Research Statement

ROBERT DICKERSON

My research is focused on finding principled solutions to practical software development challenges. Drawing from my extensive software industry experience, I approach problems related to building reliable systems from a formal perspective to develop rigorous solutions. I use insights from this formal setting to prototype tools and analyses for real-world applications. During my PhD career, I have considered two problem settings in pursuit of this overarching goal, and my future research directions will similarly be guided by this philosophy.

RELATIONAL VERIFICATION

During my career as a software engineer, several complex or error-prone tasks I encountered involved reasoning about multiple program executions. For example, when fixing a bug, only the buggy behavior should be different in the repaired code; when migrating to a new API, the post-migration code should behave the same as the pre-migration code; and when writing a multi-user service, data from one user's session should not leak to another's. Formally, these are all examples of *relational verification* tasks, which reason about multiple program executions whose states are required to stand in some relation to each other. My goal is to develop and apply automated relational verification and analysis techniques to make sure bad things cannot happen in these kinds of multi-execution situations.

Relational Program Logic. For any kind of program verification, a natural starting point is Hoare-style *program logic*. Based on foundational work by Floyd [8] and Hoare [10], program logics use an axiomatic semantics for a programming language to prove that some logical condition must hold at the end of a (single) program execution, assuming some condition held at the beginning. However, as Hoare-style logics are limited to reasoning about single program executions, they cannot express or prove the kinds of relational properties mentioned above.

Extending Hoare logic to reason about multiple program executions dates back at least 40 years to work by Francez [9]. The approach was more recently rediscovered and popularized by Nick Benton in 2004 [3], and several *relational program logics* have been subsequently proposed. When I started my PhD studies, existing relational logics dealt with *k-safety properties*; properties which describe when *no* possible executions of k programs can "go wrong." However, several important relational properties, including those needed for some of the tasks mentioned above, also have *liveness* components. Intuitively, liveness properties guarantee that *some* possible executions must "go right." For example, *program refinement* is a property which asserts the possible behaviors of one program are a subset of the possible behaviors of another; *for all* executions of one program, *there exists* an execution of another with the same behavior when given the same inputs. Program refinement is needed to reason about, e.g., the correctness of API migrations, and requires the verifier to establish the existence of some desirable program executions. Specifically, for any possible execution path of the refining program, a relational verifier must evince an execution path through the original program that yields the same result. Other properties of practical interest like *nondeterminism* and *generalized non-interference* (an information security property which says the public result of a computation using secret information could have been obtained using any other secret) also have this $\forall\exists$ shape.

Despite the importance of these $\forall\exists$ relational properties in practical verification problems, we had been missing a Hoare-style logic capable of establishing these properties. To address this gap, I introduced Relational Hoare Logic with Existentials (RHLE) [6], a program logic capable of proving

properties with a ∀∃ shape. Soundness of RHLE's reasoning rules is formally verified via the Coq proof assistant. A key insight to my approach was reifying nondeterministic execution choices as prophecy variables baked into both the reasoning rules and a novel kind of existential specification. Developing RHLE allowed me to build an automated program verifier capable of handling a diverse set of ∀∃ properties. The benchmark suite we assembled for the evaluation of RHLE has been used in the evaluation of subsequent work [4, 11].

Program Alignment. While relational program logics provide powerful frameworks for reasoning about relational properties, automated verification using these logics requires bespoke tools. An alternate approach, initially posed by Francez [9] and further developed by Barthe et al. [2], is to combine multiple program executions into a *product program*. A product program is a single program whose correctness implies the correctness of the original relational property. An advantage of this approach is that verification of the product program can directly leverage industrial-strength single program verifiers already in existence. However, constructing the product requires identifying *program alignments*, correspondences between subprograms of different executions which simplify verification. Without these simplifications, the product program can become complex, especially in its loop invariants, making verification intractable. Finding useful alignments is non-trivial, and may involve program transformations such as unrolling or duplicating a loop so that it terminates simultaneously with a corresponding loop in a different execution. Automating the search for alignments is not generally considered in existing work on this kind of product construction.

By combining a recently-developed algebra for program alignments called BiKAT [1] with advancements in efficient e-graph manipulation [12], I developed a novel data-driven approach to identifying promising alignments for product program constructions [5]. The tool I created for finding these alignments, called KestRel, uses execution traces collected from candidate alignments to drive a Markov chain Monte Carlo (MCMC) based search for BiKAT alignment rewrites with desirable semantic properties. We have found KestRel to perform well over a set of benchmark alignment tasks taken from the literature, including tasks explicitly presented as challenging cases.

SPECIFICATION INFERENCE

Scaling automated reasoning to large, real-world codebases requires *modular reasoning*; breaking verification tasks down into smaller, self-contained modules. This helps make reasoning about a large development tractable while allowing developers to focus more expensive techniques on subcomponents with the greatest return on verification effort. RHLE, for example, admits modular reasoning over both safety and liveness properties in the form of quantified method specifications; universally quantified specifications express *overapproximate* guarantees about the behavior of all invocations, while existentially quantified specifications express *underapproximate* guarantees about the existence of some desirable behaviors. Users of RHLE are therefore free to summarize expected behaviors of method calls with these specifications, then separately verify specific implementations of each method against those specifications.

