Static and dynamic analysis: synergy and duality

Michael D. Ernst
MIT Lab for Computer Science
Cambridge, MA 02139 USA
mernst@lcs.mit.edu

Abstract

This paper presents two sets of observations relating static and dynamic analysis. The first concerns synergies between static and dynamic analysis. Wherever one is utilized, the other may also be applied, often in a complementary way, and existing analyses should inspire different approaches to the same problem. Furthermore, existing static and dynamic analyses often have very similar structure and technical approaches. The second observation is that some static and dynamic approaches are similar in that each considers, and generalizes from, a subset of all possible executions.

Researchers need to develop new analyses that complement existing ones. More importantly, researchers need to erase the boundaries between static and dynamic analysis and create unified analyses that can operate in either mode, or in a mode that blends the strengths of both approaches.

1. Background

This section briefly reviews some facts about traditional static and dynamic analyses, to set the stage for the rest of the paper.

Static and dynamic analyses arose from different communities and evolved along parallel but separate tracks. Traditionally, they have been viewed as separate domains, with practitioners or researchers specializing in one or the other. Furthermore, each has been considered ill-suited for the tasks at which the other excels. This paper argues that the difference is smaller than it appears and that certain of these distinctions are unnecessary and counterproductive.

Static analysis examines program code and reasons over all possible behaviors that might arise at run time. Compiler optimizations are standard static analyses. Typically, static analysis is conservative and sound. Soundness guarantees that analysis results are an accurate description of the program's behavior, no matter on what inputs or in what environment the program is run. Conservatism means reporting weaker properties than may actually be true; the weak properties are guaranteed to be true, preserving soundness, but may not be strong enough to be useful. For instance, given a function f, the statement "f returns a non-negative value" is weaker (but easier to establish) than the statement "f returns the absolute value of its argument." A conservative analysis might report the former, or the even weaker property that f returns a number.

Static analysis operates by building a model of the state of the program, then determining how the program reacts to this state. Because there are many possible executions, the analysis must keep track of multiple different possible states. It is usually not reasonable to consider every possible run-time state of the program; for example, there may be arbitrarily many different user inputs or states of the run-time heap. Therefore, static analyses usually use an abstracted model of program state that loses some information, but which is more compact and easier to manipulate than a higher-fidelity model would be. In order to maintain soundness, the analysis must produce a result that would be true no matter the value of the abstracted-away state components. As a result, the analysis output may be less precise (more approximate, more conservative) than the best results that are in the grammar of the analysis.

Dynamic analysis operates by executing a program and observing the executions. Testing and profiling are standard dynamic analyses. Dynamic analysis is precise because no approximation or abstraction need be done: the analysis can examine the actual, exact run-time behavior of the program. There is little or no uncertainty in what control flow paths were taken, what values were computed, how much memory was consumed, how long the program took to execute, or other quantities of interest. Dynamic analysis can be as fast as program execution. Some static analyses run quite fast, but in general, obtaining accurate results entails a great deal of computation and long waits, especially when analyzing large programs. Furthermore, certain problems, such as pointer or alias analysis, remain beyond the state of the art; even exponential-time algorithms do not always produce sufficiently precise results. By contrast, determining at run time whether two pointers are aliased requires a single machine cycle to compare the two pointers (somewhat more, if relations among more than two pointers are checked).

The disadvantage of dynamic analysis is that its results may not generalize to future executions. There is no guarantee that the test suite over which the program was run (that is, the set of inputs for which execution of the program was observed) is characteristic of all possible program executions. Applications that require correct inputs (such as semantics-preserving code transformations) are unable to use the results of a typical dynamic analysis, just as applications that require precise inputs are unable to use the results of a typical static analysis. Whereas the chief challenge of building a static analysis is choosing a good abstraction function, the chief challenge of performing a good dynamic analysis is selecting a representative set of test cases (inputs to the program being analyzed). (Efficiency concerns affect both types of analysis.) A well-selected test suite can reveal properties of the program or of its execution context; failing that, a dynamic analysis indicates properties of the test suite itself, but it can be difficult to know whether a particular property is a test suite artifact or a true program property.

