Lecture 22: Dynamic Analysis

17-355/17-665/17-819: Program Analysis

Rohan Padhye

April 14, 2022

* Course materials developed with Jonathan Aldrich and Claire Le Goues

institute for SOFTWARE RESEARCH | Carnegie Mellon University School of Computer Science

What is dynamic analysis?

- Observe program behavior during *execution* on *one or more inputs*.
- Examples:
  - Code coverage (→ Greybox fuzzing, fault localization)
  - Performance Profiling
    - Code profiling, memory profiling, algorithmic profiling
  - Invariant Generation
  - Concolic Execution
  - Data structure analysis
  - Concurrency analysis: Race detection
  - Concurrency analysis: Deadlock detection
  - Taint Analysis (→ Security & Privacy)
  - … (many many more)

Motivation: Invariants for Program Verification

Proof of a Program: FIND

C. A. R. HOARE
Queen's University, Belfast, Ireland*

A proof is given of the correctness of the algorithm "Find." First, an informal description is given of the purpose of the program and the method used. A systematic technique is described for constucting the program proof during the process of coding it, in such a way as to prevent the intrusion of logical errors. The proof of termination is treated as a separate exercise. Finally, some conclusions relating to general programming methodology are drawn.

KEY WORDS AND PHRASES: proofs of programs, programming methodology, program documentation, program correctness, theory of programming
CR CATEGORIES: 4.0, 4.22, 5.21, 5.23, 5.24

1. Introduction

In a number of papers [1, 2, 3] the desirability of proving the correctness of programs has been suggested and this has been illustrated by proofs of simple example programs. In this paper the construction of the proof of a useful, efficient, and nontrivial program, using a method based on invariants, is shown. It is suggested that if a proof is constructed as part of the coding process for an algorithm, it is hardly more laborious than the traditional practice of program testing.

sort the whole array. If the array is small, this would be a good method; but if the array is large, the time taken to sort it will also be large. The Find program is designed to take advantage of the weaker requirements to save much of the time which would be involved in a full sort.

The usefulness of the Find program arises from its application to the problem of finding the median or other quantiles of a set of observations stored in a computer array. For example, if N is odd and f is set to $(N + 1)/2$, the effect of the Find program will be to place an observation with value equal to the median in $A[f]$. Similarly the first quartile may be found by setting f to $(N + 1)/4$, and so on.

The method used is based on the principle that the desired effect of Find is to move lower valued elements of the array to one end—the "left-hand" end—and higher valued elements of the array to the other end—the "right-hand" end. (See Table I(a)). This suggests that the array be scanned, starting at the left-hand end and moving rightward. Any element encountered which is small will remain where it is, but any element which is large should be moved up to the right-hand end of the array, in exchange for a small one. In order to find such a small element, a separate scan is made, starting at the right-hand end and moving leftward. In this scan, any large element encountered remains where it is; the first small element encountered is moved down to the left-hand end in exchange for the large element already encountered in the rightward scan. Then both scans can be resumed until the next exchange is necessary. The process is repeated until the scans meet somewhere in the middle of the array. It is then known that all elements to the left of this meeting point will be small, and all elements to the right will be

Finding invariants manually

- Hoare's post-condition:

The required result is:

$$\forall p, q(1 \leq p \leq f \leq q \leq N \supset A[p] \leq A[f] \leq A[q])$$

[Found]

- Some intermediate invariants:

$$m \leq f \,\&\, \forall p, q(1 \leq p < m \leq q \leq N \supset A[p] \leq A[q])$$

[m-invariant]

Similarly, n is intended to point to the rightmost element of the middle part; it must never be less than f, and there will always be a split just to the right of it:

$$f \leq n \,\&\, \forall p, q(1 \leq p \leq n < q \leq N \supset A[p] \leq A[q])$$

[n-invariant]

