Daikon: Dynamic Discovery of Likely Invariants

Testing, Quality Assurance, and Maintenance
Winter 2019

Prof. Arie Gurfinkel

based on slides by Prof. Marsha Chechik

UNIVERSITY OF WATERLOO

Discovering Specifications

Help software developers to document the design decisions at an higher level of abstraction than the code

There is one complaint about specifications

- software developers do not like to write them

It is all about cost/benefit ratio

- If there is enough benefit in writing specifications
  - reduced development time, more reliable programs, etc.
- then people will write specifications

Maybe current tools and techniques used in software development do not provide enough benefit for writing specifications

- This may change in the future based on all the techniques and tools we discussed in this course

The Challenge: Discovering Likely Invariants

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…

AKA Machine Learning / Big Data Approach

- guess/generalize based on samples collected during program execution

Dynamically Discovering Likely Program Invariants

Discovered properties are not stated in any part of the program

- They are discovered by monitoring the execution of the program on a set of inputs (a test set)
- The only thing that is guaranteed is that the discovered properties hold for all the inputs in the test set
- No guarantee of soundness or completeness

Dynamic invariant discovery is a re-documentation technique

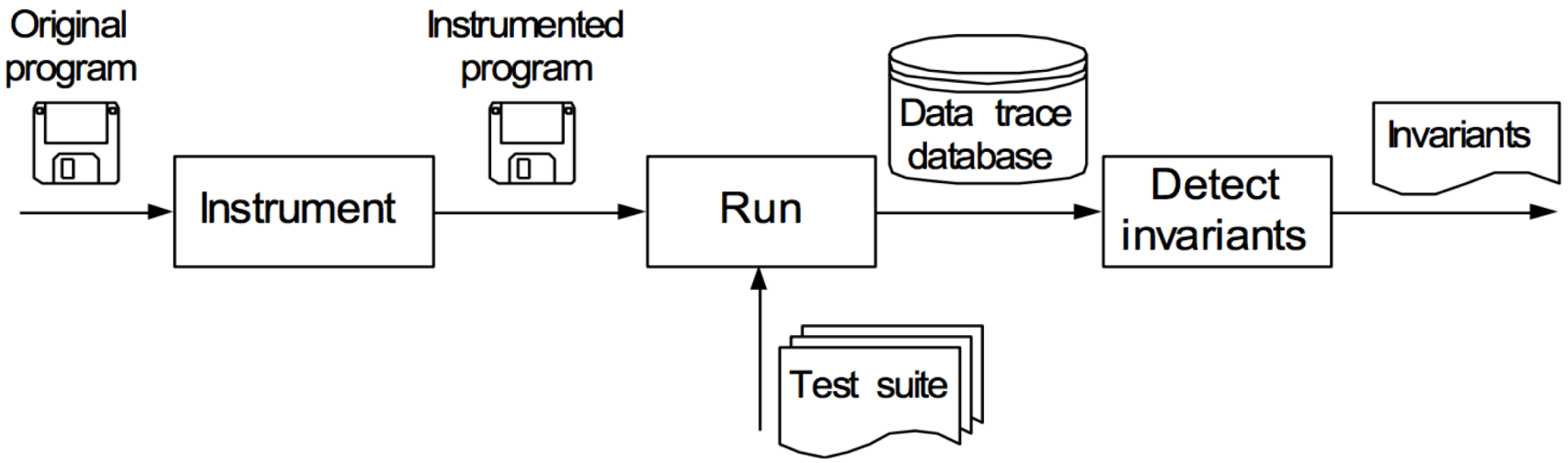
- It does not generate an abstraction of the program (this is design recovery technique)