While modular reasoning is a critical technique for practical verification, it fundamentally depends on formal specifications on module boundaries which are often not given. This can be especially problematic for "black-box" libraries, whose source code is unavailable, rapidly changing, or difficult to analyze. To address this problem, my collaborators and I developed a data-driven approach to inferring specifications for black-box library code [13]. Given a client of a black-box library and a safety property over that client, our approach observes concrete program executions to hypothesize candidate specifications of the library methods. These specifications are then refined using SMT-provided counterexamples until specifications are found which match the observed library behavior while being sufficient to verify the given safety property. We used

this approach to build an automated verification tool called Elrond, which we found capable of inferring specifications for a wide range of realistic OCaml data structure libraries. This tool was recognized with a distinguished artifact award at OOPSLA 2021.

ONGOING AND FUTURE RESEARCH DIRECTIONS

Moving forward, I plan to continue my approach of applying rigorous, principled techniques to practical software development challenges. I see research as a fundamentally collaborative activity, and I will actively seek students and colleagues to include in my work. A core component of my research philosophy is prototyping practical applications of novel theoretical contributions. I believe this pragmatic aspect of my approach is especially promising ground for finding collaboration with undergraduate students. When developing RHLE, for example, I worked with an undergraduate to teach them the mechanics of the relational logic; this student was then able to make a substantive contribution to the verifier's Haskell codebase, appearing as a coauthor on the RHLE paper [6]. I see this experience as a model for including undergraduate students in future research.

Targeting real-world applications in my research often means reasoning over large sets of possible program behaviors. Prohibitively large search spaces are a common hurdle in program verification and synthesis, but we may be able to make these approaches tractable by combining them data-driven techniques. A common thread in my existing research is using observations of real behavior to drive formal reasoning; KestRel uses execution traces of C programs to inform cost metrics in an MCMC-based e-graph extraction, and Elrond observes executions to refine candidate specifications of black-box OCaml APIs. Going forward, I am interested in investigating data-driven approaches to automated reasoning in other real-world applications, including API migrations, code merging, and ports of programs between languages.

Another challenge I hope to address is the recent rising interest in employing large-language models (LLMs) in software engineering [7]. Software developers want to use LLMs as a labor-saving device, but how can this technology be integrated into software engineering practices in a safe and principled way? I believe one answer may be formal reasoning over the programs which LLMs create. As we have no basis for trusting these LLM-generated programs, our faith in their correctness must come from a rigorous, well-defined foundation. My previous work in relational reasoning is especially applicable in situations when LLMs are used to synthesize code against some reference implementation; for example, in optimizing a piece of code or in translating it from one language to another. In these cases, the reference implementation serves as a natural specification for the correctness of the LLM's output, and guaranteeing that correctness constitutes a relational verification task. To successfully integrate formal reasoning into the LLM pipeline, we need new techniques for reasoning about and repairing LLM-generated code, as well as tools which leverage these techniques in real-world settings.

REFERENCES

[1] Antonopoulos, T., Koskinen, E., Le, T. C., Nagasamudram, R., Naumann, D. A., and Ngo, M. An algebra of alignment for relational verification. *Proceedings of the ACM on Programming Languages 7*, POPL (2023), 573–603.

[2] Barthe, G., Crespo, J. M., and Kunz, C. Relational verification using product programs. In *FM 2011: Formal Methods: 17th International Symposium on Formal Methods, Limerick, Ireland, June 20-24, 2011. Proceedings 17* (2011), Springer, pp. 200–214.

[3] Benton, N. Simple relational correctness proofs for static analyses and program transformations. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 2004), POPL '04, ACM, pp. 14–25.

[4] Dardinier, T., Li, A., and Müller, P. Hypra: a deductive program verifier for hyper Hoare logic. *Proceedings of the ACM on Programming Languages 8*, OOPSLA (2024).

[5] Dickerson, R., Mukherjee, P., and Delaware, B. Kestrel: Relational verification using e-graphs for program alignment. *arXiv preprint arXiv:2404.08106* (2024).

[6] Dickerson, R., Ye, Q., Zhang, M. K., and Delaware, B. RHLE: modular deductive verification of relational ∀∃ properties. In *Asian Symposium on Programming Languages and Systems* (2022), Springer, pp. 67–87.
[7] Fan, A., Gokkaya, B., Harman, M., Lyubarskiy, M., Sengupta, S., Yoo, S., and Zhang, J. M. Large language models for software engineering: Survey and open problems. In *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)* (2023), IEEE, pp. 31–53.
[8] Floyd, R. W. Assigning meanings to programs. *Proceedings of Symposium on Applied Mathematics 19* (1967), 19–32.
[9] Francez, N. Product properties and their direct verification. *Acta informatica 20* (1983), 329–344.
[10] Hoare, C. A. R. An axiomatic basis for computer programming. *Commun. ACM 12*, 10 (Oct. 1969), 576–580.
[11] Niessen, T., and Weissenbacher, G. Finding counterexamples to ∀∃ hyperproperties.
[12] Willsey, M., Nandi, C., Wang, Y. R., Flatt, O., Tatlock, Z., and Panchekha, P. egg: Fast and extensible equality saturation. *Proceedings of the ACM on Programming Languages 5*, POPL (2021).
[13] Zhou, Z., Dickerson, R., Delaware, B., and Jagannathan, S. Data-driven abductive inference of library specifications. *Proceedings of the ACM on Programming Languages 5*, OOPSLA (2021), 1–29.