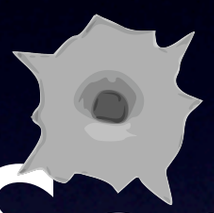


# Seeding Bugs to Find Bugs

Andreas Zeller
Saarland University

with David Schuler and Valentin Dallmeier

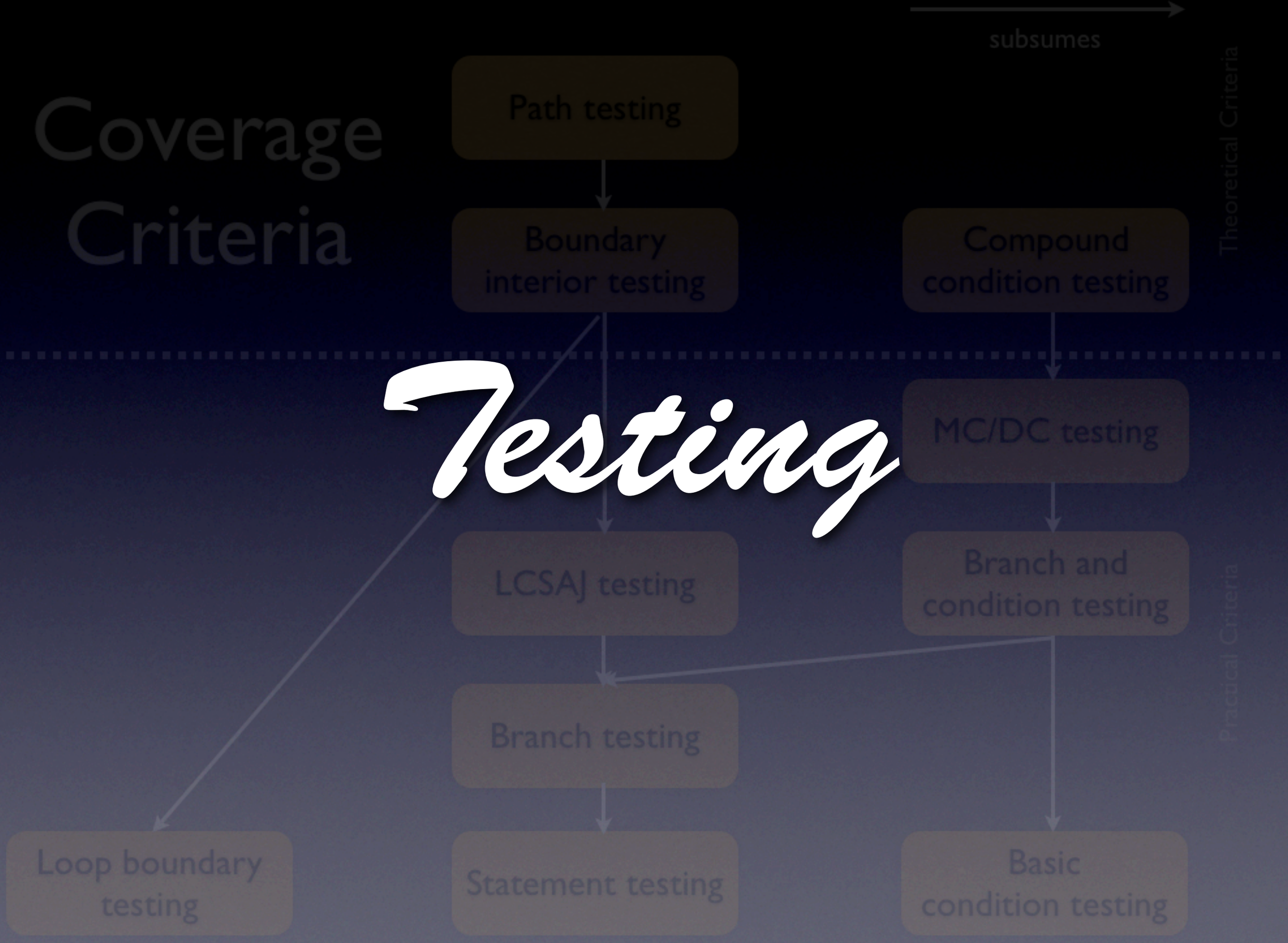
# Saarbrücken

# Saarbrücken

# Saarbrücken


5508781300Ø-A522 17

# Coverage Criteria




# Testing

# Test Quality

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

“test”

int cgi_decode(char *encoded, char *decoded)

{ char *eptr = encoded;
  char *dptr = decoded;
  int ok = 0;

while (*eptr) {

False

True

char c;
c = *eptr;
if (c == '+') {

False

True

elseif (c == '%') {

*dptr = ' ';
}

E

False

True

else
*dptr = *eptr;
}

int digit_high = Hex_Values[*(++eptr)];
int digit_low = Hex_Values[*(++eptr)];
if (digit_high == -1 || digit_low == -1) {

G

False

True

else {
*dptr = 16 * digit_high + digit_low;
}

H

ok = 1;
}

I

++dptr;
++eptr;
}

*dptr = '\0';
return ok;
}

int cgi_decode(char *encoded, char *decoded)

{ char *eptr = encoded;
  char *dptr = decoded;
  int ok = 0;

“test”

“a+b”

while (*eptr) {

False

True

char c;
c = *eptr;
if (c == '+') {

False

True

elseif (c == '%') {

*dptr = ' ';
}

False

True

else
*dptr = *eptr;
}

int digit_high = Hex_Values[*(++eptr)];
int digit_low = Hex_Values[*(++eptr)];
if (digit_high == -1 || digit_low == -1) {

