My research interests span Formal Methods and Logic and their applications to Programming Languages, Software Engineering and Cyber-Physical Systems. Software systems are becoming increasingly more complex, are being deployed at massive scales, and are entering all facets of our lives. As a result, engineering such systems is getting extremely challenging and critical bugs routinely occur in production-level software, compromising their reliability, and often leading to severe consequences. An overarching theme of my research has been to re-investigate the foundations of traditional solutions for intractable problems arising in testing, verification and synthesis of software systems. My PhD research has, so far, primarily focused on (a) designing scalable and effective dynamic program analysis algorithms for exposing concurrency bugs in multi-threaded programs, and (b) expanding the theoretical boundaries in decidable program verification.

Despite decades of research, concurrency bugs like data races still remain notoriously hard to detect. Traditional approaches for automatically finding data races miss many data races, fail to scale due to high performance overhead or report many false alarms. In my research I have developed new partial orders and algorithms thereof [1, 3] and investigated new algorithmic paradigms like compression [2] that provably address these challenges. I have extended these insights to detect broader classes of concurrency bugs [14, 4].

Automating program verification is another problem in Computer Science that is well-known to be intractable (in fact, undecidable). I have identified the first class of programs called coherent programs [5], which work over infinite domains and can be verified completely automatically without the use of any loop invariants. I have further generalized this class of programs [7] and have verified commonly occurring heap-manipulating library routines using my techniques, without any manual intervention [6]. I also showed that coherent programs can be synthesized completely automatically [15], making it the first decidability result for synthesis of programs with loops working over infinite data domains.

My research has laid solid new foundations for building scalable techniques for ensuring trustworthiness of software systems. The techniques I have developed are applicable to a broad class of systems and areas and to domains beyond testing and verification. In the future, I plan to enhance these techniques to meet the challenging demands of large-scale software development environments, develop new techniques for automatic repair of software bugs and novel frameworks for automating and boosting verification of large scale software.

## Dynamic Analyses for Concurrent Programs

Concurrency is an indispensable paradigm in modern software applications, but it can be extremely tricky to get right. Concurrency bugs inevitably find their way into production-level software despite rigorous development-time testing. Data races are arguably the most insidious amongst concurrency bugs [25] and are particularly hard to detect. Traditional dynamic race detection techniques, such as those based on Lamport’s happens-before (HB) partial order [26], fall short on many fronts. I have developed novel insights and dynamic race detection techniques to address these shortcomings. I have further extended these insights to design algorithms that can be used to detect other concurrency bugs like deadlocks and atomicity violations. Each of these techniques are carefully designed to ensure the following desired properties. First, all of these techniques are sound, i.e., they do not report false alarms; absence of soundness implies that developers need to spend time manually confirming every reported bug, rendering unsound techniques impractical. Second, these techniques have high predictive power (high recall), i.e., they can effectively uncover hard-to-find bugs. Finally, these algorithms have been developed with scalability in mind — they work in a streaming setting, processing events as soon as they are generated, and run in linear time; any algorithm that works in super-linear time is unlikely to scale to industrial applications that typically generate billions of events. My techniques are the first to simultaneously achieve soundness, linear time and high predictive power, after four decades of research in this area.

**Contributions.** Traditional HB-based race detection techniques have poor predictive power and miss many subtle data races. Software architects need to run race detectors on their programs multiple times to achieve even modest levels of assurances. In each run, the race detector observes a different program execution and attempts to find an answer to the question — “Does the observed execution have a data race?”. I developed a novel partial order Weak Causal Precedence (WCP) [1] and a race detection algorithm based on WCP to instead answer the predictive analogue to this question — “Can we reorder the observed execution to expose a data race?”. WCP-based race prediction is sound, and more importantly, has higher predictive power than HB — WCP based race prediction requires fewer executions than HB to uncover the same set of data races. Further,
WCP admits a linear time streaming vector clock algorithm and scales to real world applications. In fact, WCP is the first (and so far the only) race prediction technique that simultaneously achieves both soundness and scalability. I extended these insights to develop the first linear time sound algorithm for the deadlock prediction problem [14] — “Can the observed execution be reordered to expose deadlocks?” and achieved an average speed-up of more than 16,000× over the prior state-of-the-art deadlock prediction technique [18].

