Symbiosis of Static Analysis and Program Testing

Michal Young

University of Oregon, Dept. of Computer Science
michal@cs.uoregon.edu

1 Introduction

The fundamental fact about verifying properties of software, by any means, is that almost anything worth knowing is undecidable in principle. The limitations of software testing, on the one hand, and static analysis on the other, are just different manifestations of this one basic fact. Because both approaches to verification are ultimately doomed, neither is likely to supplant the other in the foreseeable future. On the other hand, each can complement the other, and some of the most promising avenues of research are in combinations and hybrid techniques.

2 Limits of Dynamic Testing

The basic theory underlying testing is almost entirely negative. One can of course resort to randomized testing if the objective is to measure rather than to improve the product, but the number of test cases required to obtain high levels of confidence is astronomical [5]. Moreover, statistical inferences are valid only if one has a valid statistical model of use, which is rare. More often testing is systematic, not random, and aimed at finding faults rather than estimating the prevalence of failures.

Systematic testing exercises divides the set of possible executions into a finite number of classes, and inspects samples from each class on the hope that classes are sufficiently homogeneous to raise the likelihood of fault detection. Thus systematic testing, whether based on program structure or specification structure or something else, is based on a *model* of software and a *hypothesis* that faults are somehow localized by the model. This hypothesis is not verifiable except individually and after the fact, by observing whether some test obligation induced from the model was in fact effective in selecting fault-revealing test cases.

3 Limits of Static Analysis

The unsatisfying foundations of testing make exhaustive, static analyses seem more attractive. Isn't it better to achieve some kind of result that is sound, even if we must accept some spurious error warnings, or restrict the class of programs

M. Pezzè (Ed.): FASE 2003, LNCS 2621, pp. 1–5, 2003.

to be analyzed, or check properties that are simpler than those we really want to analyze? Indeed, some properties can be incorporated into the syntax or static semantics of a programming language and checked every time we compile a program, like Java's requirement to initialize each variable before use and to explicitly declare the set of exceptions that can be thrown or propagated by a program unit.

Exhaustive static analyses are necessarily based on abstract models of the software to be checked. The static checks that have been built into Java are based on simple models derived from program syntax. The "initialize before use" rule is based on a simple, local control flow model, and crucially it sidesteps the fundamental limitation of undecidability by not distinguishing between executable and unexecutable program paths. It is acceptable for a Java compiler to reject a program with a syntactic path on which the first use of a variable appears before it is assigned a variable, even if that path can never be taken, because the restriction is easy to understand and the "fault" is easy to repair. The rule regarding declaration of exceptions is likewise based on a simple, easily understandable call-graph model. Because the rules can be understood in relation to these models, we accept them as being rather strict but precise, rather than viewing them as producing spurious error reports.

More sophisticated program checks – for example, checking synchronization structure to verify absence of race conditions – place much heavier demands on extraction of an appropriate model. If the model is too simple, the analysis is apt to be much too pessimistic, and unlike the Java rules mentioned above there may be no reasonable design rule to prevent spurious error reports. If the model is sufficiently expressive to avoid spurious error reports, an exhaustive analysis is likely to be unacceptably expensive. The remaining option is to use an expressive model, but limit analysis effort by other means, such as using a non-exhaustive (and unsound) analysis, in which case the "static" analysis becomes a kind of symbolic testing.

Sometimes the model is produced by a human, and in that case again the limits on analysis are not too onerous if failures are reported in the form of counter-examples whose cause is easily diagnosed. Crafting a model with appropriate abstractions to efficiently verify properties of interest is a challenging design task, but arguably at least the effort is recouped in establishing a clearer understanding of the software system [16]. A remaining problem is that one cannot be certain of the correspondence between the model and the actual software system it represents. Verifying their correspondence has, roughly speaking, the same difficulty as extracting models directly from software, and raises essentially the same issues.

Despite the surge of interest in static analysis of models derived directly (with varying degrees of automation) from actual software source code [8,14,11,2], the fundamental limitation imposed by undecidability ensures that static analysis will not supplant dynamic testing anytime soon. In some application domains, it may be possible to impose enough restrictions on programs that testing is relegated to a minor role. One can imagine severe restrictions on programming

style in software destined for medical devices, for example. In the larger world of software development, the trends are in the wrong direction. Dynamically reconfigurable systems, programs that migrate across a network, end-user programmable applications, and aspect-oriented programming (to pick a few) all widen the gap between the kinds of programs that developers write, and the kinds of programs that are amenable to strong static analysis.

4 Symbiotic Interactions

Testing has limitations, and static analysis has limitations, but it does not follow immediately that some combination or hybrid of testing and analysis should be better than either one independently. One could imagine that, while neither is perfect, one dominates the other. But this does not turn out to be the case, and there are many current examples of symbiotic interactions between static analysis and testing as well as additional opportunities for fruitful combinations.

One interesting class of combination is an individual technique that combines aspects of both, exploring selected scenarios like testing but using a symbolic representation of program state more like a pure static analysis technique. Symbolic testing techniques have the same ultimate limitation as conventional testing, but using symbolic representations they can more effectively search for particular classes of program fault. Howden developed a symbolic testing technique more than 25 years ago [13], but program analysis technology and computing power was perhaps not ready for it then. Lately the technique has been revived and elaborated, notably in Pincus' Prefix [4] and Engler's Metal [7].