Document the invariants in this function

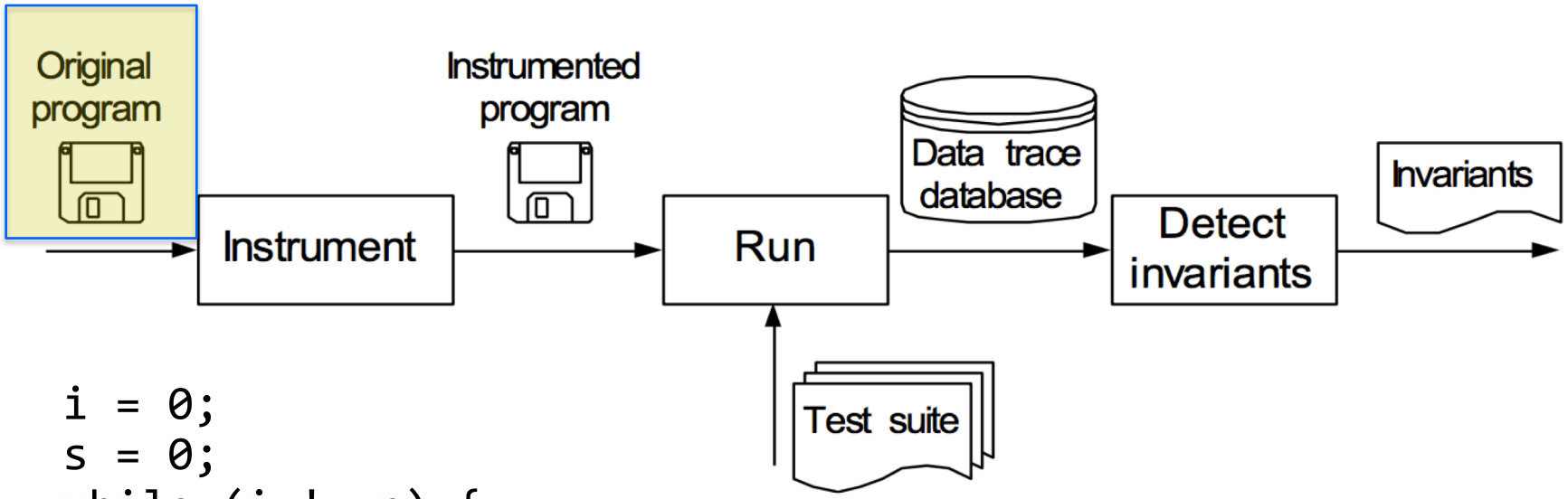
```
// Sum array b of length n into s
int sumArray(b, n) {
  i = 0;
  s = 0;
  while (i != n) {
    s = s + b[i];
    i = i+1;
  }

  return s;
}
```

Dynamic Invariant Detection



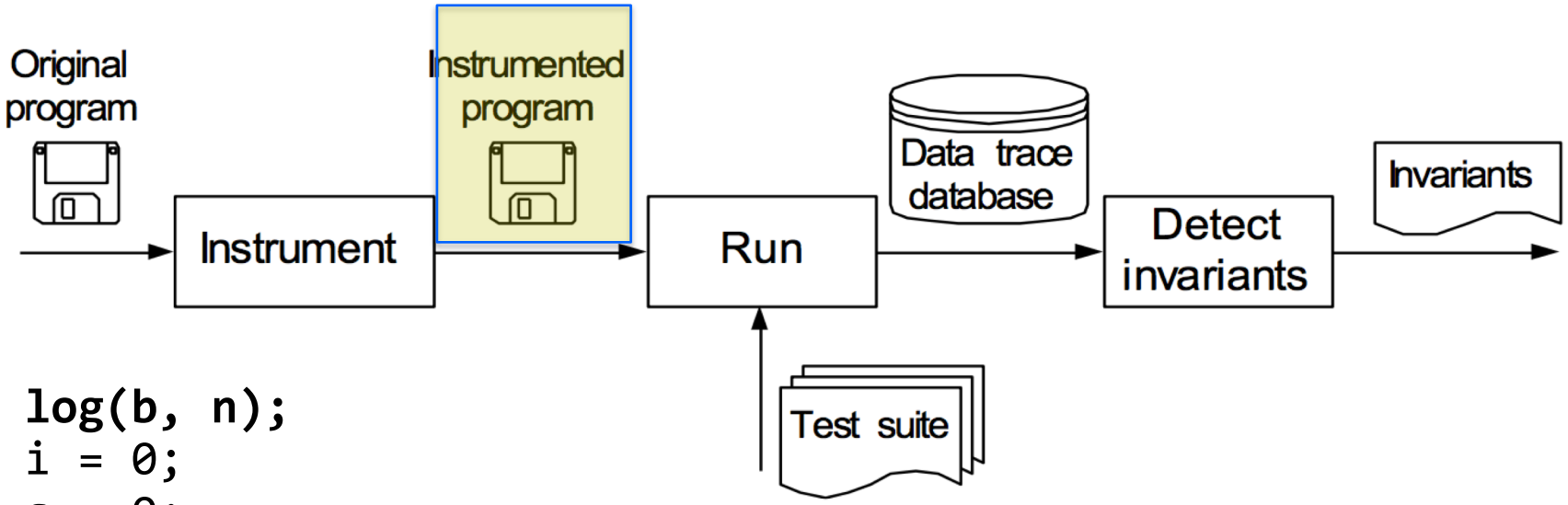
Original Program




```
i = 0;
s = 0;
while (i != n) {
  s = s + b[i];
  i = i+1;
}
```

