Detecting Anomalies

Andreas Zeller

1

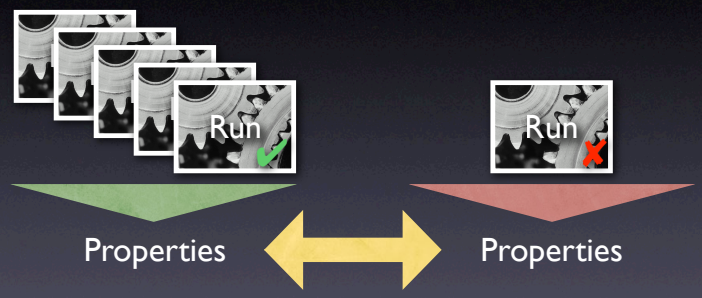
What's abnormal?

- Suppose we determine common properties of all *passing* runs.
- Now we examine a run which *fails* the test.
- Any difference in properties *correlates with failure* – and is likely to hint at failure causes

2

2

Detecting Anomalies



Differences correlate with failure

3

3

Properties

Data properties that hold in all runs:

- "At f(), x is odd"
- "$0 \leq x \leq 10$ during the run"

Code properties that hold in all runs:

- "f() is always executed"
- "After open(), we eventually have close()"

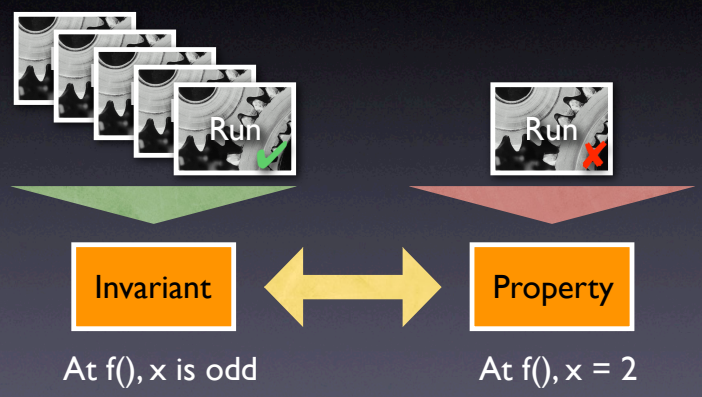
Techniques

Dynamic Invariants	Value Ranges	Sampled Values

Techniques

Dynamic Invariants	Value Ranges	Sampled Values

Dynamic Invariants



Daikon

- Determines *invariants* from program runs
- Written by Michael Ernst et al. (1998–)
- C++, Java, Lisp, and other languages
- analyzed up to 13,000 lines of code

Daikon

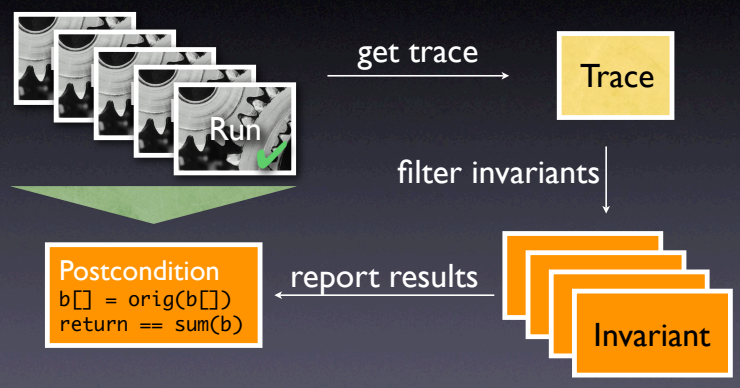
```
public int ex1511(int[] b, int n)
{
    int s = 0;
    int i = 0;
    while (i != n) {
        s = s + b[i];
        i = i + 1;
    }
    return s;
}
```

Precondition
```
n == size(b[])
b != null
n <= 13
n >= 7
```

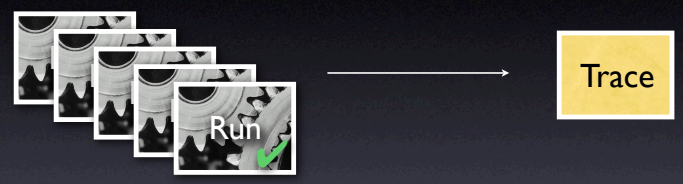
Postcondition
```
b[] = orig(b[])
return == sum(b)
```

- Run with 100 randomly generated arrays of length 7–13

Daikon



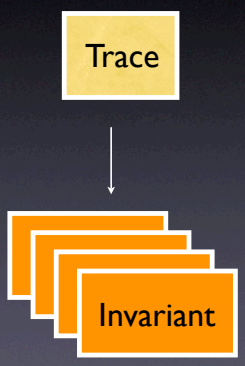
Getting the Trace



- Records all variable values at all function entries and exits
- Uses VALGRIND to create the trace

Filtering Invariants

- Daikon has a library of *invariant patterns* over variables and constants
- Only matching patterns are preserved



