Research Statement

Swarat Chaudhuri

September 21, 2020

1 Research Goals

Machine learning plays an outsized role in the world we live in. The last decade has seen remarkable progress on intelligent systems that learn from data, in areas ranging from computer vision to natural language processing to robotics and control. A world in which systems powered by machine learning drive our cars, assist us in household tasks, pervade every sphere of our professional lives, and accelerate our scientific discoveries no longer seems far away.

With this increased stature come obligations. Most existing work on machine learning focuses on improving *performance* in algorithmic tasks such as image classification or natural language translation. However, given their impact on human lives, the intelligent systems of tomorrow must also be *reliable*, *transparent*, and *secure*. This means that a system must maintain some essential correctness invariants on unexpected or adversarial inputs. Statistical models that learn from experience must work well with traditional, human-designed software systems. Humans should be able to *understand* the decisions that a learning model makes and to *communicate their intent*, and essential knowledge about the world, to the model.

My current research agenda is to develop an approach to the design of intelligent systems, complementary to mainstream machine learning, that seeks to realize these goals. I am an established researcher in the areas of formal methods (FM) and programming languages (PL), with a deep background in algorithmic verification and synthesis of programs. Over the last few years, I have become interested in artificial intelligence and have come to believe that ideas from PL and FM can play a critical role in achieving the dream of reliable, transparent, and secure machine learning. However, for this to happen, we need to go beyond the straightforward application of PL/FM ideas to machine learning, and aim for a deeper synthesis of logical approaches from PL/FM and statistical methods from machine learning. My current research goal is to help create such a synthesis.

In more detail, my current agenda is grounded in the following beliefs.

1. A statistical model, and more generally an intelligent system, is ultimately a *program*. There are reasons to represent these programs, or at least significant facets of these programs, using high-level languages that have features such as compositionality and rich data types and support automatic reasoning about correctness.

2. Methods from *program verification* can be used to reason about reliability and security of intelligent systems. In particular, such methods can be used to analyze correctness properties, such as robustness, fairness, and privacy, of statistical models. They can also be used to reason about end-to-end correctness properties of intelligent systems that use statistical models as well as traditional software modules.

3. Machine learning can be viewed as a form of *program synthesis*. There is a large body of work on program synthesis in the PL and FM communities. Properly coupled with approaches from machine learning, these methods can lead to novel learning algorithms that, for example, learn compositionally, or modulo correctness properties that capture facets of reliability and security. Achieving these goals need not require a significant sacrifice in prediction accuracy. If anything, high-level programming langauges can be a helpful way of injecting human-held knowledge into a learning process, and this domain knowledge can *improve* a learner's performance.

To elaborate on the first point, the key difference between an intelligent system and a traditional program is that the former uses statistical models as components. However, a statistical model is also a function a.k.a. program, albeit one whose parameters are discovered automatically from examples. For example, a deep neural network is a program in a low-level, domain-specific programming language, in which real tensors are the only allowed data type, and program statements implement linear and nonlinear transformations applied to tensors.

However, there is no fundamental reason to exclusively stick to "low-level" programming primitives — such as neural networks — while describing models. Modern programming languages have numerous desirable features, for example compositional idioms for expressing computations and data structures, and type systems that tightly couple programming and reasoning about correctness. There are benefits to using such abstractions, in conjunction with statistical machinery such as neural networks, to express learning models. For example, while neural networks excel at perception tasks, symbolic programs often outperform them at higher-order algorithmic reasoning. One can imagine "hybrid" models in which neural modules are used for basic vision tasks such as object recognition, and symbolic abstractions are used to compose these modules into complex architectures that perform tasks that include perception as well as algorithmic reasoning.

