Quickly Detecting Relevant Program Invariants

Michael Ernst, Adam Czeisler,
Bill Griswold (UCSD), and David Notkin
University of Washington

http://www.cs.washington.edu/homes/mernst/daikon

Overview

Goal: improve dynamic invariant detection
[ICSE 99, TSE]

Relevance improvements:

- add desired invariants (2 techniques)
- eliminate undesired ones (3 techniques)

Experiments validate the success

Program invariants

Detect invariants (as in **`assert`**s 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

Uses for invariants

- Write better programs [Gries 81, Liskov 86]
- Document code
- Check assumptions: convert to **`assert`**
- Maintain invariants to avoid introducing bugs
- Locate unusual conditions
- Validate test suite: value coverage
- Provide hints for higher-level profile-directed compilation [Calder 98]
- Bootstrap proofs [Wegbreit 74, Bensalem 96]

Dynamic invariant detection is accurate

Recovered formal specifications, found bugs

Target programs:

- *The Science of Programming* [Gries 81]
- Program checkers [Detlefs 98, Xi 98]
- MIT 6.170 student programs
- *Data Structures and Algorithm Analysis in Java* [Weiss 99]

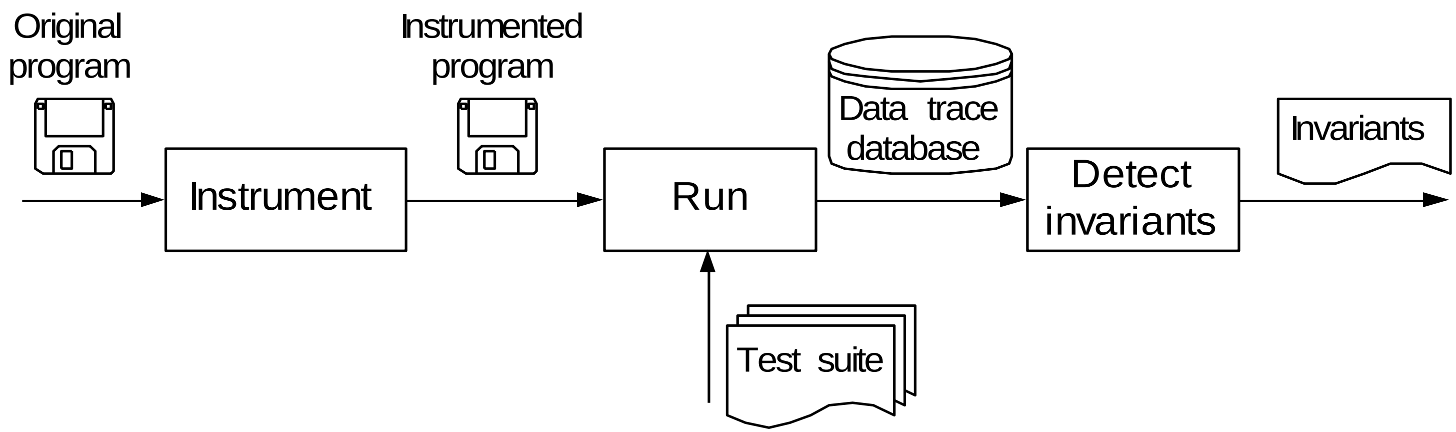
Dynamic invariant detection is useful

563-line C program: regexp search & replace

[Hutchins 94, Rothermel 98]

- Explicated data structures
- Contradicted expectations, preventing bugs
- Revealed bugs
- Showed limited use of procedures
- Improved test suite
- Validated program changes

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

Checking invariants

For each potential invariant:

- instantiate
  (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, each cheap

Relevance

Usefulness to a programmer for a task

Contingent on task and programmer

We manually classified invariants

Perfect output is unnecessary (and impossible)

Improved invariant relevance

Add desired invariants:

1. Implicit values
2. Unused polymorphism

Eliminate undesired invariants (and improve performance):

3. Unjustified properties
4. Redundant invariants
5. Incomparable variables

1. Implicit values

Goal: relationships over non-variables

Examples:

- for array a: length(a), sum(a), min(a), max(a)
- for array a and scalar i: a[i], a[0..i]
- for procedure p: #calls(p)

Derived variables

Successfully produces desired invariants

Adds many new variables

Potential problems:

- slowdown: interleave derivation and inference
- irrelevant invariants: techniques 3–5, later in talk

2. Unused polymorphism

Variables declared with general type, used with more specific type

Example: given a generic list that contains only integers, report that the contents are sorted

Also applicable to subtype polymorphism

Unused polymorphism example

```
class MyInteger { int value; … }
class Link { Object element; Link next; … }
class List { Link header; … }

List myList = new List();
for (int i=0; i<10; i++)
  myList.add(new MyInteger(i));
```

Desired invariant: in class **`List`**,

`header.`closure(**`next`**) is sorted by ≤ over key **`.element.value`**

Polymorphism elimination

Daikon respects declared types

Pass 1: front end outputs object ID, runtime type, and all known fields

Pass 2: given refined type, front end outputs more fields

Sound for deterministic programs

Effective for programs tested so far

3. Unjustified properties

Given three samples for x:

