Daikon: Dynamic Analysis for Inferring Likely Invariants

Reading: ***Dynamically Discovering Likely Program Invariants to Support Program Evolution***

17-654/17-765

Analysis of Software Artifacts

Jonathan Aldrich

What is an Invariant?

- A logical formula that is always true at a particular set of program points
- Uses
  - Function contracts with pre-/post-conditions
  - Correctness of loops and recursion
  - Correctness of data structures

Invariants and Correctness

void sum(int *b,int n) {

pre: $n \geq 0$

i, s := 0, 0;

inv: $0 \leq i \leq n \wedge s = \sum_{0 \leq j < i} b[j]$

do i ≠ n →

i, s := i+1, s+b[i]

post: $s = \sum_{0 \leq j < n} b[j]$

}

- Correctness of sort
  - Given arguments that satisfy precondition, yields result that satisfies postcondition
- Loop invariant
  - True on entry to loop
  - If loop taken, true after loop body executes
  - After loop exits, we know both the invariant and the exit condition hold
    - e.g., in sort if i=n then inv implies the postcondition: s holds the sum of the complete array

Invariants and Correctness

void sum(int *b,int n) {

pre: $n \geq 0$

i, s := 0, 0;

inv: $0 \leq i \leq n \wedge s = \sum_{0 \leq j < i} b[j]$

do i ≠ n →

i, s := i+1, s+b[i]

post: $s = \sum_{0 \leq j < n} b[j]$

}

- Proof technique
  - Dijkstra: Strongest post-condition
  - Put assertions between every two program statements
  - Step through program, ensuring that assertion + next statement implies next assertion

Invariants and Correctness

```
void sum(int *b,int n) {
    pre: n ≥ 0
    i, s := 0, 0;
    inv: 0 ≤ i ≤ n ∧ s = ∑0≤j<i b[j]
    do i ≠ n →
        i, s := i+1, s+b[i]
    post: s = ∑0≤j<n b[j]
}
```

- **i, s := 0, 0;**
  - assume $n \geq 0$
  - yields $n \geq 0$, $i=0$, $s=0$
  - clearly $0 \leq i \leq n$ and $s = \sum_{0 \leq j < i} b[j]$

Invariants and Correctness

void sum(int *b,int n) {

pre: $n \geq 0$

i, s := 0, 0;

inv: $0 \leq i \leq n \wedge s = \sum_{0 \leq j < i} b[j]$

do i ≠ n →

i, s := i+1, s+b[i]

post: $s = \sum_{0 \leq j < n} b[j]$

}

- **do i ≠ n → …**
  - true branch
    - assume $0 \leq i < n$ and $s = \sum_{0 \leq j < i} b[j]$
    - yields $0 < i \leq n$ and $s = \sum_{0 \leq j < i} b[j]$
    - implies inv again
  - false branch
    - assume $i = n$ and $s = \sum_{0 \leq j < i} b[j]$
    - Implies post

The Challenge

- Invariants are useful, but a pain to write down
- What if analysis could do it for us?
  - Problem: guessing invariants with static analysis is hard
  - Solution: guessing invariants by watching actual program behavior is easy!
    - But of course the guesses might be wrong…

Dynamic Analysis

A technique for inferring properties of a program based on execution traces of that program

- PREfix
  - Can be viewed as dynamic analysis because it simulates execution along some paths
  - Can be viewed as static analysis because the simulation is abstract
- Daikon
  - Infers invariants from program traces

Inferring *i* ≤ *n* in Loop Invariant

void sort(int *b,int n) {

pre: $n \geq 0$

i, s := 0, 0;

inv: $0 \leq i \leq n \wedge s=\sum_{0\leq j<i} b[j]$

do i ≠ n →

i, s := i+1, s+b[i]

post: $s=sum(b[j], 0\leq j<n)$

}

- Possible relationships:

$i<n$ $i\leq n$ $i=n$ $i>n$ $i\geq n$

- Cull relationships with traces

Trace: n=0

n	i

Inferring $i \le n$ in Loop Invariant

void sort(int *b,int n) {

pre: $n \ge 0$

i, s := 0, 0;

inv: $0 \le i \le n \wedge s=\sum_{0\le j<i} b[j]$

do i ≠ n →

i, s := i+1, s+b[i]

post: $s=sum(b[j], 0\le j<n)$

}

- Possible relationships:

$\cancel{i<n}$ $i\le n$ $i=n$ $\cancel{i>n}$ $i\ge n$

- Cull relationships with traces

Trace: n=0

n	i
0	0

Inferring *i* ≤ *n* in Loop Invariant

```
void sort(int *b,int n) {
   pre: n ≥ 0
   i, s := 0, 0;
   inv: 0 ≤ i ≤ n ∧ s=∑0≤j<i b[j]
   do i ≠ n →
        i, s := i+1, s+b[i]
   post: s=sum(b[j], 0≤j<n)
}
```

- Possible relationships:

