Seeding Bugs to Find Bugs

Andreas Zeller
Saarland University

with David Schuler and Valentin Dallmeier

1

<title>
Seeding Bugs to Find Bugs: Mutation Testing Revisited
</title>

<abstract>
How do you know your test suite is "good enough"? One of the best ways to tell is _mutation testing_. Mutation testing seeds artificial defects (mutations) into a program and checks whether your test suite finds them. If it does not, this means your test suite is not adequate yet.

Despite its effectiveness, mutation testing has two issues. First, it requires large computing resources to re-run the

Saarbrücken

2

Hans-Peter is moving into this building – actually, he built it, too. He's worried that everything might be okay. But he's not that worried.

3

If you're building not a building, but a piece of software, you have many more reasons to be worried.

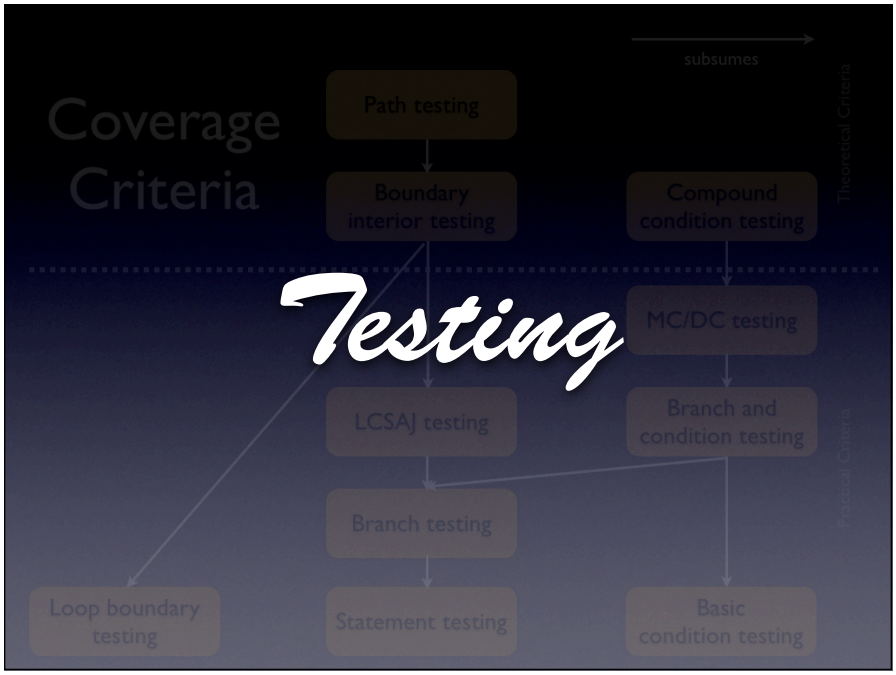
4

It's not like this is the ultimate horror…

5

…but still, this question causes fear, uncertainty and doubt in managers

6

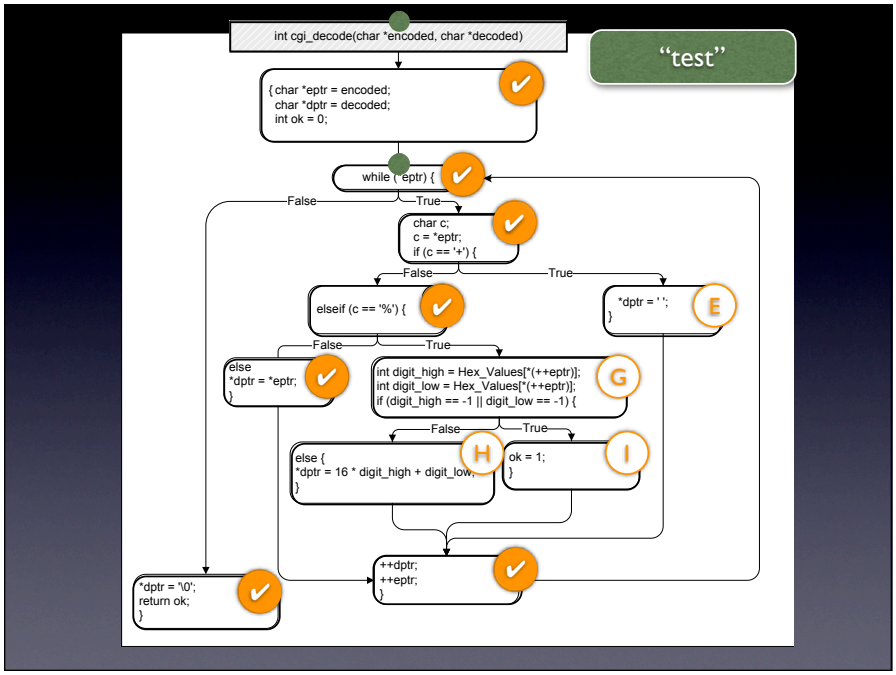


7

Test Quality

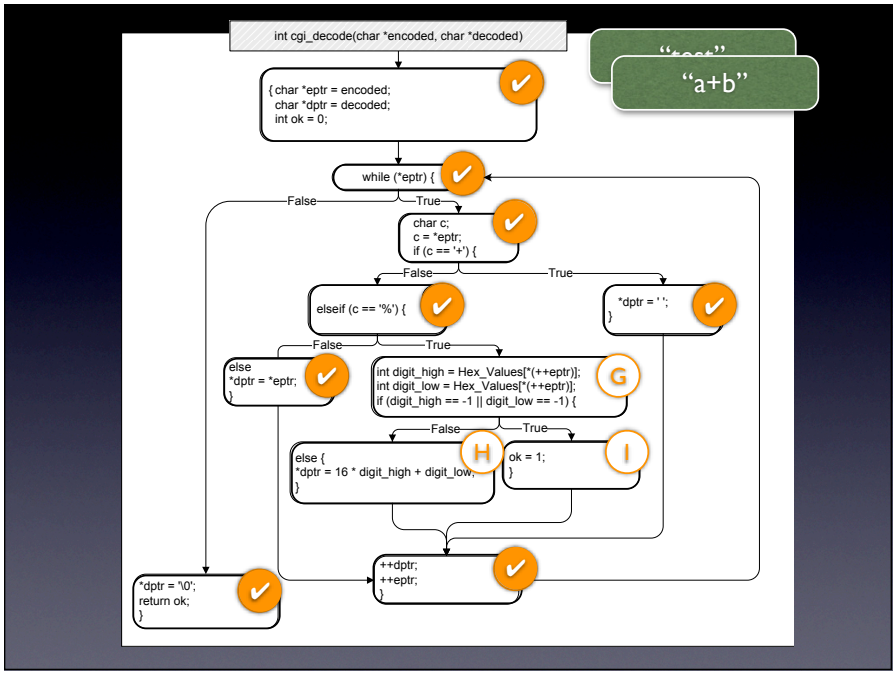
- You want your program to be well tested
- How do we measure "well tested"?

8



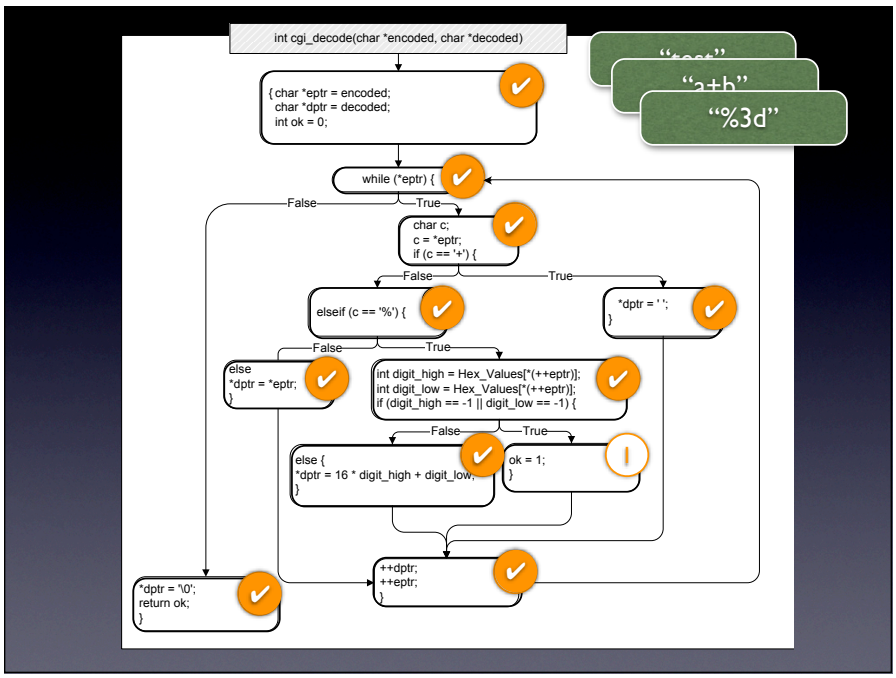
While the Programm is executed, one statement (or basic block) after the other is covered – i.e., executed at least once – but not all of them. Here, the input is "test"; checkmarks indicate executed blocks.