G

False

True

else {
*dptr = 16 * digit_high + digit_low;
}

H

ok = 1;
}

I

++dptr;
++eptr;
}

*dptr = '\0';
return ok;
}

int cgi_decode(char *encoded, char *decoded)

{ char *eptr = encoded;
  char *dptr = decoded;
  int ok = 0;

“test”

“a+b”

“%3d”

while (*eptr) {

False

True

char c;
c = *eptr;
if (c == '+') {

False

True

elseif (c == '%') {

*dptr = ' ';
}

False

True

else
*dptr = *eptr;
}

int digit_high = Hex_Values[*(++eptr)];
int digit_low = Hex_Values[*(++eptr)];
if (digit_high == -1 || digit_low == -1) {

False

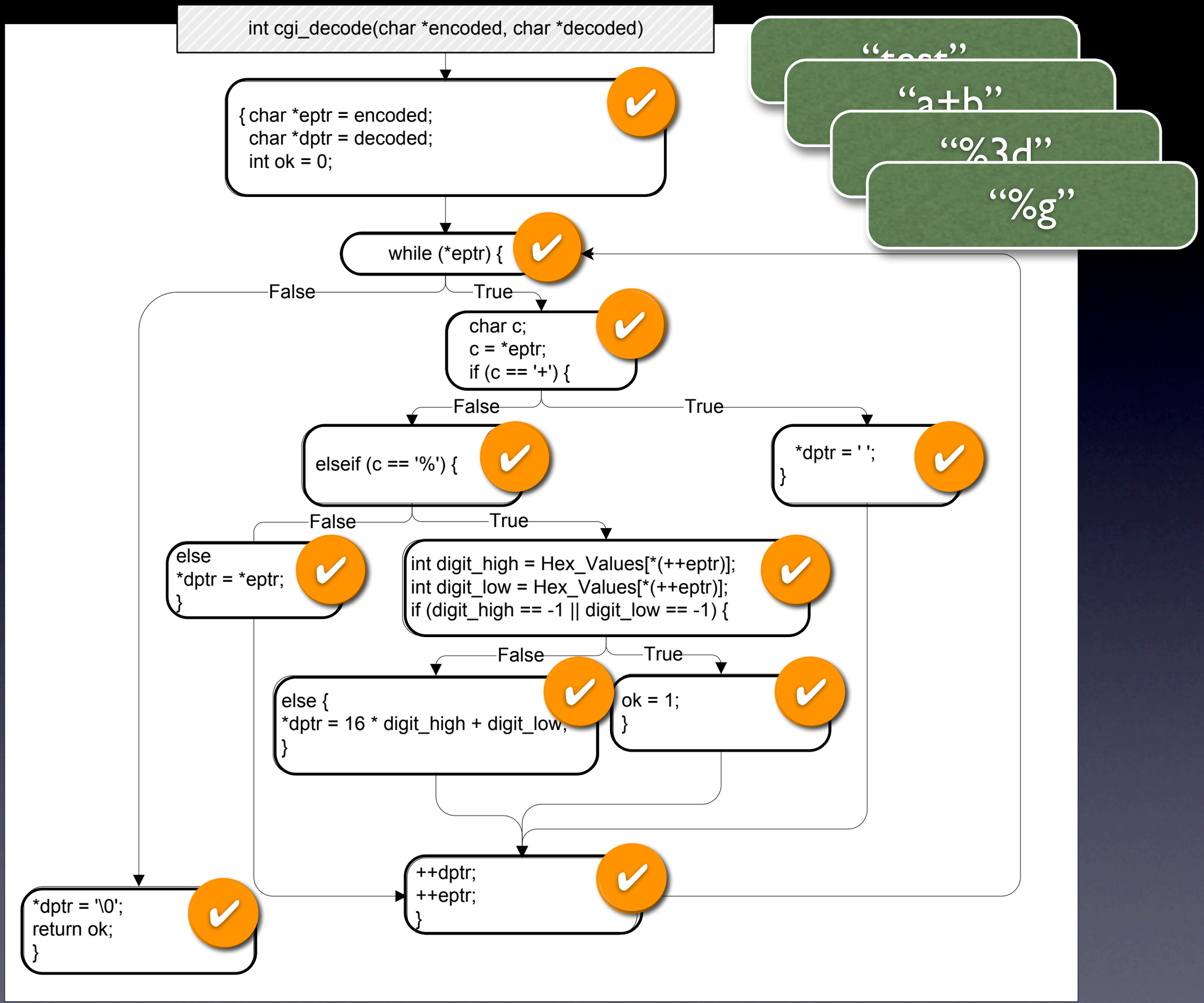
True

else {
*dptr = 16 * digit_high + digit_low;
