Research Statement Lauren Pick

Software systems carry out important tasks and make important decisions. To do this, these systems often interface with others: simply logging into a bank account involves interactions between network protocols, distributed systems, and encryption schemes. Given their critical role in society, it is important that we understand how such systems behave, how they interact, and what behaviors emerge from their composition.

Formal specifications can help address the challenge of understanding these systems. These specifications are precise, high-level descriptions of system behaviors, and they can be composed to understand the behaviors that emerge from compositions of the systems they describe. To aid in understanding software systems, I develop automated techniques both for *synthesizing* formal specifications and for *verifying* that such specifications are correct for given systems. There is an interplay between verification and synthesis, where synthesis techniques may rely on verification to ensure correctness of synthesized results, and verification techniques may generate sub-problems that can be solved by synthesis.

Unfortunately, software systems are large and their interactions complex, making it difficult to derive formal specifications. This is true even for systems composed solely of manually-constructed components such as distributed systems, where concurrency leads to many possible (and sometimes unintended) behaviors. A further challenge arises when considering machine learning (ML) systems, whose behaviors, rather than being determined by a human programmer, are learned from data. While these systems' effectiveness has led to their widespread adoption, they are challenging to reason about – they are often massive, with modern models having billions of learned parameters.

While synthesis and verification provide a promising way of understanding software systems and ensuring their correctness, techniques often have scalability or expressivity-related limitations that prevent their application to real software systems. I have tackled these limitations by exploiting the structure of programs and properties to avoid redundant work in verification and to learn useful specifications, which has had applications in ensuring secure information flow [3–5] and in verifying distributed systems [2]. Going forward, I will continue working toward the broader goal of automating reasoning about large and complex software systems. With this aim in mind, in the near future, I plan to develop new synthesis and verification techniques for distributed applications and ML systems.

Security Properties

We rely on software systems to handle sensitive personal data. As unintended leakage of this information to unauthorized parties can be disastrous, we would like to prove that these systems exhibit certain security and privacy properties. These properties can be formulated as *relational* ones that relate k executions of a single program; in particular, secure information flow properties, which formalize the notion that high-security (private) inputs do not leak information to low-security (public) outputs, are formulated as 2-safety properties – properties over *two* executions – of the underlying system. Inferring information-flow specifications can help us reason about the overall security of a system and compositions of systems. My research in this area has improved the scalability k-safety property verification and provided a way to infer information-flow specifications of procedures automatically.

Synchrony and Symmetry for Scaling k-safety Verification [3] Verification of k-safety properties involves the challenge of having to reason about k executions of a single program. A key idea used to address this challenge is that of *synchrony*, where the aim is to explore behaviors of corresponding parts of the executions in lockstep as much as possible. I leveraged the structure of k-safety properties to propose a novel algorithm to increase synchrony by detecting maximal sets of loops that can be executed in lockstep. Use of this algorithm helps execute more loops in lockstep, leading to fewer invocations of expensive invariant synthesis procedures during verification.

I also noticed that k-safety properties are often commutative for properties of interest, resulting in many verification subtasks being symmetric with respect to which execution of the program variables come from, which led me to develop an algorithm to prune these redundant subtasks: I adapted previous work in SAT solving on discovering and breaking symmetries of propositional logic formulas to discover symmetries in k-safety properties expressed in first-order logic. In my CAV paper, I showed experimentally that these techniques were very effective at reducing the runtime of an underlying state-of-the-art verifier.

$$\boxed{\forall x_1, x_2, y_1, y_2. x_1 = x_2 \Rightarrow f(x_1, y_1) = f(x_2, y_2)}$$

Figure 1: A secure information flow (noninterference) property. Note the symmetry with respect to variable indices.