9



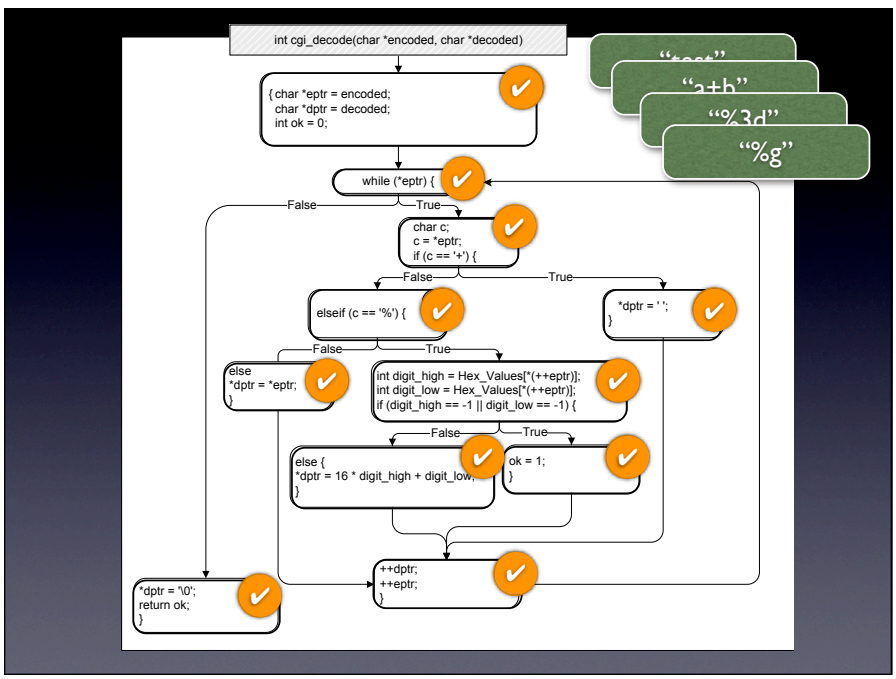
We'd like to test every statement, so we come up mit more test cases.

10



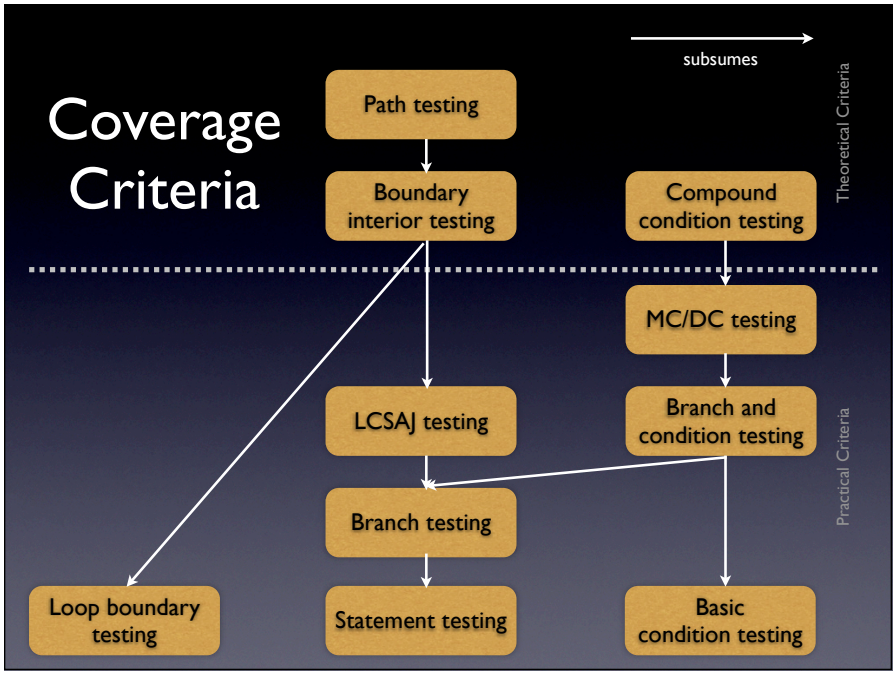
We'd like to test every statement, so we come up mit more test cases.

11



This is an interesting boundary test case, as it may cause non-deterministic behavior. Can you see why?

12



And this is the summary of structural testing techniques.

13

Weyuker's Hypothesis

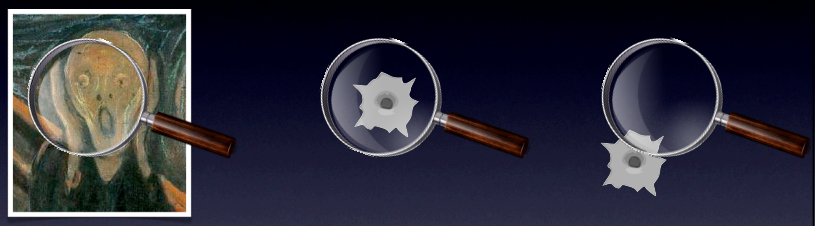
The adequacy of a coverage criterion can only be intuitively defined.

Established by a number of studies done by E. Weyuker at AT&T. "Any explicit relationship between coverage and error detection would mean that we have a fixed distribution of errors over all statements and paths, which is clearly not the case".