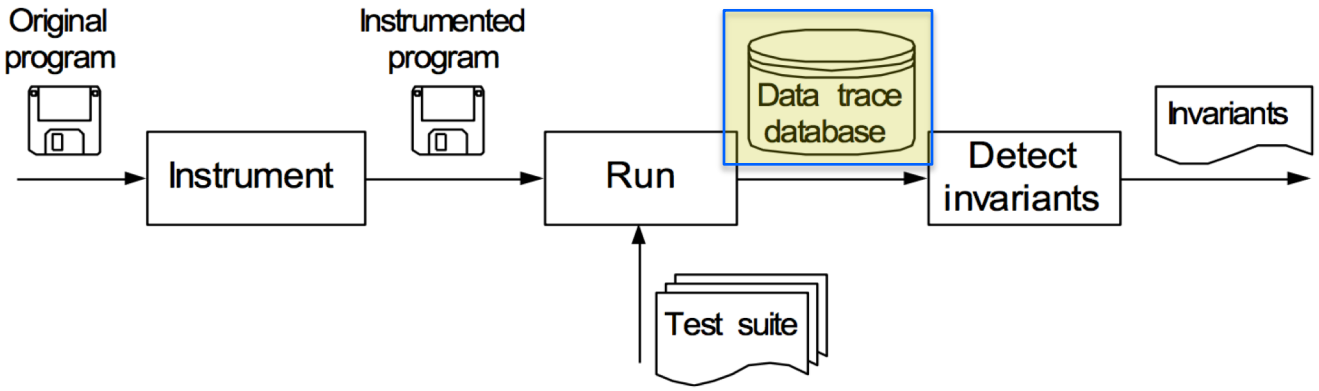
```
// Sum array b of length
// n into variable s
```

Instrumented Program




```
log(b, n);
i = 0;
s = 0;
while (i != n) {
  log(i, s, n, b[i]);
  s = s + b[i];
  i = i+1;
}
```

Trace File



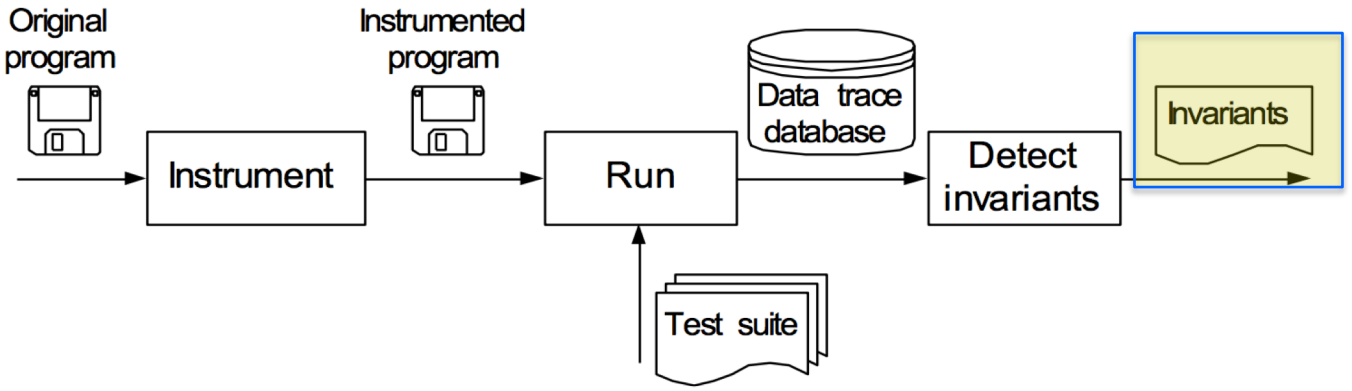

```
log(b, n);
i = 0;
s = 0;
while (i != n) {
  log(i, s, n, b[i]);
  s = s + b[i];
  i = i+1;
}
```

```
15.1.1:::ENTER
B = 92 56 -96 -49 76 92 -3 -88, modified
N = 8, modified

15.1.1:::LOOP
B = 92 56 -96 -49 76 92 -3 -88, modified
N = 8, modified
I = 0, modified
s = 0, modified

15.1.1:::LOOP
B = 92 56 -96 -49 76 92 -3 -88, unmodified
N = 8, unmodified
I = 1, modified
S = 92, modified
```

