Research Statement

John Sarracino

I am a programming languages and formal methods researcher. **My agenda is to build high-assurance certified systems and to make certified programming available to mainstream software developers.** Correctness is an overarching concern in software system development and it is increasingly important as software security becomes more critical. A powerful technique for building high-assurance systems is software development using a *proof assistant*. In this style of *certified programming*, a proof engineer implements their program in the assistant, writes a formal specification for the program within the assistant, and then *interactively* uses the assistant to *prove* that the program satisfies the specification. Certified programming provides a high level of assurance; program correctness depends only on the implementation of the proof assistant. But, the programmer must become sufficiently versed in the proof assistant and proofs require a significant amount of manual effort.

My current research is on developing *proof automation* for drastically reducing manual effort for common proofs. While proof assistants are not a realistic tool for software development today, in an increasingly security-critical world, formal machine-checked specifications must become as commonplace as linters, unit tests, and code documentation. **My research vision is to make certified software development as seamless and easy as current assurance techniques, such that developers can add and prove formal specifications as simply as they can add and check unit tests.** I approach this challenge with a human-computer interaction and programming languages background. I will first describe how my thesis work in human-computer interaction and program synthesis motivated my current research on formal methods and proof automation, and then I will conclude with some directions for future research.

Thesis work: synthesis for end-users. I developed several techniques for enabling end-users to create software without writing code. Along the way I gained an appreciation for the power of applied programming language techniques and the fundamental usability challenges of state-of-the-art techniques.

My first foray into applied programming languages was a technique for authoring interactive physical diagrams [CHI '17]. Prior to this work, interactive physical diagrams were in broad use by educators, but they required significant programming expertise to develop. This led to a mismatch between end-users of the tool, who desired material that they could adapt and customize to their curriculum, and the diagrams, which were inaccessible as many educators do not program. The key insight of the work is to represent the interactivity of these diagrams as a system of linear constraints. Using this representation, I developed an algorithm for searching over the space of constraint configurations, which is significantly smaller than the space of all possible diagrams. To make the search algorithm useable by educators, I developed a technique for *visualizing* the search; each candidate configuration is simulated and previewed, and the end-user simply needs to pick the desired configuration from a series of self-animating previews. I am particularly proud of this work. We evaluated the tool in the wild with a group of high-school and college educators. Most end-users could use the tool to quickly and easily develop diagrams.

For interactive diagrams, I traded off *expressiveness* of the technique for *automation*. While this technique is fully automated, there are many interaction modalities that it cannot express. This motivated me to explore *program synthesis*, which searches for much more expressive and complex programs (at the cost of intractability). I developed a technique for automatically generating dynamic web layouts from input-output examples of their behavior [FSE '21]. This work targets layout designers, who generally are skilled at building layout mockups, but for whom implementing the dynamic scaling logic for resizing to different screen layouts is difficult. The main contributions of our technique were to adapt SMT solvers to *noisy* input-output examples using Bayesian inference, and to *scale* layout inference up to realistic webpages through a novel divide-and-conquer algorithm, resulting in the ability to synthesize layouts with roughly 30X the size of previous work.

Postdoctoral work: formal methods and proof automation. My interest in formal methods developed after using state-of-the-art SMT solvers in the algorithm for webpage layout. I did an internship at Galois on formal methods for reasoning about parsers, which eventually lead to my first experiences with interactive theorem provers in which I developed a Coq mechanization of data-dependent parsers [LangSec '22]. The key idea is to generalize regular languages to include a *monadic bind* operator, which can express many common patterns in real-world parsers. This work was relatively theoretical and to broaden the impact of the work, I next collaborated with networking experts on reasoning about networking protocol parsers.

Networking parsing is performance critical and parser developers use very low-level abstractions that are close to the hardware model (such as P4 or eBPF). A common idiom is to take a simple parser for a format and *adapt* it to the hardware constraints of a switch, significantly transforming the structure of the parser to fully exploit switch-specific architectures. These transformations are done manually by programmers and automatically during parser compilation, and transformation correctness is crucial as parser errors are ripe sources of security

