangle$. $E(X, Y')$ is SAT. Let $\delta \models E(X, Y')$, and $\delta = \langle x_1 \leftrightarrow 0,\; x_2 \leftrightarrow 0,\; x_3 \leftrightarrow 0,\; y'_1 \leftrightarrow 0,\; y'_2 \leftrightarrow 1\rangle$. Furthermore, $\varphi(X, Y) \wedge (X \leftrightarrow \delta[X])$ is also SAT, let $\pi : \langle x_1 \leftrightarrow 0,\; x_2 \leftrightarrow 0,\; x_3 \leftrightarrow 0,\; y_1 \leftrightarrow 1,\; y_2 \leftrightarrow 1\rangle$ be the satisfying assignment. Let the candidate to repair is y_2, and corresponding G_2 formula is $G_2 := \varphi(X, Y) \wedge (x_2 \leftrightarrow 0) \wedge (x_3 \leftrightarrow 0) \wedge (y_2 \leftrightarrow 1)$. As $H_1 \not\subseteq H_2$, the formula G_2 is not allowed to constrain on y_1. G_2 turns out SAT, suggesting that we should try to repair y_1 instead of y_2, but as y_1 is also not allowed to depend on y_2, the formula G_1 would also be SAT. Therefore, Manthan3 is unable to repair candidate $\boldsymbol{f}$ to fix counterexample the σ. Manthan3 would not be able to synthesize Henkin functions for such a case. Hence, Manthan3 is not complete.

We now illustrate Algorithm 9 for Manthan3 through an example.

Example 8.2. Let $X = \{x_1, x_2, x_3\}$, $Y = \{y_1, y_2, y_3\}$ in $\forall X \exists^{H_1} y_1 \, \exists^{H_2} y_2 \exists^{H_3} y_3 \varphi(X, Y)$ where $\varphi(X, Y)$ is $(x_1 \vee y_1) \wedge (y_2 \leftrightarrow (y_1 \vee \neg x_2)) \wedge (y_3 \leftrightarrow (x_2 \vee x_3))$, and $H_1 = \{x_1\}, H_2 = \{x_1, x_2\}$, and $H_3 = \{x_2, x_3\}$.

Data Generation : Manthan3 generates training data through adaptive sampling. Let us assume the sampler generates data as shown in Figure 8.1.

Candidate Learning : As $H_1 \subset H_2$, Manthan3 adds a dependency constraint that y_1 can not depend on y_2. Manthan3 now attempts to learn candidates and calls

x_1	x_2	x_3	y_1	y_2	y_3
0	0	0	1	1	0
0	0	1	1	1	1
1	1	0	0	0	1

Figure 8.1: Henkin synthesis by Manthan3 example: Data generations.

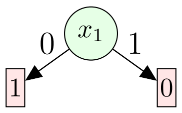


Figure 8.2: Henkin synthesis by Manthan3 example: Learning candidate for y_1 with feature set $\{x_1\}$.

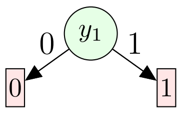


Figure 8.3: Henkin synthesis by Manthan3 example: Learning candidate for y_2 with feature set $\{x_1, x_2, y_1\}$.

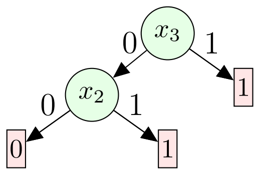


Figure 8.4: Henkin synthesis by Manthan3 example: Learning candidate for y_3 with feature set $\{x_2, x_3\}$.

CandidateSkF for each y_i. As $H_3 \not\subseteq H_1$, y_1 can not depend on y_3, the feature set for y_1 only includes H_1. The decision tree construction uses the samples of $\{x_1\}$ as features and samples of $\{y_1\}$ as labels. The candidate function f_1 is constructed by taking a disjunction over all the paths that end in leaf nodes with label 1. As shown in Figure 8.2, f_1 is $\neg x_1$.

As $H_1 \subset H_2$, the feature set for y_2 includes H_2 and y_1, however it can not include y_3 as $H_3 \not\subseteq H_2$. So, the decision tree construction uses the samples of $\{x_1, x_2, y_1\}$ as features and samples of $\{y_2\}$ as labels. The candidate function f_1 is constructed by taking a disjunction over all paths that end in leaf nodes with label 1: as shown in Figure 8.3, f_2 is synthesized as y_1. Similarity, for y_3, the feature set is H_3, and a decision tree is constructed as shown in Figure 8.4 with samples of x_2, x_3 as features and samples of y_3 as label. We get $f_3 := x_3 \vee (\neg x_3 \wedge x_2)$. At the end of CandidateSkF, we have $f_1 := \neg x_1,\ f_2 := y_1,\ f_3 := x_3 \vee (\neg x_3 \wedge x_2)$. Let us assume the total order returned by FindOrder is $\mathit{TotalOrder} = \{y_3, y_2, y_1\}$.

Verification : We construct $E(X, Y') = \wedge \neg\varphi(X, Y') \wedge (Y' \leftrightarrow \boldsymbol{f})$, which turns out to be SAT, and let $\delta = \langle x_1 \leftrightarrow 1,\ x_2 \leftrightarrow 0,\ x_3 \leftrightarrow 0,\ y_1' \leftrightarrow 0,\ y_2' \leftrightarrow 0,\ y_3' \leftrightarrow 0\rangle$

be a satisfying assignment of $E(X, Y')$. Next, Manthan3 checks if the X valaution can be extended to Y in order to satisfy the specification. It checks satisfiability of $\varphi(X, Y) \wedge (x_1 \leftrightarrow \delta[x_1]) \wedge (x_2 \leftrightarrow \delta[x_2]) \wedge (x_3 \leftrightarrow \delta[x_3])$, which turns out to be SAT, let π be the satisfying assignment, $\pi = \langle x_1 \leftrightarrow 1,\ x_2 \leftrightarrow 0,\ x_3 \leftrightarrow 0,\ y_1 \leftrightarrow 1,\ y_2 \leftrightarrow 1,\ y_3 \leftrightarrow 0\rangle$. Let $\sigma = \langle x_1 \leftrightarrow 1,\ x_2 \leftrightarrow 0,\ x_3 \leftrightarrow 0,\ y_1 \leftrightarrow 1,\ y_2 \leftrightarrow 1,\ y_3 \leftrightarrow 0,\ y'_1 \leftrightarrow 0,\ y'_2 \leftrightarrow 0,\ y'_3 \leftrightarrow 0\rangle$ be a counterexample to fix.

Candidate Repair : In order to find the candidates to repair, MaxSAT solver is called with $\varphi(X, Y) \wedge (x_1 \leftrightarrow \sigma[x_1]) \wedge (x_2 \leftrightarrow \sigma[x_2]) \wedge (x_3 \leftrightarrow \sigma[x_3])$ as hard constraints and $(y_1 \leftrightarrow \sigma[y'_1]) \wedge (y_2 \leftrightarrow \sigma[y'_2]) \wedge (y_3 \leftrightarrow \sigma[y'_3])$ as soft constraints in MaxSATList. Let MaxSATList returns $ind = \{y_2\}$.

Repair synthesis commences for f_2 with a satisfiability check of $G_2 = \varphi(X, Y) \wedge (x_1 \leftrightarrow \sigma[x_1]) \wedge (x_2 \leftrightarrow \sigma[x_2]) \wedge (y_1 \leftrightarrow \sigma[y'_1]) \wedge (y_2 \leftrightarrow \sigma[y'_2])$. Notice, here we can constrain G_2 with y_1 as $H_1 \subset H_2$. The formula is unsatisfiable, and Manthan3 calls FindCore, which returns variable $\neg x_2$, since the constraints $(x_2 \leftrightarrow \sigma[x_2])$ and $(y_2 \leftrightarrow \sigma[y'_2])$ are not jointly satisfiable in G_2. As the output of candidate f_2 for the assignment σ must change from 0 to 1, f_2 is repaired by disjoining with $\neg x_2$, and we get $f_2 := y_1 \vee \neg x_2$ as the new candidate.

The updated candidate vector $\boldsymbol{f} : \langle f_1 := \neg x_1,\ f_2 := y_1 \vee \neg x_2,\ f_3 := x_3 \vee (\neg x_3 \wedge x_2)\ \rangle$ passes the verification check, that is, the formula $E(X, Y')$ is UNSAT. Thus, Manthan3 returns $\boldsymbol{f}$ as a Henkin function vector.

8.2 Experimental Results

It is widely recognized that different techniques exhibit varying performance across different classes of benchmarks, particularly in the context of NP-hard problems. The practical adoption often employs a portfolio approach [DPV21, HPSS18, XHHLB08]. Consequently, when assessing the impact of a new technique, it is common to evaluate its influence on the existing portfolio.

To evaluate the effectiveness of our algorithm on instances that current algorithms struggle with, we focus on the **Virtual Best Synthesizer** (VBS). VBS represents the portfolio of the best-performing algorithms currently available. If at least one tool within the portfolio can successfully synthesize Henkin functions for a given instance, it is considered to be synthesized by VBS. In other words, VBS is at least as capable as each individual tool within the portfolio. The time taken by VBS to synthesize Henkin functions for a specific instance is determined by the minimum time required by any tool in the portfolio to synthesize a function for that instance. Therefore, the objective of the experimental evaluation is to determine whether Manthan3 was able to push the boundaries of VBS by synthesizing Henkin functions for instances that could not be handled by any existing tools.

Benchmarks: We performed an extensive comparison on 563 benchmarks consisting of a union of benchmarks from the DQBF track of QBFEval18, 19, and 20 [qbfc], which encompass equivalence checking problems, synthesis of controller, and succinct DQBF representations of propositional satisfiability problems.

Setup: Manthan3 employs Open-WBO [MML14] for MaxSAT queries, PicoSAT [Bie08] to find UNSAT cores, ABC [LG] to represent and manipulate Boolean functions, CMSGen to generate the required samples [GSCM21], UNIQUE [Sli20] to extract definition for uniquely defined variables, and Scikit-Learn [skl] is used to generate decision trees to learn candidate functions.

All our experiments were conducted on a high-performance computer cluster with each node consisting of a E5-2690 v3 CPU with 24 cores and 96GB of RAM, with a memory limit set to 4GB per core. All tools were run in a single core with a timeout of 7200 seconds for each benchmark.

Tools compared with: We performed a comparison vis-a-vis the prior state-of-the-art techniques, HQS2 [GWR+15] and Pedant [RSS21]. Note that we compared Manthan3 with the tools that can synthesize Henkin functions for True DQBF; the rest all the DQBF solvers, including DepQBF [LB10], DQBDD [Sìč20] do not synthesize such functions. The DQBF preprocessor HQSpre [WRMB17] is invoked implicitly by HQS2. We found that the performance of Pedant degrades with the preprocessor HQSPre; therefore, we consider the results of Pedant without preprocessing. Manthan3 is used without HQSpre.

Results: Figure 8.5 depicts the cactus plot comparing the Virtual Best Synthesizer (VBS) of HQS2 and Pedant against the VBS comprising HQS2, Pedant, and Manthan3. The results show that with the inclusion of Manthan3, the VBS synthesizes functions for 204 instances, whereas the VBS without Manthan3 only manages to synthesize functions for 178 instances. This signifies an improvement of 26 instances in the VBS when utilizing Manthan3. Out of a total of 563 benchmarks, Henkin functions are successfully synthesized for 204 instances by at least one of the three tools.

Furthermore, Manthan3 demonstrates the fastest synthesis time on 42 benchmarks, which includes 26 instances where none of the other tools were able to synthesize Henkin functions. This highlights the effectiveness and efficiency of Manthan3 in tackling challenging synthesis tasks.

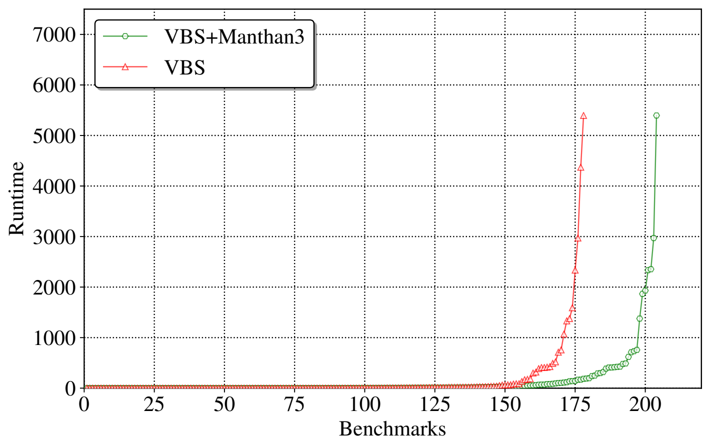


Figure 8.5: Henkin synthesis: Virtual Best Synthesizing Henkin functions with/without Manthan3. VBS in the plot represents VBS of HQS2 and Pedant.

In Figure 8.6, VBS with HQS2 and Pedant is presented on x-axis and Manthan3 is presented on y-axis. Figure 8.6 highlights that the performance of Manthan3 is orthogonal to existing tools. Furthermore, as shown in green area of Figure 8.6, for

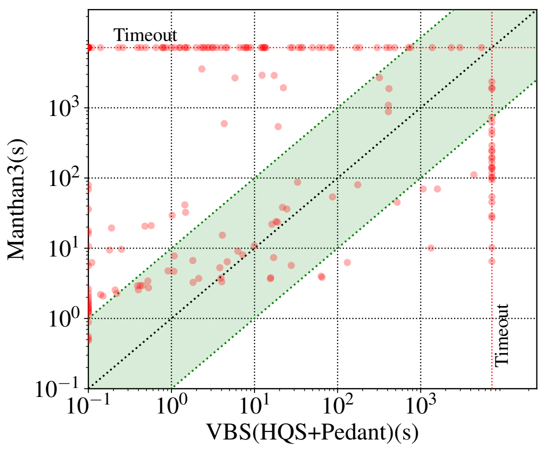


Figure 8.6: Henkin synthesis: Manthan3 vs. VBS(HQS2+Pedant).

