


CSE403: Software Engineering

David Notkin
Winter 2009

Program invariants

- Invariants can aid in the development of correct programs
  - The invariants are defined explicitly as part of the construction of the program
- Invariants can aid in the evolution of software as well
- In particular, programmers can easily make changes that violate unstated invariants
  - The violated invariants are often far from the site of the change
  - These changes can cause errors
  - The presence of invariants can reduce the number of or cost of finding these violations

But…

- …most programs have few invariants explicitly written by programmers
- Ernst's idea: trace multiple executions of a program and apply machine learning to discover likely invariants (such as those found in assert statements or specifications)
  - *x > abs(y)*
  - *x = 16*y + 4*z + 3*
  - *array a contains no duplicates*
  - *for each node n, n = n.child.parent*
  - *graph g is acyclic*
  - …

Example: Recover formal specification

```
// Sum array b of length n into
// variable s
i := 0; s := 0;
while i ≠ n do
  { s := s + b[i];  i := i + 1 }
```

- Precondition: $n \geq 0$
- Postcondition: $S = \Sigma_{0 \leq j < n} b[j]$
- Loop invariant:
  $0 \leq i \leq n$ and $S = \Sigma_{0 \leq j < i} b[j]$

Test suite: first guess

- 100 randomly-generated arrays
  - length uniformly distributed from 7 to 13
  - elements uniformly distributed from –100 to 100

Inferred invariants

```
ENTRY:
   N = size(B)
   N in [7..13] ♦
   B: All elements in [-100..100]
EXIT:
   N = I = orig(N) = size(B)
   B = orig(B)
   S = sum(B) ♦
   N in [7..13]
   B: All elements in [-100..100]
```

Inferred loop invariants

```
LOOP:
   N = size(B)
   S = sum(B[0..I-1]) ♦
   N in [7..13]
   I in [0..13] ♦
   I <= N♦
   B: All elements in [-100..100]
   B[0..I-1]: All elements in [-100..100]
```


Example: Code without explicit invariants

- 563-line C program: regular expression search & replace [Hutchins][Rothermel]
- Task: modify to add Kleene +
- Complementary use of both detected invariants and traditional tools (such as grep)

Programmer use of invariants

- Helped explain use of data structures
  – regexp compiled form (a string)
- Contradicted some maintainer expectations
  – anticipated *lj* < *j* in `makepat`
  – queried for counterexample
  – avoided introducing a bug
- Revealed a bug
  – when *lastj* = **j* in `stclose`, array bounds error

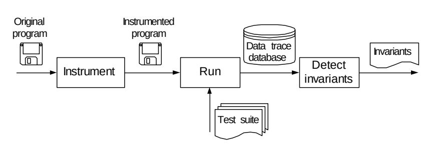
More invariant uses

- Showed procedures used in limited ways
  – `makepat`
    start = 0 and *delim = '\0'*
- Demonstrated test suite inadequacy
  – *#calls(in_set_2) = #calls(stclose)*
- Changes in invariants validated program changes
  – `stclose`: **j = orig(*j)+1*
  – `plclose`: **j ≥ orig(*j)+2*

Experiment 2 conclusions

- Invariants
  – effectively summarize value data
  – support programmer's own inferences
  – lead programmers to think in terms of invariants
  – provide serendipitous information
- Additional useful components of Daikon
  – trace database (supports queries)
  – invariant differencer

Dynamic invariant detection



- Look for patterns in values the program computes
  – Instrument the program to write data trace files
  – Run the program on a test suite
  – Invariant engine reads data traces, generates potential invariants, and checks them
- Roughly, machine learning over program traces

Requires a test suite

- Standard test suites are adequate
- Relatively insensitive to test suite (if large enough)
- No guarantee of completeness or soundness
- Complementary to other techniques and tools

Sample invariants

- *x,y,z* are variables; *a,b,c* are constants
- Invariants over numbers
  - unary: $x = a$, $a \le x \le b$, $x \equiv a (mod\ b)$, …
  - n-ary: $x \le y$, $x = ay + bz + c$, $x = max(y, z)$, …
- Invariants over sequences
  - unary: sorted, invariants over all elements
  - with sequence: subsequence, ordering
  - with scalar: membership

