Research Statement—Federico Mora (fmora@berkeley.edu)

Designing and implementing distributed systems is still an enormous task for software engineers. Much of this challenge stems from the fact that bugs can arise from complex combinations of machine failures and message interleavings that are difficult for humans to reason about manually [Gabrielson 2019]. As distributed systems become increasingly critical infrastructure, engineers will need more and more computational support to correctly build and deploy them.

Automated reasoning engines promise to be the computational support that engineers need. These engines can solve tedious, mission-critical logic problems billions of times a day (see e.g., Rungta [2022]). In the domain of distributed systems specifically, these tools have helped find bugs and prove the correctness of industrial systems (see e.g., Newcombe et al. [2015]). Unfortunately, despite their power and flexibility, adoption of automated reasoning engines remains low for one fundamental reason. Today, successful users must be experts in their application domain *and* in automated reasoning—a rare combination. Specifically, users must be able to precisely describe their problem in a formal language, like first-order logic, and know enough about automated reasoning to make sure their encoded problem is practically solvable.[1]

My research focuses on making domain-specific automated reasoning practical. During my Ph.D., I focused on the domain of distributed systems verification. The fundamental principle behind my work is that automated reasoning tools should be tailored to the needs of engineers and not the other way around. To achieve this, I focus on making modeling—the process of encoding into a formal language—easier; and improving the performance of solvers on the logical queries generated by modeling languages in this domain.

1 DOMAIN-SPECIFIC LANGUAGE FOR FORMAL VERIFICATION

Modern programming languages provide software engineers with a choice of different distributed systems programming paradigms. Two popular paradigms include the actor model, supported natively by e.g., Erlang, Scala, and Rust; and the model of communicating sequential processes, supported natively by e.g., Go. Today, engineers that want to formally verify their distributed systems must first model their systems in formal languages that support neither of these popular paradigms. For example, a software engineer working with the actor model will likely turn to IVy [McMillan and Padon 2020], TLA$^+$ [Lamport 1999], or Dafny [Leino 2010], for formal verification. IVy works in the paradigm of synchronous reactive programs; TLA$^+$ is explicitly designed to not resemble any programming language [Lamport 2021]; and Dafny targets imperative programs similar to those written in Java.

In our OOPSLA '23 paper [Mora et al. 2023], we took a language for modelling communicating state machines, called P, and added support for formal verification. P is more similar to the actor model than any existing verification language, making it easier for engineers to model their distributed systems. Before our work, P had support for explicit-state model checking (similar to testing) and engineers inside Amazon were successfully using it to find bugs in real systems [Desai 2023; Terry 2024]. **Our work allows these same engineers to write proofs of correctness in the modeling language they are already using.** Writing proofs of correctness is a hard job. Doing it in a language that is not meant for formal verification is even harder, usually requiring more lemmas and complicated reasoning. Our main contribution was to extend P with syntax, abstractions, and automation that make writing proofs easier.

The main challenge in verifying P programs is that it explicitly models message passing. Proofs, therefore, have to consider the low-level details of single messages. To remedy this, we

[1]"We cannot expect all AWS users to be experts in formal methods, have the time to be trained in the use of formal methods tools, or even be experts in the cloud domain" [Rungta 2022]. "The full power of automated reasoning is not yet available to everyone because today's tools are either difficult to use or weak" [Cook 2019].

provide users with language support for reasoning about sequences of logically related messages, called *message chains*. Message chains are related to message sequence charts [ITU-T 2011] from the software engineering community, and choreographic programming languages [Montesi 2014] from the programming languages community. Both message sequence charts and choreographic programming offer developers a way to reason about explicit message passing but with the context of how messages flow through the system in question. We use message chains to bring these same ideas to formal verification.

We evaluated our verification framework by verifying systems from related work and two systems inside Amazon. We found that, with message chains, proofs require a similar number of lemmas than more abstract frameworks, verification time is comparable, and some lemmas can be automatically learned using classic learning algorithms, like Angluin [1980]. Since publication, we have invested significant engineering time to improve modular reasoning, add parallel solving, and design a novel proof caching mechanism. These extensions increase scalability and are supporting an exciting ongoing verification effort inside of Amazon today.