47 instances Manthan3 took less than or equal to additional 10 seconds to synthesize Henkin functions than by the VBS with HQS2 and Pedant.

Table 8.1: Henkin synthesis: Manthan3 vs other state-of-the-art tools.

	HQS2	Pedant	Manthan3
Synthesized Functions	148	138	116
Manthan3(more)	40	37	-

We present a detailed pairwise comparison in Table 8.1. The first row of Table 8.1 presents the number of instances for which HQS2, Pedant and Manthan3 could synthesize a Henkin functions. And, the second row of Table 8.1 presents the number of instance for which a function vector was synthesized by Manthan3, but not by HQS2 and Pedant in Column 2 and 3 respectively.

Figure 8.7 (resp. Figure 8.8) represents scatter plot for Manthan3 vis-a-vis with HQS2 (resp. Pedant). The distribution of the instances for which functions are synthesized shows that all three tools are incomparable. There are many instances where only one of these tools succeeds, and others fail. In total there are 148, 138 and 116 instances for which HQS2, Pedant and Manthan3 could synthesize Henkin functions respectively. Moreover, there are 40 instances for which Manthan3 could synthesize Henkin functions, whereas HQS2 could not produce a Henkin function vector. Similarly, there are 37 instances for which Manthan3 was able to come up with a Henkin function vector and Pedant could not synthesize Henkin functions.

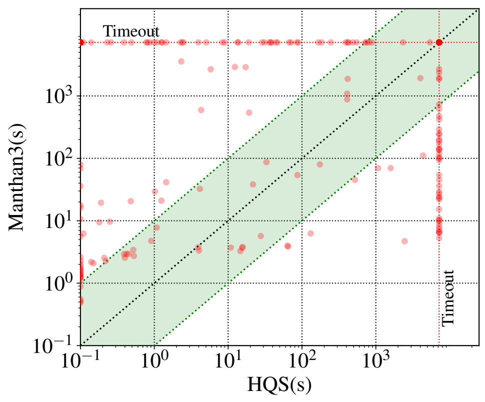


Figure 8.7: Henkin synthesis: Manthan3 vs.HQS2.

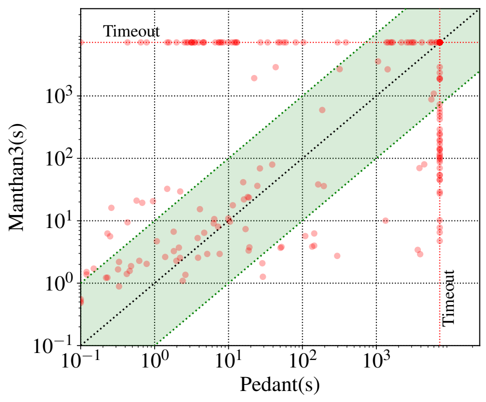


Figure 8.8: Henkin synthesis: Manthan3 vs. Pedant.

A point $\langle x, y \rangle$ implies that the synthesizer on $\langle x \rangle$ axis took x seconds while the synthesizer on $\langle y \rangle$ axis tooks y seconds to synthesize Henkin functions for an instance.

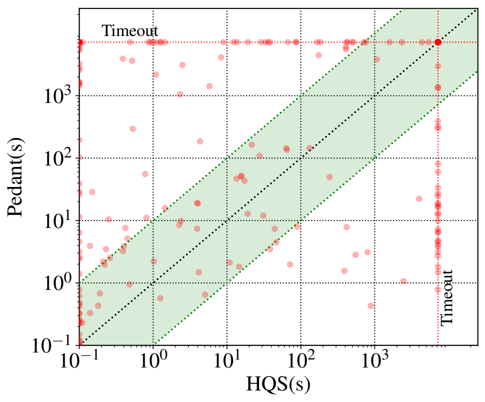


Figure 8.9: Henkin synthesis: Pedant vs.HQS2.

Figure 8.9 shows that there is no best tool even amongst the existing tools, Pedant and HQS2. Although both tools could synthesize functions for (almost) the same number of instances, the instances belong to different classes.

The results show that different approaches are suited for different classes of benchmarks, and Manthan3 [GRM23] pushes the envelope in Henkin synthesis by handling instances for which none of the state-of-the-art tools could synthesize Henkin functions.

Part IV

Generalization from Functional Synthesis

This Part is dedicated to exploring the generalization of functional synthesis in different settings. In Chapter 9, we establish a crucial link between functional synthesis and program synthesis. We demonstrate how program synthesis can be effectively achieved by reducing it to dependency quantified formulas modulo an underlying theory, thus enabling program synthesis through functional synthesis.

Furthermore, in Chapter 10, we introduce a comprehensive approach that combines machine learning and formal methods. This approach is designed to address real-world scenarios where constraints do not have equal priority. In these practical settings, the system must satisfy certain constraints, while also attempting to satisfy other constraints as much as possible. By integrating machine learning and formal methods, our approach enables the effective handling of diverse constraints, striking a balance between the different priorities imposed by the constraints.

Chapter 9

Program Synthesis as Dependency Quantified Formulas

A crucial ingredient in the *NP revolution* was the reduction of key problems such as planning [KS92] and bounded model checking [BCCZ99] to SAT. Such reductions served as a rich source of practical instances, and at the same time, planning and bounded model checking tools built on top of SAT achieved fruits of the progress in SAT solving and thereby leading to even wider adoption, and contributing to a virtuous cycle [MSLM09].

Our investigation aims to explore various applications of functional synthesis. Specifically, we focus on program synthesis, a key problem in programming languages, and examine its relationship with DQF($\mathbb{T}$), which refers to the problem of finding a witness of a Dependency Quantified Formula Modulo Theory. In program synthesis, given a specification $\varphi(X, Y)$ that defines the desired behavior of a program in terms of inputs (X) and outputs (Y), the goal is to synthesize a program that satisfies the specification.

The earliest work on program synthesis can be traced back to Church [Kol32], but the computational complexity of the problem posed challenges in developing practical techniques. A significant breakthrough came with the introduction of

Syntax-Guided Synthesis (SyGuS) [ABJ+13b]. In the SyGuS formulation, in addition to the specification φ, the input also includes a grammar that defines the set of allowed implementations of the program f. The grammar serves to restrict the space of possible implementations, enabling the development of efficient techniques for enumerating over the grammar, primarily to assist the underlying solver by constraining the search space [ABJ+13b, ARU17, BCD+11].

Recent studies have emphasized that for a wide range of applications, grammars are used solely to enhance solver efficiency. Therefore, there have been suggestions to use more expressive grammars for a given SyGuS instance [PMNS19]. In many cases, end users are primarily interested in finding any function that can be expressed using a specific theory $\mathbb{T}$. To clarify, we refer to this class of program synthesis problems as "$\mathbb{T}$-constrained" synthesis. It is worth noting two observations: First, $\mathbb{T}$-constrained synthesis is a subclass of SyGuS, meaning that every instance of $\mathbb{T}$-constrained synthesis is also an instance of SyGuS. Notably, recent work has focused on developing specialized algorithms specifically tailored for $\mathbb{T}$-constrained synthesis, such as the counterexample-guided quantifier instantiation algorithm in [RDK+15].

This thesis establishes a connection between Theory-constrained synthesis and DQF($\mathbb{T}$) [GRM21]. In particular following:

From $\mathbb{T}$-constrained synthesis to DQF($\mathbb{T}$). We present an efficient reduction of $\mathbb{T}$-constrained synthesis to DQF($\mathbb{T}$). DQF($\mathbb{T}$) lifts the notion of DQBF from the Boolean domain to general Theory $\mathbb{T}$. We view the simplicity of the reduction from $\mathbb{T}$-constrained synthesis to DQF($\mathbb{T}$) as a core strength of the proposed approach.

Efficient $\mathbb{T}$-constrained synthesizers for $\mathbb{T}$=bitvectors. The reduction to DQF($\mathbb{T}$) opens up new directions for further work. As a first step, we focus on the case when the $\mathbb{T}$ is restricted to bitvector theory, denoted by BV. We observe that the resulting DQF(BV) instances can be equivalently specified as a DQBF instance. We demonstrate that our reduction to DQBF allows us to simply plug-in the state of the art DQBF solvers [FKBV14, GWR+15, RT15, Sìč20].

The Power of DQBF. The remarkable progress in DQBF over the past few years is evident in our observation that DQBF-based synthesizers perform significantly better than domain-specific techniques that focus on utilization of the grammar to achieve efficiency. Our investigations were motivated by the classical work of Kautz and Selman [KS92] that showcased the power of general purpose SAT solvers in achieving advances over domain specific techniques. In this spirit, our work showcases the power of reducing synthesis to DQF($\mathbb{T}$). One question that we did not address so far is the choice of DQF($\mathbb{T}$) over $\mathbb{T}$-constrained synthesis. To this end, we again focused on the case when $\mathbb{T}$=BV, and we transform DQBF benchmarks to program synthesis, and perform a comparison of program synthesis techniques vis-a-vis DQBF techniques. We observe that DQBF techniques significantly outperform the program synthesis techniques for these benchmarks; these results highlight the versatility of DQBF techniques and provide evidence in support of our choice of DQBF as the representation language.

Role of Grammars. Since DQBF-based synthesis techniques perform better than techniques that rely on grammar for efficiency, we would like to put forth the thesis of the usage of grammar as a specification tool rather than to guide the search strategy of the underlying synthesis engines, i.e., evolution of syntax-guided synthesis to syntax-constrained synthesis.

9.1 Reduction of $\mathbb{T}$-Constrained Synthesis to DQF($\mathbb{T}$)

We propose the reduction of $\mathbb{T}$-constrained synthesis to DQF($\mathbb{T}$), i.e, to the problem of finding a witness of a dependency quantified formula modulo theory. A key strength of the reduction is its simplicity.

Before discussing the reduction in detail, let us establish some notations and vocubarly needed for the reductions. Let us first discuss the well-known formulation of syntax-guided synthesis (SyGuS). In this formulation, the constraints over the functions to be synthesized are specified in the vocabulary of a given background theory $\mathbb{T}$ along with the function symbols. Notice that the background theory specifies the domain of values for each variable type along with the interpretation for the function(s) and predicate symbols in the vocabulary.

Definition 9.1 ([ABJ+13b]). *Given a background theory $\mathbb{T}$, a set of typed function symbols $\{f_1, f_2, \dots f_k\}$, a specification φ over the vocabulary of $\mathbb{T} \cup \{f_1, f_2, \dots f_k\}$, a set of expressions $\{L_1, \dots L_k\}$ over the vocabulary of $\mathbb{T}$ such that L_i is of the same type as f_i, the problem of syntax-guided synthesis (SyGuS) is to find a set of expressions $\{e_1 \in L_1, e_2 \in L_2, \dots e_k \in L_k\}$ such that the formula $\varphi[f_1/e_1, f_2/e_2, \dots f_k/e_k]$ is valid modulo $\mathbb{T}$. Note that $\varphi[f_i/e_i]$ denotes the result of substitution of f_i with expression e_i such that the bindings of inputs to f_i is ensured.*

If every invocation of f in φ has an identical argument-list, then $|\mathsf{CallSigns}(f)| = 1$, and we refer to φ as a *single-callsign* instance. Otherwise, φ is a *multiple-callsign* instance. We use $args(f)$ to get the argument lists of function f, and $f(args)$ to represents the invocation of f with its *args*.

We are interested in the subclass of SyGuS where L_i corresponds to the *complete* vocabulary of $\mathbb{T}$; we call such a class as $\mathbb{T}$-constrained synthesis discussed in Chapter 2, again defined formally below:

Definition 9.2. *Given a background theory $\mathbb{T}$, a set of typed function symbols $\{f_1, f_2, \dots f_k\}$, a specification φ over the vocabulary of $\mathbb{T} \cup \{f_1, f_2, \dots f_k\}$, the problem of $\mathbb{T}$-constrained synthesis is to find the set of expressions $\{e_1, e_2, \dots e_k\}$ defined over vocabulary of $\mathbb{T}$ such that the formula $\varphi[f_1/e_1, f_2/e_2, \dots f_k/e_k]$ is valid modulo $\mathbb{T}$.*

Algorithm 10 formalizes the desired reduction of φ to DQF($\mathbb{T}$) formulation where φ is a specification over the vocabulary of background theory $\mathbb{T}$ with a set of typed function symbols $\{f_1, f_2, \dots f_m\}$ such that for all f_i, $|\mathsf{CallSigns}(f_i)| = 1$. The important point to note is that the Henkin quantifiers must be carefully constructed so that each f_i depends only on the set of variables that appear in its argument-list.