Method Specifications

using *primitive data*

$x = 6$	$x \in \{2, 5, -30\}$	$x < y$
$y = 5x + 10$	$z = 4x + 12y + 3$	$z = fn(x, y)$

using *composite data*

A subseq B	$x \in A$	sorted(A)

checked at method entry + exit

13

13

Object Invariants

string.content[string.length] = '\0'
node.left.value ≤ node.right.value
this.next.last = this

checked at entry + exit of public methods

14

14

Matching Invariants

```
public int ex1511(int[] b, int n)
{
    int s = 0;
    int i = 0;
    while (i != n) {
        s = s + b[i];
        i = i + 1;
    }
    return s;
}
```

A == B

Pattern

s size(b[])
sum(b[]) n
orig(n)
return …

Variables

15

15

Matching Invariants

==	s	n	size(b[])	sum(b[])	orig(n)	ret
s		✘			✘	
n	✘			✘		✘
size(b[])						
sum(b[])		✘				
orig(n)	✘					✘
ret		✘			✘	

run 1

A == B

Pattern

s size(b[]) sum(b[]) n orig(n) return …

Variables

16

16

Matching Invariants

==	s	n	size(b[])	sum(b[])	orig(n)	ret
s		✘	✘		✘	
n	✘			✘	✘	✘
size(b[])	✘			✘		✘
sum(b[])		✘	✘		✘	
orig(n)	✘	✘		✘		✘
ret		✘	✘		✘	

run 2

A == B

Pattern

s size(b[]) sum(b[]) n orig(n) return …

Variables

17

17

Matching Invariants

==	s	n	size(b[])	sum(b[])	orig(n)	ret
s		✘	✘		✘	
n	✘			✘	✘	✘
size(b[])	✘			✘		✘
sum(b[])		✘	✘		✘	
orig(n)	✘	✘		✘		✘
ret		✘	✘		✘	

run 3

A == B

Pattern

s size(b[]) sum(b[]) n orig(n) return …

Variables

18

18

Matching Invariants

==	s	n	size(b[])	sum(b[])	orig(n)	ret
s		✘	✘		✘	
n	✘			✘	✘	✘
size(b[])	✘			✘		✘
sum(b[])		✘	✘		✘	
orig(n)	✘	✘		✘		✘
ret		✘	✘		✘	

s == sum(b[])

s == ret

n == size(b[])

ret == sum(b[])

Matching Invariants

```
public int ex1511(int[] b, int n)
{
    int s = 0;
    int i = 0;
    while (i != n) {
        s = s + b[i];
        i = i + 1;
    }
    return s;
}
```

s == sum(b[])

s == ret

n == size(b[])

ret == sum(b[])

Enhancing Relevance

- Handle polymorphic variables
- Check for derived values
- Eliminate redundant invariants
- Set statistical threshold for relevance
- Verify correctness with static analysis

polymorphic variables: treat “object x” like “int x” if possible
derived values: have “size(…)” as extra value to compare against
redundant invariants: like x > 0 => x >= 0
statistical threshold: to eliminate random occurrences
verify correctness: to make sure invariants always hold

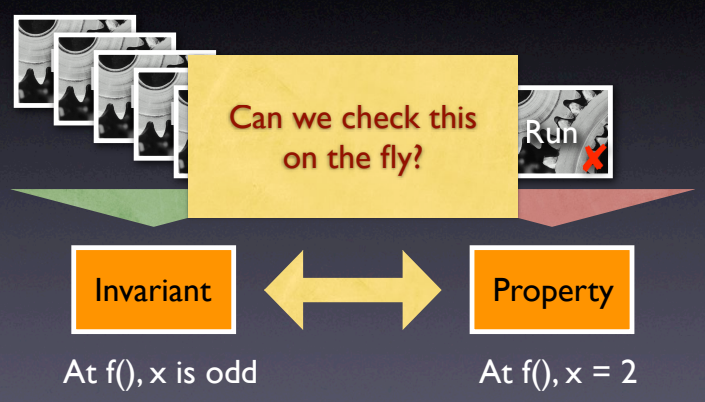
Daikon Discussed

- As long as some property can be observed, it can be added as a pattern
- Pattern vocabulary determines the invariants that can be found ("sum()", etc.)
- Checking all patterns (and combinations!) is expensive
- Trivial invariants must be eliminated