2 SEMI-AUTOMATED FORMAL MODELING

Even with a domain-specific language (DSL), formally modeling distributed systems can be tedious. In fact, a DSL could make modeling more challenging if users need to learn the target language from scratch, or modern code intelligence tools, like LLM-based coding assistants, perform poorly on the target language. The latter case is common: LLMs struggle to generate code in low-resource programming languages, like DSLs for formal modeling. In our NeurIPS '24 paper [Mora et al. 2024], **we present a neuro-symbolic tool that makes it possible for LLMs to generate programs in languages that are not represented in their training data.**

The first key idea behind our approach comes from natural programming elicitation, a kind of study that helps programming language designers understand how programmers naturally approach problems from a given programming domain [Myers et al. 2004]. Programming language designers use the results of these studies to create languages that are aligned with the expectations of users, leading to less programming friction and more effective developers. We borrow this idea for the setting where LLMs are the "users." Akin to uncovering what human users find natural for a given domain, we uncover what LLMs find "natural." Specifically, our first insight is to embrace LLM's tendencies and design an intermediate language that aligns with the LLM's output distribution for tasks in our domain.

Our first key idea makes it possible for LLMs to generate *reasonable* programs but it presents a new challenge: different languages have different features, like type systems, that make translation difficult. The second key idea behind our approach is to use an automated reasoning engine—a MAX-SMT solver in this case—to identify the minimal set of locations in the intermediate program that prevent a correct translation to the target language. We then repair these locations using a combination of deductive techniques and LLM calls, repeating the process as necessary. **This symbolic component guarantees that generated programs pass all compiler checks, including type checking, for the target DSL.**

We implemented a prototype of our approach in a tool called Eudoxus. Eudoxus translates natural language text into UCLID5 programs—a formal modeling language and verification framework whose development I help lead [Polgreen et al. 2022]. UCLID5 is used in a number of verification projects and it is the backend for our P verification framework. In our NeurIPS '24 paper, we show that Eudoxus outperforms fine-tuning, self-repair, few-shot prompting, and chain-of-thought prompting over test problems collected from three well-known textbooks. Specifically, our tool correctly solved 33% of textbook problems while the next best approach solved only 3%. We are currently extending this work with a focus on semantic correctness—the current approach guarantees that output programs pass all compiler checks.

3 FASTER DOMAIN-SPECIFIC AUTOMATED REASONING ENGINES

The P verification framework and our text-to-formal-model tool use automated reasoning engines to answer satisfiability queries. For the P verifier, these questions ask "is there a counterexample to induction?" For the text-to-formal-model tool, these queries ask "is my intermediate program consistent?" Both sets of queries contain algebraic data types (ADTs), which also appear in other distributed systems verification projects (see e.g., Zhang et al. [2024]). In our AAAI '24 paper [Shah et al. 2024], we presented a new automated reasoning engine, called Algaroba, for answering queries with algebraic data types. We found Algaroba to be the fastest solver available. Later, **Algaroba won the quantifier-free algebraic data types track at the Satisfiability Modulo Theories Competition** [Bromberger et al. 2024].

Algaroba takes an *eager* approach to solving queries. Specifically, it takes a quantifier-free query containing ADTs and translates it to a quantifier-free query without ADTs that can then be solved by existing non-ADT solvers. Algaroba eliminates ADTs by replacing them with *uninterpreted sorts and functions* along with a finite number of quantifier-free axioms. The key technical challenges with eagerly solving ADT queries is that, at first glance, this kind of translation seems like it should require an unbounded number of axioms (e.g., lists can be arbitrarily long but no list is equal to any of its sub-lists). Despite this, we proved that our finite translation is sound and complete.

Most other automated reasoning engines in this space take a *lazy* approach. In the lazy approach, theory axioms are introduced as needed, instead of up front. Since Algaroba takes a fundamentally different approach to existing tools, its performance profile is quite unique: Algaroba solves many queries that no other solver succeeds on; and other solvers succeed on queries for which Algaroba fails on. This phenomenon is common across SMT theories. The problem is that, for any given query, it is hard to know ahead of time what solver will succeed.

In our FM '21 paper [Mora et al. 2021], we defined three solving algorithms for satisfiability queries over strings, and provided a fixed classifier for deciding when to use what algorithm. In our SAT '21 paper [Pimpalkhare et al. 2021], we generalized that work to an online learning scheme that supports any logical theories. Our tool, called MedleySolver, takes a stream of queries from a given domain. For each query, MedleySolver picks a solver to execute and learns from the result. **Over time, without requiring any human expertise, MedleySolver automatically discovers the best solver for queries in the target domain.** Specifically, MedleySolver solves more queries than any individual solver using significantly less time.