Now, let us turn our attention to the case when there exist f_i such that $|\mathsf{CallSigns}(f_i)| > 1$. In such cases, we pursue a Ackermannization-style technique that transforms φ

Algorithm 10 Reducing a single-callsign instance φ to DQF($\mathbb{T}$)

Input A background theory $\mathbb{T}$, a set of typed function symbols $\{f_1, f_2, \dots f_m\}$, a specification φ over the vocabulary of $\mathbb{T}$

Output $\forall X \exists^{H_1} y_1.\ \exists^{H_2} y_2\ \dots \exists^{H_m} y_m \varphi(X, Y)$

```
1: Let $X = \bigcup_{f_i} \{h \mid h \in \mathsf{CallSigns}(f_i)\}$
2: Substitute every invocation of $f_i$ with a fresh variable $y_i$ in $\varphi$
3: Define $H_i = Set(h)$ as $\{h | h \in \mathsf{CallSigns}(f_i)\}$
```

into another specification $\hat{\varphi}$ such that every function f_i in $\hat{\varphi}$ has $|\mathsf{CallSigns}(f_i)| = 1$ (Algorithm 11). Note that this transformation allows the subsequent use of Algorithm 10 with $\hat{\varphi}$ to complete the reduction to DQF($\mathbb{T}$). The proposed transformations in Algorithm 11 are linear in the size of the formula like the transformation introduced in [Rab17], however Algorithm 11 introduces lesser number of new variables.

Algorithm 11 Reducing multiple-callsign to single-callsign instance

Input A background theory $\mathbb{T}$, a set of typed function symbols $\{f_1, f_2, \dots f_m\}$, a specification φ over the vocabulary of $\mathbb{T}$ such that $\ell_i = |\mathsf{CallSigns}(f_i)|$

Output A set of typed function symbols $\{f_1^0, f_1^2, \dots f_1^{\ell_1}, \dots, f_m^0 \dots f_m^{\ell_m}\}$, a specification $\hat{\varphi}$ over the vocabulary of $\mathbb{T}$ such that $\forall i, j$ we have $|\mathsf{CallSigns}(f_i^j)| = 1$.

```
1: for $i = 1$ to $m$ do
2:     if $|\mathsf{CallSigns}(f_i)| > 1$ then
3:         Add a fresh (ordered) set of variables $Z_i$ such that $|Z_i| = |\mathsf{CallSigns}(f_i)[0]|$
4:         for $j \in [0 \dots (\ell_i - 1)]$ do
5:             Replace every $f_i$ whose $args(f_i) = \mathsf{CallSigns}(f_i)[j]$ with $f_i^j$
6:             Add constraint $(args(f_i^j) = Z_i) \rightarrow f_i^j(args) = f_i^{\ell_i}(Z_i)$ to $\varphi$
7:         $\mathsf{CallSigns}(f_i) \leftarrow \mathsf{CallSigns}(f_i) \cup \{Z_i\}$
```

The essence of Algorithm 11 is captured in the following two transformations:

(Line 5). We substitute instances of every call signature of f_i with fresh function symbols f_i^j (that corresponds to the j^{th} call signature of f_i). This reduces the formula from multiple-callsign to a single-callsign instance.

(Line 6). Introduction of an additional constraint for each f_i that forces all the functions f_i^j (introduced above) to mutually agree on every possible instantiation of arguments. Specifically, it introduces a fresh function symbol $f_i^{l_i}$ and a set of fresh variables $z_1^i, \dots, z_n^i \in Z_i$ such that, for all argument lists $args(f_i^j)$, we have

$(args(f_i^j) = Z_i) \implies f_i^j(args) = f_i^{\ell_i}(Z_i)$ where $j \in [0 \ldots l_{i-1}]$.

9.1.1 When $\mathbb{T}$ is Bitvector (BV):

When the specification $\varphi(X, Y)$ is in BV, we solve the problem in the following steps:

1. *Reduction to single-instance:* If $\varphi(X, Y)$ is a multiple-callsign instance, we use Algorithm 11 to covert it to a single-callsign instance $\hat{\varphi}(\hat{X}, \hat{Y})$.
2. *Reduction to DQF(BV):* We use Algorithm 10 to generate the DQF(BV) instance of $\hat{\varphi}(\hat{X}, \hat{Y})$ as $\forall \hat{X} \exists^{H_1} \hat{y}_1. \exists^{H_2} \hat{y}_2 \ \ldots \exists^{H_m} \hat{y}_m \hat{\varphi}(\hat{X}, \hat{Y})$.
3. *Solving DQF(BV):* We solve the DQF(BV) instance by compiling it down to a DQBF instance, thereby allowing the use of off-the-shelf DQBF solvers. We detail this step in the following discussion.

For DQBF compilation, we perform *bit-blasting* over $\hat{\varphi}$ to obtain $\hat{\varphi}'$.

$$\forall \hat{X} \exists^{H_1} \hat{y}_1 \quad \ldots \exists^{H_m} \hat{y}_m \hat{\varphi}(\hat{X}, \hat{Y}) \quad \equiv \quad \forall X' \exists^{X'} V. \ \ \exists^{H_1'} Y_1' \ldots \exists^{H_m'} Y_m' \varphi'(X', Y') \quad (9.1)$$

where X', Y_i', H_i' are the (bit-blasted) *sets* of propositional variables mapping to the bitvector variables $\hat{X}, \hat{y}_i, H_i$ respectively. Furthermore, V is the set of auxiliary variables introduced during bit-blasting, which are allowed to depend on all the input variables X'. From an efficiency perspective, one can record the corresponding Henkin functions for the auxiliary variables during bit-blasting; we leave such optimizations to future work. We employ off-the-shelf SMT solvers for bit-blasting.

As the formula on the right-hand side in Eq. 9.1 is an instance of DQBF, we can simply invoke an off-the-shelf *certifying DQBF* solvers to generate the Henkin functions for Y'. A careful reader will observe that the Henkin functions corresponding to y_i' variables will be constructed in the propositional theory and not in BV, but note that one can simply convert a formula in propositional theory into an equivalent formula in bitvector theory defined over $\hat{X}, \hat{y}_1, \ldots \hat{y}_m$ with only a linear size increase in the representation.

9.2 Experimental Results

The primary objective of empirical evaluation is to show how general purpose DQBF techniques can be transformed to effective synthesis engines. In particular, our empirical evaluation sought answers to the following questions:

1. How does theory constrained synthesis compare with syntax-guided synthesis?
2. Can general purpose DQBF solver-based synthesis framework match the efficiency of domain specific synthesis tools?
3. Are BV-constrained synthesis an efficient approach to DQBF solving?

To this end, we perform an evaluation over an extensive suite of benchmarks and tools, which we describe in detail below.

Tools under evaluation: Given a program synthesis instance defined over BV, we sought to compare three different possibilities: executing a SyGuS tool, executing a BV-constrained synthesis, and a DQBF-augmented synthesis framework. To this end, we experimented with the following tools (Table 9.1):

Syntax-guided synthesis: SyGuS tools for bitvector theory spanning symbolic, stochastic and enumerative solvers, like CVC4 [BCD+11], EUSolver [ARU17], ESolver [URD+13], DryadSynth [HQSW20], Stochpp [ABJ+13b], Symbolic [ABJ+13b]. Since, we employ multiple versions of CVC4, we will refer to the SyGuS-based variant of CVC4 as CVC4$_{enum}$.

BV-constrained synthesis: It has been observed that any SyGuS tool can be converted into a theory-constrained tool by a straightforward pipeline that modifies the grammar of the input instance to encompass the entire vocabulary of the corresponding theory. However, our empirical evaluation has shown that this transformation is ineffective, as the tools undergoing this process consistently perform worse compared to their counterparts in syntax-guided mode. Apart from the SyGuS-based BV-constrained tools, we also incorporate the cutting-edge synthesis engine CVC4 [RDK+15] in the counter-example-guided quantifier instantiation mode, which can be considered a native approach to BV-constrained synthesis. We use the term CVC4$_{cex}$ to refer to CVC4 invoked with counterexample-guided quantifier instantiation.

DQBF-based synthesis: The set of underlying DQBF solvers that we have employed in DQBF-based synthesis framework span CADET [Rab19], Manthan [GRM20, GSRM21], DepQBF [LB10], DCAQE [TR19] and DQBDD [Sič20]. A careful reader might observe that CADET and Manthan are 2-QBF solvers, i.e., they can only handle the special case when the existentially quantified variables are allowed to depend on all the universally quantified variables.

Table 9.1: Program synthesis via functional synthesis: Tools used in evaluation.

Syntax-Guided	BV-Constrained	DQBF-based
CVC4$_{enum}$, ESolver EUSolver, DryadSynth, Stochpp	CVC4$_{cex}$	CADET, DCAQE, Manthan, DepQBF, DQBDD

Benchmarks: The benchmark suite consisted of instances from two sources: SyGuS competition and QBF competition. We use 645 general-track bitvector (BV) theory benchmarks from SyGuS competition 2018, 2019 [1] to evaluate the performance of SyGuS tools. To employ our DQBF-based framework, we used Z3 [DMB08] to convert the instances from SyGuS to DQBF. Furthermore, we considered 609 QBF benchmarks from QBFEval competition 17,18[2], disjunctive decomposition and arithmetic set [AAC+19, Rab19] and converted them to SyGuS instances. We considered

[1]https://sygus.org/
[2]http://www.qbflib.org/index_eval.php

each propositional variable as a bitvector of size 1, and allow the synthesized function to depend on all the universally quantified variables. The associated grammar for these benchmarks is the entire BV-vocabulary.

Implementation and Setup: Our reduction is implemented in a tool, DeQuS. All our experiments were conducted on a high-performance computer cluster with each node consisting of a E5-2690 v3 CPU with 24 cores and 96GB of RAM, with a memory limit set to 4GB per core. All tools were run in a single-threaded mode on a single core with a timeout of 900s.

Results: Table 9.2 represents the instances solved by the virtual best solver for SyGuS, BV constrained, and DQBF tools. The first row of Table 9.2 lists the number of instances solved by different synthesis techniques: SyGuS based, BV-constrained synthesis, and DQBF based synthesis for SyGuS instances, and the second row represents the same for DQBF instances.

Table 9.2: Program synthesis via functional synthesis: Number of SyGuS and DQBF instances solved using different synthesis techniques. Timeout 900s.

	Total	SyGuS-tools	BV-constrained	DQBF-based
SyGuS	645	513	606	610
DQBF	609	-	2	276
All	1254	513	608	886

As shown in Table 9.2, with syntax guided synthesis, we could synthesize the functions for 513 out of 645 SyGuS instances only, whereas, with BV-constrained synthesis, we could solve 606 such instances. Surprisingly, BV-constrained synthesis performs better than the syntax-guided synthesis.

Table 9.2 also shows that the DQBF based synthesis tools perform similar to BV-constrained synthesis tools for SyGuS instances; this provides strong evidence that the general purpose DQBF solvers can match the efficiency of the domain specific synthesis tools. Furthermore, BV-constrained synthesis tools perform poorly with DQBF as representation language providing support for the efficacy of DQBF as a representation language.

Performance analysis for SyGuS instances We used SyGuS instances to evaluate the performance with different synthesis strategies: DQBF based synthesis, BV-constrained synthesis and syntax-guided synthesis. We further divide the SyGuS instances into four sub-categories: single-function-single-callsign, single-function-multiple-callsign, multiple-function-single-callsign, multiple-function-multiple-callsign.

Since the single-function-single-callsign instances can be converted into QBF instances (instead of DQBF), we employ the state-of-the-art QBF solvers CADET [Rab19] and DepQBF [LE17], Manthan [GRM20] for these instances.

Table 9.3: Program synthesis via functional synthesis: The top three tool for each category are listed in the order of their performance, and the number (in bracket) represents the number of instances solved.

	Single-CallSign	Multiple-CallSign
Single-Function	Total instances: 609 **CADET(605)**, CVC4$_{cex}$(602), Manthan(592)	Total instances: 9 **CVC4$_{enum}$(8)**, DQBDD(2), DCAQE(2)
Multiple-Functions	Total instances: 7 **CVC4$_{enum}$(5)**, CVC4$_{cex}$(4), DQBDD(3)	Total instances: 19 **CVC4$_{enum}$(12),** DQBDD(0), DCAQE(0)

With respect to utility of syntax for efficiency, it turns-out that CVC4 performs better with BV-constrained synthesis than with syntax-guided synthesis, as CVC4$_{cex}$ could solve 602 instances whereas CVC4$_{enum}$ could solve only 488. However, CVC4$_{enum}$ outperforms the other state-of-the-art SyGuS tools significantly. The second best SyGuS tool was EUSolver which could solve only 236 instances.

Table 9.3 represents the overall analysis for all four categories of SyGuS instances. As shown in Table 9.3, DQBF solvers and CVC4$_{cex}$ have similar performance in terms of number of instances solved. Therefore, concerning domain specific vs DQBF based synthesis, DQBF solvers perform on par to domain specific synthesis tools, in-fact they perform slightly better for single-invocation-single-callsign category as CADET could synthesize a function for 3 more instances than CVC4$_{cex}$.

Performance analysis for 2-QBF instances We performed an experiment with the SyGuS language as a representation language instead of QBF. We considered QBF instances instead of DQBF as majority of our synthesis benchmarks could reduced to QBF. As CVC4$_{cex}$ performed the best amongst the different tools for BV-constrained synthesis for SyGuS instances, we considered the CVC4$_{cex}$ to evaluate the performance over 609 SyGuS language representation of QBF instances. Table 9.4 represents the performance analysis. We performed experiments with two settings: the same timeout (900s) as used for the tools in Chapter 9.2, and a more relaxed timeout of 7200s. With the 900s timeout, CVC4$_{cex}$ could solve only 2 instances out of 609 total instances, whereas, Manthan preformed the best by synthesizing the functions for 276 instances. With the 7200s timeout, CVC4$_{cex}$ could not solve any new instances while Manthan solved another 80 instances. Hence, BV-constrained synthesis is not an efficient approach for DQBF solving, which answers question related to efficacy of DQBF as representation language.

Table 9.4: Program synthesis via functional synthesis: Instances solved for BV-constrainted synthesis of QBF benchmarks. Total Benchmarks:609.