Because they use higher-level symbolic abstractions, such hybrid models are likely to be more human-interpretable than purely neural models. The abstractions are also likely to lead to more compact models, simplifying automated symbolic reasoning about worst-case behavior. Also, symbolic, programmatic elements are a good way of introducing human-held prior knowledge into a learning process. Such symbolic priors can be especially beneficial in complex settings such as reinforcement learning or statistical generation of structured objects, in which purely statistical methods tend to perform poorly or unreliably, for example, by violating commonsense invariants, needing an unreasonable number of training samples, or failing to generalize to scenarios not seen during training. Finally, being compositional, programming language abstractions can facilitate the transfer of pretrained modules across learning settings.

The other points naturally follow from this view of *machine learning as a programming paradigm*. There is a vast literature on logics for specifying correctness of programs. Such logical notations, and corresponding methods for testing and verification, can also be used to specify, certify, and find counterexamples to properties of statistical models, such as fairness, robustness, and privacy.

Also, if statistical models are programs, then machine learning amounts to the automatic discovery, or *synthesis*, of programs that optimally fits a dataset of example inputs and outputs. The machine learning community views this problem through the lens of optimization and solves it, typically, using gradient-based algorithms. On the other hand, the PL and FM communities formulate the problem as a *search* for a program that satisfies a formal requirement. This search problem is solved using symbolic techniques for exploring combinatorial spaces. These two perspectives can be unified by framing synthesis as a *constrained optimization* problem, in which the goal is to find programs that satisfy a logical requirement and also optimize a quantitative objective, for example to fidelity to a data set.

Solving such constrained optimization problems requires a tight coupling of algorithmic ideas from PL/FM and machine learning. For instance, ideas in this space can include learning algorithms that invoke automatic reasoning engines (for example SMT-solvers) as subroutines, and algorithms for program and proof search that are guided by statistical models.

Developing a framework of *integrated learning and verification* that combines algorithmic ideas from PL/FM and machine learning in the above ways is the central thrust of my current research agenda. Over the last couple of years, I have begun to implement this vision through a series of papers. For example, in an ICML 2018 paper [22], we propose a version of reinforcement learning in which policies are represented as programs in a high-level functional language. These programmatic policies can express complex control operations but are easier to interpret and verify than the neural policies used in deep reinforcement learning. In subsequent papers [10, 21] published at ICML 2019 and NeurIPS 2019, we show that programmatic policies have advantages even from the point of view of learning performance. Specifically, programmatic policies can be used as a *regularizers* for neural policies, and "hybrid" policies that combine neural and symbolic elements can handle certain tasks that deep learning cannot handle and generalize better to new environments. In a NeurIPS 2018 paper [20], we represent neural architectures in a higher-order functional language and use the compositionality of this representation to transfer knowledge across learning tasks. The algorithms in these papers tightly couple ideas from deep learning and PL/FM, for instance using a refinement type system to prune a search over deep architectures [20], or using a program synthesizer to imitate a deep neural network [22].

Much of my work in "core" PL/FM is synergistic with the above agenda. Specifically, traditional approaches to program verification and synthesis expect the user to provide complete, logical specifications of the program to be verified or synthesized. This assumption is a serious limitation in the kinds of applications targeted by intelligent systems, which often cannot be fully specified in logic. For a long time, my work has sought to break away from this assumption by considering *quantitative* and *ambiguous* specifications. In particular, I have been studying versions of *quantitative program synthesis*, or constrained optimization over programs, for some time [2]. I have also studied *statistical program synthesis* [16], in which one statistically learns specifications and branching heuristics that drive a search for programs; *quantitative program analysis* [4, 6], which seeks to statically estimate various quantitative facets

of program behavior; and *statistical program analysis* [15], in which bug finding in programs is cast as statistical anomaly detection. Much of this work can be connected to problems in intelligent systems, such as verification of neural networks and learning modulo correctness constraints. I also have a large body of work [23, 11, 24, 17] on logic-based formal methods for robotics, a key category of intelligent systems.

2 Specific Projects

Now I describe a few of my research projects in more detail.

