**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

**Example: Recover formal specification**

```c
// 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:** \( n ≥ 0 \)
- **Postcondition:** \( S = \sum_{0 ≤ j ≤ n} b[j] \)
- **Loop invariant:** 
  \[ 0 ≤ i ≤ n \text{ and } S = \sum_{0 ≤ j ≤ i} b[j] \]

**Inferred invariants**

**ENTRY:**

- \( N = \text{size}(B) \)
- \( N \in [7..13] \#
- \( B: \text{All elements in } [-100..100] \)

**EXIT:**

- \( N = I = \text{orig}(N) = \text{size}(B) \)
- \( B = \text{orig}(B) \)
- \( S = \text{sum}(B) \#
- \( N \in [7..13] \#
- \( B: \text{All elements in } [-100..100] \)

**Test suite: first guess**

- 100 randomly-generated arrays
  - length uniformly distributed from 7 to 13
  - elements uniformly distributed from -100 to 100
Inferred loop invariants

```plaintext
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 \( \ell < j \) 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 = "\"`
- Demonstrated test suite inadequacy
  - `#calls(in_set_2) = #calls(stclose)`
- Changes in invariants validated program changes
  - `stclose: \( \ell = \text{orig}(\ell) + 1 \)`
  - `plclose: \( \ell \geq \text{orig}(\ell) + 2 \)`

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: \(x = a, a \leq x \leq b, x = a \text{ mod } b\), ...
  - n-ary: \(x \leq y, x = ay + bz + c, \ x = \max(y, z)\), ...
- 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:
  - \(x \neq 0\)
  - \(x \leq 22\)
  - \(x \geq -42\)
Statistically check hypothesized distribution

- Probability of no zeroes (to show \( x \neq 0 \)) for \( v \) values
  of \( x \) in range of size \( r \)
- Range limits (e.g., \( x \leq 22 \))
  - same number of samples as neighbors (uniform)
  - more samples than neighbors (clipped)

Duplicate values

- Array sum program:
  \[
  i := 0; \quad s := 0; \\
  \text{while } i \neq n \text{ do } \\
  \quad \{ \quad s := s + b[i]; \quad i := i + 1 \quad \}
  \]
- \( b \) is unchanged inside loop
- Problem: at loop head
  - \(-88 \leq b[n - 1] \leq 99 \)
  - \(-556 \leq \text{sum}(b) \leq 539 \)
- 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 \( 0 \leq i \leq j \)
- Redundant
  \[
  a[j] \in a[0..j] \\
  \text{max}(a[0..j]) < \text{max}(a[0..j])
  \]
- 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

```c
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

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/