Automatic Property Checking for Software: Past, Present and Future

Sriram K. Rajamani

Microsoft Research India

1 Overview

Software validation is a very hard problem. Traditionally, most validation in our industry has been done by testing. Testing is the process of running software on representative inputs and checking if the software behaves as intended. There are various granularities in which testing is performed —ranging from unit tests that test small units of the system, to system-wide tests.

Over the past decade, automatic property checking tools that use static analysis have started providing a complementary approach to software validation. These tools are intended to augment, rather than replace, testing. These tools do not typically ensure that the software implements intended functionality correctly. Instead, they look for specific kind of errors more throughly inside the program by analyzing how control and data flow through the program.

This short paper surveys the state of the art in property checking tools and presents the author's personal perspective on future research in this area.

2 Background

Finding errors using static analysis is not a new idea. Perhaps the earliest static analysis tool that has been widely used is the Unix utility Lint [22]. The fundamental difficulty in using any kind of static analysis to detect program errors is that the problem is undecidable and equivalent to Turing's halting problem. This implies that there will never be a perfect static analysis tool. Any approach to building error detection tools using static analysis needs to necessarily consider engineering tradeoffs, in addition to the science of static analysis.

Modern static analyzers have innovated primarily in two main areas:

- **Specification language.** Early tools like Lint check for common errors that can be characterized at the level of the programming language, such as referencing an uninitialized variable. Modern tools allow users to state the kind of errors they are looking for, such as expected orderings of API calls. Thus, modern tools have specification languages such as Metal's pattern language [20, 14], SLAM's SLIC language [2] and ESP's OPAL language [11], in which the property to be checked is stated. All of these languages have some notion of a state machine, associate patterns in program text with events that transition the state machine, and specify bad states that the state machine should not reach. The static analyzer then looks for code paths that drive the state machine to the bad states.
- **Engineering tradeoffs.** Modern static analysis tools have explored various engineering tradeoffs (such as precision, scalability, soundness, completeness and usability) for analyzing very large systems and focused on providing value to their users. A property checking tool is *sound* if it detects all violations of a property in a program. A tool is *complete* if it does not report any spurious errors. Due to undecidability, no tool can be both sound and complete. Testing, for example, is complete but not sound, since it misses behavior. Typical static analysis tools based on abstraction are sound but not complete. Several practical static analysis tools are heuristic in nature —they are neither sound nor complete, but have proved to be useful nevertheless.

Traditionally, attaching preconditions and postconditions to method boundaries has been widely advocated as the preferred method of writing specifications. The primary advantage of this approach is modularity and (consequently) scalability [13, 15]. However, in addition to the input and return parameters, most function calls have side-effects resulting from the use and update of global state. Reasoning with such programs requires object invariants, and there has been recent progress in methodologies and tools for stating and checking object invariants [4].

3 Heuristic analyzers

Heuristic analyzers such as PREFix [6, 24], PREFast [24] and Metal [14] do not attempt to cover all paths. Further, along each path they do approximations. However, they manage to exercise code paths that are difficult to exercise using testing. Thus they are able to detect property violations that remain undetected after testing. Due to their heuristic nature, they are neither sound nor complete. They manage false errors by using filtering mechanisms to separate high-quality error reports, and statistical techniques to rank error reports.

However, these tools have provided impressive utility to their users. PREFix and PREFast have been successful in reporting useful errors over tens of millions of Windows code, and are now used routinely as part of the Windows build process. Metal has similarly found useful errors over several millions of lines of open source code.

4 Sound analyzers

Sound analyzers explore the property state machine using a conservative abstraction of the program. Usually, the abstraction used is the control flow graph, augmented with the state machine representing the property. Thus, the analyses explores all the feasible executions of the program, and several more infeasible executions. However the analyses do not explore individual paths. Instead, they explore abstract states. The complexity of the analysis is typically the product of the number of nodes of the property state machine and the size of the control flow graph of the program. Thus, for a 100,000 line program, and a 5-state property, the analysis can be done in 500,000 steps which is very feasible on modern processors.

However, sound analyzers are necessarily incomplete, and consequently report false errors. A promising technique to reduce false errors is counterexample driven refinement [23, 9, 1]. Here, abstract counterexamples are simulated in the original program to check if they are true errors. If they are not true errors, then the analysis automatically adds more state to track in the abstraction. Counterexample driven refinement has been used to building tools that have a very low false error rate [1, 21, 7].

Expressive type systems have also been used to state and check properties [12, 16]. Since types are integrated into the programming language, the approach has several advantages. Recent approaches allow enhanced programmability of properties using types [8]. While type based approaches are very natural for specifying protocols on one object at a time, they have difficulties specifying protocols that involve multiple objects.

Abstract interpretation [10] is a generic theory for building sound static analysers. Tools based on abstract interpretation have been tuned using domain knowledge to produce very few false errors in large safety critical software [5].

5 Future

The future for property checking tools remains very bright. We predict that they will be distributed and used widely. For instance, the PREFast tool has just been released as part of Microsoft's Visual Studio, and the Static Driver Verifier tool will be released as part of Windows Vista's driver development kit. We also predict that several more light-weight property checkers will be built, since there are a variety of programming languages and environments. For example, in the domain of scripting languages and web programming, such tools are beginning to be built.