TO	CVC4$_{cex}$	DepQBF	CADET	Manthan
900s	2	33	274	276
7200s	2	39	280	356

Chapter 10

Satisficing Synthesis

Most approaches for synthesis work on *all-or-nothing* paradigm — that is, they attempt to synthesize a system that must satisfy *all* the constraints in the given underlying specification, and if the synthesis engine could not come up with such a system within the given resources such as memory or time, they would fail and not return a system.

Recently, Peleg and Polikarpova [PP20] conjectured that all-or-nothing paradigm is a critical reason for not achieving mainstream software development, and we should let go of this conventional wisdom to handle real-world instances. Furthermore, they proposed an idea for *best-effort* synthesis to synthesize systems that satisfy at least a portion of given underlying constraints. Our work takes a further step in this direction and considers real-world scenarios in which a set of constraints requires strict guarantees, whereas the remaining constraints could be considered desirable behaviors, which could be violated if needed.

In a real-world setting, not all constraints have equal priorities. Some constraints might be of higher priority and require strict guarantees than the other constraints. For example, the synthesized controller of an autonomous vehicle must ensure that the vehicle does not hit a bystander; arguably, one would consider such a constraint a high-priority constraint. On the other hand, while we would ideally prefer the vehicle not to have a sudden change in the speed and lane changes, such constraints are desirable behaviors. Such scenarios also arise in other contexts: consider a financial institution that needs to decide whom to give a bank loan. On one hand, the institute must follow mandatory regulations such as non-discrimination based on protected attributes. Furthermore, the financial models can inform the likelihood of the loan being defaulted and therefore, an ideal system should agree as much as possible with such financial models.

This provides motivation for a general notion wherein the end user has different classes of constraints: *hard constraints* that must not be violated while satisfying *soft constraints* as much as possible. In general, one would expect to have a quantifiable measure that captures the degree to which the system satisfies soft constraints. A synthesis engine would be required to ensure that the system is able to meet satisfying measures over soft constraints to a given threshold. We define the notion of SatisficingSynth to capture the domain-agnostic synthesis with hard and soft constraints as follows: Given a set of hard constraints, soft constraints, satisficing measure, and threshold, synthesize a system such that:

- it provably satisfies the hard constraints,
- and achieves at least threshold level of satisficing measure over soft constraints.

As is evident, it would be desirable to allow different modalities for the specification of hard and soft constraints: we allow them to be specified via data and logical formulas.

A natural question would be how to design an algorithmic technique for SatisficingSynth. To this end, we build on recent progress in guess-check-repair paradigm in functional synthesis [ACG$^+$18, GRM20, GSRM21], and design a algorithmic framework, called HSsynth, which can handle soft and hard constraints in different modalities. HSsynth relies on advances in automated reasoning and formal methods for a proof-guided repair and provable verification.

10.1 Overview

We provide a high-level overview of a framework to mitigate the problem of SatisficingSynth. SatisficingSynth deals with a satisficing measure, which is defined as $SM()$ that takes two inputs, (i) a specification (say, $\psi(X, Y)$), and (ii) a function (say, $G(X)$) to compute the following:

$$SM(\psi(X, Y), G(X)) := \frac{ModelCount((\psi(X, Y) \land (Y \leftrightarrow G(X)))_{\downarrow X})}{ModelCount(\psi(X, Y)_{\downarrow X})} \tag{10.1}$$

Now, let us define the problem under consideration, SatisficingSynth, formally as follows:

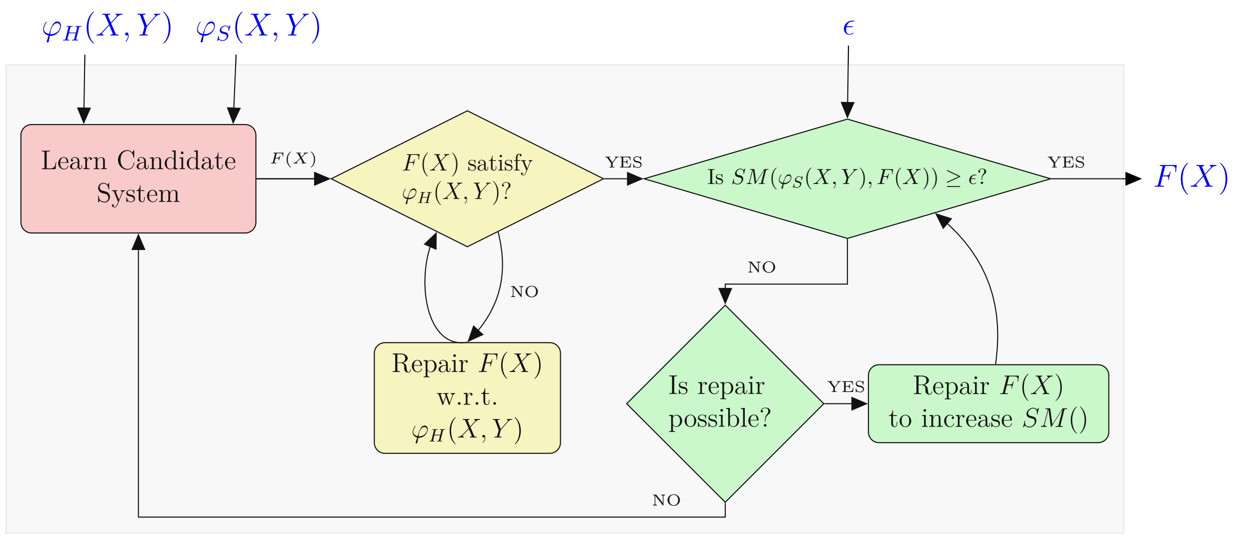
Given (i) hard constraints $\varphi_H(X, Y)$, (ii) soft constraints $\varphi_S(X, Y)$, and (iii) a satisficing threshold ε, where X is a set of inputs and Y is a set of outputs, the objective is to synthesize a system $F(X)$ such that following holds:

- $\forall X(\exists Y \varphi_H(X, Y) \leftrightarrow \varphi_H(X, F(X)))$,
- $SM(\varphi_S(X, Y), F(X)) \geq \varepsilon$.

The satisficing measure $SM()$ essentially captures the number of input valuations for which the soft constraints $\varphi_S(X, Y)$ are satisfied by $F(X)$, which could be computed as per Equation 10.1.

There are many different scenarios can be encoded in our problem setting. Now, the question is *"how do we develop a system that must satisfy the given hard constraints and achieves satisficing measure more than a certain threshold?"*. Towards this, we propose a framework called HSsynth to synthesize such systems. The overview of HSsynth is shown in the Figure 10.1.

As shown in Figure 10.1, HSsynth first learns a candidate system $F(X)$ with $\varphi_S(X, Y)$ and $\varphi_H(X, Y)$. HSsynth then needs to check whether the candidate system satisfies the hard constraints $\varphi_H(X, Y)$ or not. If it does not satisfy the hard constraints, HSsynth needs to repair the candidate system. Otherwise, HSsynth proceeds with checking whether satisficing measure $SM()$ is greater than the threshold



$\varphi_H(X,Y)$
$\varphi_S(X,Y)$
ϵ
Learn Candidate System
$F(X)$
$F(X)$ satisfy $\varphi_H(X,Y)$?
YES
Is $SM(\varphi_S(X,Y), F(X)) \geq \epsilon$?
YES
$F(X)$
NO
Repair $F(X)$ w.r.t. $\varphi_H(X,Y)$
NO
Is repair possible?
YES
Repair $F(X)$ to increase $SM()$
NO

Figure 10.1: HSsynth for satisficing synthesis:

ε or not. If it is greater than ε, then HSsynth returns $F(X)$. Otherwise, we need to check if a repair exists that does not violate the hard constraints and could increase $SM()$; if yes, the candidate system $F(X)$ needs to undergo repair, else HSsynth attempts to learn a different candidate system.

We now provide a high-level description of different components of HSsynth to highlight the technical challenges.

Learn Candidate $F(X)$: As discussed earlier, we can use advances in constrained sampling and machine learning to learn the initial guess of system. Moreover, we can use different underlying synthesizers to produce the candidate system $F(X)$. Considering the setting in which $\varphi_S(X,Y)$ is given as data, we can learn a candidate system using a decision-tree-based classifier [GRM20, GRM21]. Similarly, when $\varphi_S(X,Y)$ is given as formulas, we can first generate the data as the satisfying assignments of relation $\varphi(X,Y)$, where $\varphi(X,Y)$ is the union of $\varphi_S(X,Y)$ and $\varphi_H(X,Y)$ to use the similar decision-tree based approach to learn a candidate system [GRM20]. We can also use different approximate synthesis engines to come up with an initial $F(X)$ [SAC+20].

Does $F(X)$ satisfy $\varphi_H(X,Y)$?: Once we have a candidate system $F(X)$, the task is now to verify whether $F(X)$ satisfies given the hard constraints or not. If candidate system $F(X)$ satisfies the hard constraints, $\varphi_H(X, F(X))$ must be True wherever $\varphi_H(X, Y)$ is True. To verify, HSsynth makes a SAT call on the following:

$$V(X, Y, Y') := \varphi_H(X, Y) \wedge \neg\varphi_H(X, Y') \wedge (Y' \leftrightarrow F(X)) \tag{10.2}$$

The second and third terms in $V(X, Y, Y')$ aim to determine a valuation of X where the resulting output of the candidate system fails to meet the given constraints. Additionally, the first term guarantees the existence of a valuation of Y that satisfies the hard constraints for the valuation of X. Hence, if $V(X, Y, Y')$ is UNSAT, it indicates that $F(X)$ complies with the hard constraints. On the other hand, if $V(X, Y, Y')$ is SAT, we have identified a counterexample that requires fixing.

Repair $F(X)$ with respect to $\varphi_H(X, Y)$: Let $\sigma \models V(X, Y, Y')$ be the counterexample corresponding to which HSsynth needs to repair $F(X)$. We would like to know the *reason* behind candidate system $F(X)$ not satisfying hard constraints with input valuations $\sigma[X]$. Towards this, HSsynth applies a proof-guided strategy to repair $F(X)$. The unsatisfiability core of formula $M(X, Y)$ is employed to capture the reason.

$$M(X, Y) := \varphi_H(X, Y) \wedge (Y \leftrightarrow F(X)) \wedge (X \leftrightarrow \sigma[X]) \tag{10.3}$$

HSsynth uses unit clauses occurring in the UnsatCore of formula $M(X, Y)$ that corresponds to input literals to either strengthen or weaken the system $F(X)$.

Is $SM(\varphi_S(X, Y), F(X)) \geq \varepsilon$?: HSsynth now computes the satisficing measure $SM(\varphi_S(X, Y), F(X))$ of candidate system $F(X)$ using Equation 10.1. HSsynth checks if $SM()$ is lower bounded by given threshold ε or not.

Is repair possible?: If $SM(\varphi_S(X, Y), F(X))$ is less than ε, $F(X)$ needs to be repaired to satisfy more soft constraints in order to improve the satisficing measure. We need a valuation of inputs X such that $F(X)$ satisfies the hard constraints, but do not satisfy $\varphi_S(X, Y)$. Furthermore, we need to ensure that for the corresponding valuation of X, there exists a valuation for Y that satisfy soft constraints. Therefore, HSsynth makes an SAT call on the following:

$$U(X, Y, Y') := \varphi_S(X, Y) \wedge \neg\varphi_S(X, Y') \wedge \varphi_H(X, Y') \wedge (Y' \leftrightarrow F(X)) \tag{10.4}$$

If $U(X, Y, Y')$ is turnout to be UNSAT, there does not exists a valuation of X to repair $F(X)$ without falsifying the hard constraints. In the case $U(X, Y, Y')$ is SAT, HSsynth has a counterexample to repair $F(X)$ in attempt to increase the satisficing measure.

Repair $F(X)$ to increase $SM()$: HSsynth has a counter example π, where $\pi \models U(X, Y, Y')$. HSsynth enumerates different UnsatCores of formula $N(X, Y)$ in order to attempt a proof-guided repair.

$$N(X, Y) := \varphi_S(X, Y) \wedge \varphi_H(X, Y) \wedge (Y \leftrightarrow F(X)) \wedge (X \leftrightarrow \pi[X]) \tag{10.5}$$

HSsynth has different prospects for repair corresponding to each different UnsatCores of $N(X, Y)$. First, we must ensure that the proposed repair does not violate hard constraints. Towards this, HSsynth makes a satisfiability call on Formula 10.2 with each repair option. HSsynth would discard a repair if the corresponding Formula 10.2 turned out to be SAT. Finally, HSsynth computes the satisficing measure for the remaining repair options. HSsynth has a greedy approach to increase the satisficing measure as it picks the repair for which $F(X)$ has the maximum satisficing measure among others.

10.2 Approach

In this chapter, we provide a detailed algorithmic description HSsynth, whose pseudocode is presented in Algorithm 12. HSsynth takes soft constraint $\varphi_S(X, Y)$, hard constraints $\varphi_H(X, Y)$ and a satisficing threshold ε.

