Research Statement

Mukund Raghothaman

My research is in the areas of program synthesis, formal verification, and static analysis. As a result of advances in verification technology, programmers are increasingly relying on static analysis tools to ensure code quality, not just in safety-critical domains such as avionics and medical devices, but also within companies such as Microsoft, Google, and Facebook. Famously, the SLAM project from Microsoft Research was used to verify safety properties of Windows device drivers, and the Infer static analyzer is being used by Facebook to verify properties their mobile apps. Verification and synthesis tools are also being used by end users: FlashFill is a popular feature recently added to Microsoft Excel which automatically synthesizes string processing macros, thereby easing the laborious task of spreadsheet data entry.

Despite these successes, important challenges remain: synthesis tools are challenging to design and cannot scale to large target programs and the design of program analyzers involves a notorious balancing act between scalability, false alarms, and missed bugs. My previous research has addressed these challenges in three ways:

- *First*, I have developed foundational **program synthesis** infrastructure including the SyGuS framework [FMCAD13], back-end solvers [TR17], user-facing synthesizers [HVC14, CAV15], and code search tools [ICSE16]. The SyGuS initiative has since evolved into a community, similar to SMT for verification, with diverse applications, many competing solvers, and an annual competition to benchmark improvements in synthesis technology.

- *Second*, I have worked on **combining logical and probabilistic methods** in program analysis tools that automatically learn from user feedback: the resulting system, Bingo, produces much fewer false alarms and is dramatically more effective in finding bugs [PLDI18, MLP18, TR18b]. Bingo was developed as part of the DARPA MUSE program, and has been subsequently selected for an extended grant so it can be transitioned for public use to the GitHub platform.

- *Finally*, I have designed new domain-specific languages and programming abstractions for **stream-processing systems** [ESOP16, Ths17], characterized their expressive power [LICS13, LICS14, TR18a], and designed fast resource-efficient evaluation algorithms [POPL15, PLDI17].

I will now elaborate on each of these contributions and outline my plans for future research.

Program Synthesis: Tools and Applications

Program synthesis is the process of producing executable code from diverse descriptions of user intent including logical specifications and input-output examples. The field has roots in the classical problem of formal verification, and raises the prospect of developing software that is correct-by-construction. Over the years, improvements in constraint solving technology have enabled the application of synthesis tools to problems such programming-by-examples, grading student programming assignments, and even in challenge problems such as automatic program repair.

Syntax-guided synthesis (SyGuS). Synthesis has classically been viewed as an instance of deductive theorem proving ("*for each input* x*, there exists an output* y *such that* $\varphi(x, y)$ *holds*"). However, its application always involves additional restrictions on the artifacts being synthesized: machine instructions in the case of code generation, specific arithmetic operations when generating invariants, certain primitives in the case of string transformations, etc. As a result, previous solvers were deeply integrated into their respective application domains, and researchers were forced to repeatedly design synthesizers from scratch. Solver development consequently required great expertise and time, and made benchmarking progress difficult.

These difficulties motivated us to formalize the *syntax-guided* synthesis problem: "*Generate an expression* $f(x)$ *from a context-free grammar* G*, such that for all inputs* x*,* $\varphi(x, f(x))$ *holds*" [FMCAD13, DSSE15]. Just as satisfiability modulo theory (SMT) provides a common framework to express and solve many different problems in verification, SyGuS provides a uniform structure for a large class of program synthesis problems, and decouples the tasks of application design and solver development.

The SyGuS competition (`http://www.sygus.org/`), organized annually since 2014, has enabled the community to make great advances in solver design. Simultaneously, domain experts have integrated synthesis technology into new applications, such as the implementation of constant-time cryptographic circuits, data wrangling systems, string transformations, and some projects from our own group in the design of distributed protocols [HVC14, CAV15].

Program synthesis with "*Big Code*." The emergence of large open-source code corpora such as GitHub and BitBucket has the potential to revolutionize the state-of-the-art in developer tooling. These repositories provide data to mine statistical patterns and coding idioms, and can serve as benchmarks for synthesis, repair, and verification technology. One exciting application of these repositories is as a source of API-related trivia, for e.g., "*How do I match a regular expression?*". This forms an important class of questions asked by developers when they move to new programming languages, frameworks and software projects. I have contributed to a system called SWIM ("Synthesize What I Mean") which combines coding idioms mined from GitHub with click-through data from the Bing search engine and answers API-related natural language queries with short idiomatic snippets of C# code [ICSE16].

