RESEARCH STATEMENT

XIAOKANG QIU

Computer software has revolutionized our daily lives, but software developers’ lives have not advanced commensurately. Programming remains “one of our most demanding intellectual activities” as E.W. Dijkstra said three decades ago [5]. As an error-prone, human process, it has led to defective programs that pose a severe threat to our society, including financial repercussions, security vulnerabilities, and even life-threatening hazards. For instance, at least 89 people were killed [1] as a result of bugs in Toyota’s electronic throttle control system (ETCS), reported in 2010 [6]. My research mission is to move away from the existing error-prone process to a computer-aided, easier, more reliable and more productive process.

The last decade has seen significant advances in the analysis, verification and synthesis of programs, leading to emerging tools that help programmers in various contexts, ranging from static checkers of run-time errors to interactive synthesizers that produce string-manipulations. However, one of the least understood classes of programs is heap-manipulating programs. Manipulations of dynamically allocated memory are pervasive in low-level computer systems: garbage collectors, OS kernels, device drivers, mobile browsers, etc. Moreover, the functional correctness of these programs is highly desirable, as they should provide a secure and trustworthy platform for higher-level applications. Unfortunately, building programs in this class eludes existing automatic techniques and poses one of the greatest challenges in software verification and synthesis. Successfully tackling heap-manipulating programs requires very close coordination of multiple techniques such as constraint solving, verification and synthesis. Each of them primarily focuses on a distinct challenging problem and is traditionally inharmonious with other techniques. Thus I envision a holistic approach that encompasses multiple research thrusts and pushes the boundaries of the tools in each of the seemingly conflicted dimensions, including expressivity, automaticity and performance.

To make my research vision become a reality, I have developed four systems that span the full spectrum of computer-aided heap programming. Each system contributes by developing a novel synergy of multiple distinct but complementary themes: STRAND [15, 16] is an expressive logic that combines graph-logic and data-logic, resulting in the first decision procedures that can prove properties of tree-like data-structures. Natural Proofs [17] is a radically new proof methodology that breaks the dichotomy between powerful but interactive verifiers and automatic but weak verifiers. VCDRYAD [20, 19] is a verifier in which separation logic, natural proofs and SMT (Satisfiability Modulo Theories) solvers are brought together, and can automatically verify the full correctness of a wide variety of challenging C programs, including a large number of well-known routines from open-source libraries (GTK library, OpenBSD, etc.) and operating systems (Linux kernel and ExpressOS [18], a secure mobile OS developed at UIUC). IMPSYNT [21] is a program synthesizer that enables natural synthesis, a novel verification-synthesis integration, producing provably-correct data-structure manipulations.

Besides the above systems, my research has also been leveraged by other researchers to develop techniques in areas beyond my own expertise. For example, a new class of automata [7] was defined on top of STRAND, and used to learn loop invariants for programs [8]; new proof strategies [4] following the Natural Proofs methodology were proposed for concurrent programming and successfully verified the Windows Phone USB driver. In what follows, I briefly review the representative four research projects and present an outline of my plan for current and future research.
**Representative Research Projects**

**STRAND: Decidable logics combining heap structures and data.** One of the challenges of reasoning with heap-manipulating programs is the lack of powerful decision procedures that answer queries involving properties that combine structures and data. Although several such decision procedures already exist, they cannot handle even the sorted-ness of a binary search tree: for every two nodes \( n_1 \) and \( n_2 \), if \( n_2 \) is in \( n_1 \)'s left/right subtree, the key stored in \( n_2 \) is less/greater than the key stored in \( n_1 \). Expressing this simple property requires a graph-logic (for structural relations such as “left subtree”) and a data-logic (for integer arithmetic such as “less-than”) that deeply interact with each other (using quantifiers such as “for-every”). Motivated by solving these kinds of queries, I proposed the STRAND (“STRucture ANd Data”) logic [15]. As its name suggests, STRAND has been designed to combine a powerful graph-logic with an arbitrary data-logic.

To allow both the graph-logic part and the data-logic part to express complex arithmetic properties involved in heap-manipulating programs, I allowed the \( \exists^* \forall^* \)-quantifier prefix, as it is powerful enough for many common data properties, e.g., sorted-ness described above. Then I identified a decidable fragment of STRAND [15] which admits a small model property—if a formula is satisfiable, it is always satisfied by a small model—and developed a decision procedure for it by combining automata theory and an SMT solver. Later I also identified a smaller decidable fragment that admits a much more efficient decision procedure (up to 1000X faster) [16]. Although more restrictive, the smaller fragment is equally expressive in practice.