14

Mutation Testing

DeMillo, Lipton, Sayward 1978



Program

15

A Mutation

```
class Checker {
    public int compareTo(Object other)
    {
        return 1;
    }
}
```

not found by AspectJ test suite

16

Mutation Operators

id	operator	description	constraint
Operand Modifications			
crp	constant for constant replacement	replace constant $C1$ with constant $C2$	$C1 \neq C2$
scr	scalar for constant replacement	replace constant C with scalar variable X	$C \neq X$
acr	array for constant replacement	replace constant C with array reference $A[I]$	$C \neq A[I]$
scr	struct for constant replacement	replace constant C with struct field S	$C \neq S$
svr	scalar variable replacement	replace scalar variable X with a scalar variable Y	$X \neq Y$
csr	constant for scalar variable replacement	replace scalar variable X with a constant C	$X \neq C$
asr	array for scalar variable replacement	replace scalar variable X with an array reference $A[I]$	$X \neq A[I]$
ssr	struct for scalar replacement	replace scalar variable X with struct field S	$X \neq S$
vie	scalar variable initialization elimination	remove initialization of a scalar variable	
car	constant for array replacement	replace array reference $A[I]$ with constant C	$A[I] \neq C$
sar	scalar for array replacement	replace array reference $A[I]$ with scalar variable X	$A[I] \neq X$
cnr	comparable array replacement	replace array reference with a comparable array reference	
sar	struct for array reference replacement	replace array reference $A[I]$ with a struct field S	$A[I] \neq S$
Expression Modifications			
abs	absolute value insertion	replace e by `abs(`e`)`	$e < 0$
aor	arithmetic operator replacement	replace arithmetic operator ψ with arithmetic operator ϕ	$e_1 \psi e_2 \neq e_1 \phi e_2$
lcr	logical connector replacement	replace logical connector ψ with logical connector ϕ	$e_1 \psi e_2 \neq e_1 \phi e_2$
ror	relational operator replacement	replace relational operator ψ with relational operator ϕ	$e_1 \psi e_2 \neq e_1 \phi e_2$
uoi	unary operator insertion	insert unary operator	
cpr	constant for predicate replacement	replace predicate with a constant value	
Statement Modifications			
sdl	statement deletion	delete a statement	
sca	switch case replacement	replace the label of one case with another	
ses	end block shift	move } one statement earlier and later	

Pezzé and Young, Software Testing and Analysis

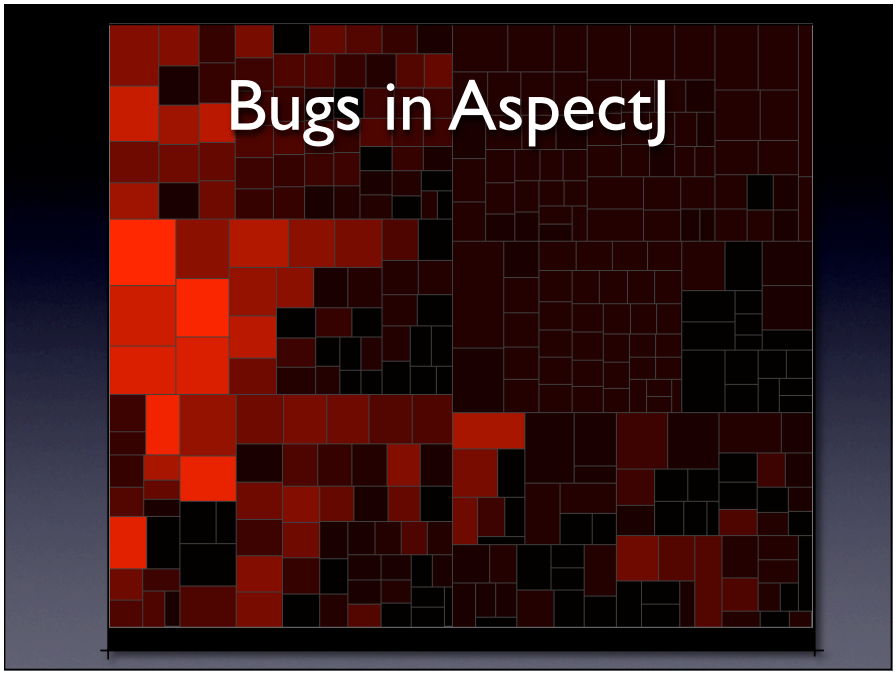
17

from Pezze + Young, “Software Testing and Analysis”, Chapter 16
If one ever needed a proof that testing is a destructive process – here it is

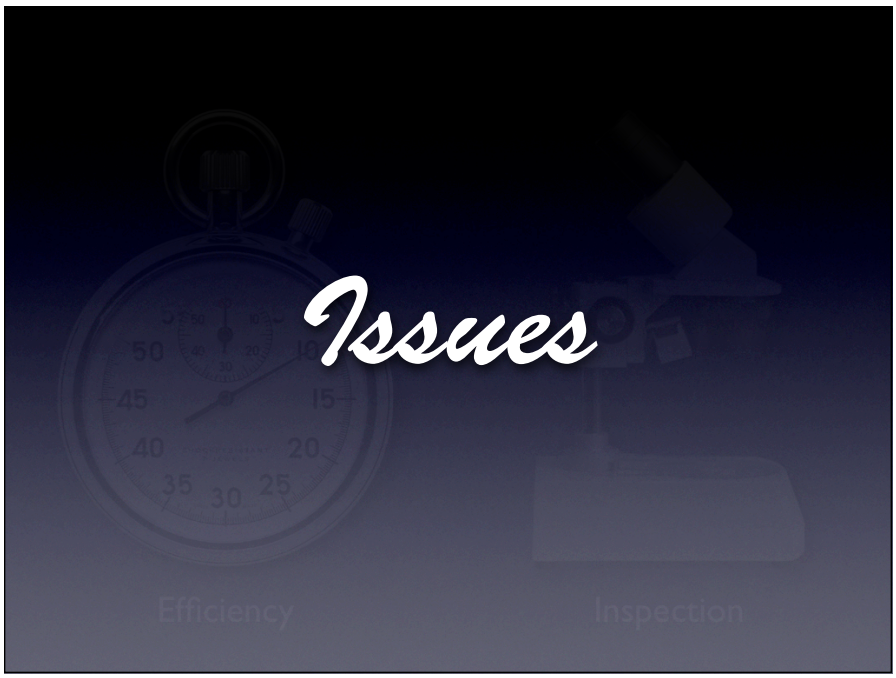
Does it work?