Modern integrated development environments provide incremental compilation that presents syntax errors to the programmer as the program is being typed. There have been recent attempts to integrate property checking tools at this level to present semantic errors in the same fashion [3]. We predict that this trend will continue, and that property checking tools will be routinely available in common programming environments.

Property checking can also be done at run-time using the same specifications used for static analysis, by compiling the specifications into run-time monitors. Once this is done, testing techniques can be used to generated test inputs that lead to the error state in the run-time monitor. We predict that static verification tools for property checking will be combined with such testing tools, since the two approaches nicely complement each other. Testing tries to find inputs and executions that demonstrate violations to the property. Static verifiers try to find proofs that all executions of the program satisfy the property. Testing works when errors are easy to find, and verification works when proofs are easy to find. Recent work has started combining static verification and testing in various interesting ways. Light-weight symbolic execution has been used to do directed test case generation and improve path coverage [17]. Similarly, methods that improve efficiency of static analysis using testing are being explored [28]. Recently, we have proposed a new algorithm that does counterexample driven refinement using both static analysis and testing techniques [18].

Combination of static analysis with testing will necessarily force abstract domains used in static analysis to model concrete executions more closely. Current tools for doing counterexample driven refinement use predicate abstraction. While this works well for control dominated properties of programs, checking properties about heap data structures remains challenging. There have been attempts to make analyses of heap data structures property driven [25]. Recently, we have proposed a refinement algorithm that works with any abstract interpretation [19].

Most static analysis tools analyze sequential programs, or even if they analyze concurrent programs, consider only one thread of execution at a time. There have been recent efforts in using static analysis to build practical tools for detecting concurrency errors [27, 26], and we believe that this trend will continue.

References

[1] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN 01: SPIN Workshop*, LNCS 2057. Springer-Verlag, 2001.

[2] T. Ball and S. K. Rajamani. Slic: A specification language for interface checking (of C). Technical Report MSR-TR-2001-21, Microsoft Research, 2001.

[3] M. Barnett, B. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO 05: Formal Methods for Components and Objects*, 2005.

[4] M. Barnett, R. DeLine, M. Fahndrich, K. R. M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004.

[5] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Mine, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *PLDI 03: Programming Language Design and Implementation*, pages 196–207, 2003.

[6] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *Software-Practice and Experience*, 30(7):775–802, June 2000.

[7] S. Chaki, E. M. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. *IEEE Transactions on Software Engineering*, 30(6):388–402, 2004.

[8] B. Chin, S. Markstrum, and T. Millstein. Semantic type qualifiers. In *PLDI 05: Programming Language Design and Implementation*, pages 85–95. ACM, 2005.

[9] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV 00: Computer-Aided Verification*, LNCS 1855, pages 154–169. Springer-Verlag, 2000.

[10] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for the static analysis of programs by construction or approximation of fixpoints. In *POPL 77: Principles of Programming Languages*, pages 238–252. ACM, 1977.

[11] M. Das, S. Lerner, and M. Seigle. ESP: Path-sensitive program verification in polynomial time. In *PLDI: Programming Language Design and Implementation*, pages 57–69. ACM, 2002.

[12] R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *PLDI 01: Programming Language Design and Implementation*. ACM, 2001.

[13] D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Extended static checking. Technical Report Research Report 159, Compaq Systems Research Center, December 1998.

[14] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *OSDI 00: Operating System Design and Implementation*. Usenix Association, 2000.

[15] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *PLDI 02: Programming Language Design and Implementation*, pages 234–245, 2002.

[16] J. S. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *PLDI 02: Programming Language Design and Implementation*, pages 1–12. ACM, 2002.

[17] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *PLDI 05: Programming Language Design and Implementation*, pages 213–223, 2005.

[18] B. Gulavani, T. A. Henzinger, Y. Kannan, A. Nori, and S. K. Rajamani. Synergy: A new algorithm for property checking. In *FSE 06: Foundations of Software Engineering (to appear)*, 2006.

[19] B. S. Gulavani and S. K. Rajamani. Counterexample driven refinement for abstract interpretation. In *TACAS 06: Tools and Algorithms for Construction and Analysis of Systems*, LNCS 3920. Springer-Verlag, 2006.

[20] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific static analyses. In *PLDI 02: Programming Language Design and Implementation*, pages 69–82, 2002.

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

[22] S. C. Johnson. Lint: A C program checker. Technical Report 65 (Unix Programmers Manual), AT&T Bell Laboratories, 1978.

[23] R.P. Kurshan. *Computer-aided Verification of Coordinating Processes*. Princeton University Press, 1994.

[24] J. R. Larus, T. Ball, M. Das, R. DeLine, M. Fahndrich, J. Pincus, S. K. Rajamani, , and R. Venkatapathy. Righting software. *IEEE Software*, 21(3):92–100, 2004.

[25] A. Loginov, T. W. Reps, and S. Sagiv. Abstraction refinement via inductive learning. In *CAV 05: Computer-Aided Verification*, pages 519–533, 2005.

[26] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. In *PLDI 06: Programming Language Design and Implementation*, 2006.

[27] S. Qadeer and D. Wu. KISS: Keep it simple and seqeuential. In *PLDI 04: Programming Language Design and Implementation*, pages 14–24. ACM, 2004.

[28] G. Yorsh, T. Ball, and M. Sagiv. Testing, abstraction, theorem proving: better together. In *ISSTA 06: Software Testing and Analysis*. ACM, 2006.