Algorithm 12 HSsynth($\varphi_S(X,Y), \varphi_H(X,Y), \varepsilon$)

```
1: $F(X) \leftarrow$ LearnCandidate($\varphi_S(X,Y), \varphi_H(X,Y)$)
2: **while** ret = UNSAT **do**
3:     ret, $\sigma \leftarrow$ CheckSat($\varphi_H(X,Y) \wedge \neg\varphi_H(X,Y') \wedge (Y' \leftrightarrow F(X))$)
4:     **if** ret = SAT **then**
5:         $F(X) \leftarrow$ RepairHC($\varphi_H(X,Y), F(X), \sigma$)
6: **while** True **do**
7:     **if** $SM(\varphi_S(X,Y), F(X)) < \varepsilon$ **then**
8:         res, $\pi \leftarrow$ CheckSat($\varphi_S(X,Y) \wedge \neg\varphi_S(X,Y') \wedge \varphi_H(X,Y') \wedge (Y' \leftrightarrow F(X))$)
9:         **if** res = SAT **then**
       $F(X)$, pos $\leftarrow$ RepairGM($\varphi_S(X,Y), F(X), SM(,), \pi$)
10:        **if** res =UNSAT or pos =False **then** $\varphi_S(X,Y) \leftarrow \varphi_S(X,Y) \wedge (Y \neq F(X))$
   Goto line 1
11:    **elsereturn** $F(X)$
```

Algorithm 12 assumes access to following subroutines:

1. LearnCandidate: It takes $\varphi_S(X,Y), \varphi_H(X,Y)$ as input and outputs the initial candidate system $F(X)$. Let $\varphi(X,Y)$ is union of $\varphi_S(X,Y)$ and $\varphi_H(X,Y)$. If $\varphi(X,Y)$ is provided as data, HSsynth uses a decision tree based method to learn the candidate system. HSsynth considers the valuation of X on $\varphi_S(X,Y)$ as features and valuation of each y of Y as label to learn a decision tree for each y. Candidate system $F(X)$ is disjunction of each path in learned decision tree with leaf node **1**. If $\varphi(X,Y)$ is given as a logical specification, HSsynth learns a candidate system using functional synthesis approach [GRM20, GRM21]. HSsynth first generates the data using a constraint sampler. HSsynth consider the uniformly generated satisfying assignments of $\varphi(X,Y)$ as data to learn a candidate system using decision tree based method.

2. CheckSat: It takes a formula as input and returns the outcome of a satisfiability check on the formula. If the formula is SAT, CheckSat returns outcome as SAT, and in addition to that it returns a satisfying assignment of the formula. If the formula is UNSAT, it returns outcome as UNSAT and an empty list.

3. RepairHC: It takes the hard constraints $\varphi_H(X,Y)$, and the candidate system $F(X)$ and a counterexample σ as inputs and returns a candidate system re-

paired with respect to the hard constraints. RepairHC is discussed in detailed in Algorithm 13.

4. RepairGM: It takes the relational specification $\varphi_S(X, Y)$, and the candidate system $F(X)$ and a counterexample π as inputs. If there exists a repaired candidate system with respect to $\varphi_S(X, Y)$ that does not falsify $\varphi_H(X, Y)$ and increase SM(,), RepairGM returns repaired $F(X)$, and True. Otherwise, it returns $F(X)$ and False. RepairGM is discussed in detailed in Algorithm 14.

Algorithm 12 starts off by learning a candidate $F(X)$ using given $\varphi_S(X, Y)$ and $\varphi_H(X, Y)$ at line 1. At line 3, Algorithm 12 ask for a counterexample by calling subroutine CheckSat on the formula $\varphi_H(X, Y) \wedge \neg\varphi_H(X, Y') \wedge (Y' \leftrightarrow F(X))$. CheckSat subroutine returns the outcome of satisfiability check and a satisfying assignment (say σ) of the formula. Algorithm 12 repairs the candidate function $F(X)$ to fix counterexample σ by calling the subroutine RepairHC with input $\varphi_H(X, Y)$, $F(X)$, and σ at line 5.

Algorithm 12 has another loop from line 6 to line 11, and it comes out of the loop by either returning system $F(X)$ which satisfies the hard constraints and has $SM()$ greater than ε at line 11 or asking for another candidate system $F(X)$ at line 10. Algorithm 7 measures if the satisficing measure $SM()$ of candidate system $F(X)$ is greater than threshold ε at line 7, and if it does, Algorithm 12 returns a system $F(X)$ at line 11. Else, Algorithm 12 at line 8 again calls CheckSat with input $\varphi_S(X, Y) \wedge \neg\varphi_S(X, Y') \wedge \varphi_H(X, Y') \wedge (Y' \leftrightarrow F(X))$. If CheckSat returns SAT with satisfying assignment π, Algorithm 9 repairs candidate system $F(X)$ by calling subroutine RepairGM with inputs $\varphi_S(X, Y)$, $F(X)$ and π at line 9. If CheckSat returns UNSAT or if RepairGM returns False, Algorithm 12 goes back to line 1 with relation $\varphi_S(X, Y) \wedge (Y \not\leftrightarrow F(X))$. Algorithm 13 presents RepairHC. It assumes access to following subroutine.

1. FindCore: It takes a formula as input and extracts the unsatisfiable core of the formula. It returns a list of unit clauses C in the unsatisifiable core.

Algorithm 13 RepairHC($\varphi_H(X,Y), F(X), \sigma$)

```
1: $C \leftarrow \mathsf{FindCore}(\varphi_H(X,Y) \wedge (Y \leftrightarrow F(X)) \wedge (X \leftrightarrow \sigma[X]))$
2: $\beta \leftarrow \bigwedge_{l \in C} l$
3: if $F(\sigma[X]) == 1$ then $F(X) \leftarrow F(X) \wedge \neg\beta$
4: else$F(X) \leftarrow F(X) \vee \beta$
5: return $F(X)$
```

Algorithm 13 first calls FindCore with formula $\varphi_H(X,Y) \wedge (Y \leftrightarrow F(X)) \wedge (X \leftrightarrow \sigma[X])$ at line 1. Algorithm 13 creates a formula β as conjunction of clauses in C, and checks the output of system $F(X)$ with valuation of X as per σ at line 3. If $F(\sigma[X])$ is 1, then repaired $F(X)$ is $F(X)$ with conjunction of negation of β, else repaired $F(X)$ is $F(X)$ with disjunction of β.

Algorithm 14 presents RepairGM. It assumes access to following subroutines:

1. CoreEnum: It takes an unsatisfiable formula and an integer as input K to enumerate K many unsatisfiable cores of the given formula. Let the list of unit clauses in each unsatisfiable core is C. CoreEnum adds C corresponding to each unsatisfiable core to $Clist$ and finally returns $Clist$.

2. FindRepair: It take a list of tuple as input where tuple is $\langle float\ F, Formula\ S\rangle$ and returns formula S corresponding to which F has maximum value among all other tuples.

Algorithm 14 RepairGM($\varphi_S(X,Y), F(X), \pi$)

```
1: $Clist \leftarrow CoreEnum(\varphi_S(X,Y) \wedge \varphi_H(X,Y) \wedge (Y \leftrightarrow F(X)) \wedge (X \leftrightarrow \pi[X]), K)$ ▷ Enumeration of K unsat cores $Sys \leftarrow \langle \emptyset, \emptyset \rangle$
2: for each $C \in Clist$ do
3:     $\beta \leftarrow \bigwedge_{l \in C} l$
4:     $rF(X) \leftarrow ite((F(\pi[X]) == 1), F(X) \wedge \neg\beta, F(X) \vee \beta)$
5:     ret, $\sigma \leftarrow \mathsf{CheckSat}(\varphi_H(X,Y) \wedge \neg\varphi_H(X,Y') \wedge (Y' \leftrightarrow rF(X)))$
6:     if ret = UNSAT then $Sys \leftarrow Sys.add(\langle SM(\varphi_S(X,Y), rF(X)), rF(X)\rangle)$
7: if $Sys = \langle \emptyset, \emptyset \rangle$ then
8:     return $F(X), False$
9: $F(X) \leftarrow FindRepair(Sys)$
10: return $F(X)$, $True$
```

Algorithm 14 first calls subroutine CoreEnum at line 1 with formula $(\varphi_S(X,Y) \wedge$

$\varphi_H(X,Y) \wedge (Y \leftrightarrow F(X) \wedge (X \leftrightarrow \pi[X])))$ and an integer K to get a list of different UnsatCores. Algorithm 14 then has a loop from line 2 to line 6 that iterates over different UnsatCores of formula. Algorithm 14 at line 4 repairs $F(X)$ by either strengthening or weakening depending on the valuation of $F(\pi[X])$ with respect to each UnsatCore. Line 5 makes a CheckSat call to the formula $\varphi_H(X,Y) \wedge \neg\varphi_H(X,Y') \wedge (Y' \leftrightarrow rF(X))$, where $rF(X)$ represent an option for repaired $F(X)$ with respect to an unsatcore. If CheckSat returns UNSAT, line 6 adds the SM($\varphi_S(X,Y)$,$F(X)$) measure with repaired $F(X)$ and the repaired $F(X)$ as a tuple to Sys.

If Sys is empty or not, RepairGM returns $F(X)$ and False. Otherwise, Sys has $F(X)$ repaired with respect to different UnsatCore and their corresponding measure respectively. Algorithm 14 finds repaired $F(X)$ that has maximum SM($\varphi_S(X,Y)$,$F(X)$) measure by calling subroutine FindRepair with Sys as input at line 9. Finally, it returns $F(X)$ and True.

Algorithmic optimizations. When we have $|Y| > 1$, we could learn a system for each y_i in terms of input X and $Y \setminus y_i$. However, we need to ensure that there is no cyclic dependencies among Y variables, that is, if f_i system corresponding to y_i is dependent on y_j, then f_j is not allowed to depend on y_i. Note that, we can have the synthesized function finally in terms of only X as we can expand functions corresponding to y_j's in the definition of y_i. Let $y_i \prec_d y_j$ denotes that f_i depends on y_j. A system vector F where f_i depends on y_j is valid vector if there exists a partial order $\prec_d$ over $\{y_1, \ldots, y_{|Y|}\}$. We can obtain a valid linear extension, *TotalOrder*, of partial order $\prec_d$ in accordance to F.

In such a case, we need to follow the *TotalOrder* while finding candidates and repairing them. We employ approach proposed by Golia et al. [GRM20], and use MaxSAT solver to find the candidates to repair. For RepairHC, we use $\varphi_H(X,Y) \wedge (X \leftrightarrow \sigma[X])$ as hard constraints and $(Y \leftrightarrow \sigma[Y'])$ as soft constraints to use a MaxSAT solver to find candidates, where σ is a satisfying assignment of formula 10.2. All Y variables for which $(Y \leftrightarrow \sigma[Y'])$ are dropped by MaxSAT solver to find a satisfying assignments, function corresponding to which are candidates that may need repair. Similarly, for RepairGM, we use $\varphi_S(X,Y) \wedge (X \leftrightarrow \pi[X])$ as hard constraint and $(Y \leftrightarrow \pi[Y'])$ as softconstraint to call MaxSAT solver, where π is a satisfying assignment of formula 10.4. Let *ind* represents Y variables whose corresponding functions may needed to be repaired.

Furthermore, when $|Y| > 1$, Formula 10.3 and 10.5 should be changed to ensure the acyclic dependencies among Y variables by constraining over $\hat{Y}$, which is the subset of Y variables and $TotalOrder[index(y_i)+1], \ldots, TotalOrder[|Y|]$. Formula 10.3 is modified as follows:

$$M(X,Y) := \varphi_H(X,Y) \wedge (\hat{Y} \cup y_i \leftrightarrow F(X)) \wedge (X \leftrightarrow \sigma[X]) \qquad (10.6)$$

Similarly, formula 10.5 is modified as:

$$N(X,Y) := \varphi_S(X,Y) \wedge \varphi_H(X,Y) \wedge (\hat{Y} \cup y_i \leftrightarrow F(X)) \wedge (X \leftrightarrow \pi[X]) \qquad (10.7)$$

Now, modified formulas 10.6 and 10.7 could be SAT, and in that case, we need to look for other candidates to repair. To this end, assume that while reparing candidates to satisfy the hard constraints, Formula 10.6 turns out to be SAT, and let ρ be the satisfying assignment, then all y_j variables for which $\rho[y_j] \neq \sigma[y_j]$ and $y_j \notin Ind$ are added to the queue of potential candidates to repair. Similarly, additional candidates are considered to repair if while improving satisficing measure, Formula 10.7 turned out to be SAT.

If the formulas are UNSAT, then we extract the UnsatCore and use the core to repair the candidate as discussed in Algorithm 13 and 14.

We now illustrate our algorithm through an example.

Example 10.1. *Let us consider* $X = \{x_1, x_2\}$ *as inputs and* $Y = \{y_1, y_2\}$ *as outputs. Given following: (i)* $\varphi_S(X,Y) : y_1 \leftrightarrow x_1 \vee x_2$*, (ii)* $\varphi_H(X,Y) : y_2 \leftrightarrow ((x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2))$*, and (iii) satisficing threshold* $\varepsilon : 80\%$*, that is,* ε *is set to 80%, synthesize* $F(X)$ *that satisfies* $\varphi_H(X,Y)$*, and have* $SM(\varphi_S(X,Y), F(X)) \geq \varepsilon$*.*

Learn candidate: HSsynth first comes up with a candidate system for y_1 and y_2.

As both hard and soft constraints are given as specification, we can generate samples of the specification, which would be considered as data to learn the candidate functions using decision trees. Let us consider the initial system as $F(X) : \langle f_1(X) : x_1,\ f_2(X) : \neg x_1 \wedge x_2 \rangle$.

Does F(X) satisfy $\varphi_H(X, Y)$?: HSsynth makes a CheckSat subroutine call on formula $y_2 \leftrightarrow ((x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2)) \wedge \neg(y'_2 \leftrightarrow ((x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2))) \wedge (y'_2 \leftrightarrow (\neg x_1 \wedge x_2))$. CheckSat returns SAT and a satisfying assignment σ as $\langle x_1 \leftrightarrow 1, x_2 \leftrightarrow 0, y_1 \leftrightarrow 1, y_2 \leftrightarrow 1, y'_1 \leftrightarrow 1, y'_2 \leftrightarrow 0\rangle$.