Unsound dynamic analysis has been traditionally denigrated by the programming languages community. Semantics-preserving program transformations such as compiler optimizations require correct information about program semantics. However, unsoundness is useful in many other circumstances. Dynamic analysis can be used even in situations where program semantics (but not perfect program semantics) are required. More importantly, humans are remarkably resilient to partially incorrect information [10], and are not hindered by its presence among (a sufficient quantity of) valuable information. Since in most domains human time is far more important than CPU time, it is a better focus for researchers. As a result, and because of its significant successes, dynamic analysis is gaining credibility.

2. Static and dynamic analysis: synergies

As noted in Section 1, static and dynamic analysis have complementary strengths and weaknesses. Static analysis is conservative and sound: the results may be weaker than desirable, but they are guaranteed to generalize to future executions. Dynamic analysis is efficient and precise: it does not require costly analyses, though it does require selection of test suites, and it gives highly detailed results regarding those test suites.

The two approaches can be applied to a single problem, producing results that are useful in different contexts. For instance, both are used for program verification. Static analysis is typically used for proofs of correctness, type safety, or other properties. Dynamic analysis demonstrates the presence (not the absence) of errors and increases confidence in a system.

This section considers the use of static and dynamic analysis in tandem, to complement and support one another. First, static and dynamic analyses enhance each other via pre- or post-processing. Second, existing static and dynamic analyses can suggest new analyses. Third, static and dynamic analyses should be combined into a hybrid analysis.

2.1. Performing both static and dynamic analysis

Static or dynamic analyses can enhance one another by providing information that would otherwise be unavailable. Performing first one analysis, then the other (and perhaps iterating) is more powerful than performing either one in isolation. Alternately, different analyses can collect different varieties of information for which they are best suited.

This well-known synergy has been and continues to be exploited by researchers and practitioners alike. As one simple example, profile-directed compilation [1] uses hints about frequently executed procedures or code paths, or commonly observed values or types, to transform code. The transformation is meaning-preserving, and it improves performance under the observed conditions but may degrade it in dissimilar conditions (the correct results will still be computed, only consuming more time, memory, or power). As another example, static analysis can obviate the collection of certain information by guaranteeing that collecting a smaller amount of information is adequate; this makes dynamic analysis more efficient or accurate.

2.2. Inspiring analogous analyses

Both static and dynamic analysis can always be applied to a particular program, though possibly at different cost, and their results have different properties. Whenever only one of the analyses exists, it makes sense to investigate the other, which may be able to use the same technical approach. In many cases, both approaches have already been implemented by different parties.

One simple example is static and dynamic slicing [14]. Slicing indicates which parts of a program (may) have contributed to the value computed at, or the execution of, a particular program expression or statement. Slicing can operate statically, dynamically, or both.

As a more substantive example, Purify [8] and LCLint [6] are tools for detecting memory leaks and uses of dead storage. (Each has capabilities missing from the other, but this discussion considers only the intersection of their capabilities.) Purify performs a run-time check, essentially by use of tagged memory. Each byte of memory used by the program is allocated a 2-bit state code indicating whether that memory is unallocated, uninitialized, or initialized; at each memory access, the memory's state is checked and/or updated by instructions that Purify inserts in the executable.

LCLint operates statically, checking user-supplied annotations that indicate assumptions. It performs a dataflow analysis whose abstract state contains includes definedness and allocation state; each program operation has particular requirements on its inputs and produces certain results. The rules and abstract states used by Purify and LCLint are essentially identical: they perform the same analysis, Purify dynamically and LCLint statically.