Extremely large trace sizes in applications like web browsers, coupled with high runtime overhead due to dynamic analyses warrants that analysis be performed offline to avoid performance degradation. Storing large traces, however, demands expensive warehousing and compressing these traces is a natural first resort. Can we detect data races on compressed traces? I have developed algorithms [2] and tools [19] to detect data races directly from compressed traces, without needing to first decompress them. These algorithms leverage grammar-based lossless compression schemes (like those used in zip or tar) and work in a compositional manner — finding a data race in a trace can be reduced to finding data races in smaller sub-traces. Grammar-based compression schemes can give exponential compression ratios. Further, my algorithms run in time that is linear in the size of the compressed traces, suggesting that compression can, in fact, be used as an algorithmic principle to speed-up race detection in an offline setting. The evaluation of the algorithms suggested that compression can leverage repetition in code, giving a speed up of up to 2500×, when compared to race detection on uncompressed traces. This work received a Distinguished Paper Award at ESEC/FSE 2018.

Race detection based on happens-before (HB) is widely believed to not report false races. Standard race detectors based on HB, including Google’s ThreadSanitizer [27], however, end up reporting many false alarms — I found that about 50% of race reports were false positives when testing real world software applications like web and database servers [3]. This apparent contradiction to the theoretical soundness guarantee of HB arises because HB only guarantees the correctness of the first race it reports; beyond this, HB does not guarantee soundness. Can we report multiple races without any false alarms? I proposed a new partial order - schedulable happens-before (SHB) that answers this question — SHB reports all true HB races without reporting any false race [3]. SHB is thus the truly sound version of HB and is, so far, the only technique ensuring soundness beyond the first race. I proposed a linear time algorithm for detecting races using SHB which is so similar to the original HB vector clock algorithm that existing HB-based race detectors can very easily patch their implementations to regain their soundness guarantees.

I demonstrated the effectiveness of vector clocks for yet another problem — checking atomicity violations [4]. Developers often annotate code blocks to be @atomic. The underlying runtime may, however, not respect these annotations and may arbitrarily context switch while executing these nominally annotated code blocks. Checking atomicity dynamically amounts to checking if an observed execution can be reordered to obtain an equivalent execution which executes @atomic code blocks serially. Prior to my work, the only sound and complete algorithm known for this problem was a graph based quadratic time algorithm [28] that fails to scale to real world executions. I developed AeroDrome [4], the first linear time algorithm using vector clocks for detecting atomicity violations. AeroDrome scales to executions with billions of events, giving a speed-up of up to 48,000× over prior state-of-the-art on a comprehensive suite of software benchmarks.

Decidable Program Verification

The increasing complexity in the design of software applications, execution environments, and desired specifications renders software testing inadequate to ensure robustness of software systems. Ensuring the complete absence of bugs entails formal software verification by establishing mathematical proofs of correctness against specifications. Manually writing these proofs, however, requires substantial effort and expertise, both of which are unreasonable to expect from software developers. Completely automatic verification, on the other hand, is undecidable in general. Nevertheless, automation has been the key propellant of the industry adoption of formal verification [29, 30, 31, 32]. In my research on verification of uninterpreted programs, I have investigated the boundaries of decidable verification and have identified the first class of infinite domain programs that can be verified completely automatically.

Contributions. Prior to my work [5], the only decidability results known in the context of verification were limited to programs with finite state spaces. In my research on uninterpreted programs, I proposed the first class of programs that work over infinite domains and yet admit decidable verification. Uninterpreted programs are imperative programs that contain loops, conditionals, and assignments and use function symbols that do not have a fixed interpretation. Here, the verification question asks if a given uninterpreted program meets a desired specification over any domain (finite or infinite) it works on, no matter what interpretations
we assign to the function symbols used in the program. While this question is undecidable in general, the subclass of coherent uninterpreted programs that I have identified in my research gives hope — it admits decidable verification despite working over arbitrary infinite domains [5]. I also proposed the notion of $k$-coherence, a systematic (but incomplete) procedure to transform non-coherent programs to coherent programs by adding $k$ auxiliary program variables that do not affect program semantics.