In ongoing work, I am using similar statistical regularities in syntax to accelerate program repair in Leon, a verification and synthesis system for Scala [TR17]. All of these techniques involve casting the program synthesis problem as an instance of combinatorial search over a space of expressions: Inspired by the success of gradient descent and related numerical optimization techniques in machine learning, I am also investigating continuous program embeddings [TR18c]. By transforming the program synthesis problem into an appropriate instance of numerical optimization, we can accelerate program synthesis and also provide best-effort solutions when a perfect solution cannot be found.

Combining Logical and Probabilistic Methods in Program Reasoning

The ideal program analysis tool—inerrant, omniscient, and arbitrarily scalable—has the potential to greatly improve programmer productivity. Unfortunately, most analysis tasks are also undecidable, and the scale of software projects (from a few thousand to a few million lines of code) imposes strict complexity requirements on potential analysis algorithms. The job of the analysis designer then consists principally of devising trade-offs between scalability, frequency of false positives, and the possibility of missed bugs. In several recent projects, I have designed techniques to improve analysis accuracy by incorporating user feedback [PLDI18, TR18b], through data-driven analysis design [TR18c], and by synthesizing correctness proofs using reinforcement learning and graph neural networks [NeurIPS18].

Automatically generalizing from user feedback. Existing analysis tools typically run in a so-called *batch mode*, where they analyze the program and output a set of alarms. These alarms share portions of their derivation trees, so that multiple bugs often have the same root cause, and multiple false alarms are caused by the tool being unable to prove some shared intermediate fact about the program. Motivated by these correlations, we developed a tool called Bingo [PLDI18] which, instead of running in the batch mode, engages in an interaction loop with the user: as the user triages each alarm and reports its ground truth, Bingo responds by suppressing or prioritizing the remaining alarms in the program.

Essentially, Bingo quantifies the incompleteness of each analysis rule as the probability of its generating a false conclusion about the program. The derivation trees at fixpoint naturally induce a Bayesian network where the random variables indicate the event that the corresponding conclusion is true. This probabilistic model then allows us to rank alarms according to their probabilities, and respond to user interaction by computing conditional probabilities. In experiments on sophisticated analyses and real-world Java programs, Bingo reduced the alarm inspection burden by approximately 62%, and by up to 98% on certain benchmarks. In work presently in submission, we are extending these ideas to the setting of *continuous integration*, where the user repeatedly runs the analysis on different versions of the same program [TR18b]. This system, Drake, reduces the alarm inspection burden by an *additional* 50% over Bingo in its traditional interaction mode.

As part of a DARPA grant extension, I am currently mentoring a group of undergraduates integrate Bingo and Drake into the GitHub platform. We are also in the process of curating a corpus of historical bugs in open-source software: this will not only inform our future research, but also provide a metric by which the community can measure progress in bug finding technology.

Synthesizing loop invariants by reinforcement learning. One of the hardest parts of reasoning about imperative programs is the construction of loop invariants. Most previous techniques either look for invariants of a specific syntactic form, or use hand-crafted features in searching for the correct invariant. On the other hand, programs can be naturally represented as graphs, such as abstract syntax trees or control flow graphs. Inspired by the success of graph neural networks in combinatorial optimization, we have recently developed Code2Inv [NeurIPS18], a reinforcement learning system which operates over the AST of the program: it iteratively concretizes the syntactic structure of the invariant, and learns from feedback provided by an automatic theorem prover. On a suite of benchmark problems from the previous literature, Code2Inv learns invariants for more problem instances than the previous state-of-the-art, makes fewer queries to the expensive theorem prover, and successfully transfers knowledge from one program to the next.

Programming Abstractions for Processing Data Streams

My Ph.D. thesis [Ths17] was on the topic of processing data streams of the kind which arise from sources such as sensors, medical devices, program logs, financial markets and while managing network traffic. In these settings, runtime monitoring, verification and decision making often involve computing numerical summaries of the data, such as the number of occurrences of a pattern or the mean time between occurrences of an event. Given the quantity of data being processed, there are strict requirements over the amount of memory available and the time within which each subsequent element of the stream must be processed. There has traditionally been limited language support to express such computations and programmers are forced to write low-level imperative code to achieve their goals.