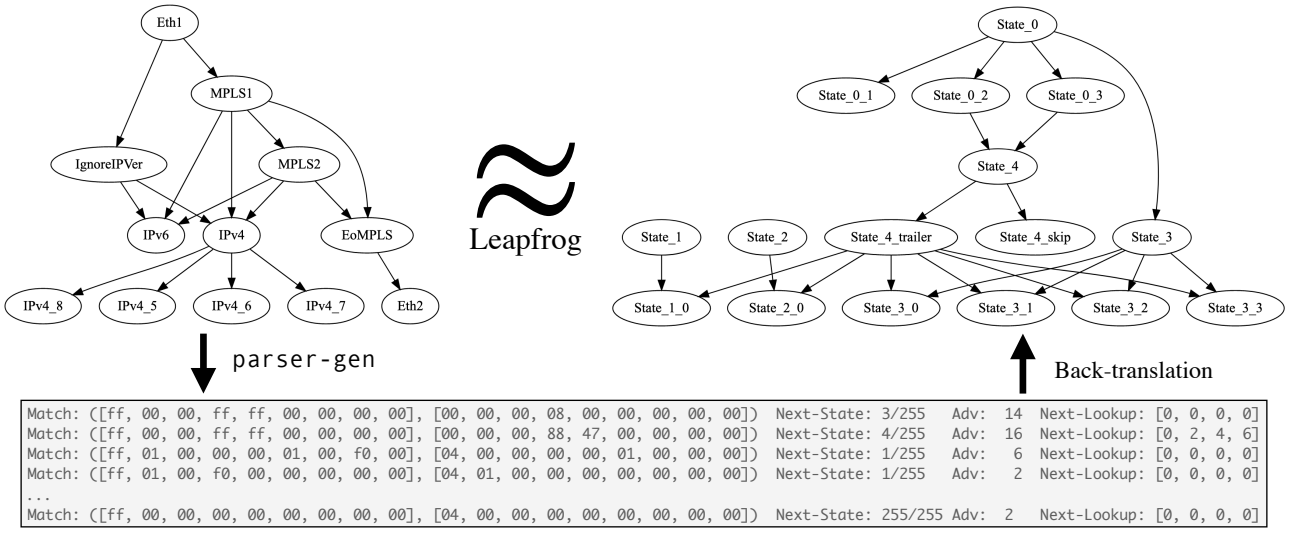

Figure 1: Translation validation case-study, in which we used a parser equivalence analysis (Leapfrog) to certify that an optimizing compiler (parser-gen) correctly generated the parser for a networking protocol.

vulnerabilities. I developed a parser analysis termed Leapfrog for automatically verifying the equivalence of two different protocol parsers [PLDI '22], i.e., checking the correctness of a parser transformation. The key insight is to represent protocol parsers using symbolic deterministic automata and the challenge is to formalize a classic automata equivalence-checking algorithm in a proof assistant. As a result we adapt prior theory to packet parsers and obtained an *automatic equivalence algorithm*. Moreover we can *prove* guarantees about the analysis so that end-users have a high-degree of confidence in the result. This way, we combined the best of both the formal methods and foundational verification, and we used our tool to certify correct the optimizing behavior of an experimental open-source protocol compiler (Figure 1).

The main challenge in Leapfrog is to handle the large state-space of protocol parsers, which is a challenge for both traditional automata equivalence algorithms and state-of-the-art tactics for solving bitvector verification conditions (VCs) in Coq. **My main contribution to Leapfrog is showing that our VCs are in a tractable fragment of SMT logic, and developing a systematic technique to soundly and robustly discharge the VCs to an off-the-shelf SMT solver.**

During Leapfrog development, I gained an appreciation for the ability for interactive theorem provers to prove very high-level, expressive specifications, such as the soundness of Leapfrog's weakest-precondition procedure for generating VCs. At the same time I also became frustrated at the proof burden required for building realistic certified systems. Manual equivalence proofs for even simple parsers took around a week of manual effort; by contrast, after we developed the proof automation, such proofs became completely push-button and finish in seconds. My most recent research, under preparation [Preprint 1], broadens Leapfrog's push-button proof automation from a domain-specific class of bitvector VCs to general Coq verification tasks.

This work, termed MirrorSolve, uses metaprogramming to move proof search from Coq to SMT solvers. To see why this is necessary, it is helpful to consider how proofs are written in Coq. Most Coq proofs are written using *tactics* which have three broad categories. First, decidable tactics completely solve goals within a small fragment of logic (such as omega/lia, ring, and congruence); such tactics are powerful but limited by their respective logics to the goals that they solve, and do not compose. For example a proof that requires reasoning about an inductive constructor and arithmetic will not be solveable by either congruence or lia alone. A second category leverages a particular class of SMT formulas and *reconstructs* proof terms from the output of the SMT solver (such as sniper/SMTCoq and itauto). These tactics are more flexible but are still limited to the set of SMT theories for which the respective algorithm can perform reconstruction. In particular the state-of-the-art SMTCoq is limited to prenex-normal quantified formulas over integers, booleans, and bitvectors. A third category does best-effort *search* for proofs using a combination of syntax-directed reasoning and heuristics (such as crush, hammer, and intuition). Such search uses decidable tactics at the leaves of a proof and is very flexible, but does not solve all goals.