Final Result

```
begin
  comment  This program operates on an array A[1:N], and a
    value of f(1 ≤ f ≤ N). Its effect is to rearrange the elements
    of A in such a way that:
      ∀p,q(1≤p≤f≤q≤N⊃A[p]≤A[f]≤A[q]);
  integer m, n;  comment
      m ≤ f & ∀p,q(1≤p<m≤q≤N⊃A[p]≤A[q]),
      f ≤ n  & ∀p,q(1≤p≤n<q≤N⊃A[p]≤A[q]);
    m := 1;  n := N;
  while m < n do
  begin integer r, i, j, w;
    comment
        m ≤ i & ∀p(1≤p<i⊃A[p]≤r),
        j ≤ n  & ∀q(j<q≤N⊃r≤A[q]);
      r := A[f];  i := m;  j := n;
    while i ≤ j do
    begin while A[i] < r do i := i + 1;
      while r < A[j] do j := j − 1
      comment  A[j] ≤ r ≤ A[i];
      if i ≤ j then
      begin w := A[i];  A[i] := A[j];  A[j] := w;
        comment  A[i] ≤ r ≤ A[j];
        i := i + 1;  j := j − 1;
      end
    end increase i and decrease j;
    if f ≤ j then n := j
  else if i ≤ f then m := i
    else go to L
  end reduce middle part;
L:
end Find
```

Insight

- Given a program location, if we could infer an invariant for that location, we could have …
  - Loop invariants (location = loop head)
  - Function pre-conditions (location = entry)
  - Function post-conditions (location = exit)
- Can we do this **automatically**?
- Two insights:
  - An invariant always holds on all executions
  - We can detect spurious false invariants

Dynamic Invariant Detection

- What if we require that the program come equipped with inputs?
  - An indicative workload
  - High-coverage test cases
- Since an invariant holds on every execution (by definition), any candidate invariant that fails even once can be tossed out!
- Plan: generate many candidate invariants, filter out the false ones!

A bad idea

- Given:
  - while b do c
- Instrument:
  - while b do (print Inv1; print Inv2; … ; c)
  - Run on all tests, filter out on false
- How many candidate invariants are there?

Templates

- Given program variables x, y, and z
  - x = c — constant
  - x != 0 — non-zero
  - x >= c — bounds
  - y = ax + b — linear
  - x < y — ordering
  - (x + y) % b = a — math functions
  - z = ax + by + c — linear
- At most three variables at a time: finite!

Ernst, M. D., Cockrell, J., Griswold, W. G., & Notkin, D. (2001).
Dynamically discovering likely program invariants to support program evolution.
IEEE transactions on software engineering, *27*(2), 99-123.

Daikon

- The Daikon invariant detection algorithm
  - For every program location
    - For all triples of in-scope variables
      - Instantiate invariant templates to obtain candidate invariants
      - Instrument program
- For every test case
  - Run instrumented program
  - Filter out any falsified candidate invariant
- Running time: cubic in in-scope variables, linear in test suite, linear in program

Ernst, M. D., Cockrell, J., Griswold, W. G., & Notkin, D. (2001).
Dynamically discovering likely program invariants to support program evolution.
IEEE transactions on software engineering, *27*(2), 99-123.

Exercise: Infer Likely Invariants

Program: (input= N >0)

```
i := 0
while i != N:
  i := i + 1
```

Loop Invariants to Evaluate

- i = 0
- i < 0
- i <= 0
- i > 0
- i >= 0
- N = 0
- N < 0
- N <= 0
- N >= 0
- N > 0
- i == N
- i < N
- i <= N
- i > N
- i >= N

Daikon: Limitations

- False Negatives
  - If your invariant does not fit a template, Daikon cannot find it
  - Example: l + u - 1 <= 2p <= l + u (bsearch pivot)
  - Example:

The required result is:

$$\forall p, q (1 \leq p \leq f \leq q \leq N \supset A[p] \leq A[f] \leq A[q])$$

[Found]

- Nothing prevents Daikon from finding these
- But each increase in the language of candidate invariants bloats the complexity

Daikon: Limitations

- False Positives from limited input
  - If you only test your sorting program on one input, [4;2;3], Daikon will learn output[0] = 2
  - But making high-coverage, high-adequacy tests is easy, no? That's why we're doing formal verification. Oh, right.
- False Positives from linguistic coincidence
  - Ex: ptr % 4 == 0
  - Ex: x <= MAX_INT
  - Not false, but not related to correctness.

Dynamic Invariant Detection (DIG)

- Daikon is ill-suited for richer languages of invariants (e.g., non-linear relations, array relations, etc.) because all candidate invariants must be listed and considered.