2.1 Program Synthesis

Automatic synthesis of programs from high-level specifications is an old problem that has had a resurgence in the last decade. In the traditional PL/FM view of the problem, the specification is a complete logical description of the desired program. More recent research has drawn on related work in the Artificial Intelligence literature and considered a wider range of specification notations, for example input-output examples and demonstrations.

I have worked on program synthesis since 2010. Over the years, my collaborators and I have developed a wide variety of tools for solving this problem. Some of these tools are grounded in logic and programming language theory. For example, we have shown how constraint-based techniques used for generating proofs of programs can be used to synthesize programs as well. We have also shown [13] that representing programs in a typed functional language, and pruning provably incorrect programs using automatic deduction, can help scale a search for programs. Finally, we have developed a body of approaches that combine logical and quantitative/statistical techniques. We have applied these ideas in many different domains, ranging from software engineering [16] to end-user-programming [13, 25, 12] to robotics [11, 24, 23] to artificial intelligence [22, 20].

Now I discuss in more detail my work on quantitative and statistical program synthesis.

Program Synthesis for Reliable and Interpretable Machine Learning The central thread of my current work explores program synthesis as an approach to reliable and interpretable machine learning. Much of machine learning concerns the discovery of functions that optimally fit a dataset (fidelity to data is defined by a quantitative objective). Our work seeks to learn representations of functions in *high-level, domain-specific programming languages*. This problem is a form of *quantitative program synthesis*: the task of discovering programs that are optimal according to a quantitative objective function, while also satisfying a Boolean correctness requirement. There are many benefits to solving this problem. First, a representation of learned knowledge in a higher-level language is more easily interpreted by humans. Second, provided the language is expressive enough, this interpretability does not come at the cost of expressiveness. Third, such representations are more amenable to automatic verification than, say, neural representations. Fourth, the syntactic and semantic constraints imposed by a programming language can ensure that the learner's inductive bias is consistent with human-held domain insights. Fifth, a programming language can be used to provide domain insights to a learning algorithm and to *regularize* the learning process. Such "programmatic" regularization can lead to learning algorithms that are less data-hungry and generalize more reliably. Finally, high-level languages permit *modular* representations, which can permit easier transfer of knowledge from one learning setting to another.

Concretely, we have been exploring these ideas in the setting of reinforcement learning (RL) in a series of recent papers. In an ICML 2018 paper [22], we posed a framework of "programmatic" RL, in which agent policies are represented as short programs in a domain-specific language (DSL). Our approach to this question, an algorithm called Neurally Directed Program Search (NDPS), uses deep RL to compute an initial approximation of the desired program, then uses this neural net as an "oracle" that guides program synthesis. We show that this approach can discover robust, interpretable policies that are easy to formally verify and clear some significant performance bars.

In a subsequent ICML 2019 paper [10], we gave a method for regularizing neural policies with programmatic policies. Specifically, we proposed a "neurosymbolic" policy class obtained through the linear combination of learned neural policies and a fixed programmatic policy, and showed that such policies can be learned more reliably than purely neural policies.

A NeurIPS 2019 paper [21] brings the ideas in these papers together through a meta-algorithm, called PROPEL, for learning programmatic policies. We observe that the because of the expressiveness of neural networks, the class of neural or neurosymbolic policies is a superset of the class of policies that have a compact representation in a given DSL. This means that the programmatic RL problem can be framed as that of finding a neurosymbolic policy

that is *constrained* to have a short programmatic representation. We approach this problem using a form of mirror descent, in which one performs gradient-based updates to a neurosymbolic policy, and periodically *projects* the current policy estimate into the programmatic space, i.e., finds a program that is behaviorally closest to the current policy. (We implement these projection steps using the NDPS algorithm.) Essentially, the PROPEL algorithm establishes a synergistic relationship between deep RL and program synthesis, using synthesized programs to regularize deep RL and using the gradients available to deep RL to improve the quality of synthesized programs. Not only are the programmatic policies synthesized this way easier to interpret and formally verify than neural policies learned using state-of-the-art deep RL algorithms, they also generalize better to new environments and are sometimes able to solve tasks that cannot be solved using deep RL.