My algorithms have been the first decision procedures that can prove full correctness of programs manipulating binary search trees. Due to the expressivity and decidability of STRAND, my decision procedures can benefit a variety of different techniques beyond deductive verification, e.g., software analysis or test generation. In fact, other researchers have followed my research and developed new classes of automata [7, 9] and algorithms that learn loop invariants [8] based on STRAND.

**Natural Proofs: Automatic, sound, and powerful reasoning for heap manipulations.** The decidability of STRAND allows a class of verifiers to boil down the whole verification task to a set of queries to the decision procedure, leading to push-button systems which are fully-automatic. Unfortunately, theoretical results suggest their complexity is high and their expressivity is limited. Another common class of verifiers are backed up by interactive, sophisticated proof systems that do not guarantee an answer and usually can only be used by experts. For building automatic yet powerful verifiers, I was motivated to break this dichotomy and reconcile these two opposites.

To this end, I proposed a radically new methodology called natural proofs [17]. The key insight behind natural proofs was that, to automate the verification process, it is unnecessary to thoroughly search for all possible proofs, which is the case for all decidability-based approaches. I studied the proofs of an almost exhaustive list of algorithms on tree-based data structures covered in the classical textbook [3], found a set of simple, deterministic proof tactics very useful, and called them natural proofs. The idea of the natural proof methodology is: a) to identify a class of natural proof tactics that mimics the human way of proving heap-manipulating programs; and b) to build algorithms that efficiently and thoroughly search this class of proofs. I have shown that many programs have natural proofs of correctness, and that the algorithm searching for only natural proofs can be implemented very efficiently, based on the fast-growing class of SMT solvers for quantifier-free theories [17].

We deployed this natural proof methodology for the specialized domain of tree-manipulating programs, developing DRYADtree logic [17], an extension of first-order logic (FOL) with recursive definitions, and building a graph-based algorithm applying the natural proof strategy. The algorithm successfully verified full functional correctness of data structures ranging from max-heaps, treaps, AVL trees, red-black trees, B-trees and binomial heaps. It was the first automatic methodology that guarantees an answer and proves such a wide variety of imperative tree-algorithms fully correct.

As a general proof methodology, natural proofs can be potentially exploited in many areas beyond heap-manipulating programs. Very recently, other researchers have explored natural proofs in the
context of verifying asynchronous, message-passing systems, and identified a set of entirely different strategies in that domain, which successfully verified the Windows Phone USB driver [4].

**VCDRYAD: Integration of structure, data, and separation.** DRYADtree demonstrated the concept of natural proofs, but it was not sufficient to handle the daily grind of real-world heap-manipulating programs. For any verification technique to scale, it needs to support local reasoning. Although Separation Logic (SL) enables effective local reasoning for heap-manipulating programs, its native solvers are usually weak or slow. Therefore I developed a more powerful, SL-based logic called DRYAD_{sep} [20] to integrate natural proofs and SL. DRYAD_{sep} extends SL with pure recursive definitions, and its semantics was carefully defined and translated into the classical logic, so that natural proofs and SMT solvers, which are significantly faster than native SL solvers, are immediately available to verify structure and data properties. We showed that the combination of separation logic, natural proofs and SMT solving is amenable to automated reasoning with structure, data and separation, and developed the first SMT-based program verifier against separation logic [20].

We also built an automated deductive verification tool called VCDRYAD [19], a verifier for C programs against DRYAD_{sep} specifications. VCDRYAD extends the VCC framework [2] and encodes natural proofs to ghost code for VCC. We developed a new set of techniques to build this framework, including a novel tool architecture that allows encoding natural proofs at a higher level and a synthesis of ghost-code annotations that captures natural proof tactics. Using VCDRYAD, we successfully verified more than 150 annotated C programs, ranging from standard data-structure manipulating routines to well-known routines from open source libraries (GTK library, OpenBSD, etc.) and operating systems (Linux kernel and ExpressOS [18], a secure mobile OS developed at UIUC). VCDRYAD is the most efficient deductive verification system that can verify the full correctness of such a wide variety of challenging C programs.

**Natural Synthesis: A marriage of natural proofs and inductive synthesis.** During my postdoc career at MIT, I have been interested in automatic synthesis of provably-correct heap-manipulating programs. In other words, the goal is, from the minimal specification of a programming task, to automatically synthesize a fully-correct concrete implementation, whose total correctness is always guaranteed. This requires us to build a system that produces not only a program, but also its proof artifacts, including loop invariants and ranking functions, such that an automatic verification can be conducted and passed. Existing synthesis systems, whether they are deductive or inductive, cannot achieve such an ambitious goal for heap-manipulating programs.