}

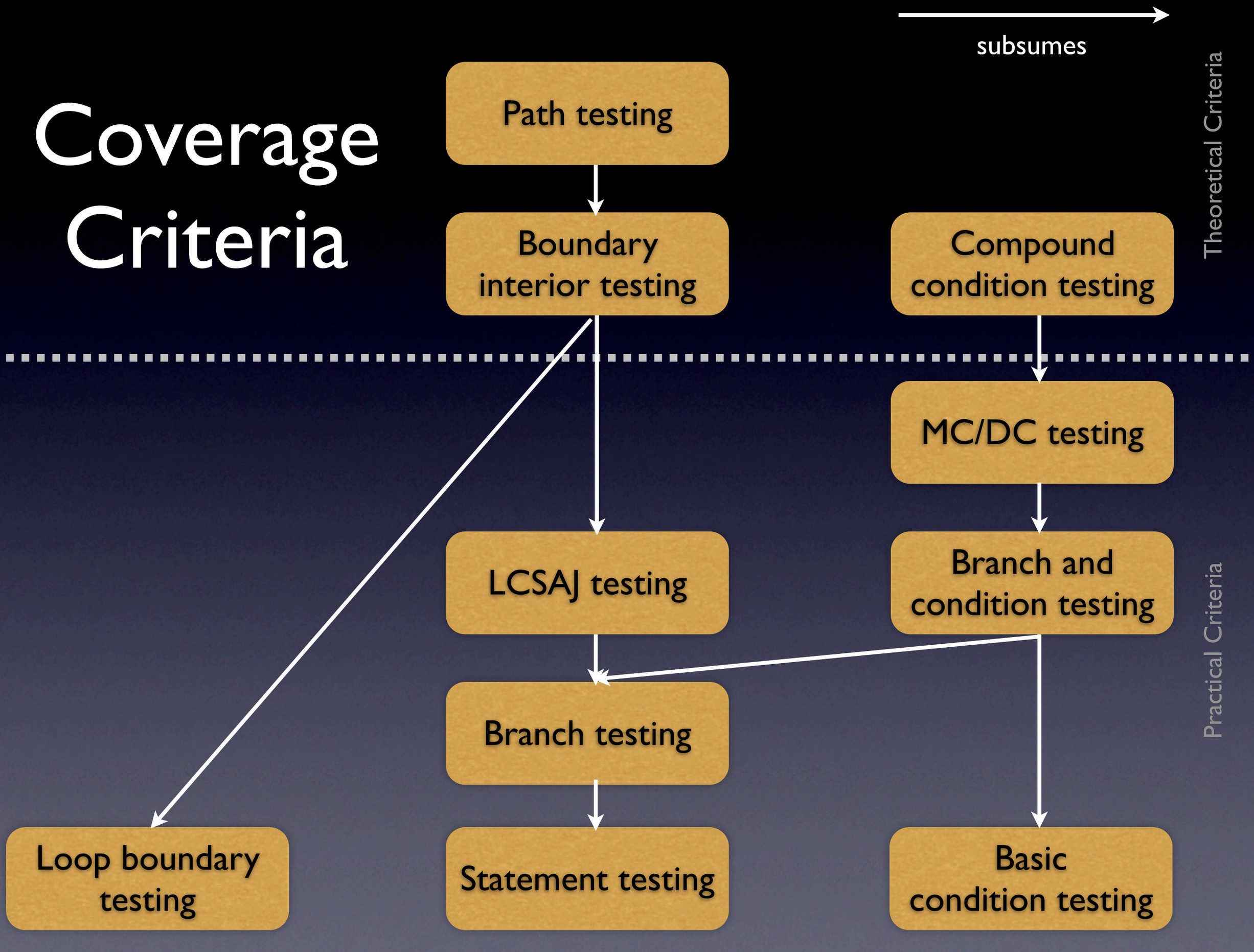
ok = 1;
}

++dptr;
++eptr;
}

*dptr = '\0';
return ok;
}


int cgi_decode(char *encoded, char *decoded)
{ char *eptr = encoded;
char *dptr = decoded;
int ok = 0;
while (*eptr) {
False
True
char c;
c = *eptr;
if (c == '+') {
False
True
elseif (c == '%') {
*dptr = ' ';
}
False
True
else
*dptr = *eptr;
}
int digit_high = Hex_Values[*(++eptr)];
int digit_low = Hex_Values[*(++eptr)];
if (digit_high == -1 || digit_low == -1) {
False
True
else {
*dptr = 16 * digit_high + digit_low;
}
ok = 1;
}
++dptr;
++eptr;
}
*dptr = '\0';
return ok;
}
“test”
“a+b”
“%3d”
“%g”