I have extended the notion of coherence to heap manipulating programs [6]. Programs that manipulate heaps while allocating and deallocating memory dynamically are known to be particularly hard to reason with, primarily because of the aliasing problem — “Do pointers $x$ and $y$ point to the same memory location?” I developed a decision procedure for verifying heap manipulating programs and used it to automatically verify common library routines for manipulating data structures like lists and trees [20].

For programs in the wild, proving correctness often requires reasoning about domain-specific properties of the function and predicate symbols they use. For example, the correctness of a sorting routine crucially relies on the property that the comparison operator, ‘$<$’, is a total order; such a routine cannot be proved correct when ‘$<$’ is assumed to be purely uninterpreted. I studied domain-specific properties of functions (associativity, commutativity, idempotence) and relations (transitivity, symmetry, reflexivity) and characterized classes of programs that retain decidability of verification in the presence of such properties [7], thereby extending the decidability results of [5] from purely uninterpreted programs to partially interpreted programs.

---

Other Work

**Automated Synthesis of Software and Cyber-Physical Systems.** The problem of automatically constructing programs from specifications, a.k.a program synthesis has gained a lot of traction in recent years, thanks to the promise of democratizing programming it offers. Nevertheless, it is an undecidable problem. A common theme to tackle this undecidability is to cast synthesis as a bounded search problem, by restricting the search space to loop-free programs of bounded size. I proposed a framework for encoding bounded program synthesis questions as satisfiability of quantifier-free first order logic formulae [8]. This framework allows one to directly leverage current advances in automated SMT-solving technology, without resorting to traditional ad-hoc enumeration-guided synthesis techniques. I used my framework to synthesize complex cryptographic protocols [22]. I proposed Eqsmt, a quantified fragment of second order logic over multiple theories that can naturally encode bounded program synthesis problems while admitting a decidable satisfiability procedure in the style of Nelson-Oppen combination. I investigated the question “When is unbounded program synthesis decidable?” and proposed the first decidability procedure for synthesis of programs, of unbounded size, containing loops and working over infinite domains [15]. I explored control-theoretic techniques to synthesize controllers for high-dimensional dynamical systems using current automated SMT-solving capabilities [10, 16, 21].

**Verification of Quantitative and Real-Time Properties.** Probabilistic models like DTMCs and MDPs are often used to model stochastic phenomena arising in biology, chemistry, computer networks and hardware engineering. State-of-the-art tools like PRISM [33] for model-checking these systems (against specifications like ‘Is the probability of reaching an error state $\geq \frac{1}{12}$?’) resort to approximations, in the absence of scalable exact model-checking techniques, and often report imprecise results. I proposed an efficient technique for exact model-checking of probabilistic systems that leverages approximate answers from existing model-checkers [11, 17, 23]. I developed algorithms and tools for quantifying information leakage by leveraging symbolic model checking [12, 24]. I proposed a class of hybrid system models for which reachability and schedulability problems become decidable [13], thus expanding the decidability boundaries in verification of cyber-physical systems.

---

**Plans for Future Research**

I envision a future where trustworthy software systems are a reality. Automated bug detection, repair and push-button verification are a key step in taking us closer to this reality. Developer tools based on traditional program analyses techniques are, however, often inadequate in realizing this vision and fail to meet the challenges posed in modern software development scenarios. I plan to extend the ideas developed in my research and create new foundations, tools and infrastructure for building reliable systems that address these challenges.