Synthesis for Security of Interprocedural Programs [4, 5] Verification of secure information flow properties in a scalable, procedure-modular way depends on having relational specifications expressing information-flow properties. In my FMCAD paper, I designed new grammar templates based on the structure of these properties, which allows a syntax-guided synthesis technique to learn these invariants *automatically*, eliminating the need for the user to provide them, as previously required. These invariants are learned in an environment-agnostic way, so they may be irrelevant to the top-level property. This observation led me to propose *property-directed invariants* in my VMCAI paper. These are generated from a grammar template that uses syntactic features of the calling environment. Invariants inferred using property-directed templates are key to handling declassification and loops, both of which are required to prove useful security properties of real-world systems. The use of these proposed grammar templates in a specification inference tool allowed it to solve benchmarks for secure information flow that prior state-of-the-art tools could not, including ones based on the industrial TLS implementation s2n.

Distributed Systems and Databases

Distributed systems underlie many applications, with distributed databases being a key application on top of which many others are built. Given their importance, we would like to be able to understand and prove properties about (1) the underlying distributed systems and (2) the distributed databases themselves.

Identifying Redundancies for Distributed Systems Verification [2] Distributed systems consist of many nodes that operate concurrently and are difficult to reason about because of their large number of possible behaviors; however, most systems involve several nodes that perform some of the same computations in response to similar messages. To eliminate redundancy, in my PLDI paper I proposed *composite value summaries*, a decomposed

representation of system states that a model checker can use to identify redundancies when computing the next frontier of states. The model checker can then avoid performing these redundant computations, even when they occur when computing transitions from *different* system states. The implementation PSYM outperforms the state-of-the-art model checker TLC on a set of open source TLA + benchmarks of common distributed protocols and scales to verify industrial protocols, including four protocols used at Amazon Web Services. It has seen industrial adoption and is now part of the P toolchain at Amazon Web Services.

Correctness of Distributed Databases [Current Work] Clients of distributed transactional database management systems (DBMSs) rely on them to provide both *semantically correct* implementations of database operations and *isolation guarantees*, which specify the visibility of writes of concurrent transactions to each other. To check correctness of black-box DBMSs in general, it is necessary to check *both* semantic correctness and isolation guarantees. As in the case of general distributed systems, checking correctness of a black-box DBMS for weaker levels of isolation presents a scalability challenge because there are many possible values that may be read by transactions, and *all* such possibilities need to be reasoned about. I formulated a notion of *observational correctness* for DBMSs and a method for checking it that relies on a novel symbolic encoding of semantic correctness and isolation guarantees respect to client observations. These encodings represent nondeterministic reads and writes symbolically, and resolve this nondeterminism is done via Satisfiability Modulo Theories (SMT) solving. The symbolic encodings also allow for inferring specifications of database states from a sequence of transactions, which can be used to help debug DBMS implementations. The implementation of this checking method has been used inside Amazon Web Services to detect two bugs in internal DBMSs currently under development.

Future Directions

Synthesis for Relaxed Distributed Data Structures

There has been much work on proposing *relaxed* concurrent data structures, which provide weaker guarantees in exchange for better performance. For example, a dequeue from a relaxed concurrent priority queue may return any one of the k highest-priority elements, reducing contention for the highest-priority element. Distributed client applications can use these relaxations to gain performance benefits when overall functional correctness is unaffected and the drop in the result quality is acceptable. If developers want to use relaxed data structures in their applications, however, they must face the challenge of reasoning about whether a particular relaxation is suitable for the application. I am interested in automating this reasoning by *synthesizing* replacements of data structures by relaxed variants.

Given hard and soft constraints capturing requirements on the client program's correctness, result quality, and desired performance, synthesis should produce a correct relaxation that achieves the desired trade-off between result quality and performance, provided such a relaxation exists. Developers can then use the synthesized relaxations to improve performance of their applications without worrying about result quality or correctness.

The synthesis techniques applicable to this problem overlap heavily with those used in work that I have been involved in for minimizing noise during quantum compilation [1,7]. In the quantum compilation work, scalability issues in synthesis were addressed by leveraging the structure of quantum circuits. For large client applications, scalability will similarly

be an issue, though in this setting, it can be addressed by developing a modular approach where *specifications are inferred for modules of client programs* that may use relaxed data structures. As these specifications constitute client-side requirements on the relaxations, they may also help inform the development of new relaxed data structures.

