Mutation Testing

Hernán Wilkinson
UBA - 10Pines
hernan.wilkinson@gmail.com

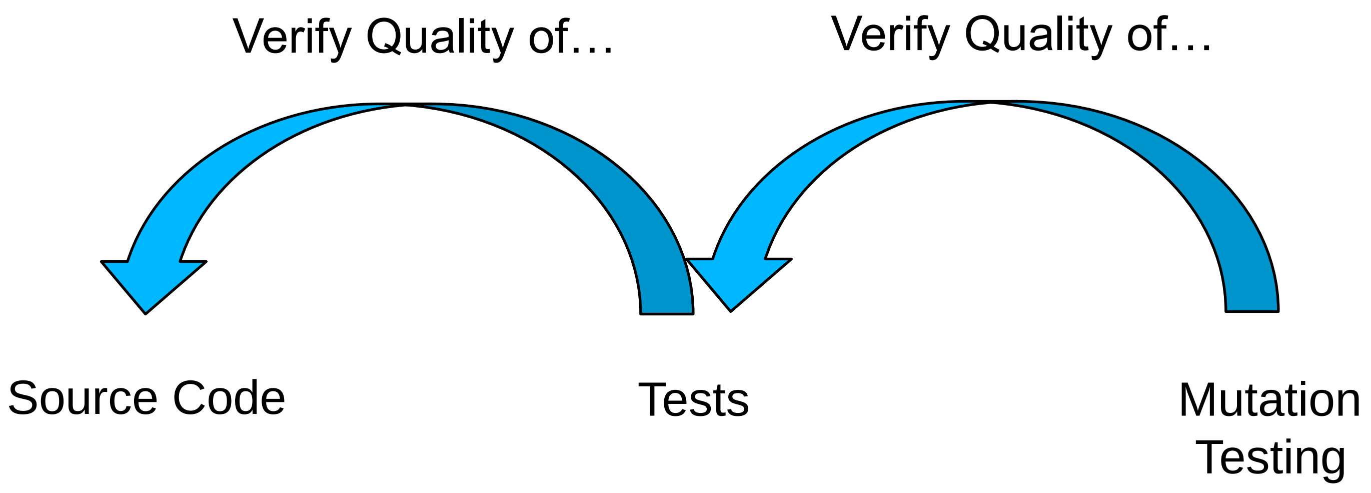
Nicolás Chillo
UBA
nchillo@gmail.com

Gabriel Brunstein
UBA
gaboto@gmail.com

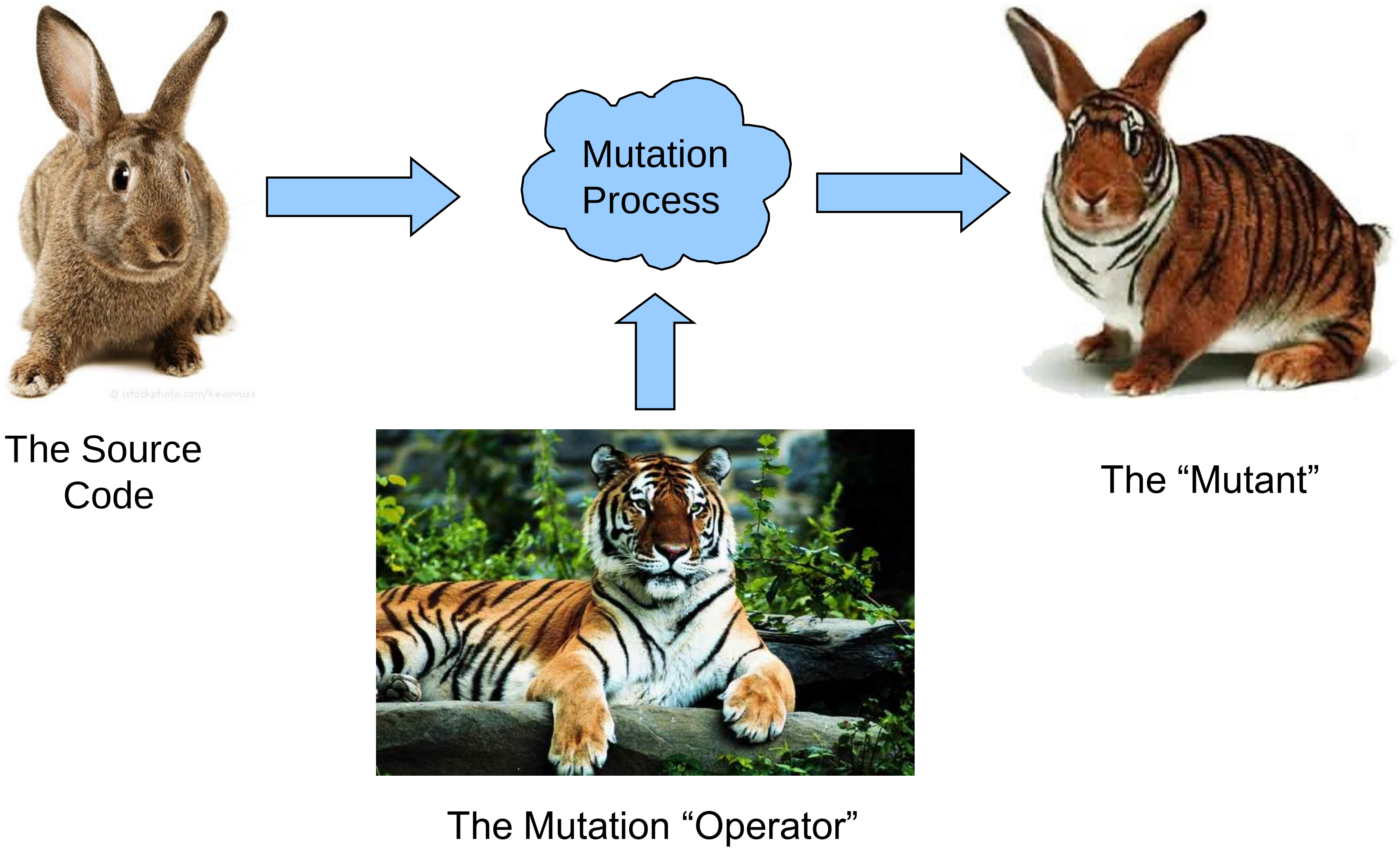
What is Mutation Testing?

Technique to verify the quality of the tests

What is Mutation Testing?



How does it work?
1st Step: Create the Mutant



Examples

```
DebitCard>>= anotherDebitCard
   ^(type = anotherDebitCard type)
    and: [ number = anotherDebitCard number ]
```

Operator: Change #and: by #or:

```
CreditCard>>= anotherDebitCard
   ^(type = anotherDebitCard type)
      or: [ number = anotherDebitCard number ]
```

Examples

```
Purchase>>netPaid
   ^self totalPaid – self totalRefunded
```

Change #- with #+

```
Purchase>>netPaid
   ^self totalPaid + self totalRefunded
```

Why?
How does it help?

How does it work?
2nd Step: Try to Kill the Mutant

The “Mutant”

A Killer
tries to kill the Mutant!

The Test Suite

All tests run → The Mutant Survives!!!

A test fails or errors → The Mutant Dies

Meaning…

The Mutant Survives → The case generated by the mutant is not tested

The Mutant Dies → The case generated by the mutant is tested

Example: The mutant survives

```
DebitCard>>= anotherDebitCard

   ^(type = anotherDebitCard type) and: [ number = anotherDebitCard number ]
```

$\downarrow$ Operator: Change #and: by #or:

```
DebitCard>>= anotherDebitCard
   ^(type = anotherDebitCard type) or: [ number = anotherDebitCard number ]
```

✓

```
DebitCardTest>>testDebitCardWithSameNumberShouldBeEqual

      self assert: (DebitCard visaNumbered: 123) = (DebitCard visaNumbered: 123).
```

Example: The mutant dies

```
DebitCard>>= anotherDebitCard

   ^(type = anotherDebitCard type) and: [ number = anotherDebitCard number ]
```