We proposed the model of quantitative regular expressions (QREs) to simplify this process [ESOP16]. Analogous to traditional regular expressions—whose operators allow patterns to be naturally described in a composable fashion—QREs provide a small collection of intuitive *combinators* by which a programmer may hierarchically express queries that simultaneously involve numerical computation and complex patterns of events in the stream being processed.

There are several challenges involved in compiling a QRE into an efficient streaming implementation. First, the underlying transducer model has to maintain numerical state corresponding to the partial results of query evaluation. Furthermore, unlike traditional regular expressions, different parse trees could lead to different numerical results, so it is essential also to verify the *unambiguity* of the pattern being parsed. By designing a type system to provide this guarantee statically, we were able to develop a single-pass evaluation algorithm whose per-element processing time and memory requirements are both polynomial in the size of the query and independent of the number of data items observed [POPL15, PLDI17].

Further continuing the parallels with regular language theory, we showed that QREs coincide in expressive power with the class of *regular cost functions* [LICS14, ESOP16], a robust class with many alternative representations, including cost register automata and as functions expressible in monadic second-order logic [LICS13].

In future work, I plan to study richer abstractions that simultaneously process multiple data streams, develop fast parallel evaluation algorithms, and apply them to problems arising in cyber-physical systems.

Future Directions: Intelligent Programming Systems

Looking ahead, I will focus my efforts on developing intelligent programming systems. I interpret this broadly as the use of various kinds of automatic reasoning, constraint solving, and optimization algorithms to improve programmer productivity.

Combining synthesis and learning. The fields of program synthesis and machine learning have, so far, largely developed in parallel. While they both aim to automatically learn programs or models from diverse forms of user intent, they traditionally use fundamentally different solution algorithms. By combining these paradigms, I see the opportunity to make foundational contributions to both areas.

The use of numerical techniques has the potential to greatly increase the scalability and applicability of program synthesis tools. The main application would be in cases where the specification is unrealizable, and the user simply wants a good approximation of the target function. Second, these techniques would enable learning functions that are difficult to express in a vectorial representation, including transformations over graphs, lists, and trees. Finally, this would also allow us to incorporate additional terms corresponding to energy efficiency, performance, or accuracy, thereby enabling new ways of applying program synthesis tools in approximate computing and energy-aware optimization. I also plan to explore connections to *differentiable programming languages* and devise highly scalable synthesis algorithms that employ hybrid search techniques.

In the other direction, the use of program synthesis and model finding procedures can inform fundamental challenges such as interpretability and the quest for explainable AI. In this context, I am part of an ongoing effort to formalize SyGuS 2.0, where I anticipate support for active learning of complex black-box functions, and the synthesis of optimum solutions for various classes of cost functions.

Data-driven program reasoning tools. I view Bingo [PLDI18, TR18b] as the first step in an over-arching program to combine deductive analysis tools with probabilistic and inductive methods [MLP18]. These approaches will allow us to classify alarms according to their anticipated severity or relevance, so that users have more information by which to focus their attention. This will also enable us to use the results of fuzz testing and dynamic analysis runs to prioritize static analysis alarms: in this sense, I envision unifying testing and verification into a single framework which automatically abduces interface guarantees, exhaustively verifies some components with respect to these properties, and subjects the remaining components to comprehensive testing to ensure conformance. Our ongoing efforts to curate a *Bug Genome*, a corpus of historical bugs in open-source projects, will also provide a rich source of training data with which to construct these tools.

References

[TR18a] Rajeev Alur, Dana Fisman, Konstantinos Mamouras, Mukund Raghothaman, and Caleb Stanford. Streamable regular transductions. Tech. rep. In submission. 2018. CoRR: `abs/1807.03865`.

[TR18b] Kihong Heo, Mukund Raghothaman, Xujie Si, and Mayur Naik. Continuous Program Reasoning via Differential Bayesian Inference. In submission. 2018.

[PLDI18] Mukund Raghothaman, Sulekha Kulkarni, Kihong Heo, and Mayur Naik. User-guided program reasoning using Bayesian inference. In: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI. ACM, 2018, pp. 722–735.