A different direction of research, initiated in a NeurIPS 2018 paper [20], uses a typed, higher-order, functional language to represent neural models, and uses program synthesis to search over models. The programs in this language can invoke a set of neural networks as reusable "library functions", and use symbolic combinators to compose these networks into complex, but end-to-end differentiable, architectures. Learning happens in two stages: the first stage synthesizes a parameterized program, and the second trains this program using gradient descent. We show that ideas from logic-based program synthesis [13], such as the use of type-based pruning, can significantly aid this learning process. We also show that the reuse of library functions across learning settings can be an effective way of transferring knowledge across learning tasks.

Smooth Interpretation and Proof Search I have also studied quantitative program synthesis outside of artificial intelligence applications — indeed, I was among the first researchers in the PL/FM communities to work on this problem. My initial efforts on this topic investigated ways of using continuous optimization techniques like gradient descent to discover optimal parameters for programs (the application domain that we considered is that of control programs; however, the ideas are applicable to many other domains as well). The fundamental challenge with such an approach is that programs often represent highly discontinuous functions, and continuity/differentiability are basic requirements for continuous optimization. Our approach, called *smooth interpretation* and described in a series of PLDI and CAV papers [7, 8, 9], attacked this problem using a combination of symbolic and numerical techniques. More precisely, we gave a novel way of using symbolic program analysis to systematically approximate programs by smooth mathematical functions. These functions were then subjected to mathematical optimization. In a subsequent paper published at POPL 2014, we extended this work [2] to *smoothed proof search*, a method for discovering optimal program parameters that also lead the program to *provably* satisfy its Boolean requirements. This work introduced a notion of smooth parameterized relaxations to provers of program correctness (specifically, abstract interpreters). The search for verifiably correct program parameters was then cast as a problem of optimizing these relaxations.

In the years since this work, a number of efforts in the deep learning community have used gradient-based methods to learn programs and prove theorems. Our work was a precursor to these efforts.

Symbolic Quantitative Synthesis for Robotics Another body of work studies logic-based program synthesis algorithms for robotics. The objective here is to synthesize robot motions from high-level specifications. These specifications include models of the robot and its environment, a logical correctness constraint, and some quantitative objectives (for example energy use) that the robot's motion should optimize. In many cases the robot's environment can only be modeled probabilistically; in this case the correctness constraints are also probabilistic.

The primary challenge in designing synthesis algorithms for robotics is that such an algorithm must simultaneously reason about two different levels of abstraction: a discrete *task level* that is concerned with the high-level goals of the robot, and a continuous *motion level* that is concerned with navigating and manipulating a physical space. Our goal is to find a "policy" for the robot that meets correctness criteria for both levels. Our solutions to this problem consist of two classes of modules: those using symbolic techniques from FM to find policies that meet task-level requirements, and those using motion planning techniques to ensure that the found policies satisfy motion-level invariants. Modules at the two levels communicate through a shared abstraction until they converge to a program that both consider acceptable. Concrete instantiations of this approach include a method for *incremental* synthesis of integrated task and motion policies [11], integrated task and motion planning modulo domain knowledge specified in programmatic notations [17], policy synthesis for robots that execute in adversarial environments [24], and policy synthesis for robots whose knowledge of their environment is modeled probabilistically [23].

Statistical Program Synthesis from Ambiguous Specifications Program synthesis from ambiguous specifications is a major thread of my recent work. Specifically, in a paper published at ICLR 2018 [16], we present a framework for synthesizing idiomatic Java code, given a "specification" that solely consists of a small amount of syntactic information about the code or some natural language text. The end application for this problem can be summarized as "automating Stack Overflow": generating examples of how to solve everyday programming tasks given a query in a "natural-ish" language.