- Generated mutants are similar to real faults
  Andrews, Briand, Labiche, ICSE 2005
- Mutation testing is more powerful than statement or branch coverage
  Walsh, PhD thesis, State University of NY at Binghampton, 1985
- Mutation testing is superior to data flow coverage criteria
  Frankl, Weiss, Hu, Journal of Systems and Software, 1997

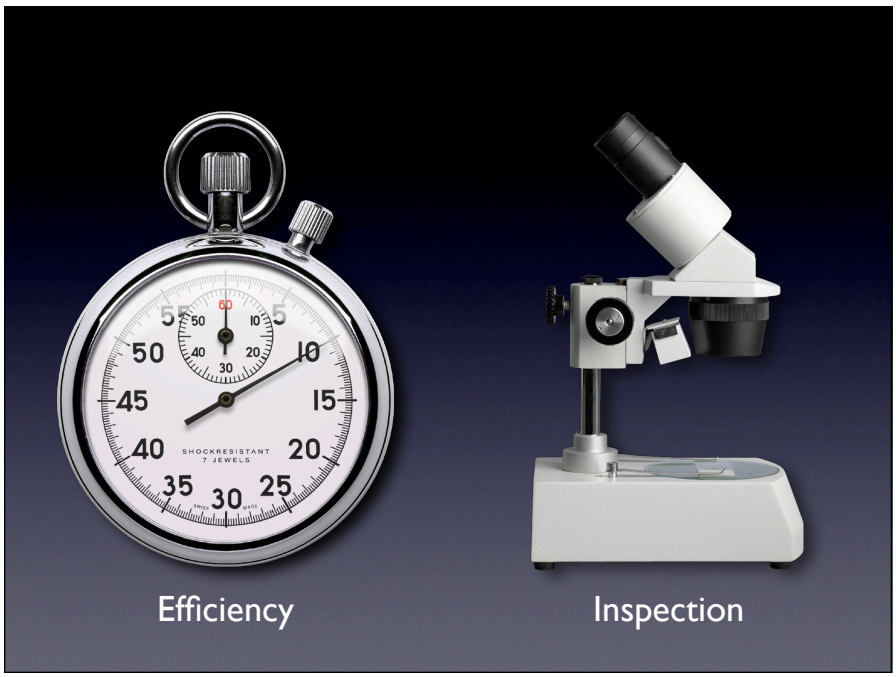
18


Bugs in AspectJ

19


Issues
Efficiency
Inspection

20


Efficiency
Inspection

21

Efficiency

- Test suite must be re-run for every single mutation
- Expensive

Efficiency

How do we make mutation testing efficient?

Efficiency

- Manipulate byte code directly
  rather than recompiling every single mutant
- Focus on few mutation operators
  - replace numerical constant *C* by *C*±1, or 0
  - negate branch condition
  - replace arithmetic operator (+ by –, * by /, etc.)
- Use mutant schemata
  individual mutants are guarded by run-time conditions
- Use coverage data
  only run those tests that actually execute mutated code

A Mutation

```
class Checker {
    public int compareTo(Object other)
    {
        return 1;
    }
}
```

not found by AspectJ test suite
because it is not executed

25

– and since we know it's not executed, we don't even apply this mutation.

Efficiency

Inspection

26

Inspection

- A mutation may leave program semantics unchanged
- These *equivalent mutants* must be determined manually
- This task is tedious.

27

An Equivalent Mutant

```
public int compareTo(Object other) {
  if (!(other instanceof BcelAdvice))
    return 0;
  BcelAdvice o = (BcelAdvice)other;

  if (kind.getPrecedence() != o.kind.getPrecedence()) {
    if (kind.getPrecedence() > o.kind.getPrecedence())
      return +2;
    else
      return -1;
  }
  // More comparisons...
}
```

no impact on AspectJ

To check this, we need to look at 50+ places!

Frankl's Observation

We also observed that […] mutation testing was *costly*. Even for these small subject programs, the human effort needed to check a large number of mutants for equivalence was *almost prohibitive*.