$\cancel{i<n}$ $\quad i \le n$ $\quad \cancel{i=n}$ $\quad \cancel{i>n}$ $\quad \cancel{i \ge n}$

- Cull relationships with traces

Trace: n=1

n	i
1	0
1	1

Inferring $i \leq n$ in Loop Invariant

```
void sort(int *b,int n) {
    pre: n ≥ 0
    i, s := 0, 0;
    inv: 0 ≤ i ≤ n ∧ s=∑0≤j<i b[j]
    do i ≠ n →
        i, s := i+1, s+b[i]
    post: s=sum(b[j], 0≤j<n)
}
```

- Possible relationships:

$\cancel{i<n}$ $\quad i \leq n$ $\quad \cancel{i=n}$ $\quad \cancel{i>n}$ $\quad \cancel{i \geq n}$

- Cull relationships with traces

Trace: n=2

n	i
2	0
2	1
2	2

Results

- Inferred all invariants in Gries' *The Science of Programming*
- Shocking to research community
  - Many people have applied static analysis to the problem
  - Static analysis is unsuccessful by comparison

Drawbacks

- Requires a reasonable test suite
- Invariants may not be true
  - May only be true for this test suite, but falsified by another program execution
- May detect uninteresting invariants
- May miss some invariants
  - Detects all invariants in a class, but not all interesting invariants are in that class
  - Only reports invariants that are statistically unlikely to be coincidental

- ***Note: easier to reject false or uninteresting invariants than to guess true ones!***

Invariants in SW Evolution

```
void stclose(pat, j, lastj)
char    *pat;
int     *j;
int     lastj;
{
    int jt;
    int jp;
    bool        junk;

    for (jp = *j - 1; jp >= lastj ; jp--)
    {
        jt = jp + CLOSIZE;
        junk = addstr(pat[jp], pat, &jt, MAXPAT);
    }
    *j = *j + CLOSIZE;
    pat[lastj] = CLOSURE;
}
```

- Guess: loop adds chars to pat on all executions of stclose
- Inferred invariant
  - lastj ≤ *j
  - Thus jp=*j-1 could be less than lastj and the loop may not execute!
- Queried for examples where lastj = *j
  - When *j>100
  - pat holds only 100 elements—this is an array bounds error

Invariants in SW Evolution

```
void stclose(pat, j, lastj)
char    *pat;
int     *j;
int     lastj;
{
    int jt;
    int jp;
    bool        junk;

    for (jp = *j - 1; jp >= lastj ; jp--)
    {
        jt = jp + CLOSIZE;
        junk = addstr(pat[jp], pat, &jt, MAXPAT);
    }
    *j = *j + CLOSIZE;
    pat[lastj] = CLOSURE;
}
```

- Task
  - Add + operator to regular expression language
- Goal
  - Don't violate existing program invariants
- Check
  - Inferred invariants for + code same as for * code
  - Except for invariants reflecting different semantics

Benefits Observed

- Invariants describe properties of code that should be maintained
- Invariants contradict expectations of programmer, avoiding errors due to incorrect expectations
- Simple inferred invariants allow programmer to validate more complex ones

Costs

- Scalability
  – Instrumentation slowdown ~10x
    - unoptimized; later on-line work improves this
  – Invariant inference
    - Scales quadratically in # vars, linearly in trace size

Invariant Uses: Test Coverage

- Problem: When generating test cases, how do you know if your test suite is comprehensive enough?
- Generate test cases
- Observe whether inferred invariants change
- Stop when invariants don't change any more
- Captures *semantic coverage* instead of *code coverage*

Harder, Mellen, and Ernst. Improving test suites via operational abstraction. ICSE '03.

Invariant Uses: Test Selection

- Problem: When generating test cases, how do you know which ones might trigger a fault?
- Construct invariants based on “normal” execution
- Generate many random test cases
- Select tests that violate invariants from normal execution

Pacheco and Ernst. Eclat: Automatic generation and classification of test inputs. ECOOP ’05, to appear.

Invariant Uses: Component Upgrades

- You're given a new version of a component—should you trust it in your system?
- Generate invariants characterizing component's behavior in your system
- Generate invariants for new component
  - If they don't match the invariants of old component, you may not want to use it!

McCamant and Ernst. Predicting problems caused by component upgrades. FSE '03.

Invariant Uses: Proofs of Programs

- Problem: theorem-prover tools need help guessing invariants to prove a program correct
- Solution: construct invariants with Daikon, use as lemmas in the proof
- Results [1]
  - Found 4 of 6 necessary invariants
  - But they were the easy ones ☹
- Results [2]
  - Programmers found it easier to remove incorrect invariants than to generate correct ones
  - Suggests that an unsound tool that produces many invariants may be more useful than a sound tool that produces few

[1] Win et al. Using simulated execution in verifying distributed algorithms. Software Tools for Technology Transfer, vol. 6, no. 1, July 2004, pp. 67-76.

[2] Nimmer and Ernst. Invariant inference for static checking: An empirical evaluation. FSE '02.