This problem is beyond the scope of traditional program synthesis, which has no principled mechanism for generalizing an ambiguous specification. Neither is it easy to synthesize programs using statistical generative models trained on real-world code: programs are highly structured objects that contain many low-level details that impede learning, and that must satisfy rich constraints like type safety that statistical models find hard to guarantee. In contrast, our solution synergistically combines symbolic and statistical methods. We train a deep generative model that, given a label, can predict sketches — or syntactic abstractions that carry just enough, but not too much, information about what a program looks like — of programs that are likely to match the label. These predicted sketches are then concretized into type-safe code using symbolic synthesis. This work was embodied in a highly visible open-source system, called BAYOU (available online at `http://www.askbayou.com`), and was featured in a number of mainstream news outlets. We also gave a tutorial on this framework at the PLDI conference. In current work, we are extending the framework to handle richer forms of ambiguous specifications, for example natural language statements, and investigating scenarios in which ambiguous specifications coexist with specifications written in formal logic.

2.2 Program Analysis

I have been working on program analysis since my Ph.D. years. I have worked on quantitative program analysis since the beginning of my first faculty appointment, in 2008.

Program Analysis for Adversarially Robust Machine Learning I have recently become interested in analyzing correctness of statistical models that learn from data, for example neural networks and probabilistic programs. Such "artificially intelligent" systems, increasingly ubiquitous, have certain unique correctness concerns. Neural networks are known to be susceptible to *adversarial examples* — small, carefully crafted perturbations to input data that cause the network to output incorrect labels. Such adversarial examples can pose security threats for AI systems such as face recognizers and spam filters. AI systems also have other problems: they can behave *unfairly*, for instance by discriminating on basis of protected attributes like race, and they can leak private information. My research seeks to automatically discover such violations and to prove their absence. This requires quantitative reasoning, as the correctness properties in question are statistical.

My work on this topic treats statistical models as programs in a specialized language, and develops program analysis techniques to reason about their correctness. The technical problems here are closely related to *robustness analysis of programs*, a topic to which I have made substantial contributions (more on this soon). In particular, in a paper at the 2018 IEEE S&P conference [14], we use abstract interpretation to show that a neural network is provably robust to all adversarial perturbations up to a bound. An subsequent paper, winner of a Distinguished Paper Award at PLDI 2019 [1], explores ways to combine optimization and verification techniques to scale this kind of analysis. Future work will extend these efforts to larger systems and consider correctness properties such as fairness.

Statistical Estimation of Program Correctness Another thread of my recent work develops a new generation of formal methods that can handle uncertainty in specification. In the setting of program analysis, this means discovering likely errors, and quantifying a program's likely correctness, without a complete definition of a program's requirements and assumptions. Our efforts along these lines cast bug finding in programs in terms of *statistical anomaly detection*. Specifically, in an FSE 2017 paper [15], we build a Bayesian statistical model that correlates the syntactic "shape" of a program with its likely behavior. This model is trained on data extracted from a large, real-world code corpus through program analysis. Given the task of analyzing a new program, we compute a statistical distance between the distribution over behaviors that the model expects from the program, and the distribution that the behaviors of the program actually follow. Is this distance is high, then the program significantly deviates from the norms followed in the corpus, and is therefore more likely to be erroneous. In current work, we are extending these ideas to settings in which statistically learned specifications coexist with logical specifications.

Reasoning about Robustness Much of my early work on quantitative program analysis was on techniques for analyzing a program's *robustness to uncertain inputs*. Such inputs — for example noisy sensor readings and results of inexact numerical computations — are common in domains such as robotics and scientific computing. Realistically, one cannot expect a program to behave correctly no matter how uncertain its inputs are. A more reasonable requirement is *robustness*: "*The program's behavior is close to its specification when its inputs are perturbed within reason*" (for a suitable definition of "close" and "within reason"). The corresponding program analysis questions are to automatically verify and discover quantitative bounds on a program's robustness, and to automatically identify violations of robustness.