P. G. Frankl, S. N. Weiss, and C. Hu.
All-uses versus mutation testing:
An experimental comparison of effectiveness.
Journal of Systems and Software, 38:235–253, 1997.

Inspection

How do we determine equivalent mutants?

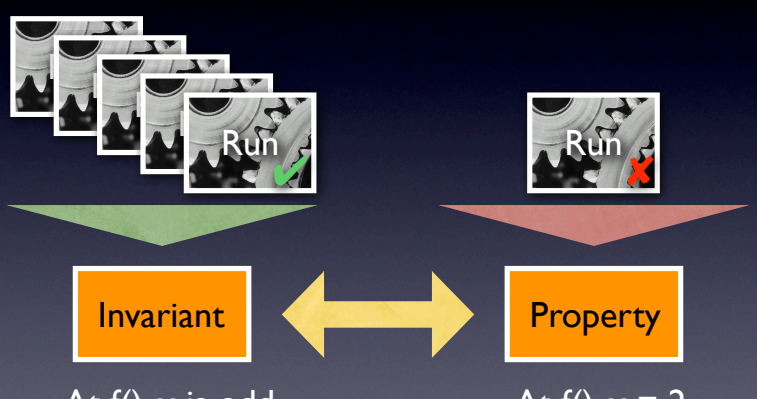
Aiming for Impact

Measuring Impact

- How do we characterize "impact" on program execution?
- Idea: Look for changes in *pre- and postconditions*
- Use *dynamic invariants* to learn these

Dynamic Invariants

pioneered by Mike Ernst's Daikon



Example

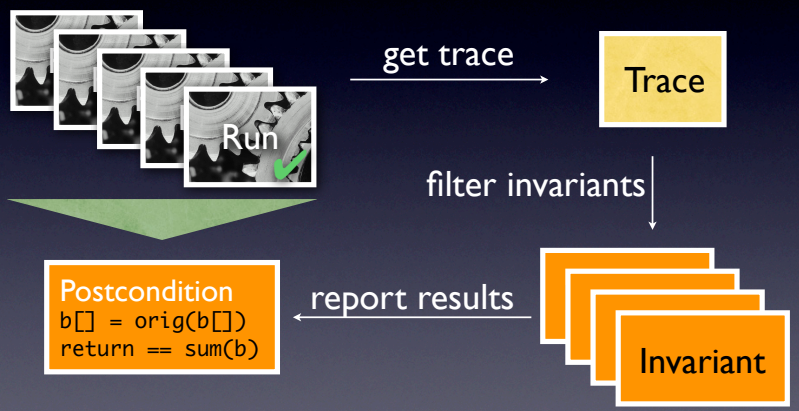
```
public int ex1511(int[] b, int n)
{
    int s = 0;
    int i = 0;
    while (i != n) {
        s = s + b[i];
        i = i + 1;
    }
    return s;
}
```

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

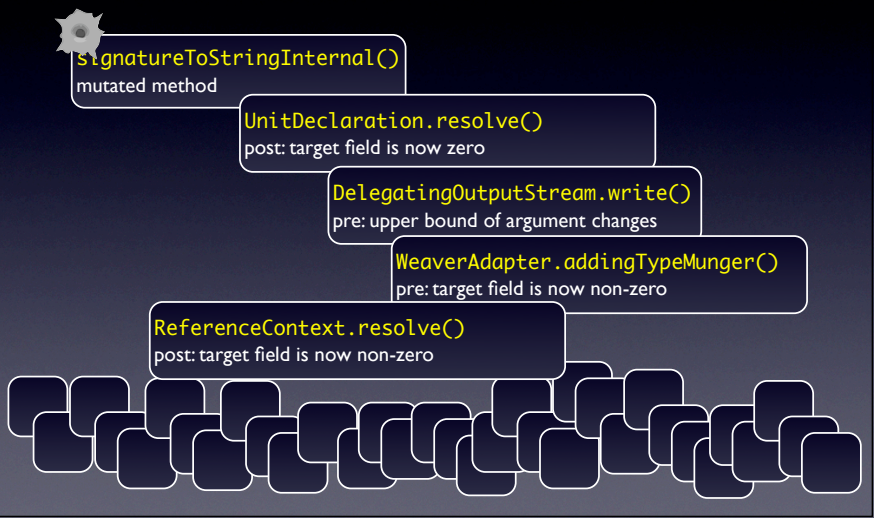
Obtaining Invariants



Impact on Invariants

```
public ResultHolder signatureToStringInternal(String signature) {
  switch(signature.charAt(0)) {
  ...
  case 'L': { // Full class name
    // Look for closing `;'
    int index = signature.indexOf(';');
    // Jump to the correct ';'
    if (index != 0 && signature.length() > index + 1 &&
        signature.charAt(index + 1) == '>')
      index = index + 2;
    ...
    return new ResultHolder (signature.substring(1, index));
  }
}
```