MirrorSolve integrates modern SMT solvers, which can compose decidable reasoning between different theories, within the tactic language of Coq. In addition, I developed automation for generating the translation from Coq into first-order logic so that the user does not have to manually configure the entire verification condition. This approach requires some preparation and manual configuration in order to set up the SMT tactic, but in turn results in a powerful and flexible push-button tactic, solving a much broader set of first-order goals than the

state-of-the-art in search tactics (crush and hammer). The two technical challenges are soundly translating from a broad class of rich Coq Propositions and (possibly recursive) functions to SMT queries, and ensuring that the automation burden is low so that the tool is practically useful. Towards those ends I developed 1) *certified reflective metaprogramming* techniques so that only the underlying SMT queries are trusted, 2) a subset of polymorphic Coq inductive types that are amenable to SMT translation, and 3) metaprogramming techniques for building an SMT theory from a set of Coq inductive types and functions. For an example of the technique, please see our github repository at https://github.com/jsarracino/mirrorsolve.

Bench	Total	Crush	Hammer	MirrorSolve
SearchTree (helper)	27	13 (48%)	18 (67%)	26 (96%)
SearchTree (original)	24	3 (12%)	6 (25%)	24 (100%)
SearchTree (total)	51	16 (31%)	24 (47%)	50 (98%)
IntSets (helper)	1	0 (0%)	0 (0%)	1 (100%)
IntSets (original)	10	0 (0%)	0 (0%)	7 (70%)
IntSets (total)	11	0 (0%)	0 (0%)	8 (73%)
Maps (helper)	10	0 (0%)	2 (20%)	10 (100%)
Maps (original)	15	5 (33%)	8 (53%)	15 (100%)
Maps (total)	25	5 (20%)	10 (40%)	25 (100%)
Groups (helper)	3	0 (0%)	3 (100%)	3 (100%)
Groups (original)	13	1 (8%)	12 (92%)	13 (100%)
Groups (total)	16	1 (6%)	15 (94%)	16 (100%)
Aggregate (helper)	41	13 (32%)	23 (56%)	40 (98%)
Aggregate (original)	62	9 (15%)	26 (42%)	59 (95%)
Aggregate (total)	103	22 (21%)	49 (48%)	99 (96%)

Table 1: Proof power of SMT-powered proof automation common Coq inductive proofs. Proofs are split between the original properties of the library (original) and necessary helper lemmas for the proofs (helper).

Some preliminary results are given in Table 1. To evaluate the benefits of integrating SMT tactics in Coq, I collected a set of benchmarks from the Verified Functional Algorithms textbook (SearchTree), the CompCert certified C compiler (IntSets, Maps), and a group theory homework assignment from Certified Programming with Dependent Types (Groups). Most of these proofs use induction, so I evaluated the ability of MirrorSolve to prove the inductive sub-cases of proofs. I compared MirrorSolve against two state-of-the-art pushbutton proof search tactics: crush and hammer. A proof is marked as solved by a tactic if the tactic solves all of the inductive subgoals of the proof. I considered all of the proofs that were fundamentally first-order (or could be transformed to be first-order), which consisted of 62 out of 79 proofs.[*] Overall I found that current search tactics can solve roughly half of real-world Coq goals, while my SMT-powered automation of MirrorSolve can handle almost all proofs.

Future work

I am fascinated by three directions of future work: 1) better integration of SMT solvers with interactive theorem provers; 2) building towards *proof library synthesis*; and 3) building certified systems in domains that have traditionally been underserved by formal methods. My research agenda is to build certified systems and to use that experience to make certified systems development easier for mainstream programmers.

Better SMT Integration in Proof Assistants. In developing MirrorSolve and Leapfrog, I found that Coq (and other proof assistants) does not have a clean way to interface with external SMT solvers. This is important because solver-aided analyses (such as symbolic execution and program synthesis) are increasingly popular but cannot be easily developed in a proof assistant. At best such analyses are modelled *axiomatically*, enabling verification of high-level properties, but precluding other Coq programs and proofs from using the analyses. In MirrorSolve, I developed a powerful technique for seamlessly converting Coq goals into SMT queries. I now want to develop the reverse: to seamlessly translate from an SMT model or refutation back to a Coq term or proof, and to design an interface around SMT theories to enable the implementation of modern solver-aided programs. One path forward is to use Coq inductive types to represent the results of a solver call so that each call to an SMT solver is guarded by a tactic application. In this way a CEGIS loop might be implemented using Coq's repeat primitive and several novel SMT tactics, with the result of the loop saved as a *constructive proof* witnessing the steps of the algorithm.