Repair F(X) with respect to $\varphi_H(X, Y)$: HSsynth calls subroutine RepairHC with $\varphi_H(X, Y)$, $F(X)$ and σ. RepairHC calls FindCore with $(y_2 \leftrightarrow ((x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2))) \wedge (y_2 \leftrightarrow (\neg x_1 \wedge x_2)) \wedge (x_1 \leftrightarrow 1) \wedge (x_2 \leftrightarrow 0) \wedge (y_1 \leftrightarrow 1) \wedge (y_2 \leftrightarrow 0)$. As FindCore returns the list of unit clauses occurred in the unsatisfiable core, let us assume FindCore returns $[x_1, \neg x_2]$. Repair formula β is $x_1 \wedge \neg x_2$, and as $\sigma[y'_2]$ is 0, the repaired f_2 is $f_2 \vee \beta$. Now, $F(X) : \langle f_1(X) : x_1, f_2(X) : (\neg x_1 \wedge x_2) \vee (x_1 \wedge \neg x_2)\rangle$.

The repaired $F(X)$ satisfies $\varphi_H(X, Y)$, and CheckSat returns UNSAT.

Is $SM(\varphi_S(X, Y), F(X)) \geq 80\%$: We use Formula 10.1 to compute $SM()$. We used model counter to compute the $ModelCount((y_1 \leftrightarrow x_1 \vee x_2) \wedge (y_1 \leftrightarrow x_1) \wedge (y_2 \leftrightarrow (\neg x_1 \wedge x_2) \vee (x_1 \wedge \neg x_2))_{\downarrow x_1, x_2})$ is 3, and $ModelCount((y_1 \leftrightarrow x_1 \vee x_2)_{\downarrow x_1, x_2})$ is 4. Therefore, $SM()$ is 75%, and we need to repair $F(X)$.

Is repair possible?: In order to check if there exists a repair that do satisfy the $\varphi_H(X, Y)$, HSsynth makes a CheckSat call on $(y_1 \leftrightarrow x_1 \vee x_2) \wedge \neg(y'_1 \leftrightarrow x_1 \vee x_2) \wedge y'_2 \leftrightarrow ((x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2)) \wedge y'_1 \leftrightarrow x_1 \wedge (y'_2 \leftrightarrow (\neg x_1 \wedge x_2) \vee (x_1 \wedge \neg x_2))$. CheckSat returns SAT with a counterexample π as $\langle x_1 \leftrightarrow 0, x_2 \leftrightarrow 1, y_1 \leftrightarrow 1, y_2 \leftrightarrow 1, y'_1 \leftrightarrow 0, y'_2 \leftrightarrow 1\rangle$.

Repair F(X) to increase $SM()$: Again, HSsynth enumerates different UnsatCore of $(y_2 \leftrightarrow (x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2)) \wedge (y_1 \leftrightarrow x_1 \vee x_2) \wedge (y_1 \leftrightarrow x_1) \wedge (x_1 \leftrightarrow$

$0) \wedge (x_2 \leftrightarrow 1)$. HSsynth picks UnsatCore as $[\neg x_1, x_2]$. Repair formula β is $\neg x_1 \wedge x_2$, and as $\pi[y'_1]$ is 0, the repaired f_1 is $f_1 \vee \beta$. The repaired candidates don't falsify the hard constraints. Now, $F(X) : \langle f_1(X) : x_1 \vee (\neg x_1 \wedge x_2), f_2(X) : (\neg x_1 \wedge x_2) \vee (x_1 \wedge \neg x_2) \rangle$.

Now, with repaired $F(X)$, the satisficing measure is greater than 80%. HSsynth returns $F(X)$.

10.3 Experimental Results

The objective of the experimental evaluation is to show that the proposed method HSsynth is adaptable in different settings. In regards to that, we considered the following underlying cases: (i) soft and hard constraints as formulas, (ii) soft constraints as data, and hard constraints as a formula. The remaining two cases, (iii) soft and hard constraints as data, (iv) soft constraints as a formula, and hard constraints as data, could be converted into the first two settings by learning a formula from the given hard constraints.

Setup: All our experiments were conducted on a high-performance computer cluster with each node consisting of a E5-2690 v3 CPU with 24 cores and 96GB of RAM, with a memory limit set to 4GB per core. All tools were run in a single-threaded mode on a single core. We considered the satisficing threshold as 75%.

Tools: We used python-sat(pysat) library [IMM18] to get a satisfying assignment and to encode learned decision trees to CNF formula. We used MUSER2 to find group minimal UnsatCore [BMS12], and MUST to enumerate different UnsatCores [BČ20]. We used ApproxMC to compute the approximate model count of a given formula [SM19]. We used CMSGen to generate the samples to learn an initial system [GSCM21]. We used Scikit learn library to learn decision trees to synthesize candidate $F(X)$ [skl, PVG$^+$].

In particular, we sought to answer the following questions for each of the aforementioned cases as applicable:

- Is the performance of HSsynth comparable to the complete synthesis of the specification, where specification includes both hard and soft constraints?
- how *good* is the system synthesized by HSsynth?

Summary: Considering both $\varphi_S(X, Y)$ and $\varphi_H(X, Y)$ as formula, HSsynth takes less time 2× time than the complete synthesis for 57% of our benchmarks. Moreover, for more than 67% of our benchmarks in this setting, the satisficing measure of the synthesized system by HSsynth is greater than 95%. For the remaining two settings, HSsynth took less than 20 seconds to repair the system, and the satisficing measure of the synthesized system was greater than 80% for *almost* all considered datasets and hard constraints.

$\varphi_S(X, Y)$ as data and $\varphi_H(X, Y)$ as formula

Setup: We considered two datasets: (i) UCI Adult [DG+17] (ii) Ricci [McG10]. The two datasets provided variety in terms of data sizes, as UCI Adult data set has 32500 data entries, whereas Ricci have 118 data entries. Adult data is about to determine whether a person makes over 50K a year or not, and it has attributes like workclass, occupation, education, capital-gain, gender, and etc. On the other hand, Ricci data set is about the firefighter promotion exam and has attributes like race, position, oral exam score, written exam score, etc. We considered different hard constraints for each considered dataset. Data entries of corresponding to each dataset are considered as soft-constraints $\varphi_S(X, Y)$.

Adult Dataset: The hard constraints $\varphi_H(X, Y)$ are considered as follows:

- HS_1: If its a Asian-par islander race female in 0-25 age, have a working spouse and works in craft repair then she must be getting $\geq$ 50K.

Table 10.1: Satisificing synthesis: Results of different dataset when $\varphi_H(X,Y)$ is considered as formulas. Time in seconds.

Dataset	Hard Constraints	$SM()$ Measure	Repair Itr	Repair Time
Adult	HS_1	75%	1	5.45
	HS_2	81.9%	2	10.97
	HS_3	81.8%	4	19.31
Ricci	HS_1	96.6 %	2	0.03
	HS_2	96.6 %	4	0.07

- HS_2: If they are a self-employed with bachelor education and have age > 50, then they must be getting $\geq$ 50K.
- HS_3: If they have workclass as local-gov, and they have occupation either as handlers-cleaners or in transportation, then they must be getting $\leq$ 50K.

Ricci Dataset: We considered the following hard constraints:

- HS_1: if the position is Lieutenant and have more than 75 in written and oral exam then must recruit.
- HS_2: if the position is Lieutenant and they have more than 75 in written, also the combine score is more than 80 then must recruit, and if they have less than 25 in written, also the combine score is less than 25 then must not recruit.

Results: Table 10.1 represents the experimental details for different data set respectively. In each table column 1 presents the hard constraints under consideration. Column 2 presents satisficing measure $SM(\varphi_S(X,Y), F(X))$. Column 4 presents the required number of repairs needed for the system. Finally, column 5 present the repair time. Satisficing measure of $F(X)$ without any repair was 81.9% and 96.6% for Adult and Ricci dataset respectively.

As shown in Table 10.1, we could repair the system to meet the hard constraints within 4 repairs for both the datasets, and it would need less than 20 seconds. Note that it is not the case that the more number of repair leads to decrease in the satisficing measure, but it is completely dependent on the underlying hard constraints—considering the case of HS_2 of adult dataset, there is no difference in the satisficing measure with or without any repair; however, the underlying system needed to be repaired to provably satisfy hard constraints.

Ricci dataset presents our proposed method in a good light, as we usually need the proposed hard constraints or rules in the recruitment purpose and the data provided to us also satisfies the given hard constraints. However, our synthesized system needs to undergo repair. As shown in Table 10.1, HS_1 and HS_2 with ricci dataset need respectively 2 and 4 repairs to ensure that the learned system must satisfy the hard constraints, however, we did not observe any change in satisficing measure of $F(X)$ with or without repair.

$\varphi_S(X,Y)$ and $\varphi_H(X,Y)$ as formulas

Setup: We considered Hacker's delight instances that are fast bit-level algorithms [syg19]. We considered interesting benchmarks that identifies the rightmost one bit and count the number of trailing zeros. We experimented with different varying input size from 8 bits to 512 bits. Three different hard constraints are considered for each instance. The following should hold on all input valuations for synthesized system:

- HS_1: middle-bit output should be correct.
- HS_2: most significant bit output should be correct.
- HS_3: five most significant bit output should be correct.

The remaining bit's output should be correct on all input valuations are considered as soft-constraint corresponding to each hard-constraint. Therefore, we experimented with 48 benchmarks (2 instances $\times$ 8 varying input bits $\times$ three hard constraints).

HSsynth compared with functional synthesis engine Manthan that generates provable correct functions for $\varphi(X,Y)$ where $\varphi(X,Y)$ is union of $\varphi_S(X,Y)$ and $\varphi_H(X,Y)$.

Table 10.2: Satisificing synthesis: Time taken by Manthan and HSsynth to synthesize function. Timeout (TO) is 1800 seconds.

Benchmarks	Manthan	HSsynth		
		HS_1	HS_2	HS_3
hd-04-8bits	2.64	3.51	3.67	3.65
hd-08-8bits	2.66	3.47	3.59	3.54
hd-08-16bits	3.43	3.82	4.04	3.91
hd-04-16bits	4.29	3.73	3.96	4.01
hd-08-32bits	4.83	4.49	5.24	5.82
hd-04-32bits	6.75	4.47	9.47	5.47
hd-08-64bits	21.19	8.7	10.85	14.76
hd-04-64bits	24.94	14.53	7.97	10.5
hd-08-128bits	77.1	36.62	25.86	50.73
hd-04-128bits	213.55	35.01	40.71	40.69
hd-08-256bits	569.08	104.56	152.33	198.47
hd-04-256bits	837.56	100.08	130.92	231.03
hd-08-272bits	841.22	129.23	157.08	177.66
hd-04-272bits	945.51	129.44	134.47	189.63
hd-04-512bits	TO	494.48	928.27	915.12
hd-08-512bits	TO	529.05	1174.05	1466.38

Results: Table 10.2 presents the results for Hacker's delight benchmarks with different input sizes. Column 2 presents the time taken to synthesize $F(X)$ by Manthan. Column 3,4,5 presents the time taken by HSsynth to synthesize $F(X)$ that have $SM(\varphi_S(X,Y), F(X)) \geq \varepsilon$ and satisfies hard constraints HS_1, HS_2, HS_3 respectively. As shown in Table 10.2 for inputs with bitsize less than 32, we observe that Manthan is faster than HSsynth, as HSsynth needs additional time to compute satisficing measure, check for repair, and like. However, as we increase the bitsize, Manthan is much slower than HSsynth. In fact, for inputs with 512 bits, Manthan was not able to synthesize the function within the given time and memory limit, whereas, HSsynth was able to return the synthesized system within 1500 seconds.

Figure 10.2 presents the satisficing measure of synthesized system by HSsynth; on $\langle y \rangle$-axis presents the satisficing measure and $\langle x \rangle$-axis presents the different benchmarks. For 21 out of 48 benchmarks HSsynth synthesized system with satisficing measure greater than 98%. Even though the satisficing threshold ε is set to 75%, the minimum measure among all synthesized system is 81.25%.

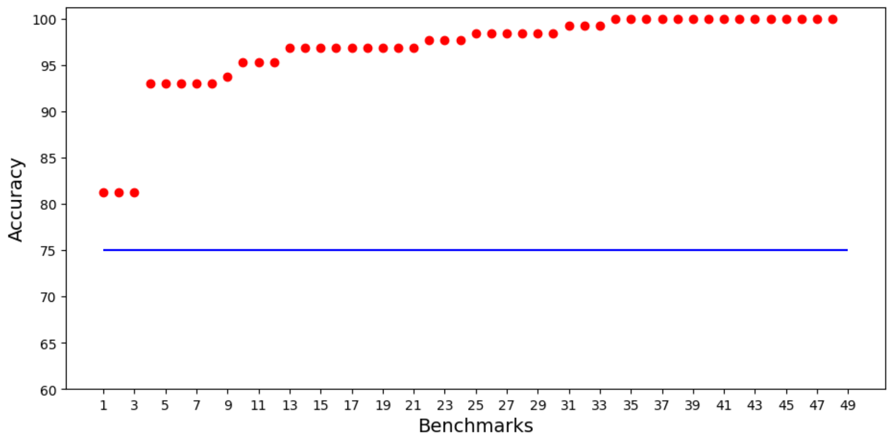


Figure 10.2: Satisificing synthesis: Showing satisficing measure of $F(X)$ synthesized by HSsynth for hard constraints HS_1, HS_2 and HS_3.

