Research Statement

Charlie Murphy, CS Department, University of Wisconsin–Madison

Software has become a ubiquitous part of modern society—e.g., the applications we use to communicate with each other, process banking transactions, operate medical systems, facilitate infrastructure, and more. Our reliance on these computerized systems in our everyday lives means that we as a society and as individuals can face large consequences when the software running these systems do not behave as intended.

> My research aims to build tools that assist software developers—both amateur and professional—in creating software the behaves as intended.

My research starts from the principle that **developing frameworks (general purpose solutions) is better than building one-off tools** for a specific task. While program verification—proving that a program behaves as intended—and synthesis—generating a program that behaves as intended—are foundational problems in computer science that have been long studied ([7, 18]). The vast majority of methodologies and solutions proposed ([11, 16, 19, 22, 23, 26]) are *one-off* solutions that are tied to a specific language and specification type (e.g., safety of C programs). My work aims to address this limitation by building *general purpose* verification and synthesis techniques on top of *language frameworks*—frameworks that are capable of expressing verification and synthesis problems across a large class of languages and specification types (e.g., the K [3] and Semantics-Guided Synthesis [14] frameworks).

My work in this area can be broadly classified into three interconnected pieces: (*i*) development of language frameworks and general purpose verification and synthesis techniques, (*ii*) generalizing domain specific techniques to language frameworks, and (*iii*) co-development of logic solvers to enable and improve reasoning about language frameworks.

1 Development of Synthesis and Verification Frameworks

While frameworks have been applied successfully to many problems in computer science, they have only recently seen use within program synthesis and verification (e.g., [3, 14, 30, 26]). This is in part due to the complexity of these problems. Developing a general-purpose framework for program verification and synthesis requires a unified representation across different programming languages, specifications, and paradigms (i.e., imperative, functional, and declarative languages). While there are many programming languages with a large variety of syntactic structures each with their own semantics, a natural lingua-franca is first-order logic (with fixed-points e.g., constrained horn clauses (CHCs))—i.e., the semantics of any program can naturally be encoded in first-order logic with fixed-points used to capture recursive behaviors. In fact, this is what many techniques for program verification (e.g., [10]) and synthesis (e.g., [1]) already do, except often on an ad-hoc program-by-program basis as opposed to at the language level via a framework.

The development of these frameworks is attractive as they allow representing verification and synthesis problems from a very general class of languages; however, there are 2 main problems to overcome to make their use feasible. First, one must represent their desired language within the framework, which requires a formal semantics for the language be available. While it has long been considered difficult to develop a formal semantics for a programming language, recent advances have made it easier and companies (e.g., Runtime Verification: `https://runtimeverification.com/`)

even offer their services in developing such semantics. Finally, the generality of the framework is a double-edged sword: while it lets one uniformly represent and solve problems in a domain and solver agnostic way, general purpose solvers must deal with the generality (often at the cost of scalability).

To date, two **language-agnostic frameworks** have been proposed for program verification and synthesis: The K framework [3] and—the one I have been involved in developing—the Semantics-Guided Synthesis framework [5]. Both frameworks share similarities—they allow a user to define a language in their framework by defining its syntax and a semantics that is inductively defined over its syntactic constructs. The key difference between the two is their focus. The K framework is focused on providing language tooling (e.g., compiler, interpreter, and type checker), while the Semantics-Guided Synthesis (SemGuS) framework is focused on representing synthesis problems in a solver and domain-agnostic way to facilitate the development of general-purpose synthesis techniques.

In the paper "Synthesizing Formal Semantics from Executable Interpreters" [17], my co-authors and I aim to reduce the burden of a user by developing a technique to **automatically synthesize a SemGuS compatible formal semantics** from an executable interpreter for a given language. We employ a counter-example guided inductive synthesis procedure that reduces the problem of synthesizing the semantics of an entire language to synthesizing first-order constraints that capture the semantics of each individual semantic construct (i.e., production of the grammar)—reducing the monolithic semantics synthesis problem to a series of simpler synthesis problems that can be solved via a syntax-guided synthesis (SyGuS) solver (e.g., CVC5).