- Idea:
  - Instead of listing invariants, list values, and induce invariants via constraint solving
  - Ex: instead of printing x>y, x<y, x>=y, etc., just print out x and y and figure out which is true later

Thanhvu Nguyen, Deepak Kapur, Westley Weimer, and Stephanie Forrest. 2014. DIG: A Dynamic Invariant Generator for Polynomial and Array Invariants. ACM Trans. Softw. Eng. Methodol. 23, 4, Article 30 (September 2014)

Dig Example: Cohen's Division

```
1  def intdiv(x, y):
2      q = 0  // quotient
3      r = x  // remainder
4      while r ≥ y:
5          a = 1
6          b = y
7          while r ≥ 2b:
8              [L] // loop invariant
9              a = 2a
10             b = 2b
11         r = r - b
12         q = q + a
13     return q
```

Thanhvu Nguyen, Deepak Kapur, Westley Weimer, and Stephanie Forrest. 2014. DIG: A Dynamic Invariant Generator for Polynomial and Array Invariants. ACM Trans. Softw. Eng. Methodol. 23, 4, Article 30 (September 2014)

Cohen's Division on input (15,2)

```
1  def intdiv(x, y):
2      q = 0   // quotient
3      r = x   // remainder
4      while r ≥ y:
5          a = 1
6          b = y
7          while r ≥ 2b:
8              [L] // loop invariant
9              a = 2a
10             b = 2b
11         r = r - b
12         q = q + a
13     return q
```

x	y	a	b	q	r
15	2	1	2	0	15
15	2	2	4	0	15
15	2	1	2	4	7

Thanhvu Nguyen, Deepak Kapur, Westley Weimer, and Stephanie Forrest. 2014. DIG: A Dynamic Invariant Generator for Polynomial and Array Invariants. ACM Trans. Softw. Eng. Methodol. 23, 4, Article 30 (September 2014)

Cohen's Division on input (4,1)

```
1  def intdiv(x, y):
2      q = 0   // quotient
3      r = x   // remainder
4      while r ≥ y:
5          a = 1
6          b = y
7          while r ≥ 2b:
8              [L] // loop invariant
9              a = 2a
10             b = 2b
11         r = r - b
12         q = q + a
13     return q
```

x	y	a	b	q	r
15	2	1	2	0	15
15	2	2	4	0	15
15	2	1	2	4	7
4	1	1	1	0	4
4	1	2	2	0	4

Thanhvu Nguyen, Deepak Kapur, Westley Weimer, and Stephanie Forrest. 2014. DIG: A Dynamic Invariant Generator for Polynomial and Array Invariants. ACM Trans. Softw. Eng. Methodol. 23, 4, Article 30 (September 2014)

Cohen's Division Desires

```
1  def intdiv(x, y):
2      q = 0   // quotient
3      r = x   // remainder
4      while r ≥ y:
5          a = 1
6          b = y
7          while r ≥ 2b:
8              [L] // loop invariant
9              a = 2a
10             b = 2b
11         r = r - b
12         q = q + a
13     return q
```

x	y	a	b	q	r
15	2	1	2	0	15
15	2	2	4	0	15
15	2	1	2	4	7
4	1	1	1	0	4
4	1	2	2	0	4

$$\{b = ya, x = qy + r, r \geq 2ya\}$$

"Geometric Invariant Inference"

Thanhvu Nguyen, Deepak Kapur, Westley Weimer, and Stephanie Forrest. 2014. DIG: A Dynamic Invariant Generator for Polynomial and Array Invariants. ACM Trans. Softw. Eng. Methodol. 23, 4, Article 30 (September 2014)

Dynamic Analysis: Recap

- Observe program behavior during *execution* on *one or more inputs*.
- Examples:
  - Code coverage (→ Greybox fuzzing, fault localization)
  - Performance Profiling
    - Code profiling, memory profiling, algorithmic profiling
  - Invariant Generation
  - Concolic Execution
  - Data structure analysis
  - Concurrency analysis: Race detection
  - Concurrency analysis: Deadlock detection
  - Taint Analysis (→ Security & Privacy)
  - … (many many more)