$\varphi_S(X,Y)$ and $\varphi_H(X,Y)$ as data

Setup: We considered two datasets: (i) UCI Adult [DG+17] (ii) Ricci [McG10], and divided them into (i) soft constraints and (ii) hard constraints. We first learned a overfitted system with only hard constraint data and considered the equivalent formula of learned model as our $\varphi_H(X,Y)$ formula. We considered two cases (i) HS_1: 10% of data, and (ii) HS_2: 20% of data as hard constraints.

Table 10.3: Satisificing synthesis: Results of Adult and Ricci dataset with hard constraints as data. Time in seconds.

Dataset	Hard Constraints	$SM()$ Measure	Repair Itr	Repair Time
Adult	HS_1	81.4%	3	16.35
	HS_2	82.1%	3	13.30
Ricci	HS_1	86.7 %	1	0.017
	HS_2	86.6%	1	0.036

Results: Table 10.3 shows the results with Adult dataset and Ricci dataset. HSsynth is able to repair learned system to satisfy formula learned from data considered as hard constraint. Even through the threshold is set to 75%, we get the satisficing measure more than 80% for both the dataset. HSsynth was able to repair the system in less than 20 seconds for all the cases.

Part V

Conclusion

Chapter 11

Concluding Remarks

This thesis focuses on functional synthesis, a fundamental problem in computer science with a wide range of applications including but not limited to: hardware synthesis, software synthesis, synthesis in formal verification, and synthesis in control system. Functional synthesis has been studied for over 150 years, dating back to Boole in 1850's, yet scalability remains a core challenge. To tackle the scalability challenge, we relied on artificial intelligence and formal methods.

We developed a scalable data-driven approach, Manthan, that combines advances in formal methods and machine learning to significantly improve upon the state-of-the-art. Our approach uses constrained sampling to generate data, which is then fed into a machine learning pipeline to generate an initial candidate system. We leverage automated reasoning to repair the candidate system and synthesize a final system that provably satisfies the given specification.

Manthan was able to synthesize functions for 509 instances out of a total of 609 standard suites of instances — to give a perspective, the prior state-of-the-art techniques ranged from 210 to 280 instances. Manthan improved the state-of-the-art by solving an additional 40% of instances. Furthermore, we lift the data-driven approach to contrive modular designs that enabled Manthan to push the envelope in synthesis with explicit input dependencies, and it could handle additional 26 instances for which the state-of-the-art tools could not synthesize a system.

Motivated by the impressive scalability, we turned our attention to program synthesis. We demonstrated that the problem of program synthesis reduces to functional synthesis when there are no syntactic restrictions. We investigated the use of $\mathbb{T}$-constrained synthesis where $\mathbb{T}$ stands for theory, and showed its reduction to DQF($\mathbb{T}$). Moreover, we focused on the special case of $\mathbb{T} = BV$ and its reduction to DQBF. We also note that grammar can serve both as a tool for improving solver efficiency and as a means for specifying certain properties, such as information flow leakage. Our empirical analysis indicates that $\mathbb{T}$-constrained synthesis can improve performance compared to syntax-guided program synthesis approaches, and DQF($\mathbb{T}$) solvers perform similarly to domain-specific techniques. These results suggest the potential benefits of further research into DQB($\mathbb{T}$) as a general-purpose representation language for program synthesis.

Additionally, as a generalization from functional synthesis, we introduced HSsynth, a general purpose framework for synthesizing systems that provably satisfy hard constraints, with a satisficing measure of the synthesized system in regards to soft constraints greater than a certain threshold. The analysis shows that HSsynth is scalable and can achieve a satisficing measure greater than 80% for all considered settings of constraints.

In summary, the thesis makes several novel contributions to the field of functional synthesis. The proposed data-driven approach is new and innovative techniques that can greatly improve the performance of functional synthesis both with or without explicit dependencies. The Henkin synthesizer and the framework for synthesizing systems that provably satisfy hard constraints are significant contributions. Overall, the proposed approach and tools have a wide range of applications and could open up several exciting possibilities for future research at the intersection of machine learning, constrained sampling, and automated reasoning.

Chapter 12

Future Directions

We end the thesis by presenting a list of several promising avenues for future research. We believe that exploring these directions could be crucial and lead to significant advancements. Furthermore, progress in these areas has the potential to unlock various real-world applications.

Theoretical analysis. The immediate future direction would be to analyze the technique proposed from a theoretical point of view. While the thesis provides significant empirical evidence on the importance of different components used in the data-driven approach for functional synthesis, it is also essential to consider the theoretical analysis regarding these components. For example, in Chapter 7.2, we discussed empirically that the quality of the generated data is a critical factor in synthesizing functions efficiently. However, we still lack knowledge about the ideal distribution from which we should sample the data points. Additionally, we need to theoretically determine the bound on the number of data points required to learn 'good" candidate functions. Conducting a theoretical analysis of the empirical evidence is one of the future directions to explore.

Approximate functional synthesis. In this thesis, we have explored algorithmic improvements for functional synthesis. As we move forward, it is important to define the concept of approximate functional synthesis. One initial step in this direction was the introduction of the notion of satisficing synthesis. Now, our next objective could be to propose techniques that can provide probabilistic Approximate Correct (PAC) guarantees for approximate functional synthesis. By achieving this, we can apply these techniques to scenarios where having an error bound on the synthesized function is crucial.

Search for optimal functions. In this thesis, our focus has been on synthesizing arbitrary functions that satisfy a given specification. However, there is an exciting opportunity to formalize the notion of function quality, considering factors such as size, the use of specific gates, readability, applicability to specific domains, and more. It would be intriguing to explore whether we can guide the synthesis technique towards finding the optimal function that meets these quality criteria. By doing so, we can enhance the overall effectiveness and efficiency of the synthesis process.

Beyond synthesizing functions. In the realm of SAT solving, we initially had excellent SAT solvers that could find satisfying assignments for given specifications. Over time, these solvers evolved to not only find assignments but also count them, sample them, and perform other advanced operations. These advancements have paved the way for various applications, with functional synthesis being one of them. Following a similar trajectory, now that we have efficient synthesizers for generating single functions, it is worth considering the exploration of enumerating functions, randomly sampling functions, and counting functions. This expansion poses significant theoretical complexity challenges, but it undoubtedly holds the potential for numerous practical applications.

Functional synthesis modulo theory. Over the past two decades, we have witnessed remarkable advancements in the field of satisfiability modulo theory (SMT) solvers. These solvers have demonstrated their ability to handle various underlying theories, including strings, bitvectors, and linear real arithmetic. However, in order to extend functional synthesis beyond propositional logic and address real-world applications effectively, it is necessary to incorporate machine learning-based techniques into the process of learning and formal method based candidate repai for SMT. To accomplish this, a key area of focus should be the development of efficient constrained samplers and counters for SMT, enabling synthesis for different underlying theories.

A key contribution of the thesis is successful integration of machine learning and formal methods, resulting in trustworthy scalability. Looking ahead, it is important to explore further applications where machine learning can be leveraged to find rapid solutions, while relying on formal methods to ensure correctness. By combining the strengths of both machine learning and formal methods, we can continue to advance in the pursuit of scalable and trustworthy solutions across various domains.

References

[AAC$^+$19] S Akshay, Jatin Arora, Supratik Chakraborty, S Krishna, Divya Raghunathan, and Shetal Shah. Knowledge compilation for boolean functional synthesis. In *Proc. of FMCAD*, 2019.

[ABJ$^+$13a] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo MK Martin, Mukund Raghothaman, Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *Proc. of FMCAD*, 2013.

[ABJ$^+$13b] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo MK Martin, Mukund Raghothaman, Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *Proc. of FMCAD*, 2013.

[ACG$^+$18] S Akshay, Supratik Chakraborty, Shubham Goel, Sumith Kulal, and Shetal Shah. What's hard about boolean functional synthesis? In *Proc. of CAV*, 2018.

[ACJS17] S Akshay, Supratik Chakraborty, Ajith K John, and Shetal Shah. Towards parallel boolean functional synthesis. In *Proc. of TACAS*, 2017.

[ARU17] Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. Scaling enumerative program synthesis via divide and conquer. In *Proc. of TACAS*, 2017.

[BČ20] Jaroslav Bendík and Ivana Černá. Must: minimal unsatisfiable subsets enumeration tool. In *Proc. of TACAS*. Springer, 2020.

[BCCZ99] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *Proc. of TACAS*, 1999.

[BCD$^+$11] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovi'c, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *Proc. of CAV*, 2011.

[BCJ14] Valeriy Balabanov, Hui-Ju Katherine Chiang, and Jie-Hong R Jiang. Henkin quantifiers and boolean formulae: A certification perspective of DQBF. *Proc. of Theoretical Computer Science*, 2014.

[Bet56] Evert W Beth. On padoa's method in the theory of definition. *Journal of Symbolic Logic*, 1956.

[BGHZ15] Daniel W Barowy, Sumit Gulwani, Ted Hart, and Benjamin Zorn. Flashrelate: extracting relational data from semi-structured spreadsheets using examples. *ACM SIGPLAN Notices*, 2015.

[Bie08] Armin Biere. PicoSAT essentials. *Proc. of JSAT*, 2008.

[BJ11] Valeriy Balabanov and Jie-Hong R Jiang. Resolution proofs and skolem functions in QBF evaluation and applications. In *Proc. of CAV*, 2011.

[BJ12] Valeriy Balabanov and Jie-Hong R Jiang. Unified QBF certification and its applications. In *Proc. of FMCAD*, 2012.

[BKS14] Roderick Bloem, Robert Könighofer, and Martina Seidl. Sat-based synthesis methods for safety specs. In *Proc. of VMCAI*, 2014.

[BLS11] Armin Biere, Florian Lonsing, and Martina Seidl. Blocked clause elimination for QBF. In *Proc. of CADE*, 2011.

[BMS12] A. Belov and J. Marques-Silva. MUSer2 : An efficient mus extractor, system description. *Proc. of JSAT*, 2012.

[Boo47] George Boole. *The mathematical analysis of logic*. Philosophical Library, 1847.

[BPR16] Rohan Bavishi, Awanish Pandey, and Subhajit Roy. To be precise: regression aware debugging. In *Proc. of OOPSLA*, 2016.

[Bra89] Robert K Brayton. Boolean relations and the incomplete specification of logic networks. In *Proc. of VLSID*, 1989.

[BS89] Robert K Brayton and Fabio Somenzi. An exact minimizer for boolean relations. In *Proc. of ICCAD*, 1989.

[CFM$^+$15] Supratik Chakraborty, Daniel J Fremont, Kuldeep S Meel, Sanjit A Seshia, and Moshe Y Vardi. On parallel scalable uniform SAT witness generation. In *Proc. of TACAS*, 2015.

[CFTV18] Supratik Chakraborty, Dror Fried, Lucas M Tabajara, and Moshe Y Vardi. Functional synthesis via input-output separation. In *Proc. of FMCAD*, 2018.

[CHOP13] Krishnendu Chatterjee, Thomas A Henzinger, Jan Otop, and Andreas Pavlogiannis. Distributed synthesis for ltl fragments. In *Proc. of FMCAD*, 2013.

[CM19] Sourav Chakraborty and Kuldeep S. Meel. On testing of uniform samplers. In *Proc. of AAAI*, 2019.

[CMF19] Yanju Chen, Ruben Martins, and Yu Feng. Maximal multi-layer specification synthesis. In *Proc. of ESEC/FSE*, 2019.

[CMV13] Supratik Chakraborty, Kuldeep S Meel, and Moshe Y Vardi. A scalable and nearly uniform generator of SAT witnesses. In *Proc. of CAV*, 2013.

[CMV14] Supratik Chakraborty, Kuldeep S Meel, and Moshe Y Vardi. Balancing scalability and uniformity in SAT witness generator. In *Proc. of DAC*, 2014.

[Coo23] Stephen A Cook. The complexity of theorem-proving procedures. In *Logic, Automata, and Computational Complexity: The Works of Stephen A. Cook*, pages 143–152. 2023.

[DG$^+$17] Dheeru Dua, Casey Graff, et al. Uci machine learning repository. 2017. Available at `https://archive.ics.uci.edu/`.

[DLBS18] Rafael Dutra, Kevin Laeufer, Jonathan Bachrach, and Koushik Sen. Efficient sampling of SAT solutions for testing. In *Proc. of ICSE*, 2018.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Proc. of TACAS*, 2008.

[DPV21] Jeffrey M. Dudek, Vu H. N. Phan, and Moshe Y. Vardi. ProCount: Weighted projected model counting with graded project-join trees. In *Proc. of SAT*, 2021.

[EGSS12] Stefano Ermon, Carla P Gomes, Ashish Sabharwal, and Bart Selman. Uniform solution sampling using a constraint solver as an oracle. In *Proc. of UAI*, 2012.

[END$^+$18] P Ezudheen, Daniel Neider, Deepak D'Souza, Pranav Garg, and P Madhusudan. Horn-ICE learning for synthesizing invariants and contracts. In *Proc. of OOPSLA*, 2018.

[FG19] Grigory Fedyukovich and Aarti Gupta. Functional synthesis with examples. In *Proc. of CP*, 2019.

[FKB12] Andreas Fröhlich, Gergely Kovásznai, and Armin Biere. A DPLL algorithm for solving DQBF. *Proc. POS*, 2012.

[FKBV14] Andreas Fröhlich, Gergely Kovásznai, Armin Biere, and Helmut Veith. idq: Instantiation-based DQBF solving. In *Proc. of SAT*, 2014.

[FTV16] Dror Fried, Lucas M Tabajara, and Moshe Y Vardi. BDD-based boolean functional synthesis. In *Proc. of CAV*, 2016.