My work "Verifying solutions to Semantics-Guided Synthesis Problems" ([27]) builds upon the SemGuS framework and **develops the logical underpinnings of reasoning about SemGuS problems with logical specifications**. Previously, SemGuS solvers were limited to SemGuS problems whose specifications consisted of a (finite) set of input-output examples. In contrast, my work has enabled the development of a SemGuS solver that (for the first time) was capable of solving SemGuS problems with arbitrary (quantified) logical specifications. More concretely, my work offers a complete reduction from SemGuS verification problems to μCLP validity (a first-order fixed-point logic with least and greatest fixed-point operators [31]). Finally, my work uses the connection between SemGuS verification and μCLP to develop the SemGuS$^{\mu}$ framework (an extension of SemGuS) that is capable of representing both functional and reactive (e.g., controller) synthesis problems—two classes of synthesis problems that have long been considered disjoint.

2 Generalizing Domain Specific Techniques

While general-purpose frameworks allow one to develop techniques that work for any problem instance, general purpose techniques may scale significantly worse than corresponding domain-specific (e.g., language/specification aware) techniques. Thus, my work also aims to identify sub-classes of problem instances in which specialized techniques (possibly inspired by domain specific techniques) may be applied.

For example, in "Verifying solutions to Semantics-Guided Synthesis Problems" ([27]), I identify two sub-classes of problems in which verification can be respectively reduced to a single verification condition expressed in the satisfiability modulo theories (SMT) or CHC fragment of first-order logic—for which more scalable solvers exist (in comparison to μCLP). In addition, I embedded the developed SemGuS verifier into an enumerative synthesis algorithm to develop the first SemGuS synthesizer capable of solving SemGuS problems with logical specifications.

3 Co-development of Logic Solvers

As noted previously, first-order logic is a natural lingua-franca for expressing programs regardless of language or paradigm. As such, it is natural to expect techniques for reasoning about programs to require the use of logic/constraint-based solvers. In fact, even language and specification aware techniques for verification and synthesis typically require multiple calls to logic solvers (e.g., [2, 10, 25]). Thus, a natural way to improve and/or enable the development of such techniques is by **improving the logic solvers that underly the verification and synthesis techniques**. For example, improvements to SAT solvers in the early 2000's allowed for the development of many hardware model checking techniques [4]. Similarly, the development of SMT and CHC solvers enabled the development of many program verification, synthesis, and analysis techniques [6, 9]. And most recently, μCLP has emerged as a first-order fixed-point logic that is capable of uniformly expressing a large class of program verification problems [31].

My work on logic solvers has primarily focused on developing **techniques that naturally support the construction of proofs** that can be inspected and used in higher-level tasks (e.g., program verification and synthesis). My work in "Linear Arithmetic Satisfiability Via Fine-Grained Strategy Improvement" ([28]) develops a decision procedure for quantified linear arithmetic satisfiability (and more generally for decidable theories of a specific form). My technique takes a game theoretic approach to satisfiability and produces a winning strategy (i.e., a certificate/proof) that either proves the formula is satisfiable or unsatisfiable. In "Relational Verification via Weak Simulation" ([29]), I show how a winning strategy to an LIA satisfiability game can be used to handle complications arising from angelic non-determinism. Furthermore, in my recent work "Strategy Synthesis for Validity of the μCLP Calculus" (work in progress), I similarly take a game theoretic view to develop a semi-decision procedure for proving the validity of a μCLP query by synthesizing a winning strategy that proves the μCLP formula valid (or invalid), which allows one to use the extracted proof in higher-level tasks (e.g., using the extracted proof to generate counter-examples when a verification condition (represented in μCLP) is violated).

4 Future Directions

My current and future research is focused on co-developing logic solvers and general purpose frameworks for program synthesis and verification. As part of this research direction, last spring I submitted (with Loris D'Antoni) an NSF small grant proposal to develop the logical foundations of verification and synthesis for the Semantics-Guided Synthesis framework, which while not funded received competitive scores. My recent and current work (described above) addresses the problem of developing a logical foundation to reason about SemGuS$^{\mu}$ problems.

Software development has flourished in modern society, and **software developers write code in an ever increasing number of programming languages**—for which domain specific techniques/tools may not apply. My research directly addresses the lack of domain-specific tools for new or under-resourced languages by offering language-agnostic verification and synthesis techniques that work for any language—whether the software developer is developing a safety-critical application in a new C-like language or writing a reactive controller in some domain specific language that must satisfy a complex hyperproperty. While the developer would need to provide a formal semantics of the language, my work on automatically synthesizing formal semantics from an executable interpreter reduces the burden placed on the developer and need only be done once per language. Additionally, I plan to develop an ecosystem for the verification and synthesis framework that provides the user with semantics of common languages.

