Swarat Chaudhuri

Research statement [1]

As the numerous patches downloaded every year and news stories featuring software failures would attest, software is getting buggier even as it pervades more aspects of our lives. On one hand, traditional testing techniques have been outmatched by the global explosion of code; at the same time, the deployment of software in critical areas has rendered software reliability more important. This poses many opportunities as well as challenges for researchers. Opportunities, because formal techniques to specify, analyze, test and formally verify software increasingly seem to be not only interesting science, but practical necessities. Challenges, because a great many theoretical and experimental questions about formal methods in software are still unanswered. My aim as a researcher is to answer some of these questions.

More specifically, my research interests so far have been in mathematical logics and algorithms that can aid specification and analysis of software. The last decade has seen a flurry of research, often blurring area boundaries, on these topics. The verification community has proposed applying *model checking*, an algorithmic technique successfully applied to the verification of finite-state reactive systems over the last 25 years, to prove correctness of software. The static analysis community, historically more motivated by programming language implementation, has begun to apply its expertise in scalable, approximate analysis to this domain. Such cross-fertilization has produced a plethora of tools, some quite successful. Microsoft's Static Driver Verifier, for instance, is now part of a standard kit used by Windows device driver developers. Tools like MOPS that search for error patterns in source code have scaled to very large code bases. Developers in many Windows groups are now required to write interface specifications that are checked statically.

What challenges must we address to obtain the next generation of tools for software analysis? I believe that an answer to this question requires a scrutiny of the state of the art of formal methods from a foundational perspective. What is known about the algorithmic problems that our analyses must solve? Are their specification notations expressive enough? For the last few years, my research has tried to address these questions. While work in this space builds on the rich literature on model checking of hardware and finite-state models of programs, it has to grapple with several additional issues. A crucial one is that control flow of procedural programs depends on "contexts" defined by the call stack, so that any reasonably precise software model checker needs to be "context-sensitive." This amounts to analyzing *pushdown models* of programs, which capture the semantics of the program stack while abstracting out all other program data.

Arguing that traditional temporal logics like the μ-calculus are not the best tools for specifying procedural programs, I have proposed formalisms [4,6,7] that, on one hand, unify insights from real specification languages and directly apply to model checking, and, on the other, are more expressive that existing notations. On the algorithmic end, I have designed the first subcubic algorithm for reachability analysis of pushdown program models [2], solving a long-open question.

Logics and automata for software specification In my doctoral thesis, I address a basic question in formal methods: what are the right logical foundations for software specifications usable in model checking? The most general traditional answer has been "regular languages of trees." In this view, a program generates a tree whose paths encode all possible executions, and a specification defines the set of "acceptable" trees. In typical software model checkers, a pushdown model of a program is algorithmically analyzed so as to determine whether the tree it generates is "acceptable". The specification may be given as a tree automaton or a formula in a temporal logic like the μ-calculus.

This notion of specifications was inherited from verification of reactive systems. Is it the right notion for software? There is a case that it may not be. Consider Hoare-style pre/post-conditions: "If a lock is held at a procedure call, then it must be released by the time the *matching* return is reached." A regular language of trees, which cannot argue about the balanced-parenthesis structure of calls and returns in programs, cannot express this property. Other requirements it cannot express include data-flow properties involving local variables and stack-sensitive access control requirements in security. While context-free languages can, they are not closed under intersection and complementation and cannot be used for model checking. On the other hand, there are real-life specification languages able to capture some properties with this flavor: Microsoft's interface specification language SAL, for example, is a pre/post-condition language used to analyze Windows code. What is missing is an understanding of the trade-off between expressiveness and decidability: which of these properties can be statically checked, and why? While this question has foundational merit, an answer to it would also lead to expressive, well-understood notations for *context-sensitive software specification*.

