Program Analysis: A Hierarchy

Andreas Zeller
Lehrstuhl für Softwaretechnik
Universität des Saarlandes, Saarbrücken, Germany
zeller@acm.org

Abstract

Program analysis tools are based on four reasoning techniques: (1) deduction from code to concrete runs, (2) observation of concrete runs, (3) induction from observations into abstractions, and (4) experimentation to find causes for specific effects. These techniques form a hierarchy, where each technique can make use of lower levels, and where each technique induces capabilities and limits of the associated tools.

1. Introduction

Reasoning about programs is a core activity of any programmer. To answer questions like “what can happen?”, “what should happen?”, “what did happen?”, and “why did it happen?”, programmers use four well-known reasoning techniques:

Deduction from an abstraction into the concrete—for instance, analyzing program code to deduce what can or cannot happen in concrete runs.

Observation of concrete events—e.g. tracing, monitoring or profiling a program run or using a debugger.

Induction for summarizing multiple observations into an abstraction—an invariant, for example, or some visualization.

Experimentation for isolating causes of given effects—e.g. narrowing down failure-inducing circumstances by systematic tests.

These reasoning techniques form a hierarchy (Figure 1), in which each “outer” technique can make use of “inner” techniques. For instance, experimentation uses induction, which again requires observation; on the other hand, deduction cannot make use of any later technique.

The interesting thing about this hierarchy is that the very same reasoning techniques are also the foundations of automated *program analysis* tools. In fact, each of the reasoning techniques induces a specific class of tools, its capabilities and its limits. This is the aim of this paper: to provide a rough classification of the numerous approaches in program analysis—especially in dynamic analysis—, to show their common benefits and limits, and to show up new research directions to overcome these limits.

2. Deduction

Deduction is reasoning from the general to the particular; it lies at the core of all reasoning techniques. In program analysis, deduction is used for reasoning from the program code (or other abstractions) to concrete runs—especially for deducing what can or cannot happen. These deductions take the form of mathematical proofs: If the abstraction is true, so are the deduced properties.

Since deduction does not require any knowledge about the concrete, it is not required that the program in question is actually executed—the program analysis is *static*. Static

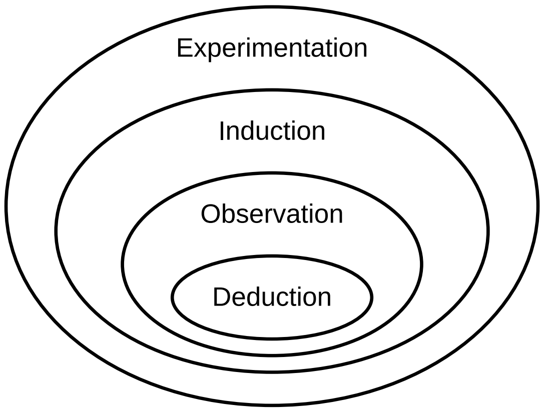

Figure 1. A hierarchy of reasoning techniques

program analysis was originally introduced in compiler optimization, where it deduces properties like

- Can this variable influence that other variable? (if not, one can parallelize their computation)
- Can this variable be used before it is assigned? (if not, there is probably an error)
- Is this code ever executed? (if not, it can be ignored)

Deduction techniques are helpful in program understanding, too—especially for computing *dependencies* between variables. A variable v' at a statement s' is dependent on a variable v at a statement s if altering v at s can alter the value of v' at s'; in other words, the value of v at s is a *potential cause* for v' at s'. By tracing back the dependencies of some variable v, one obtains a *slice* of the program—the set of all statements that could have influenced v [13, 14].

As an ongoing example, consider the following piece of C code. If `p` holds, `a` is assigned a value, which is then printed into the string `buf`.

```
3  char *format = "a = %d";
4  if (p)
5      a = compute_value();
6  sprintf(buf, format, a);
```

Let us assume that after executing this piece of code, we find that `buf` contains `"a = 0"`. However, `a` is not supposed to be zero. What's the cause of `"a = 0"` in `buf`?

By deduction, we find that the string `buf` is set by the `sprintf` function which takes `a` as an argument; hence, `buf` depends on `a` at line 5. Likewise, `a` depends on `p` at line 4 (since altering `p` may alter `a`) and on the result of `compute_value()`. To find out why `a` is zero, we must trace back these dependencies in the slice. More important than the slice itself are the statements *not* included in the slice—e.g. a statement like `c = d + e;` The analysis proves that these cannot influence `a` or `buf` in any way; hence, they can be ignored for all further analysis.

Unfortunately, proving that executing some statement cannot influence a variable is difficult. Parallel or distributed execution, dynamic loading or reconfiguration of program code, unconstrained pointer arithmetic, or use of multiple programming languages are obstacles that are hard to handle in practice.

The biggest obstacle for deduction, though, is *obscure code:* If we cannot analyze some executed code, anything can happen. The `sprintf` function above, is typically part of the C runtime library and not necessarily available as source code. Only if we assume that `sprintf` works as expected can we ensure that `buf` depends on `a`.