I tackle this challenging problem by proposing natural synthesis, a novel verification-synthesis integration, in which there are two key ingredients: natural proofs and inductive synthesis. They stem from different research areas, but turn out to be a perfect match. On the one hand, natural proofs are the state-of-the-art in heap verification, but it is not synthesis-enabled, i.e., the programmer needs to provide both a complete program and a full specification. On the other hand, inductive synthesis can effectively synthesize implementation details as well as annotations satisfied by any program execution, but it is hard to synthesize ones that can be proved by a verifier. Technically, the main idea of natural synthesis is to reduce an intractable, unbounded-size synthesis problem to a tractable, bounded-size synthesis problem, which is amenable to be handled by modern inductive synthesis engines. In essence, this reduction encodes the gist of natural proofs so that the soundness of the reduction is guaranteed.

As a result, I designed IMPSYNT, a synthesis-enabled language that allows the user to describe only a high-level skeleton of the program, and developed a natural synthesizer that can produce imperative, data-structure manipulations and their proofs in tandem, from these skeletons [21]. Experiments have shown that this natural synthesizer can efficiently produce provably-correct implementations for sorted lists and binary search trees. This is the first synthesizer that can automatically generate such sophisticated imperative programs, with minimal user inputs and such rich functional correctness guaranteed.
Research Agenda

I am excited to continue building automated programming systems. Besides continuing existing projects and expanding previous systems, I plan to investigate several important problems that need to be solved in order to develop next-generation computer-aided programming techniques.

Engage with the program. Today’s heap verification and synthesis tools still focus on small pieces of code. The next step is to go beyond these code snippets and target software of industrial scale. As modern software is built in a stack, from operating systems, libraries, and programming infrastructure, to a myriad of client applications, programming tools should also be built in multiple layers. For a higher-level tool to scale, it should rely on an abstract model of the lower-level framework, which might be too complex, evolvable, or even unavailable.

To make these layered programming tools feasible, besides developing more powerful synthesis engines by combining explicit and symbolic search techniques [14, 13], I also focus on generating abstract models for lower-level frameworks. Very recently, I have made substantial contributions to PASKET [11], a first step toward automatically generating Java framework models. PASKET takes as input the framework API, together with tutorial programs that exercise the framework, and emits framework models by instantiating design patterns to the framework API. This tool has been able to synthesize models for a subset of Swing and a subset of Android.

Once models of frameworks can be automatically synthesized, another key question arises: can we leverage the synthesized models to help develop client code? Possible usage scenarios include automatically maintaining client code when the underlying framework is updated, synthesizing algorithms by learning from a corpus of applications, etc. Developing these techniques will free programmers from concerning themselves with the framework’s subtle implementation details.

Engage with the programmer. Another limitation on the widespread usage of logic-based programming systems has been the demand for writing logical specifications. I have come to believe that for any programming tool to have a significant impact, it must be accessible to countless end-programmers who are not trained to describe programs using incomprehensible logics and mathematics. Programmers should be allowed to flexibly use a wide range of artifacts that are best-suited to their particular development tasks. For example, a sorting algorithm can be easily described by an average programmer using a declarative specification; but when it comes to rotating a binary tree, the more natural format to express the insight will be concrete input-output examples. It is a great research opportunity to develop programming tools that reconcile various artifacts, leading to a more intuitive and less error-prone way of programming.

I recently presented a new language, JSketch [12], as my first step in this direction; this language makes sketch-based synthesis directly available to Java programmers. My collaborators and I also have been developing a synthesis tool called SyntRec [10], in which the user is allowed to define spaces of programs in a highly reusable manner; this enables end-programmers to reuse pre-defined program spaces for a variety of problems with very little effort.

I also plan to explore other interesting questions. For instance, 1) When the programmer provides multimodal specifications, such as examples, templates, graphs, etc., can we build a learning algorithm that summarizes the programmer’s intention as formal pre-/post-conditions, in the form of logic formulas or abstract domains? 2) When the user-provided specification is insufficient, e.g., not inductive, can our system provide helpful hints, allowing the programmer to strengthen the specification? 3) When students submit buggy programs for an entry-level programming course, can our tools find a counterexample from a failed proof and suggest some revisions of the program? I believe the answers to these questions will lead to a highly promising future of computer-aided programming, as well as new programming systems that will drastically improve programmers’ ability to efficiently write bug-free programs.
References