Research Statement

Ranjit Jhala

The success researchers have had in improving the *performance* of computer systems has led to heavy dependence upon computing machinery. Prudence dictates that we depend only on systems which are *reliable*, as the price of bugs leading to loss of throughput, exploitable vulnerabilities, or crashes, is millions of dollars as well as human life. The question my research addresses is: How can we *guarantee the reliability* of computer systems?

Civil engineers know that bridges they have designed will not collapse by using the building blocks of those designs, the laws of mechanics, to reduce the question of reliability to a system of equations which can be mechanically solved. The building blocks with which computer systems are built are the laws of logic. The aim of my research is to harness those laws to devise methods that reduce the question of reliability to a system of equations and constraints and efficiently, mechanically analyze them.

While such mechanical "formal verification" has been a goal since the work of Dijkstra, Hoare, and Floyd in the 1960s, it is only now becoming a reality. The combination of Moore's law with new techniques for model checking using automatic abstraction, decision procedures, program analysis, type systems and a shift of focus from total correctness, which is hard to even specify, to properties common to several systems (*e.g.,* memory safety and race freedom) has resulted in practical verification methods. These, coupled with the prohibitive cost of errors have led to the acceptance of formal analysis tools like PrefAST and Static Driver Verifier (SLAM) at Microsoft and the J2EE Java Code Validator at IBM, and the widespread use of formal circuit verification in the EDA industry.

The question of guaranteed reliability motivates work on two fronts. First, developing *precise and scalable checking technology*, that can analyze large, complex systems for rich properties and either prove that they meet their specifications, or, if not, tell the programmer why by exhibiting a counterexample behavior. Precision, to ensure the analysis does not report bogus errors or claim a buggy system is correct. Scalability, so that the method works for large systems where the need for analysis is most acute. Second, to build reliable systems we cannot just perform checks *after* building the system but must *integrate* checking technology into *Software Engineering* processes by finding ways to: check systems as they are designed and built; use checking technology in traditional *testing* applications; and, build systems that are easily checked by identifying the information required for checking and making this easily inferrable from the code, *e.g.,* by finding how to effectively use interfaces to make checking modular.

Verification by Effective Abstraction

My thesis attacks the first front by giving novel techniques for precise and scalable checking of software by automatically finding *effective* abstractions, *i.e.,* abstractions coarse enough for efficient analysis and fine enough to avoid spurious counterexamples. These techniques are implemented in a tool BLAST [5] which can verify safety properties of sequential and multithreaded C programs. We have used BLAST to verify that Windows and Linux Device Drivers call kernel API functions correctly, to verify security properties of Linux programs, and to check for data races in Networked Embedded Systems applications.

Precision and scalability have hitherto been mutually exclusive. Type and flow based analyses are scalable as they track only a fixed small domain of facts, which leads to false positives (spurious counterexamples/errors) when checking complicated properties. Model Checking approaches are precise but are choked by state explosion as they track too many facts. One way to find an effective abstraction is to begin with a trivial abstraction, and iteratively tune it using false positives. Several hurdles must be crossed to make this practical.

Lazy Abstraction. For large programs, a monolithic abstraction that tracks all the facts used at different places of the program, is too detailed to be efficiently analyzed. We have devised a method *Lazy Abstraction* [1] that abstracts incrementally, on-the-fly while exploring the state space, and thus, by detecting where to track extra information, builds an abstraction that has different precisions at different places. Unlike previous analyses, lazy abstraction combines precision and scalability, by using the insight that precision can be *localized*. A critical part of any "counterexample-guided" analysis is how to systematically refine abstractions by analyzing spurious counterexamples. We provided a breakthrough by showing how to partition *proofs of unsatisfiability* of the counterexamples to refine abstractions [7]. Using these techniques, we are, for the first time, able to analyze C programs with more than 100K lines for complicated behavioral safety specifications,

and we have found several errors. For safe programs we construct small *machine-checkable proofs* that the program satisfies the property [2].

Context Inference. Multithreaded systems are notoriously difficult to program and validate as they have unexpected control flow due to thread interleavings. Our approach is to analyze such programs as a single thread executing in a *context*, an abstraction of the behavior of *all* the other, possibly infinitely many, threads. Consider the problem of checking for data races: states where multiple threads can access a variable, with at least one access being a write. A basic requirement for the correctness of concurrent systems is the absence of races. In the large body of work on this problem, the proposed solutions are limited to programs that use locks to enforce race-freedom. Contexts let us remove that restriction, and let us precisely analyze large programs using any protection mechanism. In [3, 9] we define two novel kinds of contexts and give the first algorithms to infer such contexts, by viewing them as nested fixpoints. These new techniques allowed, for the first time, precise race checking of complex networked embedded systems applications which use protection mechanisms like state variables and the sophisticated use of interrupt disabling to prevent data races.