# Coverage Criteria



# Weyuker's Hypothesis

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

# Mutation Testing

DeMillo, Lipton, Sayward 1978

Program

# A Mutation

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

not found by AspectJ test suite

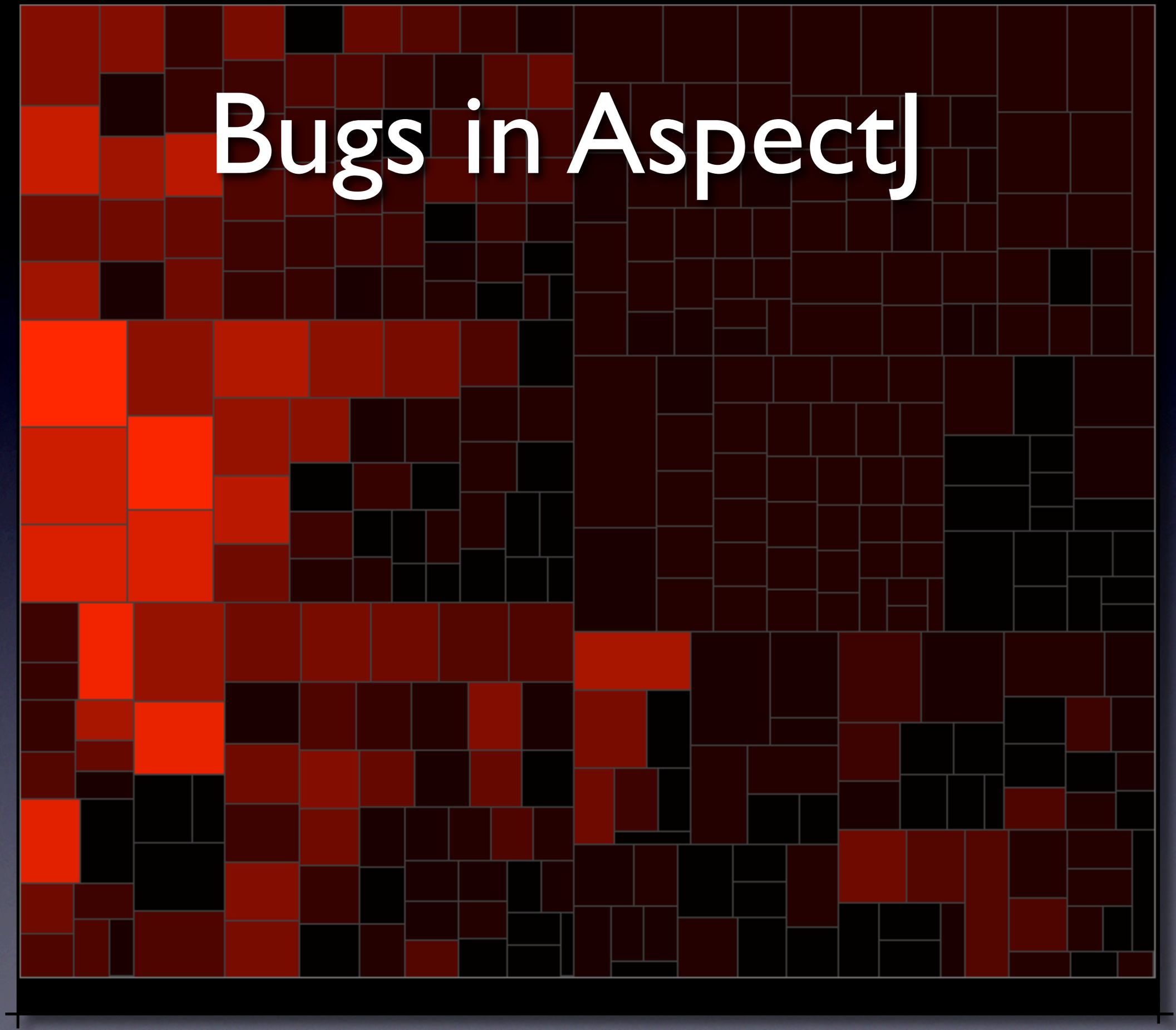
# Mutation Operators

| 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 $\psi$ with arithmetic operator $\phi$ | $e_1 \psi e_2 \neq e_1 \phi e_2$ |
| lcr | logical connector replacement | replace logical connector $\psi$ with logical connector $\phi$ | $e_1 \psi e_2 \neq e_1 \phi e_2$ |
| ror | relational operator replacement | replace relational operator $\psi$ with relational operator $\phi$ | $e_1 \psi e_2 \neq e_1 \phi e_2$ |
| 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 | |

