

Dynamically Detecting Invariants

Lecture 6a

Motivation

- Programmers have little help understanding code they didn't write
 - Because they need to extend it
 - Because they need to debug it
- Buzzwords
 - legacy code
 - software evolution
 - re-engineering

The Idea

- Gather traces of many executions of a program
- Analyze traces for invariants
 - Properties that hold across all traces
- Embodied in a tool called *Daikon*

Typical Invariants

- $x = y$
- $x > y$
- $y = 2x$
- $n = \text{size}(A[])$
- $\text{size}(B[]) = \text{size}(A[])$

- NB: All invariants are with respect to a particular program point.

Example

$i := 0;$

$s := 0;$

while $i \neq n$ {

$s := s + b[i];$

$i := i + 1;$

}

$n = \text{size}(b)$

$n \text{ in } [7..13]$

$s = \text{sum}(b[0..i-1])$

$i \leq n$

$s = \text{sum}(b)$

Comments

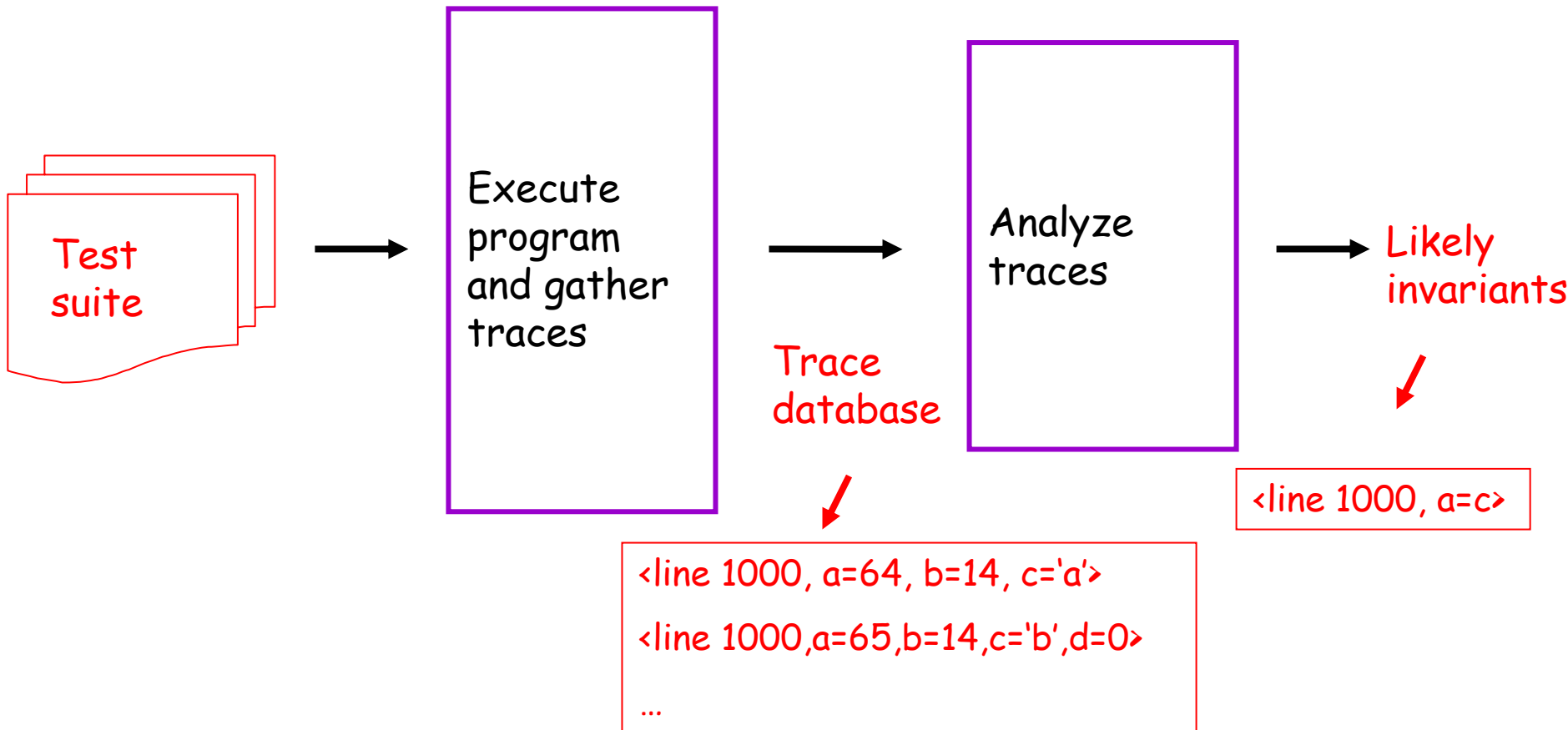
- Some invariants are trivial but useful:
 $n = \text{size}(b)$
- Some invariants are silly:
 $n \text{ in } [7..13]$
- Some invariants are truly interesting
 $s = \text{sum}(b[0..i-1])$

How Does This Work?

- By brute force.

Evaluate every possible invariant at every program point across all traces

System Architecture



Possible Invariants

- Fixed menu of invariants between scalar variables

$$x \cdot y, x = y, x = ay + b, \dots$$

- Fixed menu of invariants over arrays

min, max, non-decreasing, all the same, . . .

- “Derived variables” (particular expressions)

$$s = \text{sum}(b[0..i-1])$$

Number of Invariants

- How many possible invariants are there?
- Ans: Let V be the number of variables in scope at a program point.
 - Complexity is $O(V^3)$.
 - Because some invariants involve 3 variables.