Future Work

I eagerly look forward to exploring, over the next few years, several promising ideas that I have on improving checking technology and integrating it with Software Engineering processes.

Checking Technology. I would like to extend my work to deal with modern languages like *Java*, which are hard due to sophisticated control flow mechanisms (virtual methods, exceptions), but which provide richer type systems which can simplify analyses. Another immediate area is how to check for more complicated properties such as those which involve *heap data structures*; I believe my work on counterexample analysis [7, 4] provides a foundation on which automatic scalable heap analyses can be built. I wish to use *mixed abstractions*: overapproximations to ensure coverage and local underapproximations to reduce the number of infeasible behaviors explored by the analysis, thus making checking more efficient. Mixed abstractions can be used to analyze *heterogenous systems* made up, say, of some control software written in C, and a hardware component described better by a HDL. The same abstract state would be the conjunction of two different abstractions for the different components. Another direction is to devise analyses that return *partial counterexamples*, *i.e.,* traces with holes in them which correspond to paths through procedures that are either difficult to analyze or for which code is not available. There are many opportunities for *optimization for verification* (as opposed to efficient execution). These range from adapting methods like slicing and partial evaluation to reduce the program to be checked, to radically different approaches like automatically transforming programs with pointers into semantically equivalent programs that are free of aliasing, and hence, easier to verify.

Reliability Engineering. The ultimate success of checking technology will depend on how well it can be integrated into Software Engineering processes. In [6] we take some early steps in this direction by showing how to reuse proofs generated in prior checks to *incrementally* verify programs while they are being developed. This gives the programmer immediate feedback about what might break in the system, and what might be hard to verify. There are several ideas we have about combining verification with testing, the traditional way of validating systems. In [8] we solve the key problem of finding test vectors with guaranteed coverage by showing how they can be generated using counterexamples. Many programs use libraries which cannot be statically analyzed as the source is unavailable. This suggests *partial* verification where we statically analyze the source at hand, to determine what behavior (as a system of constraints), if any, of the library keeps the system safe. The constraints can be used to generate a test suite for the library, as well as to insert run-time checks into the program. Even without specifications, checking technology can be used to query semantic properties of programs. In a large security sensitive program, the programmer may wish to know which lines can execute with the system holding certain permissions. This leads to viewing *programs as databases*, the data being syntactic objects like lines, procedures, files, *etc.*, and the attributes being semantic state properties. Precise checkers can be used to answer queries about which locations have some semantic property, and to return a test vector which demonstrates the same. One goal of researching automatic checking technology is to learn its limits, and thus learn the minimal annotation burden that the programmer must bear and also how to build systems in ways that avoid hitting these limits. One well-known way to overcome scalability barriers is to use *interfaces* to modularize checks. I would like to explore ways to modularize checking for rich behavioral interfaces, as well as extend my work on context inference, to automatically infer interfaces from the code.

References

[1] T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL 02: 29th Symp. on Principles of Programming Languages*, pages 58–70. ACM, 2002.

[2] T.A. Henzinger, R. Jhala, R. Majumdar, G.C. Necula, G. Sutre, and W. Weimer. Temporal-safety proofs for systems code. In *CAV 02: 14th Intl. Conference on Computer-Aided Verification*, LNCS 2404, pages 526–538. Springer, 2002.

[3] T.A. Henzinger, R. Jhala, R. Majumdar, and S. Qadeer. Thread-modular abstraction refinement. In *CAV 03: 15th Intl. Conference on Computer-Aided Verification*, LNCS 2725, pages 262–274. Springer, 2003.

[4] T.A. Henzinger, R. Jhala, and R. Majumdar. Counterexample-guided control. In *ICALP 03: 30th Intl. Colloquium on Automata, Languages and Programming*, LNCS 2725, pages 262–274. Springer, 2003.

[5] T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with BLAST. In *SPIN 03: 10th Intl. Workshop on Model Checking of Software*, LNCS. Springer, 2003.

[6] T.A. Henzinger, R. Jhala, R. Majumdar, and M.A.A. Sanvido. Extreme model checking. In *International Symposium on Verification*, LNCS. Springer, 2003.

[7] T.A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In *POPL 04: 31st Symp. on Principles of Programming Languages*. ACM, 2004.

[8] A. Chlipala, T.A. Henzinger, R. Jhala, and R. Majumdar. Generating tests from counterexamples. In *ICSE 04: 29th Intl. Conference on Software Engineering*. ACM/IEEE, 2004.

[9] T.A. Henzinger, R. Jhala, and R. Majumdar. Race checking by context inference. *Submitted to PLDI 04: ACM Symp. Programming Language Design and Implementation*, 2004.