Program Synthesis for Understanding Transformer Models

ML components, and especially the transformer models that underlie large language models (LLMs), despite being in wide use, are not well-understood. Experimental frameworks can help understand some behaviors of ML systems and their interactions with other ML or non-ML components [6], but for more rigorous understanding of and automated reasoning about ML systems, we would like to infer compositional specifications that describe the behavior of ML components. The structure of the residual blocks that make up transformer layers suggests that specification inference using traditional synthesis techniques may be a promising method for achieving interpretability. To perform this specification inference, there are two problems that must be solved: (1) designing the space of specifications to infer and (2) designing the inference algorithm.

Specification format. Existing work in understanding transformers has largely addressed problem (1) in two ways. The first is by understanding the behavior of transformer models using a circuit-based approach,where models are computational graphs and circuits are subgraphs that can be viewed as modules providing a particular functionality. While such models can be low-level enough to capture all transformer behaviors, circuit construction can be difficult, and the circuits themselves may be difficult to understand. An orthogonal body of work proposes RASP, a domain-specific programming language whose constructs are modeled after the structure of the transformer architecture. RASP programs provide succinct and easier-to-understand descriptions of transformer behavior, suggesting that they would be easier to synthesize and reason about, but RASP is unable to capture certain low-level behaviors of transformers like superposition. I am interested in coming up with ways to combine and extend these modeling approaches to strike the right balance in trading off between model expressivity and feasibility of automatic inference.

Specification inference. Problem (2) meanwhile remains largely unaddressed in existing work. In the work described so far, specifications for arbitrary transformer models are provided by a human theorizing about the behavior of a transformer model or its components in the same way that a human may write candidate invariants or procedure specifications for loops or modular programs in a software verification setting. While there has been some work on circuit-based mechanistic interpretability that can be applied to transformers as well as other models, automation only goes as far as identifying subcomponents of a model that satisfy the specification (in the form of a computation graph). Similarly, for arbitrary transformers, using RASP-like specifications would require humans to provide programs describing the behavior that they would like to check an existing model against. I plan to apply insights from specification inference in other domains to automate this aspect of achieving interpretability for general transformers. For example, traditional program synthesis techniques seem well-suited to synthesize RASP or RASP-like programs.

References

[1] A. Molavi, A. Xu, M. Diges, L. Pick, S. S. Tannu, and A. Albarghouthi. Qubit mapping and routing via MaxSAT. In *55th IEEE/ACM International Symposium on Microarchitecture, MICRO 2022, Chicago, IL, USA, October 1-5, 2022*, pages 1078–1091. IEEE, 2022.

[2] L. Pick, A. Desai, and A. Gupta. Psym: Efficient symbolic exploration of distributed systems. *Proc. ACM Program. Lang.*, 7(PLDI):660–685, 2023.

[3] L. Pick, G. Fedyukovich, and A. Gupta. Exploiting synchrony and symmetry in relational verification. In *CAV (1)*, volume 10981 of *Lecture Notes in Computer Science*, pages 164–182. Springer, 2018.

[4] L. Pick, G. Fedyukovich, and A. Gupta. Automating modular verification of secure information flow. In *FMCAD*, pages 158–168. IEEE, 2020.

[5] L. Pick, G. Fedyukovich, and A. Gupta. Unbounded procedure summaries from bounded environments. In *VMCAI*, volume 12597 of *Lecture Notes in Computer Science*, pages 291–324. Springer, 2021.

[6] N. Roberts, X. Li, T. Huang, D. Adila, S. Schoenberg, C. Liu, L. Pick, H. Ma, A. Albarghouthi, and F. Sala. Autows-bench-101: Benchmarking automated weak supervision with 100 labels. In *NeurIPS*, 2022.

[7] A. Xu, A. Molavi, L. Pick, S. Tannu, and A. Albarghouthi. Synthesizing quantum-circuit optimizers. *Proc. ACM Program. Lang.*, 7(PLDI):835–859, 2023.