Furthermore, my research on language-agnostic verification and synthesis has opened up new opportunities to explore cross-language reasoning—e.g., equivalence of programs in different languages [15], verification of programs that use linked libraries [32], and proving equivalence of specifications in different specification languages [13]. Specifically, my work on program verification and synthesis frameworks allows one to reason about programs written in more than one programming language by providing the semantics of all languages involved—effectively creating a program verifier/synthesizer for the combination of languages.

Cross-language program equivalence. When programming, a common task is transpiling code from one language to a user (e.g., by using stack overflow or more recently large language models like chat-gpt) [33]. The translation task from one language to another can be error prone (even for automated techniques) [33]. Work on automatically transpiling code has either focused on (*i*) soundly translating code from one specific language to another [20] or (*ii*) on unsound methods which have typically used machine learning/large language models [33]. My work offers the ability to either (*i*) automatically verify that the transpiled program is equivalent to the original program or (*ii*) synthesize a correctly transpiled program (assuming one is willing to encode both the source and target languages within a verification/synthesis framework).

Verifying programs with linked code. Similarly, a common practice in software development is the use of linked code from different languages (e.g., a python program that includes library code that was written in C) [24]. Existing approaches to address this problem have typically required a kind of rely-guarantee style reasoning of the program and library code [21]—i.e., by requiring the user to provide a specification at the boundary of the two languages—or via compilation to some intermediate language [8]. In contrast, my research enables reasoning at the source level of both languages without the user needing to provide a specification at the language boundaries (e.g., of the library code).

Equivalence of specifications. Finally, in large scale proof and verification efforts it is common to use multiple tools/proof systems to reason about the proof/program, which can often require translating a specification from one specification language to another at the "boundaries" of the proofs—e.g., as done in IronFleet which uses both TLA+ and Dafny to prove correct a distributed system [12]. The translation of the specification at the boundaries of the proof efforts are typically done by hand (by an expert) and trusted. My work enables proving that the specifications at the boundaries of the proof effort (written in two separate specification languages) are equivalent (and thus that the proof of both parts can be soundly combined)—i.e., specification languages like programming languages can be defined by their syntax and semantics and formally reasoned about.

References

[1] Ahlgren, J., and Yuen, S. Y. Efficient program synthesis using constraint satisfaction in inductive logic programming. *The Journal of Machine Learning Research 14*, 1 (2013), 3649–3682.

[2] Beyer, D., Dangl, M., and Wendler, P. A unifying view on smt-based software verification. *Journal of automated reasoning 60*, 3 (2018), 299–335.

[3] Chen, X., and Roşu, G. A language-independent program verification framework. In *International Symposium on Leveraging Applications of Formal Methods* (2018), Springer, pp. 92–102.

[4] Claessen, K., Een, N., Sheeran, M., and Sorensson, N. Sat-solving in practice. In *2008 9th International Workshop on Discrete Event Systems* (2008), IEEE, pp. 61–67.

[5] D'Antoni, L., Hu, Q., Kim, J., and Reps, T. Programmable program synthesis. In *Computer Aided Verification: 33rd International Conference, CAV 2021, Virtual Event, July 20–23, 2021, Proceedings, Part I 33* (2021), Springer, pp. 84–109.

[6] Ferrara, P., Arceri, V., and Cortesi, A. Challenges of software verification: the past, the present, the future. *International Journal on Software Tools for Technology Transfer* (2024), 1–10.

[7] Fetzer, J. H. Program verification: The very idea. *Communications of the ACM 31*, 9 (1988), 1048–1063.

[8] Garzella, J. J., Baranowski, M., He, S., and Rakamarić, Z. Leveraging compiler intermediate representation for multi-and cross-language verification. In *Verification, Model Checking, and Abstract Interpretation: 21st International Conference, VMCAI 2020, New Orleans, LA, USA, January 16–21, 2020, Proceedings 21* (2020), Springer, pp. 90–111.

[9] Gurfinkel, A. Program verification with constrained horn clauses. In *International Conference on Computer Aided Verification* (2022), Springer, pp. 19–29.

[10] Gurfinkel, A., Kahsai, T., Komuravelli, A., and Navas, J. A. The seahorn verification framework. In *International Conference on Computer Aided Verification* (2015), Springer, pp. 343–361.

[11] Havelund, K. Runtime verification of c programs. In *International Workshop on Formal Approaches to Software Testing* (2008), Springer, pp. 7–22.