Impact on Invariants



Impact on Invariants

```
public ResultHolder signatureToStringInternal(String signature) {
  switch(signature.charAt(0)) {
  ...
  case 'L': { // Full class name
    // Look for closing `;'
    int index = signature.indexOf(';');
    // Jump to the correct ';'
    if (index != 0 && signature.length() > index + 1 &&
        signature.charAt(index + 1) == '>')
      index = index + 2;
    ...
    return new ResultHolder (signature.substring(1, index));
  }
}
```

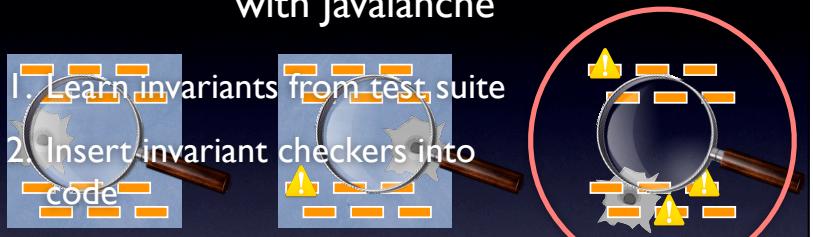
impacts 40 invariants
but undetected by AspectJ unit tests

J 

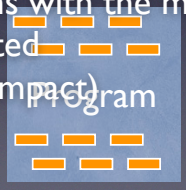
- Mutation Testing Framework for Java
  12 man-months of implementation effort
- Efficient Mutation Testing
  Manipulate byte code directly • Focus on few mutation operators • Use mutant schemata • Use coverage data
- Ranks Mutations by Impact
  Checks impact on dynamic invariants • Uses efficient invariant learner and checker

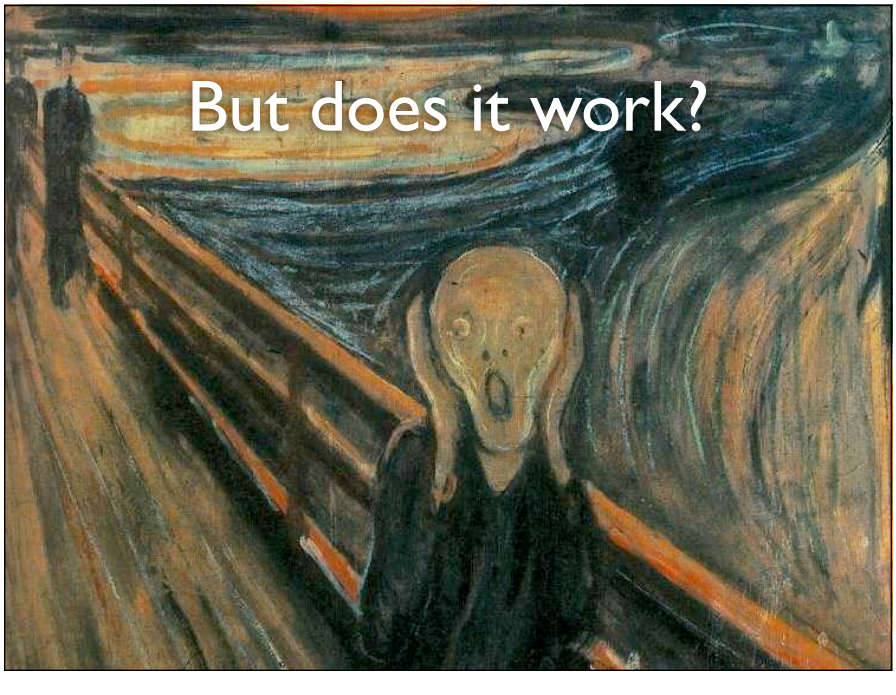
Mutation Testing
with Javalanche

Mutation Testing
with Javalanche



1. Learn invariants from test suite
2. Insert invariant checkers into code
3. Detect impact of mutations
4. Select mutations with the most invariants violated (= the highest impact)





Evaluation

1. Are mutations that violate invariants *useful?*
2. Are mutations with the highest impact *most useful?*
3. Are mutants that violate invariants *less likely to be equivalent?*

Evaluation Subjects

Name	Lines of Code	#Tests
AspectJ Core AOP extension to Java	94,902	321
Barbecue Bar Code Reader	4,837	137
Commons Helper Utilities	18,782	1,590
Jaxen XPath Engine	12,449	680
Joda-Time Date and Time Library	25,861	3,447
JTopas Parser tools	2,031	128
XStream XML Object Serialization	14,480	838

Mutations

Name	#Mutations	%detected
AspectJ Core	47,146	53
Barbecue	17,178	67
Commons	15,125	83
Jaxen	6,712	61
Joda-Time	13,859	79
JTopas	1,533	72
XStream	5,186	92

% detected means covered mutations

Performance

- Learning invariants is very expensive
  22 CPU hours for AspectJ – one-time effort
- Creating checkers is somewhat expensive
  10 CPU hours for AspectJ – one-time effort
- Mutation testing is feasible in practice
  14 CPU hours for AspectJ, 6 CPU hours for XStream

46

Are mutations that violate invariants *useful*?