# 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

# Bugs in AspectJ



# Issues

Efficiency

Inspection


Efficiency

Inspection

# 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*


Efficiency

Inspection

# Inspection

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

# 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

# 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



Invariant

Property

At f(), x is odd

At f(), x = 2

# 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


get trace

Trace

filter invariants

Invariant

report results

Postcondition

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

# 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

`signatureToStringInternal()`
mutated method

`UnitDeclaration.resolve()`
post: target field is now zero

`DelegatingOutputStream.write()`
pre: upper bound of argument changes

`WeaverAdapter.addingTypeMunger()`
pre: target field is now non-zero

`ReferenceContext.resolve()`
post: target field is now non-zero

# 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

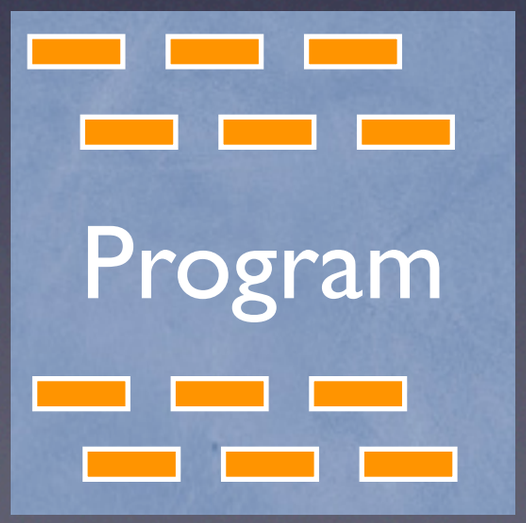
but undetected by AspectJ unit tests

# 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

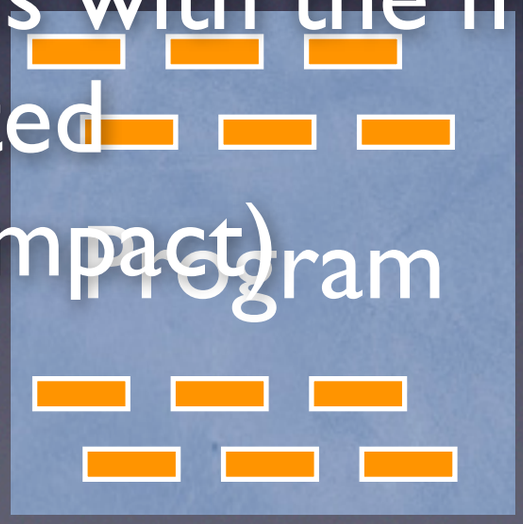
## with Javalanche



# 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)



# But does it work?

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

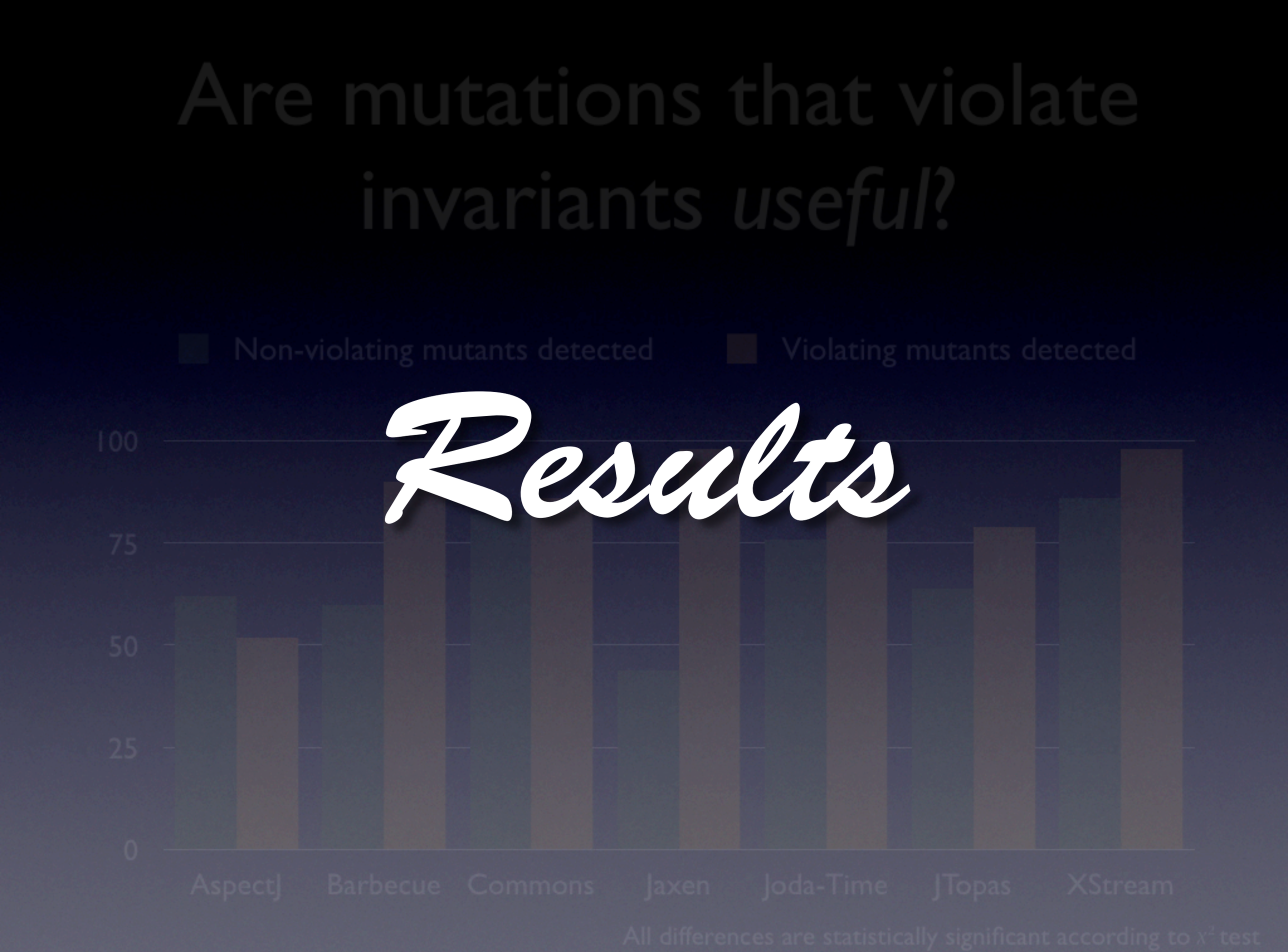
# Mutations

| 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 |

# 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

# Are mutations that violate invariants *useful*?




## *Results*

All differences are statistically significant according to $\chi^2$ test

# Evaluation

1. Are mutations that violate invariants *useful?*

What is a “useful” mutation?