4 FUTURE WORK AND VISION

Empowering distributed systems engineers with practical, domain-specific automated reasoning engines will lead to faster and safer development of globally critical infrastructure. We have taken significant steps towards this goal, but a truly transformational contribution will not come from a single verification engine, semi-automated modeling tool, or solver. Rather, we need to continuously build full-stack automated reasoning support for emerging, domain-specific programming paradigms.

Example Emerging Concurrency Paradigm. Take for example the new concurrency model supported by OCaml [Sivaramakrishnan et al. 2021]. This model, based on *algebraic effects*, allows programmers to capture non-local control (like concurrency) and is increasingly gaining popularity in programming language circles. Unfortunately, little work has been done on practical automated reasoning about algebraic effects, so engineers have few choices when it comes to tools for proving correctness of their programs. Theoretical work in this space has focused on carving out fragments of effectful languages for which certain kinds of verification problems are decidable (see e.g., Sekiyama and Unno [2024]). This is exciting work that could greatly benefit from tailored automated reasoning engines—like Algaroba is for P—and new verification abstractions—like message chains are for P. My research group will study emerging programming models, like effectful programming, from the perspective of domain-specific automated reasoning. We will provide software engineers working in these domains with practical tools for reasoning about their programs.

Semi-Automated Modeling Framework. For every new programming paradigm that we support, we want to provide good modeling automation—Like Eudoxus does for UCLID5. Similarly, any user of a domain-specific language should be able to reap the benefits of automated code generation. Currently, however, our approach for code generation is largely manual and requires engineering for every new target language, making it difficult to adopt. I seek to remedy this by studying neuro-symbolic code generation from the perspective of domain-specific automated reasoning. Just like existing frameworks for building domain-specific languages do not require users to write compilers from scratch, we should be able to automatically generate an effective neuro-symbolic code generation tool from a description of a domain-specific language.

This language design perspective is also exciting because it opens up the possibility of alternate user interfaces. Our first work only supports textual prompts, but programming, and reasoning about programs, is not always done in text. For example, distributed systems are frequently described through diagrams, like message sequence charts. Diagrams can describe models, specifications, and lemmas. If we are able to support code generation and automated modeling from alternate interfaces, engineers would be able to automatically convert their documentation to formal artifacts, fundamentally changing how programs are verified today. My research group will study how advances in machine learning, like vision language models, can enable code generation from different input modalities that are relevant in a given domain.

Declarative Eager Solvers. All of my goals described above depend on fast solvers. The problem is, we do not know what logical theories or fragments we will need to support. Even if we did, we do not know what solver optimizations will be effective for queries generated by our future tools and frameworks. This is true for the whole field of automated reasoning: the demand for solvers is growing in often unpredictable directions. My research group will face this challenge by studying how to automatically generate eager solvers, like Algaroba, from declarative specifications. The key insight for this line of work is that the internal representation and compiler-like passes of eager solvers can be modeled by *terms* and *term rewriting rules*, respectively. In fact we are already working on re-implementing Algaroba using an *equality saturation* tool [Willsey et al. 2021]. The key open question is how far we can push this approach, both in terms of expressivity and practical performance. If successful, users will be able to quickly generate new solvers to meet their domain-specific automated reasoning needs.

REFERENCES

Dana Angluin. 1980. Finding patterns common to a set of strings. *J. Comput. System Sci.* (1980). https://doi.org/10.1016/0022-0000(80)90041-0

Martin Bromberger, François Bobot, and Martin Jonáš. 2024. International Satisfiability Modulo Theories Competition. https://smt-comp.github.io/2024/results/qf_datatypes-single-query/

Byron Cook. 2019. AWS Security Profile: Byron Cook, Director of the AWS Automated Reasoning Group. https://aws.amazon.com/blogs/security/aws-security-profile-byron-cook-director-aws-automated-reasoning-group/

Ankush Desai. 2023. AWS re:Invent - Gain confidence in system correctness and resilience with formal methods. https://www.youtube.com/watch?v=FdXZXnkMDxs

Mike Dodds. 2022. Formally Verifying Industry Cryptography. *IEEE Security & Privacy* (2022). https://doi.ieeecomputersociety.org/10.1109/MSEC.2022.3153035

Jacob Gabrielson. 2019. Challenges with distributed systems. https://aws.amazon.com/builders-library/challenges-with-distributed-systems/