My collaborators and I were among the first to study robustness analysis of software. In a POPL 2010 paper [4], we proposed a formulation of robustness in terms of mathematical continuity, and gave an algorithm for verifying that a program is robust in this sense. A followup paper, published at FSE 2011 [6], defined robustness in terms of Lipschitz-continuity, and gave a method to compute a provable bound on the Lipschitz constant of a program. The analysis algorithms in these papers combined ideas from numerical error analysis and logic-based reasoning, using the former to quantify uncertainty in a program's data, and the latter to reason about the effects of uncertainty on a program's control flow. These papers have proved to be influential. The FSE 2011 paper [6], in particular, received a Distinguished Paper Award and was featured as a research highlight in *Communications of the ACM* [5]. Our subsequent work on this topic explored a number of other robustness properties, adapting definitions previously studied in the context of continuous dynamical systems and geometric algorithms [19, 18, 3]. Our algorithms for analyzing these properties used tools variously drawn from automata theory, symbolic program verification, and numerical analysis.

3 Summary

We live in an era in which Panglossian optimism about intelligent systems is punctuated by fears of these systems behaving badly. The fears are not entirely unfounded: while the predictive performance of learning algorithms has grown by leaps and bounds in the recent past, the challenge of building intelligent systems that are also reliable, transparent, and secure is very far from solved. I believe that a view of *machine learning as a programming paradigm*, and an algorithmic agenda of *integrating learning and verification*, can go a long way towards addressing this challenge. This belief has driven much of my recent research, and will continue to motivate my work in the foreseeable future.

I am particularly excited by these directions because they bring together, in a new way, the two grand research traditions of automated reasoning and optimization/machine learning. Artificial Intelligence researchers have dreamed about reconciling symbolic reasoning and inductive learning for decades, and the agenda I describe can be seen to be a continuation of that dream. Recent algorithmic advances in machine learning and PL/FM, the powerful hardware that we now have at our disposal, and new applications, including ones that I have described in this document, make the current moment an especially good time to pursue such an agenda. Even modest progress along these directions can influence multiple disciplines. For example, I have published research related to these topics at machine learning conferences such as ICLR, ICML, and NeurIPS, security conferences such as IEEE S&P, and FM, PL, and software engineering conferences like CAV, PLDI, POPL, and FSE. I expect great strides in this research area over the next few years, and hope to continue to play a leading role in it.

References

[1] Greg Anderson, Shankara Pailoor, Isil Dillig, and Swarat Chaudhuri. Optimization and abstraction: a synergistic approach for analyzing neural network robustness. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 731–744, 2019.

[2] Swarat Chaudhuri, Martin Clochard, and Armando Solar-Lezama. Bridging boolean and quantitative synthesis using smoothed proof search. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 207–220, 2014.

[3] Swarat Chaudhuri, Azadeh Farzan, and Zachary Kincaid. Consistency analysis of decision-making programs. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 555–568, 2014.

[4] Swarat Chaudhuri, Sumit Gulwani, and Roberto Lublinerman. Continuity analysis of programs. In *37th ACM Symposium on Principles of Programming Languages (POPL)*, 2010.

[5] Swarat Chaudhuri, Sumit Gulwani, and Roberto Lublinerman. Continuity and robustness of programs. *Commun. ACM*, 55(8):107–115, 2012.

[6] Swarat Chaudhuri, Sumit Gulwani, Roberto Lublinerman, and Sara NavidPour. Proving programs robust. In *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*, pages 102–112, 2011.

[7] Swarat Chaudhuri and Armando Solar-Lezama. Smooth interpretation. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*, pages 279–291, 2010.

[8] Swarat Chaudhuri and Armando Solar-Lezama. Smoothing a program soundly and robustly. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, pages 277–292, 2011.

[9] Swarat Chaudhuri and Armando Solar-Lezama. Euler: A system for numerical optimization of programs. In *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, pages 732–737, 2012.