[MLP18] Mukund Raghothaman, Sulekha Kulkarni, Richard Zhang, Xujie Si, Kihong Heo, Woosuk Lee, and Mayur Naik. Beyond deductive methods in program analysis. In: *Machine Learning for Programming*. 2018.

[TR18c] Mukund Raghothaman, Xujie Si, Kihong Heo, and Mayur Naik. Difflog: Learning Datalog programs by continuous optimization. In submission. 2018.

[NeurIPS18] Xujie Si, Hanjun Dai, Mukund Raghothaman, Mayur Naik, and Le Song. Learning loop invariants for program verification. In: *Advances in Neural Information Processing Systems 31 (Spotlight; to appear)*. 2018.

[TR17] Manos Koukoutos, Mukund Raghothaman, Etienne Kneuss, and Viktor Kuncak. On repair with probabilistic attribute grammars. Tech. rep. In submission. 2017. CoRR: abs/1707.04148.

[PLDI17] Konstantinos Mamouras, Mukund Raghothaman, Rajeev Alur, Zachary Ives, and Sanjeev Khanna. StreamQRE: Modular specification and efficient evaluation of quantitative queries over streaming data. In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI. ACM, 2017, pp. 693–708.

[Ths17] Mukund Raghothaman. Regular programming over data streams. PhD thesis. University of Pennsylvania, 2017.

[ESOP16] Rajeev Alur, Dana Fisman, and Mukund Raghothaman. Regular programming for quantitative properties of data streams. In: *Programming Languages and Systems: 25th European Symposium on Programming*. ESOP. Springer, 2016, pp. 15–40.

[ICSE16] Mukund Raghothaman, Yi Wei, and Youssef Hamadi. SWIM: Synthesizing What I Mean. Code search and idiomatic snippet synthesis. In: *Proceedings of the 38th International Conference on Software Engineering*. ICSE. ACM, 2016, pp. 357–367.

[DSSE15] Rajeev Alur, Rastislav Bodik, Eric Dallal, Dana Fisman, Pranav Garg, Garvit Juniwal, Hadas Kress-Gazit, Madhusudan Parthasarathy, Milo Martin, Mukund Raghothaman, Shambwaditya Saha, Sanjit Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In: *Dependable Software Systems Engineering*. IOS Press, 2015, pp. 1–25.

[POPL15] Rajeev Alur, Loris D'Antoni, and Mukund Raghothaman. DReX: A declarative language for efficiently evaluating regular string transformations. In: *Proceedings of the 42nd Annual Symposium on Principles of Programming Languages*. POPL. ACM, 2015, pp. 125–137.

[CAV15] Rajeev Alur, Mukund Raghothaman, Christos Stergiou, Stavros Tripakis, and Abhishek Udupa. Automatic completion of distributed protocols with symmetry. In: *Proceedings of the 27th International Conference on Computer Aided Verification*. CAV. Springer, 2015, pp. 395–412.

[LICS14] Rajeev Alur, Adam Freilich, and Mukund Raghothaman. Regular combinators for string transformations. In: *Proceedings of the Joint Meeting of the 23rd Annual Conference on Computer Science Logic and the 29th Annual Symposium on Logic in Computer Science*. CSL-LICS. ACM, 2014, 9:1–9:10.

[HVC14] Rajeev Alur, Milo Martin, Mukund Raghothaman, Christos Stergiou, Stavros Tripakis, and Abhishek Udupa. Synthesizing finite-state protocols from scenarios and requirements. In: *Hardware and Software: Verification and Testing: Proceedings of the 10th International Haifa Verification Conference*. HVC. Springer, 2014, pp. 75–91.

[FMCAD13] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo Martin, Mukund Raghothaman, Sanjit Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In: *Formal Methods in Computer-Aided Design*. FMCAD. Extended version published as [DSSE15]. IEEE, 2013, pp. 1–8.

[LICS13] Rajeev Alur, Loris D'Antoni, Jyotirmoy Deshmukh, Mukund Raghothaman, and Yifei Yuan. Regular functions and cost register automata. In: *Proceedings of the 28th Annual Symposium on Logic in Computer Science*. LICS. IEEE, 2013, pp. 13–22.