As another example, consider program specifications, which are formal mathematical abstractions of program behavior. When used to verify behavior, the standard static technique is theorem proving, which typically requires human interaction. The dynamic analog of theorem-proving is the `assert` statement, which verifies the truth of a particular formula at run time. Specifications are best written by the designer before implementation commences. When specifications are synthesized after the fact, the typical approach is a static one that proceeds by examining the program text. This task is sometimes done automatically with the assistance of heuristics, but very frequently it is done by hand. The dynamic analog to writing down a specification is generating one automatically by dynamic detection of likely invariants [4, 5]. The invariant detection technique postulates potential invariants, tests them over program executions, and then prunes them via static analysis, statistical tests, heuristics, and other techniques. As a result, its output is often close to the ideal (over its grammar) that a perfect static analysis or human would produce [11].

Dynamic invariant detection was invented as a direct result of considering the duality between dynamic and static analysis. There existed static analyses that could generate specifications (or formulas syntactically identical to specifications, if the term "specification" is reserved for human-produced formulas), but no dynamic analyses existed. (Dynamic techniques for other varieties of specifications already existed [2].) This led to a new technique that has since been applied to refactoring, bug detection, fault isolation, test suite improvement, verification, theorem-proving, detection of component incompatibilities, and other tasks. Other researchers would be well advised to look for other missing analyses, in order to inspire development of new analyses by comparison with their existing analogs. Where just one (static or dynamic) analysis exists, the other is likely to be advantageous.

2.3. Hybrid static-dynamic analysis

Presently, tool users must select between static and dynamic analysis. (Section 2.1 noted cooperative strategies that use one analysis as a prepass for the other, but the overall output is that of the final analysis.) In some cases, one or the other analysis is perfectly appropriate. However, in other cases, users may prefer not to be forced to choose between the two approaches.

A better alternative is to create new, hybrid analyses that combine static and dynamic analyses. Such an analysis would sacrifice a small amount of the soundness of static analysis and a small amount of the accuracy of dynamic analysis to obtain new techniques whose properties are better-suited for particular uses than either purely static or purely dynamic analyses.

The hybrid analyses would replace the (large) gap between static and dynamic analysis with a continuum. Users would select a particular analysis fitted to their needs: they would, in a principled way, turn the knob between soundness and precision. It seems unlikely that one extreme or the other is always the appropriate choice: users or system builders should be able to find the "sweet spot" for their application. Indeed, different analyses (both static and dynamic) already use different amounts of processing power to produce results of differing precision. This could be a starting point for the work. Another starting point could be use of only a subset of all available static information, much as already practiced by some tools [12, 7]. A third starting point is an observation about the duality of static and dynamic analysis, noted immediately below in Section 3. One potential barrier is different treatments (optimistic vs. conservative) of unseen executions.

3. Static and dynamic analysis: duals

Static and dynamic analysis are typically seen as distinct and competing approaches with fundamentally different the techniques and technical machinery. (Section 2.2 noted that in some cases, the underlying analyses are quite similar.) This section argues that the two types of analysis are not as different as they may appear; rather, they are duals that make many of the same tradeoffs.

The key observation is that both static and dynamic analysis are able to directly consider only a subset of program executions. Generalization from those executions is the source of unsoundness in dynamic analysis and imprecision in static analysis.

A dynamic analysis need not be unsound. A sound dynamic analysis observes every possible execution of a program. If a test suite contains every possible input (and every possible environmental interaction), then the results are guaranteed to hold regardless of how the program is used. This simple goal is unattainable: nontrivial programs usually have infinitely many possible executions, and only a relatively small (even if absolutely large) set of them can be considered before exhausting the testing budget (in time, money, or patience). Researchers have devised a number of techniques for using partial test suites or for selection of partial test suites [13]. These techniques are of interest solely as efficiency tweaks to an algorithm that works perfectly in theory but exhausts resources in practice.

A static analysis need not be approximate. A perfectly precise static analysis considers every possible execution of a program, maintaining, for each execution, the program's full state (or, rather, all possible states). This is not typically feasible, because there are infinitely many possible executions and the state of the program is extremely large. Researchers have devised many abstractions, primarily of state but also of executions, that permit them to consider a smaller state space or a smaller number of executions, reducing the problem to one that can often be solved on today's computers. The abstractions are of interest solely as efficiency tweaks to an algorithm that works perfectly in theory [3] but exhausts resources in practice.