[12] Hawblitzel, C., Howell, J., Kapritsos, M., Lorch, J. R., Parno, B., Roberts, M. L., Setty, S., and Zill, B. Ironfleet: proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles* (2015), pp. 1–17.

[13] Hristov, M., and Bieniusa, A. Erla$^+$: Translating tla$^+$ models into executable actor-based implementations. In *Proceedings of the 23rd ACM SIGPLAN International Workshop on Erlang* (2024), pp. 13–23.

[14] Kim, J., Hu, Q., D'Antoni, L., and Reps, T. Semantics-guided synthesis. *Proceedings of the ACM on Programming Languages 5*, POPL (2021), 1–32.

[15] Larson, M. L. *Meaning-based translation: A guide to cross-language equivalence.* University press of America, 1997.

[16] Le Charlier, B., Leclère, C., Rossi, S., and Cortesi, A. Automated verification of prolog programs. *The Journal of Logic Programming 39*, 1-3 (1999), 3–42.

[17] Liu, J., **Murphy, C.**, Grover, A., Johnson, K. J., Reps, T., and D'Antoni, L. Synthesizing formal semantics from executable interpreters. *arXiv preprint arXiv:2408.14668* (2024).

[18] Manna, Z., and Waldinger, R. A deductive approach to program synthesis. *ACM Transactions on Programming Languages and Systems (TOPLAS) 2*, 1 (1980), 90–121.

[19] Maryasov, I. V., Nepomnyaschy, V. A., Promsky, A. V., and Kondratyev, D. A. Automatic c program verification based on mixed axiomatic semantics. *Automatic Control and Computer Sciences 48* (2014), 407–414.

[20] Metere, R., Lindner, A., and Guanciale, R. Sound transpilation from binary to machine-independent code. In *Formal Methods: Foundations and Applications: 20th Brazilian Symposium, SBMF 2017, Recife, Brazil, November 29—December 1, 2017, Proceedings 20* (2017), Springer, pp. 197–214.

[21] Müller, P. *Modular specification and verification of object-oriented programs.* Springer, 2002.

[22] Pascutto, C. *Runtime verification of OCaml programs.* PhD thesis, Université Paris-Saclay, 2023.

[23] Pereira, M., and Ravara, A. Cameleer: a deductive verification tool for ocaml (extended version). *arXiv preprint arXiv:2104.11050* (2021).

[24] Shen, B., Zhang, W., Yu, A., Wei, Z., Liang, G., Zhao, H., and Jin, Z. Cross-language code coupling detection: A preliminary study on android applications. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)* (2021), IEEE, pp. 378–388.

[25] Si, X., Lee, W., Zhang, R., Albarghouthi, A., Koutris, P., and Naik, M. Syntax-guided synthesis of datalog programs. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2018), pp. 515–527.

[26] Solar-Lezama, A. The sketching approach to program synthesis. In *Asian symposium on programming languages and systems* (2009), Springer, pp. 4–13.

[27] **Murphy, C.**, Johnson, K., Reps, T., and D'Antoni, L. Verifying solutions to semantics-guided synthesis problems. *arXiv preprint arXiv:2408.15475* (2024).

[28] **Murphy, C.**, and Kincaid, Z. Quantified linear arithmetic satisfiability via fine-grained strategy improvement. In *International Conference on Computer Aided Verification* (2024), Springer, pp. 89–109.

[29] **Murphy, T. C.** *Relational Verification of Distributed Systems Via Weak Simulations.* Princeton University, 2023.

[30] Torlak, E., and Bodik, R. Growing solver-aided languages with rosette. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software* (2013), pp. 135–152.

[31] Unno, H., Terauchi, T., Gu, Y., and Koskinen, E. Modular primal-dual fixpoint logic solving for temporal verification. *Proceedings of the ACM on Programming Languages 7*, POPL (2023), 2111–2140.

[32] Wang, P., Cuellar, S., and Chlipala, A. Compiler verification meets cross-language linking via data abstraction. *ACM SIGPLAN Notices 49*, 10 (2014), 675–690.

[33] Yang, Z., Liu, F., Yu, Z., Keung, J. W., Li, J., Liu, S., Hong, Y., Ma, X., Jin, Z., and Li, G. Exploring and unleashing the power of large language models in automated code translation. *Proceedings of the ACM on Software Engineering 1*, FSE (2024), 1585–1608.