# Useful Mutations

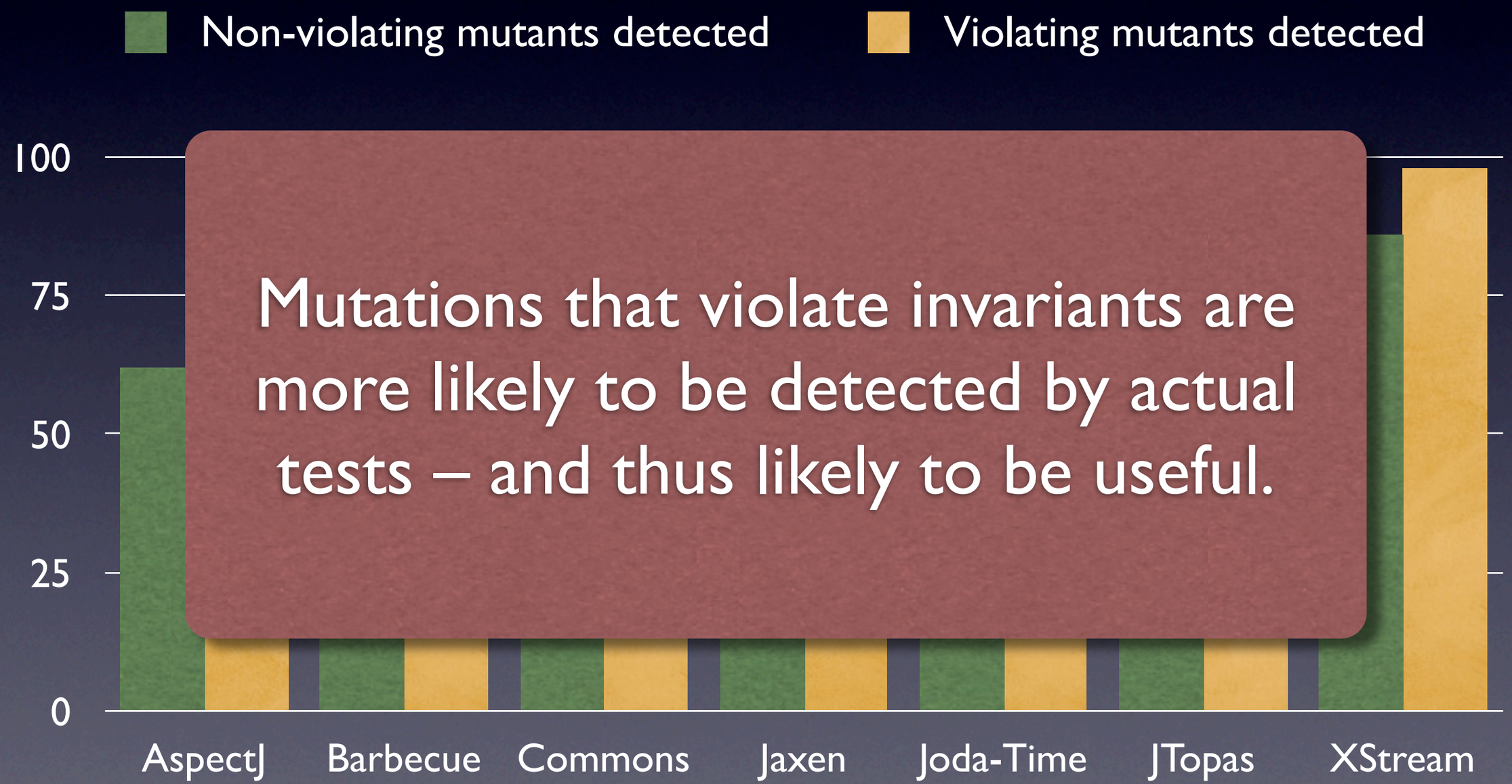
A technique for generating mutants is *useful* if most of the generated mutants *are detected:*

- 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

# Mutations we look for

| | not violating invariants | violating invariants |
|---|---|---|
| not detected by test suite | — | — |
| detected by test suite | ? | ! |

# Are mutations that violate invariants *useful*?




Mutations that violate invariants are more likely to be detected by actual tests – and thus likely to be useful.

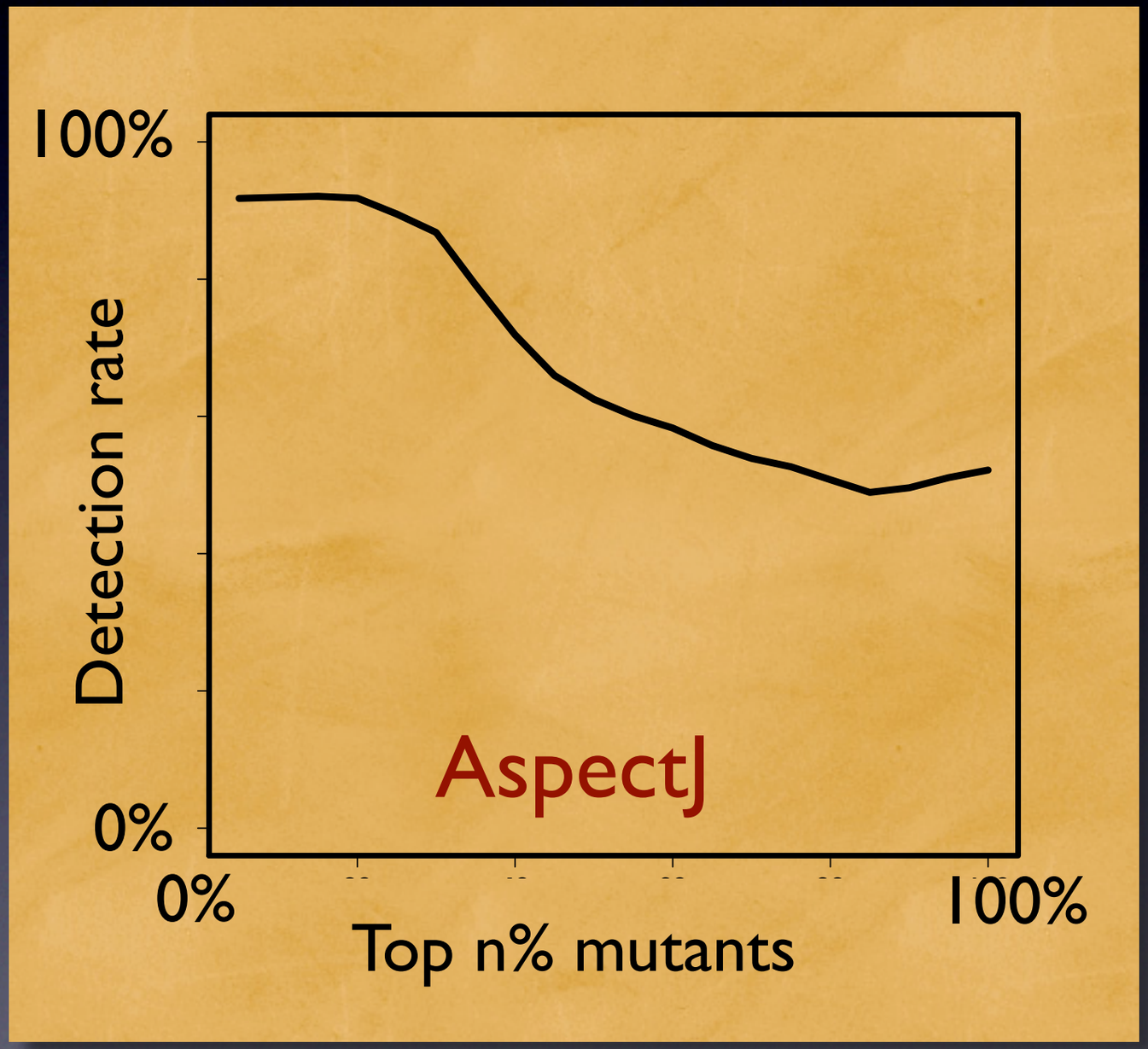
All differences are statistically significant according to $\chi^2$ test