This is where *languages of nested trees* step in. In a few recent papers [4,6,7], we have proposed to model the unfolding of a program by a *nested tree* rather than a tree (the linear-time case is handled by *nested words*). A nested tree is obtained by augmenting a tree with a set of extra edges, known as jump-edges, that connect a position where a call happens to all nodes modeling matching returns; in the special case where the underlying structure is a word, we have a nested word. As calls and returns in program executions are properly nested, jump-edges never cross. Temporal logics

[1] Citations refer to publication list in CV.

and finite automata accepting "regular" languages of such structures are now defined, and used to define specifications. The model-checking question becomes: does a pushdown model only generate a nested tree that are permitted by the specification?

Interpreting logics and automata on these structures turns out to make a major difference. On one hand, we can now state requirements involving the program stack as well as traditional temporal properties. For example, NT-μ, our fixpoint calculus on nested trees [6,7], can state pre/post-conditions, stack-sensitive access control properties, and context-sensitive dataflow requirements in addition to every property expressible in the μ-calculus (alternating automata on nested trees have the same expressive power). Moreover, these notations allow requirements on procedural programs to be specified in a more modular manner—for example, when a procedure calls another, we can state: "the invoked context satisfies property f, and on return to the present context, property g is satisfied." Thus, just as structured programming lets us compose code using procedure calls and returns, logics like NT-μ allows *structured specifications*. We argue that such specifications are, in general, more succinct and comprehensible than traditional ones.

At the same time, regular languages of nested trees have properties similar to regular tree languages, being closed under intersection and complementation and allowing product constructions with procedural programs. Most attractively, these formalisms permit model checking on pushdown models at the same cost as their classical counterparts. For example, model checking for NT-μ is not only decidable, but has the same complexity as for far weaker logics such as CTL. This is not all. In finite-state model checking, a modal fixpoint formula not only states a property, but syntactically encodes a symbolic computation. This is why hardware model-checkers like SMV use μ-calculus formulas as directives for fixpoint computation. Known model-checking algorithms for the μ-calculus on pushdown models, however, are complex and do not follow from the semantics of the formula. On the other hand, a formula in NT-μ syntactically encodes a symbolic model-checking algorithm generalizing the interprocedural fixpoint computations known in program analysis for years.

Consequently, we believe that these logics and automata form a foundational advance in software specification. A real application based on this theory is a notation—called PAL [1]—for safety monitors based on nested word automata. Given a C program and a PAL monitor, our compiler generates an instrumented C program that fails an assertion if and only if the original program violates the safety requirement. The instrumented program may then be checked for assertion failures via testing or run-time verification as well as static analysis or model-checking. While PAL belongs to a family of software instrumentation languages such as BLAST or SLIC, it can, unlike its cousins, express context-sensitive, and structured, program requirements.

Algorithms for pushdown automata The *reachability problem* for pushdown automata is to determine if a given pushdown automaton can reach a certain control state along some execution. This problem is equivalent to checking whether the pushdown model of a program violates a safety property, and is central to software model checking. It is also equivalent to Datalog chain query evaluation and a graph problem called CFL-reachability, and arises in numerous program analysis contexts, including interprocedural slicing and data-flow analysis, type-based flow analysis, and important variants of pointer and shape analysis. While a cubic algorithm for this problem is easily obtained by generalizing a dynamic programming algorithm for context-free parsing known since the 1960s, it was not, unlike parsing, known to have a subcubic algorithm. In fact, its intrinsic cubic complexity was believed to be behind the "cubic bottleneck" of its many applications.

In a recent result [2], I have obtained a new algorithm for this problem that exploits a form of caching to speed up the underlying dynamic program, and runs in (slightly, but asymptotically) subcubic time. In the same paper, I also give better algorithms for reachability analysis of two classes of pushdown models with limited recursion, identifying a gradation in the complexity of program analysis as recursion is constrained. One of these algorithms is obtained via a new algorithm for transitive closure of graphs that matches the best non-algebraic algorithms for this problem, is particularly suited to sparse graphs and partial closure computation, and may be of independent interest. In current work, I am applying the main ideas behind these algorithms to design heuristics for alias analysis.

Other research Imagine a scenario where the interactions of a program with its environment are being observed by an eavesdropper, and consider the question: "can the eavesdropper infer some of the program's secret data from its source code and these observations?" Such information-flow requirements have previously been expressed using type systems. In a recent paper [3], we introduce a model checking approach suitable for these questions. It turns out that traditional temporal logics are not able to express requirements as above, and a new class of logics are necessary.

