Dynamically Discovering Likely Program Invariants to Support Program Evolution

Michael D. Ernst, Jake Cockrell, William G. Griswold, David Notkin

(presented by David Hovemeyer for SysChat, September 13, 2002)

http://pag.lcs.mit.edu/~mernst/pubs/invariants-tse.pdf

http://pag.lcs.mit.edu/daikon/

Outline

- Introduction
- Detecting Invariants Dynamically
  - ▷ Types of Invariants
  - ▷ Instrumentation
  - ▷ Inferring Invariants
- Using Daikon
- Applications
- Links

Introduction

- All programs have invariants
  - ▷ Preconditions, postconditions, loop invariants
  - ▷ Establish correctness conditions
  - ▷ Useful in understanding how program works
  - ▷ Violation of invariant $\equiv$ Bug
- Programmers generally don't write invariants explicitly
- The paper investigates the possibility of discovering invariants dynamically, based on observed program states

Outline

- Introduction
- Detecting Invariants Dynamically
  - ▷ Types of Invariants
  - ▷ Instrumentation
  - ▷ Inferring Invariants
- Using Daikon
- Applications
- Links

Detecting Invariants Dynamically

- Basic idea: instrument program to output values of live variables at selected program points
- Postprocess trace data to infer likely invariants based on observed values
- Automatic tool: "Daikon"
- Incomplete, unsound
  - ▷ In practice, it does find genuine and useful invariants

Types of Invariants

Variables x, y, z, constants a, b, c

- For any variable:
  - ▷ $x = a$
  - ▷ $x =$ uninit
  - ▷ $x \in \{a, b, c\}$
- For single numeric variables:
  - ▷ $x \geq a, x \leq b, a \leq x \leq b$
  - ▷ $x \neq 0$
  - ▷ $x \equiv (a \text{ mod } b)$, $x \not\equiv (a \text{ mod } b)$

Types of Invariants (continued)

- For two numeric variables:
  - ▷ $y = ax + b$
  - ▷ $x < y$, $x \leq y$, $x > y$, $x \geq y$, $x = y$, $x \neq y$
  - ▷ $y = f(x)$ (for various functions f)

- For three numeric variables:
  - ▷ $z = ax + by + c$
  - ▷ $z = g(x, y)$ (for various functions g)

- For single sequence variables:
  - ▷ Range (min and max values)
  - ▷ Ordering (increasing, decreasing, etc.)
  - ▷ Invariants over all elements

Types of Invariants (continued)

- For two sequence variables:
  - ▷ Elementwise linear relationship: $y = ax + b$
  - ▷ Elementwise comparison
  - ▷ Subsequence
  - ▷ Reversal

- For sequence and number variables:
  - ▷ Membership: $i \in s$

Instrumentation

- At program points of interest:
  - ▷ Function entry points
  - ▷ Loop heads
  - ▷ Function exit points

- Output values of all 'interesting' variables
  - ▷ Scalar values (locals, globals, array subscript expressions, etc.)
  - ▷ Arrays of scalar values
  - ▷ Object addresses/ids
  - ▷ More kinds of invariants checked for numeric types

Inferring Invariants

- All invariants can be checked quickly (no theorem proving)
  - ▷ For example: Values for a, b, c in $z = ax + by + c$ can be found once 3 linearly independent samples for x, y, z have been encountered

- Potential invariants are discarded when falsified

- Derived Variables
  - ▷ Synthetic array subscript expressions (not occurring in source)
  - ▷ Sum of array elements
  - ▷ Number of function invocations
  - ▷ Others...

Invariant Confidence

- To make the tool useful, invariants must be supported by statistically significant number of different values

- Daikon checks likelihood that invariant would occur by chance; lower number means increased confidence

- Invariants filtered based on a minimum confidence parameter

Efficiency

- Efficiency of instrumentation
  - ▷ Values of tracked variables are output at each instrumentation point
  - ▷ Significant program slowdown, large amounts of trace data produced

- Efficiency of analysis
  - ▷ Potentially cubic in number of variables at any program point
  - ▷ Influenced more strongly by size of trace data

Outline

- Introduction
- Detecting Invariants Dynamically
  - ▷ Types of Invariants
  - ▷ Instrumentation
  - ▷ Inferring Invariants
- Using Daikon
- Applications
- Links

Using Daikon

- From the paper:
  - ▷ Found a bug in a previously-studied program with a large test suite (array bounds exceeded)
  - ▷ Revealed a condition that the test suite (with thousands of tests) did not exercise
  - ▷ Quality of detected invariants dependent on completeness of test suite

Using Daikon (continued)

- On trivial programs, Daikon can produce gigabytes of trace data, cause slowdowns on the order of 100x, and require hours to infer invariants[1]
- Paper mentions that compute-bound programs typically become I/O-bound when instrumented by Daikon
- I tried it on a simple merge sort program written in Java

[1] Chadd Williams, private communication.

Using Daikon (continued)

Merge sort results

Elements	Run time (orig)	Run time (instr)	Trace file size	Time to check
1,000	1.461 s	3.208 s	1,269,078	17.886 s
10,000	4.050 s	12.747 s	14,951,294	102.559 s
100,000	12.605 s	120.394 s	172,015,722	354.514 s

- Slowdown is not too bad for this program, but trace file size is significant

- Given the ease of producing huge trace files for simple programs, Daikon is not practical for real systems

Using Daikon (continued)

- What can be done to detect invariants more efficiently? Paper suggests:
  - ▷ Adjust granularity of instrumentation
  - ▷ Instrument only 'interesting' parts of program

- Other ideas:
  - ▷ On-line compression of trace data
    - ◦ Gzip reduced trace file for 100,000 element merge sort by factor of 5.69
  - ▷ Decrease sampling frequency (as in Arnold, PLDI 2001)
  - ▷ Dynamically recompile code to remove instrumentation once enough data has been collected (i.e., in JVM)

Outline

- Introduction
- Detecting Invariants Dynamically
  - ▷ Types of Invariants
  - ▷ Instrumentation
  - ▷ Inferring Invariants
- Using Daikon
- Applications
- Links

Applications

- Paper describes use of generated invariants as aid to understanding of an undocumented program
  - ▷ A more recent paper by Nimmer and Ernst uses output of Daikon to feed ESC/Java, a static specification checker based on theorem-proving
- Recent research uses runtime failures of statically or dynamically detected invariants to detect probable bugs (anomolous behavior) [Engler et. al. SOSP 01, Hangal and Lam ICSE 2002]
  - ▷ The paper suggests this as well (including the original conference paper at ICSE 1999)

Outline

- Introduction
- Detecting Invariants Dynamically
  - ▷ Types of Invariants
  - ▷ Instrumentation
  - ▷ Inferring Invariants
- Using Daikon
- Applications
- Links

Links

- Ernst, et. al., *Dynamically Discovering Likely Program Invariants to Support Program Evolution*, http://pag.lcs.mit.edu/~mernst/pubs/invariants-tse.pdf
- Daikon home page: http://pag.lcs.mit.edu/daikon/
- Engler et. at., *Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code*, http://www.stanford.edu/ engler/deviant-sosp-01.pdf
- Hangal and Lam, *Tracking Down Software Bugs Using Automatic Anomaly Detection*, http://suif.stanford.edu/papers/Diduce.pdf