Both dynamic and static analyses consider only a subset of all possible executions, but that subset is chosen differently. (Executions not in the set may be dealt with differently, as well. In particular, the unobserved executions may be treated conservatively and pessimistically or may be treated optimistically, which often means simply ignoring them. This distinction between sound and unsound analysis is important but is omitted for reasons of space and because it is orthogonal to the main point.)

The set of executions considered by a dynamic analysis is exactly those that appear in the test suite or that were observed during execution. This set is very easy to enumerate and may characterize a particular environment well; however, the set may be difficult to formalize in mathematical notation. The set of executions considered by a static analysis is those that induce executions of a certain variety. For instance, the k-limiting [9] abstraction considers in detail only the executions that create data structures with pointer-directed paths of length no more than k; another popular abstraction considers only executions that traverse each loop either zero or one times [6, 7].

Each of the descriptions is simpler in some respects and more complicated in others. Given a data-structure-centric description like those used for static analysis, it is difficult to know what executions induce the data structures or whether particular programs or execution environments will suffer degradation of analysis results. Given a set of inputs or executions, analysis is required to understand what parts of a program are exercised, and in what ways.

Recognition of this duality — both analyses consider a subset of executions — should make it easier to translate approaches from one domain to the other and to combine static and dynamic analyses, or at least lead to a better understanding of the gap between them.

4. Conclusion

This paper has listed some widely-recognized distinctions between static and dynamic analysis, notably soundness versus precision. It noted ways that static and dynamic analysis can interact: by augmenting one another, by inspiring new analyses, and by creating hybrid analyses that combine them. Some of these seem to have been overlooked by previous authors. Finally, it noted a duality between static and dynamic analysis, both of which consider (differently-specified) subsets of program executions. We encourage other researchers to join us in bringing these research ideas to fruition.

References

[1] B. Calder, P. Feller, and A. Eustace. Value profiling. In *MICRO-97*, pages 259–269, Dec. 1–3, 1997.

[2] J. E. Cook and A. L. Wolf. Event-based detection of concurrency. In *FSE*, pages 35–45, Nov. 1998.

[3] P. M. Cousot and R. Cousot. Automatic synthesis of optimal invariant assertions: Mathematical foundations. In *ACM Symposium on Artificial Intelligence and Programming Languages*, pages 1–12, Aug. 1977.

[4] M. D. Ernst. *Dynamically Discovering Likely Program Invariants*. PhD thesis, U. Wash. Dept. of Comp. Sci. & Eng., Seattle, Washington, Aug. 2000.

[5] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE TSE*, 27(2):1–25, Feb. 2001.

[6] D. Evans. Static detection of dynamic memory errors. In *PLDI*, pages 44–53, May 21–24, 1996.

[7] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *PLDI*, pages 234–245, 2002.

[8] R. Hastings and B. Joyce. Purify: A tool for detecting memory leaks and access errors in C and C++ programs. In *Winter 1992 USENIX Conference*, pages 125–138, Jan. 1992.

[9] N. D. Jones and S. S. Muchnick. Flow analysis and optimization of Lisp-like structures. In *Program Flow Analysis: Theory and Applications*, chapter 4, pages 102–131. Prentice-Hall, Englewood Cliffs, N.J., 1981.

[10] G. C. Murphy, D. Notkin, and E. S.-C. Lan. An empirical study of static call graph extractors. In *ICSE*, pages 90–99, Mar. 1996.

[11] J. W. Nimmer and M. D. Ernst. Automatic generation of program specifications. In *ISSTA*, pages 232–242, July 2002.

[12] PREfix/Enterprise. `www.intrinsa.com`, 1999.

[13] G. Rothermel and M. J. Harrold. Empirical studies of a safe regression test selection technique. *IEEE TSE*, 24(6):401–419, June 1998.

[14] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, 1995.