I have also worked on the theory of tree automata. While studying automata on nested trees, we realized that pushdown tree automata as defined in the literature cannot express combinations of branching and pushdown properties. In a recent paper [5], we propose an alternative, known as *branching pushdown tree automata*, that remedies this limitation in expressiveness while retaining decidability.

In the less recent past, I have worked on state-space search heuristics for model checking [8] and compression algorithms [9].

Future work In the next few years I want to continue working on software analysis, with a particular focus on analysis of concurrent software and new algorithmic techniques. While doing so, I hope to create and mix techniques in static analysis and formal methods as well as logic, concurrency theory, and algorithms. While I want to obtain theoretical results on these topics, I will also, jointly with students, build systems that implement them.

Among the problems that excite me the most currently, *analysis of concurrent software* is at the top. While concurrency has become a critical issue of late because of the emergence of the web and multicore architectures, debugging or analyzing concurrent software is notoriously difficult. An error in a multithreaded program may be triggered by scheduler behavior that is impossible to replicate while debugging, and algorithmic verification becomes undecidable very easily. Can these difficulties be overcome?

My angle on this would be lightweight, automated formal methods exploiting the structure of the class of programs to be analyzed. What kind of program properties do we need to prove? What kind of program abstractions would be meaningful here? Can decidability results in concurrency theory be applied meaningfully? Can we use SAT-solvers, lightweight theorem provers, and symbolic search intelligently? To answer these questions, I will examine concurrent programs from specific domains—for example, device drivers or browser plugins—and try to understand the insights needed to debug and verify them. Concomitantly, I will dig into the rich literature on concurrency, abstract interpretation, and formal methods, and look for insights applicable in this world.

Another question is analysis support for emerging languages for concurrent programming. Believing that the present metaphors for concurrent programming will not scale to a world where concurrency is pervasive, many researchers are designing new languages for concurrent programming. This raises a question: how do we specify and statically reason about programs in these languages? On one hand, the extra structure offered by these languages may cause some hard analysis problems to become tractable. On the other, we may have to produce new formal models and algorithms for such analysis. Either way, this question offers theoretical challenges as well as practical impact, especially as these languages are at an embryonic stage and insights about what can be analyzed will possibly affect their design.

Another general direction is to design better *algorithms and heuristics for software analysis*. My current work in applying the main idea of my CFL-reachability algorithm would fall in this category. Also challenging would be to find new, creative ways to use SAT-solvers, decision procedures, and BDD-based search, or to design algorithms with better cache performance. I am also interested in *modular and incremental software analysis*. Can we segregate analyses for different software components, so that "certificates" summarizing these analyses can be composed to certify the whole system? My graduate research may be of interest here: our model-checking algorithm for NT-μ proves that a property holds at a procedural context by composing proofs for contexts invoked via procedure calls. As for incremental analysis, can we efficiently maintain an analysis of a program that gets modified constantly, perhaps as the programmer works on it in an IDE? How do the foundations of such analysis relate to the substantial literature on dynamic graph algorithms?

Another broad direction involves applying *model checking and static analysis in software security*. My work on model checking secrecy properties of programs would belong to this category—also, one of the important applications of the work on nested trees is to state access control requirements. What other kinds of security vulnerabilities can we capture using formal logics? Can we expand these formalisms into a notation for structured, realistic security policies and statically prove that programs do not violate such policies?

Another problem, on the theoretical end, is to find *new complexity measures* for program analysis algorithms. Many algorithms in this domain are known to perform better in practice than their worst-case complexity would predict. Is there a way to quantify the structure in real-life programs that causes this phenomenon? A related problem is to build a complexity model for algorithms that use symbolic data structures or calls to SAT-solvers. Can we classify algorithms by the number of SAT instances that they generate and the complexity of these instances, for example?

I am sure that only some of these directions will bear fruits, and many more problems will present themselves as time passes. To conclude for now, I see the area of software analysis as being full of problems with practical impact as well as theoretical intrigue, and am excited by the idea of starting a research program to contribute to it.