**Bug Detection in Contemporary Software Ecosystems.** I plan to develop techniques for automatic bug detection that cater to specific challenges posed in present-day software development scenarios — continuously evolving software repositories, collaborative development, complex software designs and intricate hardware architectures, strict quality standards, diversity in desired specifications (performance, security, privacy, crash...
freedom) and execution environments (geographically distributed, networked, extreme climate) and ambitious product delivery deadlines. Concrete directions that I want to pursue are as follows:

1. **Evolution-aware analyses.** When software evolves over several small diffs submitted by a large team, developers are more likely to address bugs that arise due to their own code changes, as compared to pre-existing bugs [34, 35]. Intensive dynamic analyses (such as race detection) to pinpoint bugs can however be prohibitive to run from scratch after every diff submission, especially in very large software ecosystems with hundreds of millions of test executions. I plan to develop evolution-aware techniques for predicting concurrency bugs that leverage insights of compositionality developed in my research [2], together with other advances in static analysis and regression testing.

2. **Predictive Runtime Monitoring.** Today, software applications are required to meet strict specifications like security, privacy and performance that go far beyond traditional ‘absence of crashes’. Often, these non-functional specifications are described as temporal patterns — "memory location pointed to by x is never freed twice without an intervening reallocation", "resource R is never accessed without prior authentication", or, "a leader election protocol reaches consensus within first N rounds". Runtime monitoring has emerged as a powerful lightweight framework to find violations of such properties. However, the effectiveness of existing runtime verification techniques, in the presence of concurrency is reliant on specific thread interleavings, leading to a needle-in-a-haystack problem. As with data races [1], one can ask the predictive question — *can a single observed execution be used to infer violations of a temporal specification in some reordering of the execution?* I plan to develop new theoretical foundations and efficient algorithms for online predictive monitoring to dynamically predict bugs arising from violations of such temporal specifications.

3. **Concurrency Bugs in Diverse Architectures.** My research efforts have so far been geared towards detecting concurrency bugs in shared-memory multi-threaded programs. Concurrency in industrial scale systems can however give rise to more intricate challenges. These include relaxed behaviors such as instruction reordering and out-of-thin-air reads arising due to weak memory models, geographic distribution, crash recovery and eventual consistency in cloud systems, and unbounded parallelism in event-driven Android applications. I plan to address these challenges by extending the insights from my research on concurrency bug detection.

**Automated Repair of Software Bugs.** While automated detection of bugs is crucial in assisting developers write quality code, it only solves a part of the problem. Engineering reliable software also warrants fixing the bugs identified by these automated techniques. Patching bugs in large code bases can however be a tricky task, especially so when programs use concurrency constructs — programmers need to reason about exponentially many interleavings to check if the patch indeed fixes the problem in all thread interleavings. Naively fixing concurrency bugs can introduce other concurrency bugs or result in significant performance degradation. An ideal patch should ensure correctness as well as reasonable performance. I plan to develop techniques for assisting software engineers to generate code patches for fixing complex synchronization issues in concurrent and distributed systems. I believe that ideas from my PhD research, in conjunction with new solution paradigms like learning fixes from existing software repositories using data driven and machine learning techniques, can be effectively used to engineer automated repair technology.

**Automatic and Deductive Verification for Large Scale Software.** Large scale software systems such as those operating in a distributed setting are prone to subtle bugs that evade exhaustive testing and often surface only in production. Deductive verification — establishing program correctness using a machine-checked formal proof — has recently shown promise in developing large-scale trustworthy implementations like the CompCert compiler [36], and Project IronFleet by Microsoft Research [37], paving the way towards correct-by-construction systems. In a deductive verification pipeline, one typically annotates smaller blocks of code with proof objects or lemmas and checks if these lemmas are consistent with each other. Coming up with annotations for code blocks with loops (or loop invariants) is arguably the hardest task of this pipeline. As a result, end-to-end verification of industrial scale software requires multiple years of manual effort making it a daunting task. My research on decidable verification [5, 6, 7] can be used to automatically generate loop invariants, and thus, has the potential to speed up large scale verification efforts. The automata-theoretic decision procedures developed in my research [5] can further be used to develop abstraction based automatic verification techniques for large-scale software. I plan to gear my future research efforts in these directions.
Publications


Unpublished Manuscripts


Software Tools


Other References


[34] M. Harman and P. O'Hearn. From Start-ups to Scale-ups: Opportunities and Open Problems for Static and Dynamic Program Analysis. SCAM 2018.