ITU-T. 2011. *Message Sequence Chart (MSC).* Recommendation Z.120. International Telecommunication Union. https://www.itu.int/rec/t-rec-z.120

Leslie Lamport. 1999. Specifying concurrent systems with TLA+. *Calculational System Design* (1999). https://lamport.azurewebsites.net/pubs/lamport-spec-tla-plus.pdf

Leslie Lamport. 2021. A high-level view of TLA+. https://lamport.azurewebsites.net/tla/high-level-view.html

K Rustan M Leino. 2010. Dafny: An automatic program verifier for functional correctness. In *LPAR*. https://www.andrew.cmu.edu/course/18-330/2020s/reading/dafny.pdf

Kenneth L McMillan and Oded Padon. 2020. Ivy: a multi-modal verification tool for distributed algorithms. In *CAV*. https://doi.org/10.1007/978-3-030-53291-8_12

Fabrizio Montesi. 2014. *Choreographic programming*. IT-Universitetet i København. https://www.fabriziomontesi.com/files/choreographic_programming.pdf

Federico Mora, Murphy Berzish, Mitja Kulczynski, Dirk Nowotka, and Vijay Ganesh. 2021. Z3str4: A Multi-armed String Solver. In *FM*. https://doi.org/10.1007/978-3-030-90870-6_21

Federico Mora, Ankush Desai, Elizabeth Polgreen, and Sanjit A. Seshia. 2023. Message Chains for Distributed System Verification. In *OOPSLA*. https://doi.org/10.1145/3622876

Federico Mora, Justin Wong, Haley Lepe, Sahil Bhatia, Karim Elmaaroufi, George Varghese, Joseph E. Gonzalez, Elizabeth Polgreen, and Sanjit A. Seshia. 2024. Synthetic Programming Elicitation for Text-to-Code in Very Low-Resource Programming and Formal Languages. In *NeurIPS*. https://arxiv.org/abs/2406.03636

Brad A. Myers, John F. Pane, and Amy J. Ko. 2004. Natural programming languages and environments. *CACM* (2004). https://doi.org/10.1145/1015864.1015888

Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. 2015. How Amazon Web Services uses formal methods. *CACM* (2015). https://www.amazon.science/publications/how-amazon-web-services-uses-formal-methods

Nikhil Pimpalkhare, Federico Mora, Elizabeth Polgreen, and Sanjit A. Seshia. 2021. MedleySolver: Online SMT Algorithm Selection. In *SAT*. https://doi.org/10.1007/978-3-030-80223-3_31

Elizabeth Polgreen, Kevin Cheang, Pranav Gaddamadugu, Adwait Godbole, Kevin Laeufer, Shaokai Lin, Yatin Manerkar, Federico Mora, and Sanjit A. Seshia. 2022. UCLID5: Multi-Modal Formal Modeling, Verification, and Synthesis. In *CAV*. https://doi.org/10.1007/978-3-031-13185-1_27

Neha Rungta. 2022. A billion SMT queries a day. In *CAV*. https://doi.org/10.1007/978-3-031-13185-1_1

Taro Sekiyama and Hiroshi Unno. 2024. Higher-Order Model Checking of Effect-Handling Programs with Answer-Type Modification. In *OOPSLA*. https://doi.org/10.1145/3689805

Amar Shah, Federico Mora, and Sanjit A. Seshia. 2024. An Eager Satisfiability Modulo Theories Solver for Algebraic Datatypes. In *AAAI*. https://ojs.aaai.org/index.php/AAAI/article/view/28649

KC Sivaramakrishnan, Stephen Dolan, Leo White, Tom Kelly, Sadiq Jaffer, and Anil Madhavapeddy. 2021. Retrofitting effect handlers onto OCaml. In *PLDI*. https://doi.org/10.1145/3453483.3454039

Doug Terry. 2024. How do we sleep at night with confidence that our services are operating correctly? https://www.linkedin.com/posts/doug-terry-08b2b68_an-unexpected-discovery-automated-reasoning-activity-7258244396049997825-GlH3

Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. 2021. Egg: Fast and extensible equality saturation. In *POPL*. https://doi.org/10.1145/3434304

Tony Nuda Zhang, Travis Hance, Manos Kapritsos, Tej Chajed, and Bryan Parno. 2024. Inductive Invariants That Spark Joy: Using Invariant Taxonomies to Streamline Distributed Protocol Proofs. In *OSDI*. https://www.usenix.org/conference/osdi24/presentation/zhang-nuda