[10] Richard Cheng, Abhinav Verma, Gábor Orosz, Swarat Chaudhuri, Yisong Yue, and Joel Burdick. Control regularization for reduced variance reinforcement learning. In *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, pages 1141–1150, 2019.

[11] Neil T. Dantam, Zachary K. Kingston, Swarat Chaudhuri, and Lydia E. Kavraki. Incremental task and motion planning: A constraint-based approach. In *Robotics: Science and Systems XII*, 2016.

[12] Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. Component-based synthesis of table consolidation and transformation tasks from examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 422–436, 2017.

[13] John K. Feser, Swarat Chaudhuri, and Isil Dillig. Synthesizing data structure transformations from input-output examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 229–239, 2015.

[14] Timon Gehr, Matthew Mirman, Dana Drachsler-Cohen, Petar Tsankov, Swarat Chaudhuri, and Martin Vechev. Ai^2: Safety and robustness certification of neural networks with abstract interpretation. In *Security and Privacy (SP), 2018 IEEE Symposium on*, 2018.

[15] Vijayaraghavan Murali, Swarat Chaudhuri, and Chris Jermaine. Bayesian specification learning for finding API usage errors. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, pages 151–162, 2017.

[16] Vijayaraghavan Murali, Swarat Chaudhuri, and Chris Jermaine. Neural sketch learning for conditional program generation. In *International Conference for Learning Representations (ICLR)*, 2018.

[17] Srinivas Nedunuri, Sailesh Prabhu, Mark Moll, Swarat Chaudhuri, and Lydia E. Kavraki. Smt-based synthesis of integrated task and motion plans from plan outlines. In *2014 IEEE International Conference on Robotics and Automation, (ICRA)*, pages 655–662, 2014.

[18] Roopsha Samanta, Jyotirmoy V. Deshmukh, and Swarat Chaudhuri. Robustness analysis of networked systems. In *Verification, Model Checking, and Abstract Interpretation, 14th International Conference, VMCAI 2013, Rome, Italy, January 20-22, 2013. Proceedings*, pages 229–247, 2013.

[19] Roopsha Samanta, Jyotirmoy V. Deshmukh, and Swarat Chaudhuri. Robustness analysis of string transducers. In *Automated Technology for Verification and Analysis - 11th International Symposium, ATVA 2013, Hanoi, Vietnam, October 15-18, 2013. Proceedings*, pages 427–441, 2013.

[20] Lazar Valkov, Dipak Chaudhari, Akash Srivastava, Charles A. Sutton, and Swarat Chaudhuri. Synthesis of differentiable functional programs for lifelong learning. In *Neural Information Processing Systems (NeurIPS)*, 2018.

[21] Abhinav Verma, Hoang M. Le, Yisong Yue, and Swarat Chaudhuri. Imitation-projected programmatic reinforcement learning. In *Neural Information Processing Systems (NeurIPS)*, 2019.

[22] Abhinav Verma, Vijayaraghavan Murali, Rishabh Singh, Pushmeet Kohli, and Swarat Chaudhuri. Programmatically interpretable reinforcement learning. In *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, pages 5052–5061, 2018.

[23] Yue Wang, Swarat Chaudhuri, and Lydia E. Kavraki. Bounded policy synthesis for pomdps with safe-reachability objectives. In *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems, AAMAS 2018, Stockholm, Sweden, July 10-15, 2018*, pages 238–246, 2018.

[24] Yue Wang, Neil T. Dantam, Swarat Chaudhuri, and Lydia E. Kavraki. Task and motion policy synthesis as liveness games. In *Proceedings of the Twenty-Sixth International Conference on Automated Planning and Scheduling, ICAPS 2016, London, UK, June 12-17, 2016.*, page 536, 2016.

[25] Navid Yaghmazadeh, Christian Klinger, Isil Dillig, and Swarat Chaudhuri. Synthesizing transformations on hierarchically structured data. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, pages 508–521, 2016.