Results

All differences are statistically significant according to χ^2 test

47

Evaluation

1. Are mutations that violate invariants *useful*?

What is a "useful" mutation?

riants

48

Useful Mutations

A technique for generating mutants is *useful* if most of the generated mutants *are detected:*

- less likely to be equivalent
  because detectable mutants = non-equivalent mutants
- close to real defects
  because the test suite is designed to catch real defects

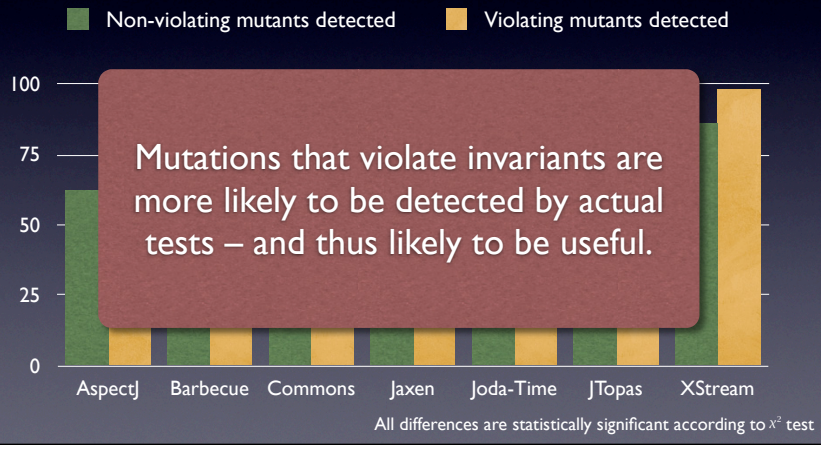
49

Mutations we look for

	not violating invariants	violating invariants
not detected by test suite	–	–
detected by test suite	?	!

50

Are mutations that violate invariants *useful*?



All differences are statistically significant according to χ^2 test

51

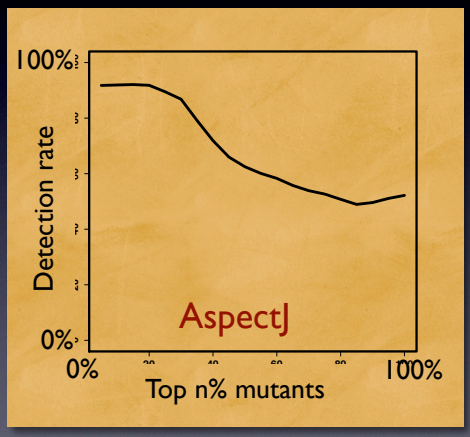
Are mutations with the highest impact *most useful?*

Mutations that violate *several* invariants are *most likely* to be detected by actual tests – and thus *the most* useful.

All differences are statistically significant according to χ^2 test

52

Detection Rates



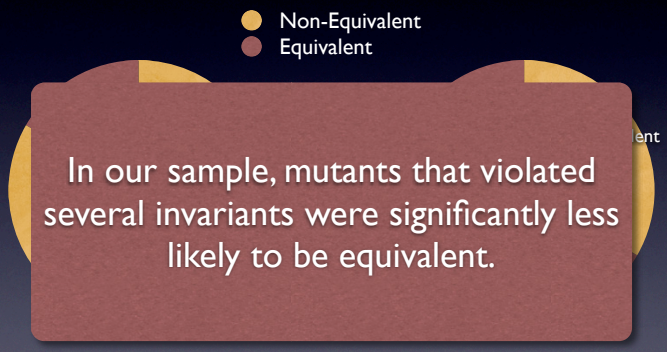
53

Are mutants that violate invariants less likely to be equivalent?

- Randomly selected non-detected *Jaxen* mutants – 12 violating, 12 non-violating
- Manual inspection: *Are mutations equivalent?*
- Mutation was proven non-equivalent iff we could create a detecting test case
- Assessment took 30 minutes per mutation

54

Are mutants that violate invariants less likely to be equivalent?



Difference is statistically significant according to Fisher test
Mutations and tests made public to counter researcher bias

55

Evaluation

1. Are mutations that violate invariants *useful?* ✔
2. Are mutations with the highest impact *most useful?* ✔
3. Are mutants that violate invariants *less likely to be equivalent?* ✔

56

Efficiency

Inspection

57

Factor 6,666 – plus full automation due to lack of inspection

58

Future Work

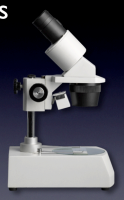
- How effective is mutation testing?
  on a large scale – compared to traditional coverage
- Predicting defects
  How does test quality impact product quality?
- Alternative impact measures
  Coverage • Program spectra • Method sequences
- Adaptive mutation testing
  Evolve mutations to have the fittest survive

59

60

Issues

Efficiency



Inspection

Mutation Testing

with Javalanche

1. Learn invariants from test suite
2. Insert invariant checkers into code
3. Detect impact of mutations
4. Select mutations with the most invariants violated (= the highest impact)

http://www.st.cs.uni-saarland.de/mutation/

Are mutations with the highest impact *most useful?*