[Gar79] Michael R Garey. A guide to the theory of np-completeness. *Computers and intractability*, 1979.

[GLMN14] Pranav Garg, Christof Löding, P Madhusudan, and Daniel Neider. ICE: A robust framework for learning invariants. In *Proc. of CAV*, 2014.

[GLST05] Orna Grumberg, Flavio Lerda, Ofer Strichman, and Michael Theobald. Proof-guided underapproximation-widening for multi-process systems. In *Proc. of POPL*, 2005.

[GNMR16] Pranav Garg, Daniel Neider, Parthasarathy Madhusudan, and Dan Roth. Learning invariants using decision trees and implication counterexamples. In *Proc. of POPL*, 2016.

[Gre81] Cordell Green. Application of theorem proving to problem solving. In *Readings in Artificial Intelligence*. 1981.

[GRM] Priyanka Golia, Subhajit Roy, and Kuldeep S Meel. Good enough synthesis with hard constraints. *Under Review*.

[GRM20] Priyanka Golia, Subhajit Roy, and Kuldeep S. Meel. Manthan: A data driven approach for boolean function synthesis. In *Proc. of CAV*, 2020.

[GRM21] Priyanka Golia, Subhajit Roy, and Kuldeep S. Meel. Program synthesis as dependency quantified formula modulo theory. In *Proc. of IJCAI*, 2021.

[GRM23] Priyanka Golia, Subhajit Roy, and Kuldeep S. Meel. Synthesis with explicit dependencies. In *Proc. of DATE*, 2023.

[GRS$^+$13] Karina Gitina, Sven Reimer, Matthias Sauer, Ralf Wimmer, Christoph Scholl, and Bernd Becker. Equivalence checking of partial designs using dependency quantified boolean formulae. In *Proc. of ICCD*, 2013.

[GSCM21] Priyanka Golia, Mate Soos, Sourav Chakraborty, and Kuldeep S. Meel. Designing samplers is easy: The boon of testers. In *Proc. of FMCAD*, 2021.

[GSRM19] Rahul Gupta, Shubham Sharma, Subhajit Roy, and Kuldeep S Meel. WAPS: Weighted and projected sampling. In *Proc. of TACAS*, 2019.

[GSRM21] Priyanka Golia, Friedrich Slivovsky, Subhajit Roy, and Kuldeep S. Meel. Engineering an efficient boolean functional synthesis engine. In *Proc. of ICCAD*, 2021.

[GWR$^+$15] Karina Gitina, Ralf Wimmer, Sven Reimer, Matthias Sauer, Christoph Scholl, and Bernd Becker. Solving DQBF through quantifier elimination. In *Proc. of DATE*, 2015.

[Hen61] Leon Henkin. Some remarks on infinitely long formulas, infinitistic methods (proc. sympos. foundations of math., warsaw, 1959), 1961.

[HLR12] Jiawei Huang, John Lach, and Gabriel Robins. A methodology for energy-quality tradeoff using imprecise hardware. In *Proc. of DAC*. IEEE, 2012.

[HPSS18] Holger H Hoos, Tomáš Peitl, Friedrich Slivovsky, and Stefan Szeider. Portfolio-based algorithm selection for circuit QBFs. In *Proc. of CP*, 2018.

[HQSW20] Kangjing Huang, Xiaokang Qiu, Peiyuan Shen, and Yanjun Wang. Reconciling enumerative and deductive program synthesis. In *Proc. of PLDI*, 2020.

[HSB14] Marijn JH Heule, Martina Seidl, and Armin Biere. Efficient extraction of skolem functions from QRAT proofs. In *Proc. of FMCAD*, 2014.

[IMM18] Alexey Ignatiev, Antonio Morgado, and Joao Marques-Silva. PySAT: A Python toolkit for prototyping with SAT oracles. In *Proc. of SAT*, 2018.

[Jan18a] Mikoláš Janota. Circuit-based search space pruning in QBF. In *Proc. of SAT*. Springer, 2018.

[Jan18b] Mikoláš Janota. Towards generalization in QBF solving via machine learning. In *Proc. of AAAI*, 2018.

[JBS$^+$07] Toni Jussila, Armin Biere, Carsten Sinz, Daniel Kröning, and Christoph M Wintersteiger. A first step towards a unified proof checker for QBF. In *Proc. of SAT*, 2007.

[JKL20] Jie-Hong R. Jiang, Victor N. Kravets, and Nian-Ze Lee. Engineering change order for combinational and sequential design rectification. In *Proc. of DATE*, 2020.

[JMF14] Satoshi Jo, Takeshi Matsumoto, and Masahiro Fujita. SAT-based automatic rectification and debugging of combinational circuits with lut insertions. *Proc. of IPSJ T-SLDM*, 2014.

[JMS11] Mikoláš Janota and Joao Marques-Silva. Abstraction-based algorithm for 2QBF. In *Proc. of SAT*, 2011.

[JSC$^+$15] Ajith K John, Shetal Shah, Supratik Chakraborty, Ashutosh Trivedi, and S Akshay. Skolem functions for factored formulas. In *Proc. of FMCAD*, 2015.

[KHDR21] Jinwoo Kim, Qinheping Hu, Loris D'Antoni, and Thomas Reps. Semantics-guided synthesis. In *Proc. of POPL*, 2021.

[KMD$^+$22] Pankaj Kumar Kalita, Sujit Kumar Muduli, Loris D'Antoni, Thomas Reps, and Subhajit Roy. Synthesizing abstract transformers. In *Proc. of OOPSLA*, 2022.

[Knu15] Donald E Knuth. *The art of computer programming, Volume 4, Fascicle 6: Satisfiability.* Addison-Wesley Professional, 2015.

[Kol32] Andrei N Kolmogorov. „zur deutung der intuitionistischen logik mathematische zeitschrift35. *English translation in VM Tikhomirov (ed.) Selected Works of AN Kolmogorov*, 1932.

[KS92] Henry A Kautz and Bart Selman. Planning as satisfiability. In *Proc. of ECAI*, 1992.

[KS00] James H Kukula and Thomas R Shiple. Building circuits from relations. In *Proc. of CAV*, 2000.

[L10] Leopold Löwenheim. Über die auflösung von gleichungen im logischen gebietekalkul. *Mathematische Annalen*, 1910.

[LB10] Florian Lonsing and Armin Biere. DepQBF: A dependency-aware QBF solver. *Proc. of JSAT*, 2010.

[LE17] Florian Lonsing and Uwe Egly. Depqbf 6.0: A search-based QBF solver beyond traditional QCDCL. In *Proc. of CADE*, 2017.

[LG] B. Logic and V Group. ABC: A system for sequential synthesis and verification. Available at http://www.eecs.berkeley.edu/~alanmi/abc/.

[LM08] Jérôme Lang and Pierre Marquis. On propositional definability. *Artificial Intelligence*, 2008.

[McG10] Ann C McGinley. Ricci v. destefano: A masculinities theory analysis. *Harv. JL & Gender*, 2010.

[MM00] Fabio Massacci and Laura Marraro. Logical cryptanalysis as a SAT problem. *Journal of Automated Reasoning*, 2000.

[MML14] Ruben Martins, Vasco Manquinho, and Inês Lynce. Open-WBO: A modular MaxSAT solver. In *Proc. of SAT*, 2014.

[MSLM09] Joao Marques-Silva, Inês Lynce, and Sharad Malik. Conflict-driven clause learning SAT solvers. In *Handbook of satisfiability*. 2009.

[MW71] Zohar Manna and Richard J Waldinger. Toward automatic program synthesis. *Communications of the ACM*, 1971.

[NLBR14] Kumud Nepal, Yueting Li, R Iris Bahar, and Sherief Reda. Abacus: A technique for automated behavioral synthesis of approximate computing circuits. In *Proc. of DATE*, 2014.

[NPL$^+$12] Aina Niemetz, Mathias Preiner, Florian Lonsing, Martina Seidl, and Armin Biere. Resolution-based certificate extraction for QBF. In *Proc. of SAT*, 2012.

[PMNS19] Saswat Padhi, Todd D. Millstein, Aditya V. Nori, and Rahul Sharma. Overfitting in synthesis: Theory and practice. In *Proc. of CAV*, 2019.

[PP20] Hila Peleg and Nadia Polikarpova. Perfect is the enemy of good: Best-effort program synthesis. *Leibniz international proceedings in informatics*, 2020.

[PRA01] Gary Peterson, John Reif, and Salman Azhar. Lower bounds for multiplayer noncooperative games of incomplete information. *Computers & Mathematics with Applications*, 2001.

[PSY18] Hila Peleg, Sharon Shoham, and Eran Yahav. Programming not only by example. In *Proc. of ICSE*, 2018.

[PVG$^+$] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *the Journal of machine Learning research*.

[qbfa] QBF solver evaluation portal 2017. Available at http://www.qbflib.org/qbfeval17.php.

[qbfb] QBF solver evaluation portal 2018. Available at http://www.qbflib.org/qbfeval18.php.

[qbfc] QBF solver evaluation portal 2020. Available at http://www.qbflib.org/qbfeval20.php.

[Qui86] J. Ross Quinlan. Induction of decision trees. *Proc. of Machine learning*, 1986.

[Rab17] Markus N Rabe. A resolution-style proof system for DQBF. In *Proc. of SAT*, 2017.

[Rab19] Markus N Rabe. Incremental determinization for quantifier elimination and functional synthesis. In *Proc. of CAV*, 2019.

[RDK$^+$15] Andrew Reynolds, Morgan Deters, Viktor Kuncak, Cesare Tinelli, and Clark Barrett. Counterexample-guided quantifier instantiation for synthesis in SMT. In *Proc. of CAV*, 2015.

[RSS21] Franz-Xaver Reichl, Friedrich Slivovsky, and Stefan Szeider. Certified DQBF solving by definition extraction. In *Proc. of SAT*, 2021.

[RT15] Markus N. Rabe and Leander Tentrup. CAQE: A certifying QBF solver. In *Proc. of FMCAD*, 2015.

[RTRS18] Markus N Rabe, Leander Tentrup, Cameron Rasmussen, and Sanjit A Seshia. Understanding and extending incremental determinization for 2QBF. In *Proc. of CAV*, 2018.

[SAC$^+$20] Ilaria Scarabottolo, Giovanni Ansaloni, George A Constantinides, Laura Pozzi, and Sherief Reda. Approximate logic synthesis: A survey. *Proceedings of the IEEE*, 2020.

[SGF13] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S Foster. Template-based program verification and program synthesis. *Proc. of STTT*, 2013.

[SGM20] Mate Soos, Stephan Gocht, and Kuldeep S. Meel. Tinted, detached, and lazy CNF-XOR solving and its applications to counting and sampling. In *Proc. of CAV*, 2020.

[SGRM18] Shubham Sharma, Rahul Gupta, Subhajit Roy, and Kuldeep S Meel. Knowledge compilation meets uniform sampling. In *Proc. of LPAR*, 2018.

[Sìč20] Juraj Sìč. Satisfiability of DQBF using binary decision diagrams. Master's thesis, Masaryk University, 2020. Available at https://is.muni.cz/th/prexv/.

[skl] sklearn.tree.decisiontreeclassifier. Available at https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html.

[Sli20] Friedrich Slivovsky. Interpolation-based semantic gate extraction and its applications to QBF preprocessing. In *Proc. of CAV*, 2020.

[SM19] Mate Soos and Kuldeep S Meel. Bird: Engineering an efficient CNF-XOR sat solver and its applications to approximate model counting. In *Proc. of the AAAI*, 2019.

[Soo19] Mate Soos. msoos/cryptominisat, 2019. Available at https://github.com/msoos/cryptominisat.

[SRSM19] Shubham Sharma, Subhajit Roy, Mate Soos, and Kuldeep S. Meel. Ganak: A scalable probabilistic exact model counter. In *Proc. IJCAI*, 2019.

[SS10] Marko Samer and Stefan Szeider. Algorithms for propositional model counting. *Journal of Discrete Algorithms*, 2010.

[syg19] SyGuS: syntax-guided synthesis competition, 2019. Available at https://sygus.org/comp/2019/.

[TR19] Leander Tentrup and Markus N Rabe. Clausal abstraction for DQBF. In *Proc. of SAT*, 2019.

[TV17] Lucas M Tabajara and Moshe Y Vardi. Factored boolean functional synthesis. In *Proc. of FMCAD*, 2017.

[URD$^+$13] Abhishek Udupa, Arun Raghavan, Jyotirmoy V Deshmukh, Sela Mador-Haim, Milo MK Martin, and Rajeev Alur. TRANSIT: specifying protocols with concolic snippets. *ACM SIGPLAN Notices*, 2013.

[VR17] Sahil Verma and Subhajit Roy. Synergistic debug-repair of heap manipulations. In *Proc. of ESEC/FSE*, 2017.

[WRMB17] Ralf Wimmer, Sven Reimer, Paolo Marin, and Bernd Becker. Hqspre–an effective preprocessor for QBF and DQBF. In *Proc. of TACAS*, 2017.

[WWSB16] Karina Wimmer, Ralf Wimmer, Christoph Scholl, and Bernd Becker. Skolem functions for QBF. In *Proc. of ATVA*, 2016.

[XHHLB08] Lin Xu, Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Satzilla: portfolio-based algorithm selection for SAT. *Proc. of JAIR*, 2008.

[XMK15] Qiang Xu, Todd Mytkowicz, and Nam Sung Kim. Approximate computing: A survey. *IEEE Design & Test*, 2015.

[ZS13] Sai Zhang and Yuyin Sun. Automatically synthesizing SQL queries from input-output examples. In *Proc. of ASE*, 2013.