Notes on “Derived Variables”

- Again, really expressions
- Set of rules for introducing derived variables
 - E.g., if E is an integer variable, $E-1$ and $E+1$ are derived variables
 - Different rules for different types
- Impose limit on size of derived variables
 - Keeps things from getting out of hand

Performance

- This is expensive
 - Lots of variables (including derived variables)
 - Lots of potential invariants
 - Lots of program points
- Naïve brute force won't work
 - We need some optimizations

Optimization 1

- Once an invariant is falsified, never check it again
- Most potential invariants are falsified early
- Thus, the running time is more related to the number of invariants discovered than the number of possible invariants.

Optimization 2

- Do not introduce derived variables unless they are useful.
- E.g., $A[j]$ is useful only if $j < \text{size}(A)$
 - No use in indexing outside of A 's range
- This is essentially multi-stage inference
 - First infer the invariant $j < \text{size}(A)$
 - Introduce derived variable $A[j]$

Optimization 3

- Don't consider every program point
- Daikon looks only at
 - Function entry
 - Function exit
 - Loop headers

Optimization 4

- Exploit relationships between invariants
- Examples
 - If $x = y$, then check invariants for either x or y , but not both
 - If $P \neq Q$, then check P first and if it holds, don't check Q
- Exploited in Daikon, but apparently somewhat ad hoc

Analysis of Performance

- Hard to tell how well all of this scales.
 - Experiments are all on small-to-miniscule programs
 - Base implementation of Daikon is not highly optimized
 - E.g., written in Python
- But
 - Intuitively, this ought to scale to big programs
 - Complexity question: How does the # of variables in scope scale with program size?

Analysis of Performance (Cont.)

- Performance is slow
 - 220 seconds/instrumented program point
- Mentioned several times that it could be made much faster with a little work

Next Topic: Extensions

- So far we handle only scalars and arrays
- Other important features to model:
 - Objects
 - Data structures

Objects

- Think Java
- If x : `Object`, what invariants can we infer for x ?
- But if dynamically, x always has type `Foo`, we can look for invariants of x 's `Foo` fields
- Requires two passes over the trace database:
 - First pass: Determine if x has a unique type `Foo` in test suite
 - Second pass: examine x 's `Foo` fields for invariants

Data Structures

- What do we do about trees, lists, etc?
- What invariants are interesting
 - for lists?
 - for trees?
 - for other data structures?

Control Flow

- Some invariants only hold conditionally
 "if $y \neq \text{null}$ then $x = y.\text{next}$ "
- Idea: Split the trace into two pieces using the condition of the invariant, and evaluate both
- But how do we come up with the splitting conditions?
 - Ans: Hard-wired heuristics

Next Topic: Confidence

- When do we have enough evidence for an invariant to report it?
- Example: If we have only one sample showing $x=y$ and none contradicting that, do we claim $x=y$ as an invariant?

Confidence (Cont.)

- Use pseudo-statistics to estimate our confidence
- Calculate odds of coincidental invariant given
 - Range of values actually observed for a variable
 - Uniform distribution on choice of values
- Report invariants that exceed user-specified confidence threshold

Next Topic: Handling Low Level Languages

- Instrumentation done as source-to-source translation
 - Argued that instrumentation could be done on binaries
 - Is this true?

Handling Low Level Languages (Cont.)

- Think C
 - The grubbiest case
- Root problem
 - Daikon evaluates expressions during program execution
 - These expressions must not cause exceptions
- Sample issues
 - Uninitialized variables
 - Buffer size
 - Avoiding overruns and underruns

Handling Low Level Languages (Cont.)

- Solution
 - Keep additional status information with each variable/array
 - E.g., whether variable is initialized
 - E.g., array size
- But this problem is inherent in any dynamic analysis technique
 - Instrumentation code can't change the program's computation

A Question

- What happens with `void *` in C?
- What invariants can we infer for it?
 - Note Java `Object` trick won't work here
- Dependence on runtime type information
 - Hints at problems in instrumenting binaries

Usability

- One usability experiment
 - Add E^+ to a regular expression compiler*
- Done by two programmers, via pattern matching and experimenting with code for E^*
- The programmers exploited invariants
 - But, apparently mostly via direct queries to the trace database, not from the automatically inferred invariants

Automatically Inferred Invariants

- How useful is the automatic inference?
- Argument in paper is "serendipity"
 - Programmer alerted to invariants they might not have thought of
 - Probably useful, but how often?

Inferring Known Invariants

- Applied Daikon to the "*Science of Programming*"
- CS text that argues for specifications in the form of invariants
 - Lots of small programs annotated with and derived from invariants
 - Hoare/Dijkstra style programming

Inferring Known Invariants (Cont.)

- Daikon did well
 - Inferred postulated invariants
 - In at least one case, filled in a crucial invariant missing from the text
- But
 - These programs are truly small
 - Finding missing invariant begs a question
 - Not even the author of the text cared enough to carefully check that he had the right invariants!

**Warning
Opinions Ahead!**

The Good News

- This seems to be a new idea
- A rare thing, indeed

Analysis

- The weakest part of Daikon is the test suite
 - Even with an extensive test suite, clearly get a number of silly invariants
 - Poor test suite \neq poor results
- Ernst argues Daikon helps identify poor test suites
 - Making lemonade?
- Other standard test metrics could be incorporated
 - Code coverage

How Important is Invariant Inference?

- Invariant inference is expensive
- In actual use, querying the trace database was at least as important as using the inferred invariants
- Are the arguments for inference compelling?
 - Serendipity
 - Uncovering problems in tests
- Is being able to query the trace database the useful thing?