# 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

Results:
- recovered formal specifications
- aided in a software modification task

Outline:
- motivation
- example
- techniques
- example
- future work



# Goal: recover invariants

Detect invariants like those in `assert` statements
- `x > abs(y)`
- `x == 16*y + 4*z + 3`
- array `a` contains no duplicates
- each node pointed to by `n`'s `child` slot contains a pointer, in its `parent` slot, back to `n`
- graph `g` is acyclic



# Uses for invariants

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: formally specified programs

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

// Sum array *b* of length *n* into variable *s*.
$i := 0; s := 0;$
**while** $i \neq n$ **do**
  **{** $s := s+b[i]; \ i := i+1$ **}**

Precondition: $n \geq 0$

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

Loop invariant: $0 \leq i \leq n$ and $s = (\Sigma j: 0 \leq j < i : b[j])$



# 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 >= -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 >= -100             (200 values)
```



## More inferred invariants

```
15.1.1:::LOOP   1107 samples
  N = size(B)                        (7 values)
  S = sum(B[0..I-1])                 (96 values)
  N in [7..13]                       (7 values)
  B                                  (100 values)
    All elements in [-100..100]      (200 values)
  I in [0..13]                       (14 values)
  sum(B) in [-556..539]              (96 values)
  B[0] nonzero in [-99..96]          (79 values)
  B[-1] in [-88..99]                 (80 values)
  B[0..I-1]                          (985 values)
    All elements in [-100..100]      (200 values)
  I <= N                             (77 values)
 Negative invariants:
  N != B[-1]                         (99 values)
  B[0] != B[-1]                      (100 values)
```

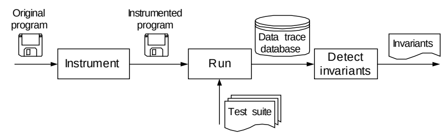


## Obtaining invariants

- Programmer-supplied
- Static analysis: examine the program text
  [Cousot 77, Gannod 96]
  - properties are guaranteed to be true
  - pointers are un-analyzable 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



## Instrumentation

Source-to-source translator

Instrument procedure entry, exit, loop heads:
output value of each variable in scope

C array sizes

C/C++, Java (in progress), Lisp



## Running the program

Requires a test suite

- what test suites are good for invariant detection?
- sensitivity to test suite

No guarantee of completeness or soundness

## Example invariants

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

Numbers:

- unary: $x = a$, $a \le x \le b$, $x \equiv a \pmod b$
- n-ary: $x \le y$, $x = ay + bz + c$, $x = \max(y, z)$

Sequences:

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



## Checking invariants

Quickly determine constants
(e.g., a and b in $y = ax + b$)

Stop checking an invariant once it is falsified



## Performance

Runtime growth:

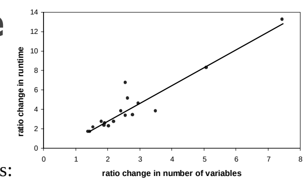
- quadratic in number of variables at a program point (linear in number of invariants checked/discovered)
- linear in number of samples or values (test suite size)
- linear in number of program points

Absolute runtime: a few minutes per program point

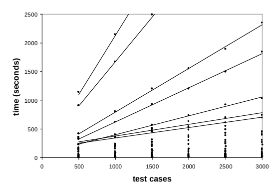


## Performance graphs

Quadratic in number of variables:




Linear in number of test cases:






## Statistical checks

Check hypothesized distribution

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

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

- 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

## Program 15.1.1 test suite 2

100 randomly-generated arrays

- Length exponentially distributed, ≥ 0
- Elements exponentially distributed, signed



## Inferred invariants (2)

```
15.1.1:::BEGIN  100 samples
   N = size(B)                       (24 values)
   N >= 0                            (24 values)

15.1.1:::END    100 samples
   B = B_orig                        (96 values)
   N = I = N_orig = size(B)          (24 values)
   S = sum(B)                        (95 values)
   N >= 0                            (24 values)
```



## More inferred invariants (2)

```
15.1.1:::LOOP   986 samples
   N = size(B)                       (24 values)
   S = sum(B[0..I-1])                (95 values)
   B                                 (96 values)
     All elements in [-6005..7680]   (784 values)
   N in [0..35]                      (24 values)
   I >= 0                            (36 values)
   sum(B) in [-15006..21144]         (95 values)
   B[0..I-1]                         (887 values)
     All elements in [-6005..7680]   (784 values)
   I <= N                            (363 values)
```



## 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 maintainer expectations

anticipated lastj < j, lj < j in **`makepat`**

Revealed a bug

when lastj = *j in **`stclose`**, array bounds error

Explicated structure of compiled regexps

regexp compiled form: string with different properties



## Experiment 2 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`**: $*j = *j_{orig}+1$     **`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: new logics

Disjunctions: `p = NULL` or `*p > I`

Predicated invariants: if *condition* then *invariant*

Temporal invariants

Global invariants (multiple program points)

Existential quantifiers



## Future work: new domains

Recursive (pointer-based) data structures

- Local invariants
- Global invariants: structure [Hendren 92], value



## More future work

Eliminate spurious invariants

- incomparable values
- statistically unsupported

User interface

- control over instrumentation
- display and manipulation of invariants

Experimental evaluation

- apply to variety of tasks
- apply to more and bigger programs
- users wanted!



## 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