Dynamically Detecting Likely Program Invariants

Michael Ernst, Jake Cockrell, Bill Griswold (UCSD), and David Notkin
University of Washington
Department of Computer Science and Engineering
http://www.cs.washington.edu/homes/mernst/
Overview

Goal: recover invariants from programs
Technique: run the program, examine values
Artifact: Daikon
Results: • recovered formal specifications
  • aided in a software modification task
Outline: • motivation
  • techniques
  • future work
Goal: recover invariants

Detect invariants like those in `assert` statements

• \( x > \text{abs}(y) \)
• \( x = 16*y + 4*z + 3 \)
• array \( a \) contains no duplicates
• for each node \( n \), \( n = n.\text{child}.\text{parent} \)
• graph \( g \) is acyclic
Uses for invariants

Write better programs [Liskov 86]
Documentation
Convert to assert
Maintain invariants to avoid introducing bugs
Validate test suite: value coverage
Locate exceptional conditions
Higher-level profile-directed compilation [Calder 98]
Bootstrap proofs [Wegbreit 74, Bensalem 96]
Experiment 1: recover formal specifications

Example: Program 15.1.1 from *The Science of Programming* [Gries 81]

// Sum array b of length n into variable s.

\[
i := 0; \ s := 0;
\]
\[
\textbf{while } i \neq n \ \textbf{do}
\]
\[
\{ \ s := s+b[i]; \ i := i+1 \}
\]

Precondition: \( n \geq 0 \)

Postcondition: \( s = (\sum_j: 0 \leq j < n : b[j]) \)

Loop invariant: \( 0 \leq i \leq n \) and \( s = (\sum_j: 0 \leq j < i : b[j]) \)
Test suite for program 15.1.1

100 randomly-generated arrays

• Length uniformly distributed from 7 to 13
• Elements uniformly distributed from -100 to 100
Inferred invariants

15.1.1:::BEGIN                      (100 samples)
    N = size(B)                      (7 values)
    N in [7..13]                     (7 values)
    B                                (100 values)
        All elements in [-100..100]    (200 values)

15.1.1:::END                        (100 samples)
    N = I = N_orig = size(B)         (7 values)
    B = B_orig                       (100 values)
    S = sum(B)                       (96 values)
    N in [7..13]                     (7 values)
    B                                (100 values)
        All elements in [-100..100]    (200 values)
Inferred loop invariants

15.1.1:::LOOP

\[\begin{align*}
N &= \text{size}(B) \quad (7 \text{ values}) \\
S &= \text{sum}(B[0..I-1]) \quad (96 \text{ values}) \\
N \text{ in } [7..13] \quad (7 \text{ values}) \\
I \text{ in } [0..13] \quad (14 \text{ values}) \\
I &\leq N \quad (77 \text{ values}) \\
B \quad (100 \text{ values}) \\
&\quad \text{All elements in } [-100..100] \quad (200 \text{ values}) \\
B[0..I-1] \quad (985 \text{ values}) \\
&\quad \text{All elements in } [-100..100] \quad (200 \text{ values})
\end{align*}\]
Ways to obtain invariants

• Programmer-supplied

• Static analysis: examine the program text
  [Cousot 77, Gannod 96]
  • properties are guaranteed to be true
  • pointers are intractable in practice

• Dynamic analysis: run the program
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
- Offline invariant engine reads data trace files, checks for a collection of potential invariants
Running the program

Requires a test suite
  • standard test suites are adequate
  • relatively insensitive to test suite

No guarantee of completeness or soundness
  • useful nonetheless
Sample invariants

$x, y, z$ are variables; $a, b, c$ are constants

Numbers:

- unary: $x = a, \ a \leq x \leq b, \ x \equiv a \ (\text{mod} \ b)$
- n-ary: $x \leq y, \ x = ay + bz + c, \ x = \max(y, z)$

Sequences:

- unary: sorted, invariants over all elements
- with scalar: membership
- with sequence: subsequence, ordering
Checking invariants

For each potential invariant:

- quickly determine constants (e.g., a and b in $y = ax + b$)
- stop checking once it is falsified

This is inexpensive
Performance

Runtime growth:

- quadratic in number of variables at a program point (linear in number of invariants checked/discovered)
- linear in number of samples or values (test suite size)
- linear in number of program points

Absolute runtime: a few minutes per procedure

- 10,000 calls, 70 variables, instrument entry and exit
Statistical checks

Check hypothesized distribution

To show $x \neq 0$ for $\nu$ values of $x$ in range of size $r$, probability of no zeroes is $\left(1-\frac{1}{r}\right)^\nu$

Range limits (e.g., $x \geq 22$):

• more samples than neighbors (clipped to that value)
• same number of samples as neighbors (uniform distribution)
Derived variables

Variables not appearing in source text

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

Enable inference of more complex relationships

Staged derivation and invariant inference

- avoid deriving meaningless values
- avoid computing tautological invariants
Experiment 2: C code lacking explicit invariants

563-line C program: regexp search & replace

[Hutchins 94, Rothermel 98]

Task: modify to add Kleene +

Use both detected invariants and traditional tools
Experiment 2 invariant uses

Contradicted some maintainer expectations
  anticipated \( l_j < j \) in `makepat`

Revealed a bug
  when \( \text{lastj} = \ast j \) in `stclose`, array bounds error

Explicated data structures
  regexp compiled form (a string)
Experiment 2 invariant uses

Showed procedures used in limited ways

\texttt{makepat: start = 0 and delim = '\0'}

Demonstrated test suite inadequacy

\texttt{calls(in\_set\_2) = calls(stclose)}

Changes in invariants validated program changes

\texttt{stclose: *j = *j_{orig} + 1} \quad \texttt{plclose: *j \geq *j_{orig} + 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

Useful tools:
- trace database (supports queries)
- invariant differencer
Future work

Logics:

• Disjunctions: \( p = \text{NULL} \text{ or } *p > i \)
• Predicated invariants: if condition then invariant
• Temporal invariants
• Global invariants (multiple program points)
• Existential quantifiers

Domains: recursive (pointer-based) data structures

• Local invariants
• Global invariants: structure [Hendren 92], value
More future work

User interface
- control over instrumentation
- display and manipulation of invariants

Experimental evaluation
- apply to a variety of tasks
- apply to more and bigger programs
- users wanted! (Daikon works on C, C++, Java, Lisp)
Related work

Dynamic inference

• inductive logic programming [Bratko 93]
• program spectra [Reps 97]
• finite state machines [Boigelot 97, Cook 98]

Static inference [Jeffords 98]

• checking specifications [Detlefs 96, Evans 96, Jacobs 98]
• specification extension [Givan 96, Hendren 92]
• etc. [Henry 90, Ward 96]
Conclusions

Dynamic invariant detection is feasible
  • Prototype implementation

Dynamic invariant detection is effective
  • Two experiments provide preliminary support

Dynamic invariant detection is a challenging but promising area for future research