Checking invariants

- For each potential invariant:
  - Instantiate
    - That is, determine constants like *a* and *b* in $y = ax + b$
  - Check for each set of variable values
  - Stop checking when falsified
- This is inexpensive
  - Many invariants, but each cheap to check
  - Falsification usually happens very early

Relevance

- Our first concern was whether we could find any invariants of interest
- When we found we could, we found a different problem
  - We found many invariants of interest
  - But most invariants we found were not relevant

Find relationships over non-variables

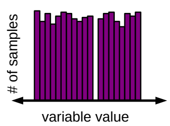
- array: *length, sum, min, max*
- array and scalar: element at index, subarray
- number of calls to a procedure
- …

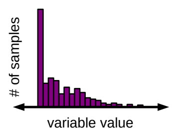
Unjustified properties

- Given three samples for **`x`**:
  - **`x = 7`**
  - **`x = -42`**
  - **`x = 22`**
- Potential invariants:
  - $x \ne 0$
  - $x \le 22$
  - $x \ge -42$

Statistically check hypothesized distribution

- Probability of no zeroes (to show x ≠ 0) for v values of x in range of size r $\left(1-\frac{1}{r}\right)^v$
- Range limits (e.g., x ≤ 22)
  - same number of samples as neighbors (uniform)
  - more samples than neighbors (clipped)





Duplicate values

- Array sum program:

```
i := 0; s := 0;
while i ≠ n do
  { s := s + b[i];  i := i + 1 }
```

- *b* is unchanged inside loop
- Problem: at loop head
  - ***–88 ≤ b[n – 1] ≤ 99***
  - ***–556 ≤ sum(b) ≤ 539***
- Reason: more samples inside loop

Disregard duplicate values

- Idea: count a value only if its variable was just modified
- Result: eliminates undesired invariants

Redundant invariants

- Given
  0 ≤ i ≤ j
- Redundant
  a[i] ∈ a[0..j]
  max(a[0..i]) ≤ max(a[0..j])
- Redundant invariants are logically implied
- Implementation contains many such tests

Suppress redundancies

- Avoid deriving variables: suppress 25-50%
  - equal to another variable
  - nonsensical
- Avoid checking invariants:
  - false invariants: trivial improvement
  - true invariants: suppress 90%
- Avoid reporting trivial invariants: suppress 25%

Unrelated variables

```
bool b;
int *p;
```

b < p

```
int myweight, mybirthyear;
```

myweight < mybirthyear

Limit comparisons

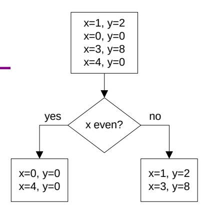
- Check relations only over comparable variables
  - declared program types: 60% as many comparisons
  - Lackwit [O'Callahan]: 5% as many comparisons; scales well
- Runtime: 40-70% improvement
- Few differences in reported invariants

Richer types of invariant

- Object/class invariants
  - `node.left.value < node.right.value`
  - `string.data[string.length] = '\0'`
- Pointers (recursive data structures)
  - `tree is sorted`
- Conditionals
  - `if proc.priority < 0 then proc.status = active`
  - `ptr = null or *ptr > i`

Conditionals mechanism

- Split the data into parts
- Compute invariants over each subset of data
- Compare results, produce implications




```
if even(x) then
  y = 0
else
  y = 2x
```


Data splitting criteria

- Static analysis
- Distinguished values: zero, source literals, mode, outliers, extrema
- Exceptions to detected invariants
- User-selected
- Exhaustive over random sample

Summary

- Dynamic invariant detection is feasible
- Dynamic invariant detection is accurate & useful
  - Techniques to improve basic approach
  - Experiments provide preliminary support
- Daikon can detect properties in C, C++, Eiffel, IOA, Java, and Perl programs; in spreadsheet files; and in other data sources.
- Easy to extend Daikon to other applications
- http://groups.csail.mit.edu/pag/daikon/