3. Observation

Observation allows the programmer to inspect arbitrary aspects of an individual program run. Since an actual run is required, the associated techniques are called *dynamic.* Observation brings in actual *facts* of a program execution; unless the observation process is flawed, these facts cannot be denied.

For observing program runs, programmers and researchers have created a big number of tools, typically called "debuggers" because they are mainly used for debugging programs. A debugger allows to inspect states at arbitrary events of the execution; advanced tools allow a database-like querying of states and events [3, 12].

The programmer uses these tools to *compare* actual facts with expected facts—as deduced from an abstract description such as the program code. This comparison with expected facts can also be conducted automatically within the program run, using special *assertion* code that checks runtime invariants. Specific invariant checkers have been designed to detect illegal memory usage or array bound violations.

By combining slicing with observation, one obtains *dynamic slicing:* a slice that is valid for a specific execution only, and hence more precise than a slice that applies for all executions [1, 6, 11]. In principle, a dynamic slicing tool does not require source code as long as it can intercept all read/write accesses to program state and thus trace actual dependencies.

As an example of dynamic slicing, assume that after the execution of the code above, we find that `buf` contains `"a = 0"` and that `p` is true. Consequently, a dynamic slice tool can deduce from the code that the value of `a` can only stem from `compute_value()`; an earlier value of `a` cannot have any effect on `buf` (that is, unless `a` is being read in `compute_value()`).

Let's now introduce a little complexity: By observation, we also find that `compute_value()` returns a non-zero value. Yet, `buf` contains `"a = 0"`. How can this be?

4. Induction

Induction is reasoning from the particular to the general. In program analysis, induction is used to *summarize* multiple program runs—e.g. a test suite or random testing—to some abstraction that holds for all considered program runs. In this context, a "program" may also be a piece of code that is invoked multiple times from within a program—that is, some function or loop body.

The most widespread program analysis tools that rely on induction are *coverage tools* that summarize the statement and branch coverage of multiple runs; such results can be easily visualized [10]. Most programming environments support coverage tracing and summarizing. In program visualization, call traces and data accesses are frequently summarized [2].

On a higher abstraction level, *invariant detection* filters a set of possible abstractions against facts found in multiple runs. The remaining abstractions hold as invariants for all examined runs [4, 7]. This approach relies only on observation of the program state at specific events; hence, it is not limited by obscure code or other properties that make static analysis hard.

Both techniques can be used to detect *anomalies:* One trains the tool on a set of correct test runs to infer common properties. Failing runs can then be checked whether they violate these properties; these violations are likely to cause the failures.

As an example, let us assume that we execute the above C code under several random inputs, flagging an error each time `buf` contains `"a = 0"`. An invariant detector can then determine that, say, `a < 2054567 || a % 2 == 1` holds at line 6 for all runs where the error occurs. This is the common abstraction for all abnormal runs: `buf` contains `"a = 0"` whenever `a` is odd or smaller than 2,054,567. Obviously, something very strange is going on.

5. Experimentation

As in our C example, most problems in program understanding can be formulated as a search for *causes:* What is the cause for `buf` containing `"a = 0"`? It may be surprising that none of the techniques discussed so far is able to find an actual cause—or, more precisely, to *prove* that some aspect of a program is actually the cause for a specific behavior. To prove actual causality, one needs two experiments: one where cause and effect occur, and one where neither cause nor effect occur. The cause must precede the effect, and the cause must be a *minimal* difference between these experiments.

Searching for the actual cause thus requires a series of *experiments,* refining and rejecting hypotheses until a minimal difference—the actual cause—is isolated. This implies multiple program runs that are *controlled* by the reasoning process.

In our C example, our earlier induction step has already refined the cause in the program state: `a` is the cause for `buf` containing `"a = 0"`, because we can alter `a` such that `buf` has a different content. However, altering `a` in an experiment to, say, `2097153`, makes `buf` contain `"a = -2147483648"`. Would we consider this non-failing?

So, we decide that `a` is sane, and turn to the `sprintf` call. Assuming that `sprintf` works as specified, the only cause that can remain is the `format` string `"a = %d"` as `sprintf` argument. Indeed, it turns out that `%d` is a format for integers, while `a` is declared as a floating-point value:

```
double a;
```

To verify that the format string is really the cause for `"a = 0"` in `buf`, we experimentally change the `format` variable from `"a = %d"` to `"a = %f"`. Our observation confirms that `buf` now has a sane value; this proves that the `format` string was indeed the cause for the failure.

Where do we obtain such alterations from? Obviously, a string like `format` can have an infinite number of possible contents. Finding the one format string that causes the bad `buf` content to turn into the correct one is left to the programmer; actually, this is part of writing a program that works as intended.