Dynamic Invariants



Determined Invariants

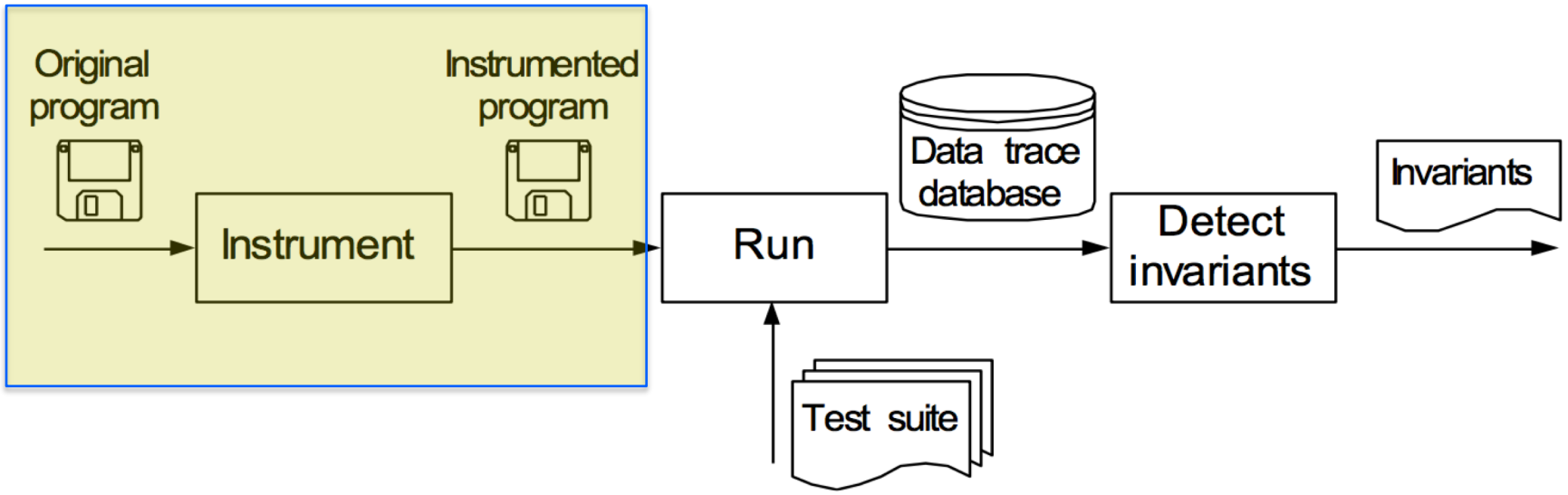
1.) n >= 0
2.) s = SUM(B)
3.) $0 \leq i \leq n$

```
15.1.1:::ENTER
B = 92 56 -96 -49 76 92 -3 -88, modified
N = 8, modified

15.1.1:::LOOP
B = 92 56 -96 -49 76 92 -3 -88, modified
N = 8, modified
I = 0, modified
s = 0, modified

15.1.1:::LOOP
B = 92 56 -96 -49 76 92 -3 -88, unmodified
N = 8, unmodified
I = 1, modified
S = 92, modified
```

Code Instrumentation



Code Instrumentation

Modifies source code to trace specific variables at points of interest:

- Function entry points (pre-conditions)
- Function exit points (post-conditions)
- Loop heads (loop invariants)