Symbolic testing, like conventional testing, is not sound: If we fail to find a violation of some property of interest, that does not constitute a proof of the property. Exhaustive static analyses, on the other hand, are typically designed to be sound, at least with respect to the abstract model on which they operate. As we noted above, this leaves the problem of discrepancies between the model and the underlying program. Relegating the problem of model conformance to dynamic testing, as in communication protocol conformance testing, is an attractive option. Separating analysis of the model from dynamic testing of model conformance makes each simpler. It makes diagnosing problems in the model much easier, and provides much more flexibility (e.g., in the use of pointers and other dynamic structures) than insisting that the programmer use only idioms that can be automatically (and soundly) abstracted to a model.

Dynamic testing often divides the execution space of a program into putative equivalence classes based on the results of some form of static analysis of program text, e.g., data flow testing based on def-use associations produced by a data flow analysis. To date, most such testing techniques have been based on conservative analyses. For example, points-to analysis typically overestimates the set of pointers that can point to the same object, resulting in spurious def-use associations; this in turn can lead to test obligations that cannot be met. While conservative analysis can be justified when used directly to find faults, it

makes more sense to use a precise but unsafe analysis to provide guidance for testing [12].

One could take this a step further: It is easy to imagine static program analyses that are unsound, but which produce as a by-product a set of unverified assumptions to be checked by dynamic testing. For example, one might choose to perform a static analysis that ignores exceptions or some aliasing relations, except for those that have been observed in dynamic tests. Or, one could produce static analysis results in which "maybe" results are distinguished from definite faults, as Chechik and Ding have done with model checking [6], and use "maybe" results as guidance to testing.

Even a simple, intraprocedural control flow graph mode model is overly conservative in that it includes program paths that cannot actually be executed, so systematic testing typically ends with a residue of unmet coverage obligations. This residue itself can be considered as a kind of model or hypothesis about behavior in actual use. If we release a product with some unmet coverage obligations, we are hypothesizing that they are either impossible to execute (an artifact of an overly conservative model) or at least so rarely executed as to be insignificant. This hypothesis can be tested in actual use, through program instrumentation [15,3].

Increased computational power and clever algorithms benefit dynamic testing and program monitoring, just as they have benefited static analysis and particularly finite-state verification techniques. One of the techniques that would have seemed implausible a decade ago is dynamically gathering "specifications" (more precisely, hypothetical properties) during program execution, as in Ernst's Daikon [9] and the "specification mining" technique of Ammons, Bodik, and Larus [1]. These can be applied directly to testing [10]. An interesting possibility is to make these highly likely but unproven properties available to static analysis techniques, again giving up soundness (when necessary) to achieve more precise analysis. It is also possible that, once identified, some of them might be statically verified.

Many years ago, a battle raged between the partisans of program verification and of dynamic testing. Thankfully, that battle is long over, and static and dynamic analysis techniques (as well as other aspects of formal methods, such as precise specification) are almost universally regarded as complementary. Models are the common currrency of static analysis and dynamic testing. By accepting the inevitability of imperfect models, we open many opportunites for synergistic combinations.

References

1. Glenn Ammons, Rastislav Bodik, and James R. Larus. Mining specifications. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 4–16. ACM Press, 2002.
2. T. Ball and S. Rajamani. Checking temporal properties of software with boolean programs. In *Proceedings of the Workshop on Advances in Verification*, 2000.

3. Jim Bowring, Alessandro Orso, and Mary Jean Harrold. Monitoring deployed software using software tomography. In *Proceedings of the 2002 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 2–9. ACM Press, 2002.
4. William R. Bush, Jonathan D. Pincus, and David J. Sielaff. A static analyzer for finding dynamic programming errors. *Software – Practice and Experience*, 30(7):775–802, 2000.
5. Ricky W. Butler and George B. Finelli. The infeasibility of quantifying the reliability of life-critical real-time software. *IEEE Transactions on Software Engineering*, 19(1):3–12, 1993.
6. M. Chechik and W. Ding. Lightweight reasoning about program correctness. Technical Report CSRG Technical Report 396, University of Toronto, 2000.
7. Benjamin Chelf, Dawson Engler, and Seth Hallem. How to write system-specific, static checkers in Metal. In *Proceedings of the 2002 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 51–60. ACM Press, 2002.
8. James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Păsăreanu, Robby, and Hongjun Zhenga. Bandera: Extracting finite-state models from Java source code. In *International Conference on Software Engineering*, pages 439–448, 2000.
9. Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. In *International Conference on Software Engineering*, pages 213–224, 1999.
10. Michael Harder. Improving test suites via generated specifications. Master's thesis, M.I.T., Dept. of EECS, May 2002.
11. Gerard J. Holzmann and Margaret H. Smith. An automated verification method for distributed systems software based on model extraction. *IEEE Transactions on Software Engineering*, 28(4):364–377, 2002.
12. J. Horgan and S. London. Data flow coverage and the C language. In *Proceedings of the Fourth Symposium on Testing, Analysis, and Verification*, pages 87–97, Victoria, Oct 1991. ACM Press.
13. William E. Howden. Symbolic testing and the DISSECT symbolic evaluation system. *IEEE Transactions on Software Engineering*, SE–3(4):266–278, July 1977.
14. D. Park, U. Stern, and D. Dill. Java model checking. In *Proceedings of the First International Workshop on Automated Program Analysis, Testing and Verification*, Limerick, Ireland, June 2000.
15. Christina Pavlopoulou and Michal Young. Residual test coverage monitoring. In *International Conference on Software Engineering*, pages 277–284, 1999.
16. Wei Jen Yeh and Michal Young. Redesigning tasking structures of Ada programs for analysis: A case study. *Software Testing, Verification, and Reliability*, 4:223–253, December 1994.