Techniques

Dynamic Invariants	Value Ranges	Sampled Values

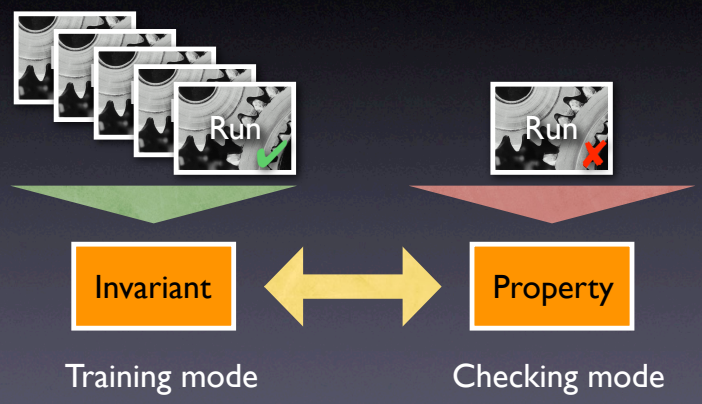
Dynamic Invariants



Diduce

- Determines *invariants* and *violations*
- Written by Sudheendra Hangal and Monica Lam (2001)
- Java bytecode
- analyzed > 30,000 lines of code

Diduce



Training Mode

- Start with empty set of invariants
- Adjust invariants according to values found during run

Invariants in Diduce

For each variable, Diduce has a pair (V, M)

- V = *initial value* of variable
- M = *range of values:* i-th bit of M is cleared if value change in i-th bit was observed
- With each assignment of a new value W, M is updated to $M := M \wedge \neg (W \otimes V)$
- *Differences* are stored in same format

Training Example

Code	i	Values		Differences		Invariant
		V	M	V	M	
i = 10	1010	1010	1111	–	–	i = 10
i += 1	1011	1010	1110	1	1111	$10 \le i \le 11 \wedge \lvert i' - i \rvert = 1$
i += 1	1100	1010	1000	1	1111	$8 \le i \le 15 \wedge \lvert i' - i \rvert = 1$
i += 1	1101	1010	1000	1	1111	$8 \le i \le 15 \wedge \lvert i' - i \rvert = 1$
i += 2	1111	1010	1000	1	1101	$8 \le i \le 15 \wedge \lvert i' - i \rvert \le 2$

During *checking*, clearing an M-bit is an anomaly

In Code, i = 10 is decimal 10; i, V, M are binary values.

Diduce vs. Daikon

- Less space and time requirements
- Invariants are computed on the fly
- Smaller set of invariants
- Less precise invariants

Techniques

Dynamic Invariants	Value Ranges	Sampled Values

Detecting Anomalies

Differences correlate with failure

Liblit's Sampling

- We want properties of runs in the field
- Collecting all this data is too expensive
- Would a *sample* suffice?
- Sampling experiment by Liblit et al. (2003)

Return Values

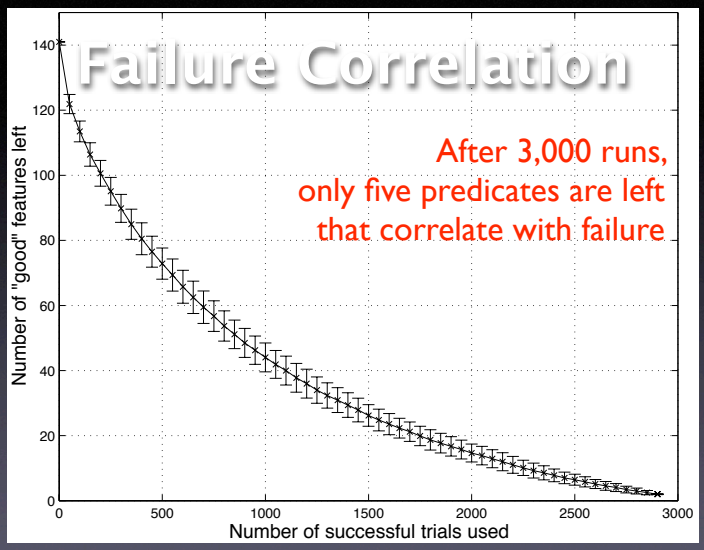
- Hypothesis: *function return values* correlate with failure or success
- Classified into positive / zero / negative

CCRYPT fails

- CCRYPT is an interactive encryption tool
- When CCRYPT asks user for information before overwriting a file, and user responds with EOF, CCRYPT crashes
- 3,000 random runs
- Of 1,170 predicates, only file_exists() > 0 and xreadline() == 0 correlate with failure

Liblit's Sampling

- Can we apply this technique to remote runs, too?
- 1 out of 1000 return values was sampled
- Performance loss <4%



37

37

Web Services

- Sampling is first choice for web services
- Have 1 out of 100 users run an instrumented version of the web service
- Correlate instrumentation data with failure
- After sufficient number of runs, we can automatically identify the anomaly

38

38

Techniques

Dynamic Invariants	Value Ranges	Sampled Values

39

39

Anomalies and Causes

- An anomaly is not a cause, but a correlation
- Although correlation ≠ causation, anomalies can be excellent hints
- Future belongs to those who exploit
  - Correlations in *multiple runs*
  - Causation in *experiments*

Concepts

- ★ Comparing data abstractions shows anomalies correlated with failure
- ★ Variety of abstractions and implementations
- ★ Anomalies can be excellent hints
- ★ Future: Integration of anomalies + causes