$x = 7$

$x = -42$

$x = 22$

Potential invariants:

$x \neq 0$

$x \leq 22$

$x \geq -42$

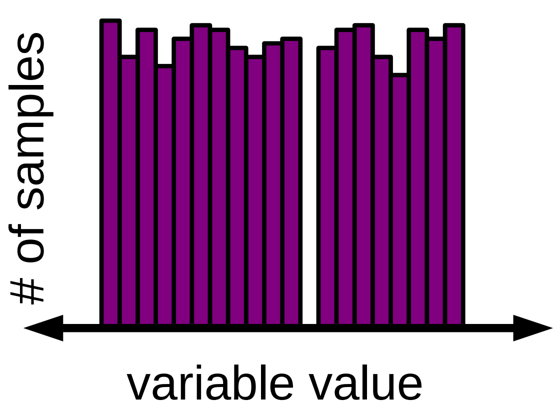
Statistical checks

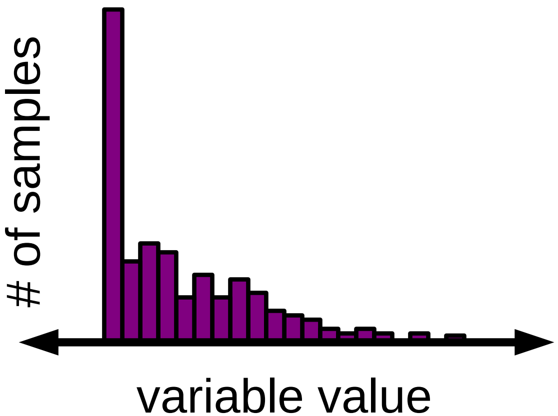
Check hypothesized distribution

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., $x \leq 22$):

- same number of samples as neighbors (uniform)
- more samples than neighbors (clipped)





Duplicate values

Array sum program:

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

b is unchanged inside loop

Problem: at loop head,

$-88 \leq b[n-1] \leq 99$

$-556 \leq \mathrm{sum}(b) \leq 539$

Reason: more samples inside loop

Disregard duplicate values

Idea: count a value if its var was just modified

Front end outputs modification bit per value

- compared techniques for eliminating duplicates

Result: eliminates undesired invariants

4. Redundant invariants

Given:

$0 \le i \le j$

Redundant:

$a[i] \in a[0..j]$

$\max(a[0..i]) \le \max(a[0..j])$

Redundant invariants are logically implied

Implementation contains many such tests

Suppress redundancies

Avoid deriving variables: suppress 25-50%

- equal to another variable
- nonsensical (a[i] when $i < 0$)

Avoid checking invariants:

- false invariants: trivial improvement
- true invariants: suppress 90%

Avoid reporting trivial invariants: suppress 25%

5. Unrelated variables

Problem: the following are of no interest

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

`b < p`

```
int myweight, mybirthyear;
```

`myweight < mybirthyear`

Limit comparisons

Check relations only over comparable variables

- declared program types
- Lackwit [O'Callahan 97]: value flow analysis based on polymorphic type inference

Comparability results

Comparisons:

- declared types: 60% as many comparisons
- Lackwit: 5% as many comparisons; scales well

Runtime: 40-70% improvement

Few differences in reported invariants

Future work

Online inference

Proving invariants

Characterize good test suites

New invariants: temporal, existential

User interface

- control over instrumentation
- display and manipulation of invariants

Further experimental evaluation

- apply to more and bigger programs
- apply to a variety of tasks

Related work

Dynamic inference

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

Static inference

- checking specifications [Detlefs 96, Evans 96, Jacobs 98]
- specification extension [Givan 96, Hendren 92]
- other [Jeffords 98, Henry 90, Ward 96]

Conclusions

Naive implementation is infeasible

Relevance improvements: accuracy, performance

- add desired invariants
- eliminate undesired invariants

Experimental validation

Dynamic invariant detection is promising for research and practice

Questions?

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
  - complementary to static techniques

Unused polymorphism example

```
class MyInteger { int value; … }
class Link { Object element; Link next; … }
class List { Link header; … }

List myList = new List();
for (int i=0; i<10; i++)
  myList.add(new MyInteger(i));
```

Desired invariant: in class **`List`**,

`header.`closure(**`next`**)**`.element.value`**: sorted by $\leq$

Comparison with AI

Dynamic invariant detection:

Can be formulated as an AI problem

Cannot be solved by current AI techniques

- not classification or clustering
- no noise
- no negative examples; many positive examples
- intelligible output

Is implication obvious?

Want:

size(**`topOfStack`**.closure(**`next`**)) =
size(orig(**`topOfStack`**.closure(**`next`**))) + 1

Get:

size(**`topOfStack.next`**.closure(**`next`**)) =
size(**`topOfStack`**.closure(**`next`**)) – 1

`topOfStack.next`.closure(**`next`**) =
orig(**`topOfStack`**.closure(**`next`**))

Solution: interactive UI, queries on variables