Proof Library Synthesis Would it not be great if proof assistants did not require manual proofs at all? Imagine a world in which a programmer implements a functional program, writes a formal specification for that program, and then a synthesizer *generates* a proof of that specification, as well as a *library* of helper lemmas (and proofs)

[*]Due to performance constraints I have evaluated only the first half of the Maps file. The entire Maps file has 84 proofs in total of which 74 are fundamentally first-order.

necessary for the proof. While proof library synthesis is currently far beyond the reach of state-of-the-art proof automation, there have been some promising advances that bring it closer to reality. There is a long line of work on *neural proof synthesis*, which use neural networks to *synthesize* a proof script. These techniques are promising but currently do not solve more than a quarter of the proofs in a realistic development. This is because they are fundamentally limited by the strongest push-button tactic that is available to the proof search; for example, the Proverbot9000 and TakTok synthesizers perform better when they have access to CoqHammer.

My work on MirrorSolve is a leap forward in proof automation as compared to CoqHammer. I would expect that MirrorSolve would similarly impact the efficacy of neural proof search. At the same time, MirrorSolve proofs differ from traditional proofs. The proof engineer must find the inductive structure of the proof and develop (and prove) inductive sublemmas before attempting the proof. In addition, for larger libraries the engineer must be careful about what lemmas are in scope for the SMT search. All these subtleties make neural search with MirrorSolve a fundamentally different challenge than traditional neural proof synthesis. I am collaborating with Alex Sanchez-Stern on a project for estimating the difficulty of a proof using learned models [FSE '23 sub], leading to 43% shorter proofs requiring 44% fewer synthesis steps, and I am excited to bring the power of machine learning to bear in the service of proof automation.

Applications of Formal Methods In Leapfrog, I collaborated with experts in networking to apply formal methods to the networking area. I am excited to continue work at this intersection. For example, I am collaborating on a project for specifying network-wide security policies using a NetKAT-style DSL Anderson et al. [2014], specializing the policies to the nodes of the network, and synthesizing correct-by-construction P4 programs for individual switches in the network. More generally, the networking community put significant thought and effort into building networks that satisfy robust security policies, but because networking is traditionally underserved by formal methods, there are no programmer tools for ensuring that high-level policies are implemented by network switches. Another potential area for impact is the use of formal methods for developing high-assurance systems software; for example, my work on certified dependent grammars could be generalized and applied to parsing programs in the linux kernel in the style of EverParse Ramananandro et al. [2019].

I am excited to continue applications like these, in areas such as networking, systems, high-performance computing, and security. Such collaborations are impactful for two reasons. First, they raise the overall level of assurance available to programmers; for example, Leapfrog provides an analysis that networking operators can use in security-critical packet parsers. Second, I use collaborations to uncover usability challenges with state-of-the-art formal methods techniques; for example, my work in Leapfrog surfaced the limitations of current SMT integration in proof assistants. **In my future work I will build more certified systems and use that experience to make certified programming a mainstream reality.**

References

Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. NetKAT: semantic foundations for networks. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, 2014.

Ryan Doenges, Tobias Kappé, **John Sarracino**, Nate Foster, and Greg Morrisett. Leapfrog: Certified equivalence for protocol parsers. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI '22, 2022.

T Ramananandro, A Delignat-Lavaud, C Fournet, N Swamy, T Chajed, N Kobeissi, and J Protzenko. EverParse: Verified secure zero-copy parsers for authenticated message formats. In *28th USENIX Security Symposium (USENIX Security 19)*, USENIX '19, 2019.

John Sarracino, Odaris Barrios-Arciga, Jasmine Zhu, Noah Marcus, Sorin Lerner, and Ben Wiedermann. User-guided synthesis of interactive diagrams. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, CHI '17, 2017.

John Sarracino, Dylan Lukes, Cora Coleman, Hila Peleg, Sorin Lerner, and Nadia Polikarpova. Synthesis of web layouts from examples. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, FSE '21, 2021.

John Sarracino, Gang Tan, and Greg Morrisett. Certified parsing of dependent regular grammars. In *2022 IEEE Security and Privacy Workshops (SPW)*, LangSec '22, 2022.

John Sarracino, Tobias Kappé, Ryan Doenges, and Greg Morrisett. Metaprogramming for practical and extensible SMT-powered Coq proof automation. In *In preparation*, 2023a. URL https://goto.ucsd.edu/~john/files/mirrorsolve.pdf.

John Sarracino, Alex Sanchez-Stern, and Abhishek Varghese. Lambdelphi: Predicting the difficulty of interactive proofs. In *In submission to ESEC/FSE '23*, 2023b. URL https://goto.ucsd.edu/~john/files/lambdelphi.