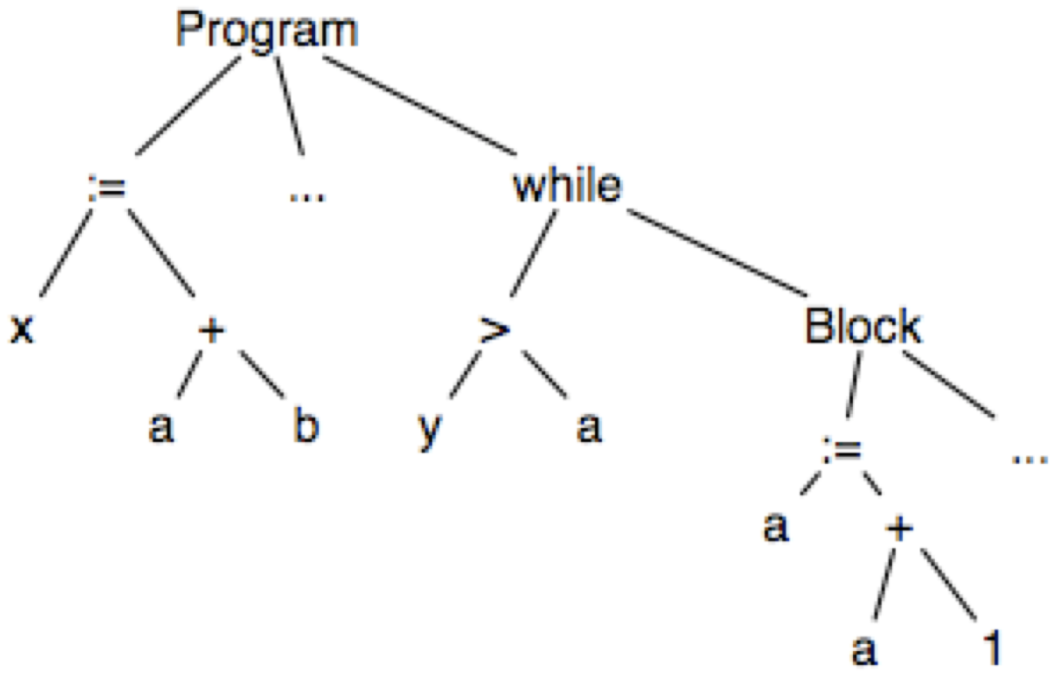
The trace data is used as input to infer invariants

Parsing the code

We need to parse the code to instrument it

Parse to an Abstract Syntax Tree (AST)

```
x := a + b;
y := a * b;
while (y > a) {
    a := a + 1;
    x := a + b
}
```



Traverse the AST

AST is used to easily traverse the code in search of interesting variables to instrument

AST helps to determine which variables are in scope at each point of interest.

- Code is inserted into program point to log the values for all variables in scope to a file in a specific format.

Status variables

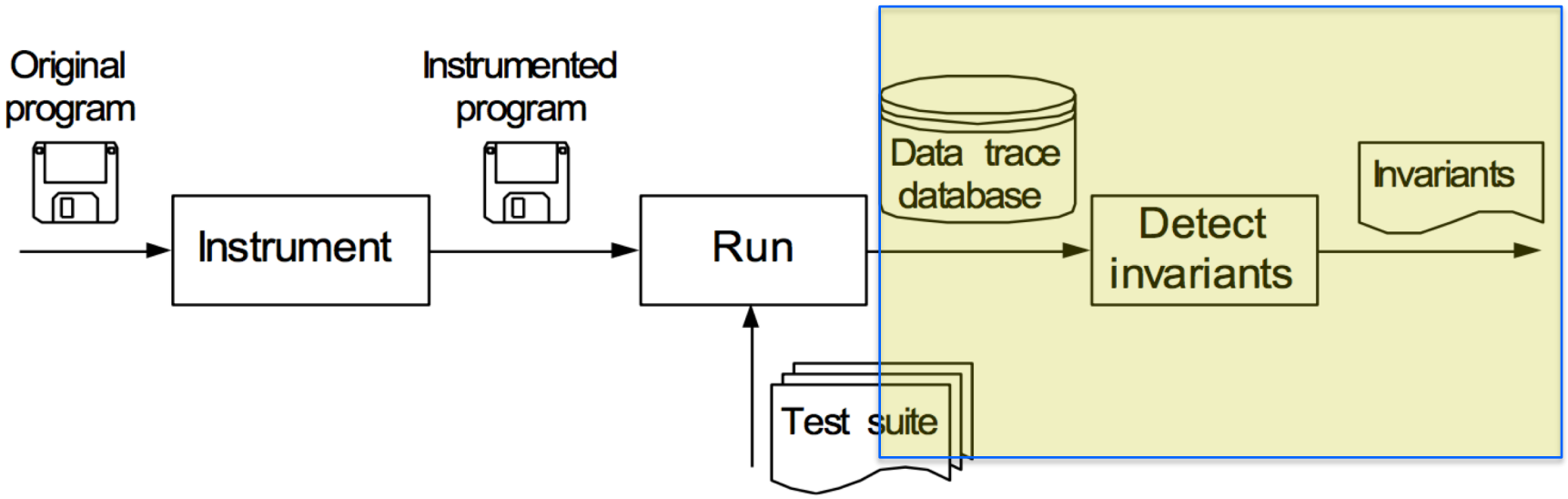
Status variables are created for each original program variable and are passed along throughout function calls.