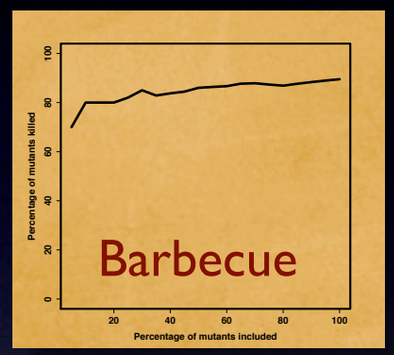
# Are mutations with the highest impact *most useful?*

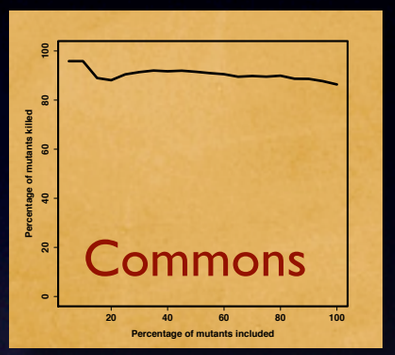

Mutations that violate *several* invariants are *most likely* to be detected by actual tests – and thus *the most* useful.

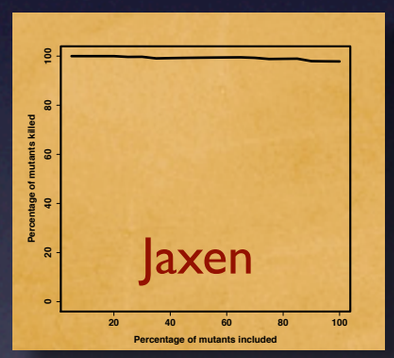
All differences are statistically significant according to $\chi^2$ test