Collecting execution info

- Runtime monitoring
  - Combination of OS-level interrupts, introspection of exec state (registers, program counter), and saved debug info
  - Examples: gdb (ddd), Java Debug Interface (VisualVM)
- Run on a specialized VM
  - e.g., valgrind
- Instrument at compile time
  - e.g., Aspects, logging
- Instrument bytecode on-the-fly
  - Python: `sys.settrace`; Java: ASM toolkit with a javaagent or classloader

Collecting execution info

- What to collect? *Only what's necessary*
- Key idea (again): *Abstraction*
- Examples:
  - Code coverage → Log branches
  - Profiling → Log loops, function calls, allocations, frees, etc.
  - Invariant generation → Log predicates over vars in scope
  - Concolic execution → Track symbolic values; log branch constraints
  - Race detection → Track locks, vector clocks; log accesses

Techniques for Instrumentation

- Choice of program representation is critical
  - Source code
  - Abstract syntax trees
  - Control-flow graph (e.g. LLVM)
  - Bytecode (e.g. JVM .class files, Python .pyc)
  - Assembly / Machine code
- Which one is best? Depends.
  - ASTs good for expression-level instrumentation (e.g. concolic testing)
  - CFGs for basic-block-level logic (e.g. branch coverage, loop profiling)
  - Bytecode/assembly for tracking low-level details sans types, etc.
    - May be the only thing available when instrumenting closed source or otherwise precompiled third-party libraries

AST-level instrumentation

- Most general form: Replace every node with a callback to a node “handler”, which interprets the expression/statement
- Example:
  - **Original**: a + (b – 1)
  - **Instrumented**: add(var(‘a’, a), sub(var(‘b’, b), const(1))
  - **Default Handler:** function add(x1, x2) { return x1 + x2; }
  - **Logging Handler:**
    - function add(x1, x2) { printf(“Adding %s + %s2”, x1, x2); return x1 + x2; }
  - **Concolic Handler**:
    - function var(name, val) { concolic.track(name, val); return concolic.get(name); }
    - function add(x1, x2) { return concolic.add(x1, x2); }
- Sample tool: Jalangi (JavaScript)

Bytecode Instrumentation

- Bytecode: Mid-to-low-level IR used by somewhat dynamic language runtimes (e.g. JVM, Python, WebAssembly)
- Often use a stack machine representation
  - Accesses and manipulates a stack of values
  - Instructions are simple and operate on stack values
  - Very easy to write an AST-to-stack-machine compiler
    - Pre-order tree traversal to emit code (”emit” operands first, then “emit” node)
  - Bytecode can be interpreted (e.g. CPython) or JIT-compiled to assembly (e.g. JVM HotSpot)

Stack Machine Bytecode

Instruction (at <label>)	Stack (before → after)
• Push <const>	• … → … <const>
• Load <var>	• … → … E(var)
• Store <var>	• … val → … // E[var ↦ val]
• Dup	• … val → … val val
• Add	• … val_1 val_2 → … (val_1+val_2)
• Invoke <func> <nargs>	• … val_1 val_2 … val_{nargs} → … result
• Jump <label'>	• … → … // PC = label'
• Jump-if-zero <label'>	• … val → … // PC = val ? PC+1 : label'

Exercise: Convert to stack-machine bytecode

- x = foo(a + bar())

Exercise: Instrument bytecode

- Replace all “a + b” with call to “my_add(a, b)”
- What to search for:
- What to replace with:

Exercise: Instrument bytecode

- For each conditional branch executed, we want to print 1 if taken and 0 if not taken.
- What to search for:
- What to replace with:

Static vs Dynamic Analysis

- Over-approximation vs Under-approximation
- When is one better than other? Tradeoffs!
  - Soundness/Completeness
    - Static analysis often “sound” for over-approximate reasoning (e.g. verification)
    - Dynamic Analysis can be ”sound” for under-approximate reasoning (e.g. hot spots or bugs).
    - Neither technique is complete in general.
  - Scalability
    - Static analysis often scales super-linearly with program size
    - Dynamic analysis *tries* to scale linearly with execution length
  - Feasibility
    - Static analysis may be impossible with incomplete information (e.g. dynamically loaded code, dependency injection, multi-language code, hardware interaction)
    - Dynamic analysis is only useful when appropriate program inputs are available