Status variables:

- Modification timestamp (Used to prevent garbage output)
- Smallest and largest indices (for arrays and pointers)
- Linked list flag

Status variables are updated when a program manipulates its associated variable.

Inferring Invariants



Inferring $i \leq n$ in Loop Invariant

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

Possible relationships:

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

Cull relationships with traces

Trace: n=0

n	i

Inferring $i \leq n$ in Loop Invariant

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

Possible relationships:

$i \leq n$ ~~X~~ $i \geq n$ $i \leq n$ $i = n$ $i \geq n$ ~~X~~

Cull relationships with traces

Trace: n=0

n	i
0	0

Inferring $i \leq n$ in Loop Invariant

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

Possible relationships:

$i<n$ ✗ $i \neq n$ ✗ $i \leq n$ $i=n$ ✗ $i \geq n$ ✗

Cull relationships with traces

Trace: n=1

n	i
1	0
1	1

Inferring $i \leq n$ in Loop Invariant

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

Possible relationships:

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

Cull relationships with traces

Trace: n=2

n	i
2	0
2	1
2	2

Inferring Invariants

There are two issues

- Choosing which invariants to infer
- Inferring the invariants

Daikon infers invariants at specific program points

- procedure entries
- procedure exits
- loop heads (optional)

Daikon can only infer certain types of invariants

- it has a library of invariant patterns
- it can only infer invariants which match to these patterns

Types of Invariants

Single Variables	
Constant Value	x = a
Uninitialized Value	x = uninit
Small Value Set	x in {a,b,c}
Single Numeric Variables	
Range Limits	x >= a, x <= b, etc…
Non-zero	x != 0
Modulus	x = a (mod b)
Non-Modulus	x != a (mod b)

Types of Invariants

Two Numeric Variables	
Linear Relationship	y = ax + b
Functional Relationship	y = f(x)
Comparison	x > y, x = y, etc…
Combinations of Single Numeric Values	x+y = a (mod b)
Three Numeric Variables	
Polynomial Relationship	z = ax + by + c

How Are Invariants Inferred?

A sample is a set of values for the traced variables at a program point, stemming from one execution of that program point.

An invariant is reverse engineered from the sample in turn.

As soon as a sample not satisfying the invariant is encountered, the invariant is known not to hold and is not checked for any subsequent samples (discarded).

Inferring Invariants

For each invariant pattern

- determine the constants in the pattern
- stop checking the invariants that are falsified

For example,

- For invariant pattern **x** ≥ **a**, we have to determine the constant **a**
- For invariant pattern **x = ay + bz + c**, we have to determine the constants **a, b, c**

Inferring Invariants (1/3)

```
int inc(int *x, int y) {
   *x += y;
   return *x;
}
```

What invariants should be inferred from this method, regardless of the test suite input?

Inferring Invariants (2/3)

As a simple example, consider the C code

```
int inc(int *x, int y) {
  *x += y;
  return *x;
}
```

At the procedure exit, value tuples might include (the first line is shown for reference):

orig(x),	orig(*x),	orig(y),	x,	*x,	y,	return
4026527180,	2,	1,	4026527180,	3,	1,	3
146204,	13,	1,	146204,	14,	1,	14
4026527180,	3,	1,	4026527180,	4,	1,	4
4026527180,	4,	1,	4026527180,	5,	1,	5
146204,	14,	1,	146204,	15,	1,	15
4026527180,	5,	1,	4026527180,	6,	1,	6
4026527180,	6,	1,	4026527180,	7,	1,	7

⋮

Inferring Invariants (3/3)

orig(x),	orig(*x),	orig(y),	x,	*x,	y,	return
4026527180,	2,	1,	4026527180,	3,	1,	3
146204,	13,	1,	146204,	14,	1,	14
4026527180,	3,	1,	4026527180,	4,	1,	4
4026527180,	4,	1,	4026527180,	5,	1,	5
146204,	14,	1,	146204,	15,	1,	15
4026527180,	5,	1,	4026527180,	6,	1,	6
4026527180,	6,	1,	4026527180,	7,	1,	7
		⋮				

We can infer from this trace that for all samples, `*x = orig(*x) + 1`.