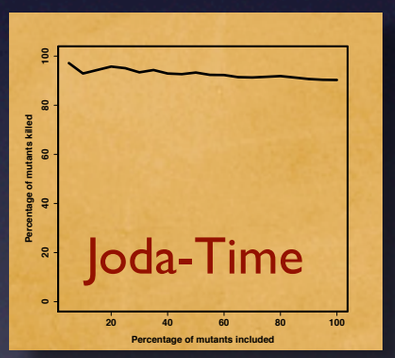
# Detection Rates

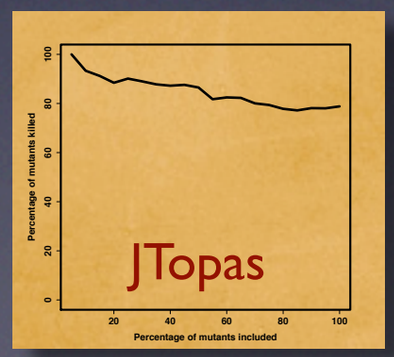


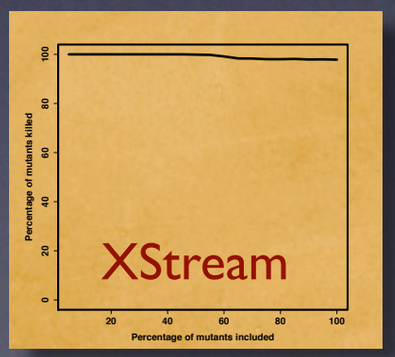



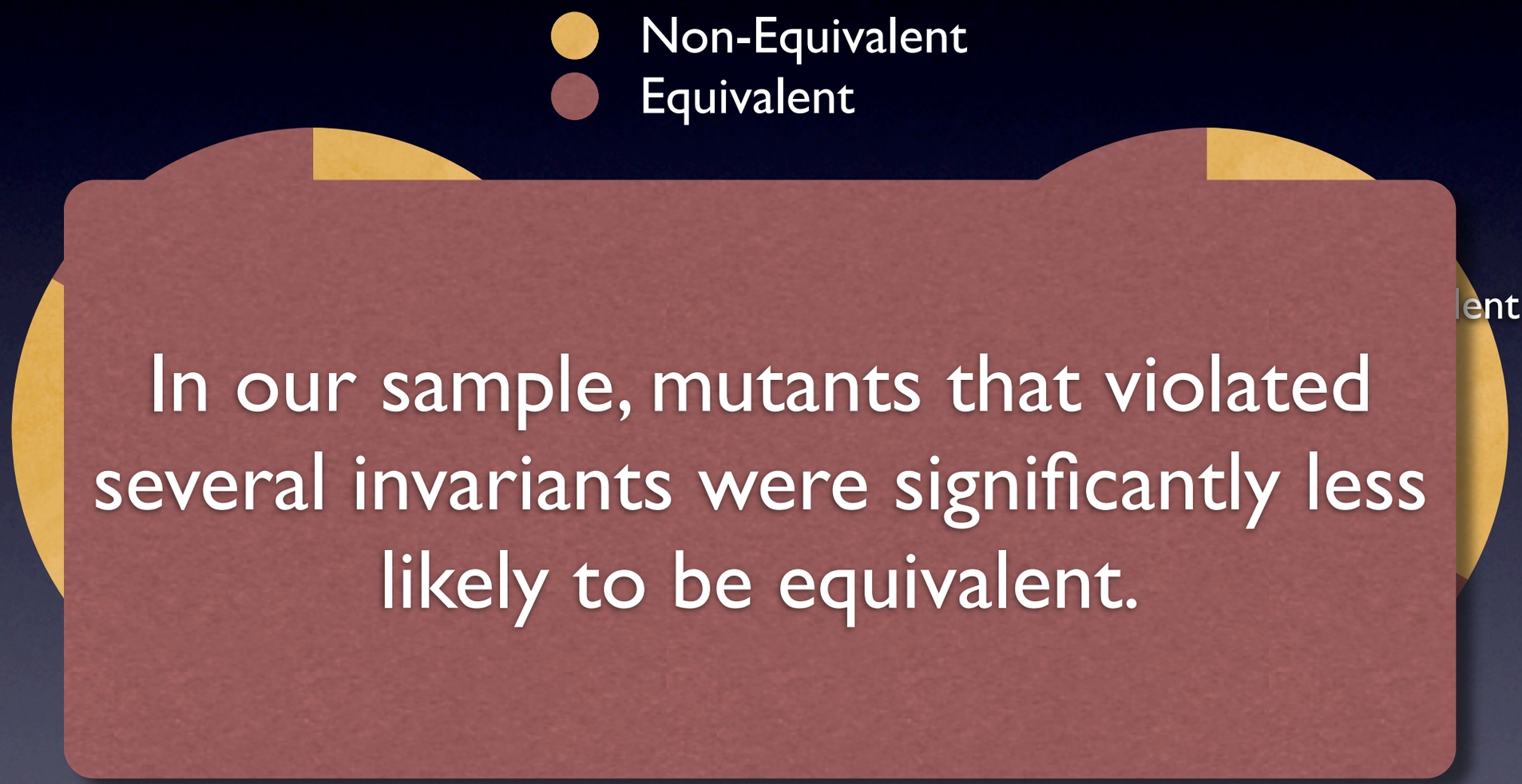



# Are mutants that violate invariants less likely to be equivalent?

- 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

# Are mutants that violate invariants less likely to be equivalent?




Violating mutants

Non-violating mutants

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

# 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?* ✔


Efficiency

Inspection

Mid
16 LOC

AspectJ
~94,000 LOC

# 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

## Issues

Efficiency

Inspection

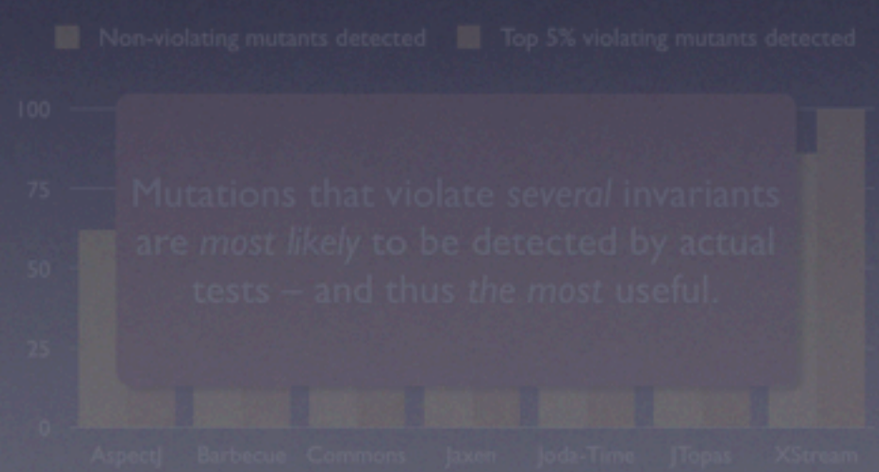
## Mutation Testing

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)

# Conclusion

## 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 *t* test

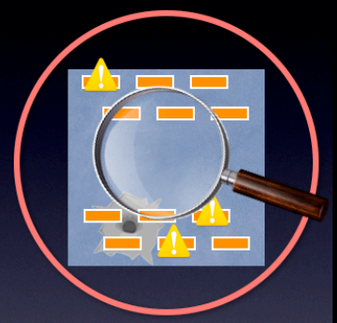
# Issues

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-sb.de/mutation/

## Are mutations with the highest impact *most useful?*