Nonetheless, even the search for causes can be automated—at least, if one has an alternate run where the effect does *not* occur. Our *delta debugging* approach can narrow down the initial difference between the two runs to the actual cause in program input [8] or program state [15]. Delta debugging creates artificial *intermediate* configurations that encompass only a part of the initial difference. Testing such configurations and assessing the outcome then allows to narrow down the actual cause.

Delta debugging has successfully isolated cause-effect chains from programs that so far had defied all kinds of deductive analysis, such as the GNU C compiler.

6. A Hierarchy of Program Analysis

By now, we have seen four techniques which are the foundation of program analysis tools. Each of these techniques induces a *class* of program analysis tools, defined by the *number of program runs* considered:

Deductive program analysis ("static analysis") generates findings *without executing* the program.

Observational program analysis generates findings from a *single execution* of the program.

Inductive program analysis generates findings from given *multiple executions* of the program.

Experimental program analysis generates findings from *multiple executions* of the program, where the executions are *controlled* by the tool.

As in Figure 1, these classes form a hierarchy where tools of each "outer" class may make use of the techniques in "inner" classes. Hence, dynamic slicing (observation) makes use of static slices (deduction); invariant detection (induction) relies on observation; delta debugging (experimentation) relies on observation and induction.

The classes also induce capabilities and limits:

- To determine causes, one needs experiments.
- To summarize findings, one needs induction over multiple runs.
- To find facts, one needs observation.
- And deduction, perhaps to some surprise, cannot tell any of these—simply because it abstracts from concrete program runs and thus runs the risk of abstracting away some relevant aspect.

However, deduction effectively proves what can and what cannot happen in the examined abstraction level; hence, it is an excellent guidance on what to observe, where to induce from and what to experiment.

7. Conclusion and Future Work

Program analysis tools can be classified into a hierarchy along the used reasoning techniques—deduction, observation, induction, and experimentation. Each class is defined by the used knowledge sources which impose capabilities and limits. This allows for a finer distinction of dynamic analysis techniques; names like observation, induction, or experimentation link directly to the techniques that programmers use in program comprehension.

While deduction and observation are quite well-understood, we have only yet begun to automate induction and experimentation techniques. Research in machine learning and data mining has produced a wealth of induction techniques. All of these can be applied to program runs in order to find patterns, rules, and anomalies—in runs and in code.

While induction works on a given set of program runs, we can use experimentation to gather more data from new, generated runs. The challenges here are when to use additional experimentation, how to generate runs that satisfy desired properties, and how to guide the experimentation process. The capability to design, run, and assess experiments automatically is unique to dynamic program analysis; we should make use of it.

Finally, program analysis can greatly benefit from further integration of "inner" tools and "outer" tools. Integrating experimentation with further inductive or deductive techniques is the main challenge in dynamic program analysis—and its greatest chance.

Acknowledgments. Silvia Breu, Holger Cleve, Jens Krinke and Tom Zimmermann provided substantial comments on earlier revisions of this paper.

References

[1] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation (PLDI)*, volume 25(6) of *ACM SIGPLAN Notices*, pages 246–256, White Plains, New York, June 1990.

[2] W. de Pauw, E. Jensen, N. Mitchell, G. Sevitsky, J. M. Vlissides, and J. Yang. Visualizing the execution of java programs. In S. Diehl, editor, *Proc. of the International Dagstuhl Seminar on Software Visualization*, volume 2269 of *Lecture Notes in Computer Science*, pages 163–175, Dagstuhl, Germany, May 2002. Springer-Verlag.

[3] M. Ducassé. Coca: An automated debugger for C. In *Proc. International Conference on Software Engineering (ICSE)*, pages 504–513, Los Angeles, California, May 1999.

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

[5] W. G. Griswold, editor. *Proc. Tenth ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-10)*, Charleston, South Carolina, Nov. 2002. ACM Press.

[6] T. Gyimóthy, Á. Beszédes, and I. Forgács. An efficient relevant slicing method for debugging. In *Proc. ESEC/FSE'99 – 7th European Software Engineering Conference / 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, volume 1687 of *Lecture Notes in Computer Science*, pages 303–321, Toulouse, France, Sept. 1999. Springer-Verlag.

[7] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In ICSE 2002 [9], pages 291–302.

[8] R. Hildebrandt and A. Zeller. Simplifying failure-inducing input. In *Proc. ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 135–145, Portland, Oregon, Aug. 2000.

[9] *Proc. International Conference on Software Engineering (ICSE)*, Orlando, Florida, May 2002.

[10] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In ICSE 2002 [9], pages 467–477.

[11] B. Korel and J. Laski. Dynamic slicing of computer programs. *The Journal of Systems and Software*, 13(3):187–195, Nov. 1990.

[12] R. Lencevicius. *Advanced Debugging Methods*. Kluwer Academic Publishers, Boston, 2000.

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

[14] M. Weiser. Programmers use slices when debugging. *Communications of the ACM*, 25(7):446–452, 1982.

[15] A. Zeller. Isolating cause-effect chains from computer programs. In Griswold [5], pages 1–10.