Is this invariant too limited?

Falsification Example

Consider the invariant pattern: **X = aY + bZ + c**

Consider sample trace for variables (X, Y, Z)

(0,1,-7), (10,2,1), (10,1,3), (0, 0,-5), (3, 1, -4), (7, 1, 1), ...

Does the Invariant: X = 2Y + Z + 5 hold?

Based on the first 3 values for X, Y, Z in the trace, we can infer the constants a, b, and c:

0 = a -7b +c, 10 = 2a + b +c, 10 = a +3b + c

If we solve these equations for a, b, c, we get: **a=2, b=1, c=5,**

Invariant: X = 2Y + Z + 5

The next two tuples (0, 0,-5), (3, 1, -4) confirm the invariant

However the last trace value (7, 1, 1) kills this invariant

- Hence, it is not checked for the remaining trace values and it is not reported as an invariant

Inferring Invariants

Determining the constants for invariants is not too expensive

- For example, three linearly independent data values are sufficient for figuring out the constants in the pattern x = ay + bz + c
- there are at most three constants in each invariant pattern

Once the constants for the invariants are determined, the check that an invariant holds for each data value is not expensive

- Just evaluate the expression and check for equality

Derived Variables

Looking for invariants only on variables declared in the program may not be sufficient to detect all interesting invariants

Daikon adds certain derived variables (which are actually expressions) and also detects invariants on these derived variables

Derived Variables

Derived from any sequence s:

- Length: size(s)
- Extremal elements: s[0], s[1], s[size(s)-1], s[size(s)-2]

Derived from any numeric sequence s:

- Sum: sum(s)
- Minimum element: min(s)
- Maximum element: max(s)

Derived Variables

Derived from any sequence s and any numeric variable i

- Element at the index: s[i], s[i-1]
- Subsequences: s[0..i], s[0..i-1]

Derived from function invocations:

- Number of calls so far

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

Drawbacks of Dynamic Invariant Detection

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

- Some may actually tell you about the test suite, not the program (still useful)

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!

Limitations

Accuracy of inferred invariants depends on quality and completeness of traces (test cases)

- Additional test cases could provide data that will lead to additional invariants to be inferred.
- Additionally, invariants may only hold true for cases found in test suite

Gigabytes of trace data, even while analyzing trivial programs.

Unstable invariants: fail on the same version of the application

Invariant Confidence

Not all unfalsified invariants should be reported

There may be a lot of irrelevant invariants which may just reflect properties of the test set

The output may become unreadable if a lot of irrelevant invariants are reported

Improving (increasing) the test set would reduce the number of irrelevant invariants

Invariant Confidence

For each detected invariant, Daikon computes the probability that such a property would appear by chance in a random input

- If that probability is smaller than a user-specified confidence parameter, then the property is reported

Daikon assumes a distribution and performs a statistical test

- It checks the probability that the observed values for the detected invariant were generated by chance from the distribution
- If that probability is very low, then the invariant is reported

Invariant Confidence

Given: integer variable x which takes values between r/2 and –(r/2)-1

Assume that $x \neq 0$ for all test cases

If the values of **x** are uniformly distributed between r/2 and –(r/2)-1, then the probability that **x** is not 0 is 1 – 1/r

Given **s** samples, the probability that **x** is never 0 is $(1-1/r)^s$

If this probability is less than a user defined confidence level then $x \neq 0$ is reported as an invariant

Cost of Inferring Invariants

The cost of inferring invariants increases as follows:

- **linear** in the number of samples or values (test set size)
- **linear** in the the number of program points
- **quadratic** in the number of variables at a program point

Typically a few minutes per procedure.

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

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.

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.

Tool: Daikon

Daikon is an implementation of dynamic detection of likely invariants, by Ernst et al.

Download: http://plse.cs.washington.edu/daikon/

Support for: C, Java, Lisp, and JavaScript

Dynamically Detecting Invariants: Summary

Useful reengineering tool

- Re-documentation

Can be used as a coverage criterion during testing

- Are all the values in a range covered by the test set?

Can be helpful in detecting bugs

- Found bugs in an existing program in a case study

Can be useful in maintaining invariants

- Prevents introducing bugs, programmers are less likely to break existing invariants

Can be VERY useful for program verification