Operator: Change #and: by #or:

```
DebitCard>>= anotherDebitCard
   ^(type = anotherDebitCard type) or: [ number = anotherDebitCard number ]
```

```
DebitCardTest>>testDebitCardWithSameNumberShouldBeEqual

       self assert: (DebitCard visaNumbered: 123) = (DebitCard visaNumbered: 123).
```

✓

```
DebitCardTest >>testDebitCardWithDifferentNumberShouldBeDifferent

       self deny: (DebitCard visaNumbered: 123) = (DebitCard visaNumbered: 789).
```

✗

Example: The mutant survives

```
Purchase>>netPaid
   ^self totalPaid – self totalRefunded
```

Change #- with #+

```
Purchase>>netPaid
   ^self totalPaid + self totalRefunded
```

```
Purchase>>testNetPaid
       | purchase |
       purchase := Purchase for: 20 * euros.
       self assert: purchase netPaid = (purchase totalPaid – purchase totalRefunded)
```

✓

Example: The mutant dies

```
Purchase>>netPaid
   ^self totalPaid – self totalRefunded
```

↓ Change #- with #+

```
Purchase>>netPaid
   ^self totalPaid + self totalRefunded
```

```
Purchase>>testNetPaidWithOutRefunds ← Renamed!
       | purchase |
       purchase := Purchase for: 20 * euros.
       self assert: purchase netPaid = (purchase totalPaid – purchase totalRefunded)
```

✓

```
Purchase>>testNetPaidWithRefunds
       | purchase |
       purchase := Purchase for: 20 * euros.
       purchase addRefundFor: 10 * euros.
       self assert: purchase netPaid = (purchase totalPaid – purchase totalRefunded)
```

✗

How does it work? - Summary

- Changes the original source code with special “operators” to generate “Mutants”
- Run the test suite related to the changed code
  - If a test errors or fails → Kills the mutant
  - If all tests run → The Mutant survives
- Surviving Mutants show not tested cases

The Important Thing!

MuTalk

Mutation Testing Tool for Smalltalk (Pharo and Squeak)

Demo

MuTalk – How does it work?

- Runs the test to be sure that all run
- For each method *m*
  - For each operator *o*
    - Changes *m* AST using *o*
    - Compiles mutated code
    - Changes method dictionary
    - Run the tests

MuTalk – Operators

- Boolean messages
  - Remove #not
  - Replace #and: with #eqv:
  - Replace #and: with #nand:
  - Replace #and: with #or:
  - Replace #and: with #secondArgResult:
  - Replace #and: with false
  - Replace #or: First Condition with false
  - Replace #or: Second Condition with false
  - Replace #or: with #and:
  - Replace #or: with #xor:

MuTalk – Operators

- Magnitude messages
  - Replace #'<=' with #<
  - Replace #'<=' with #=
  - Replace #'<=' with #>
  - Replace #'>=' with #=
  - Replace #'>=' with #>
  - Replace #'~=' with #=
  - Replace #< with #>
  - Replace #= with #'~='
  - Replace #> with #<
  - Replace #max: with #min:
  - Replace #min: with #max:

MuTalk – Operators

- Collection messages
  - Remove at:ifAbsent:
  - Replace #reject: with #select:
  - Replace #select: with #reject:
  - Replace Reject block with [:each | false]
  - Replace Reject block with [:each | true]
  - Replace Select block with [:each | false]
  - Replace Select block with [:each | true]
  - Replace detect: block with [:each | false] when #detect:ifNone:
  - Replace detect: block with [:each | true] when #detect:ifNone:
  - Replace do block with [:each |]
  - Replace ifNone: block with [] when #detect:ifNone:
  - Replace inject:aValue into:aBlock with aValue
  - Replace sortBlock:aBlock with sortBlock:[:a :b| true]

MuTalk – Operators

- Number messages
    - Replace #* with #/
    - Replace #+ with #-
    - Replace #- with #+
    - Replace #/ with #*

MuTalk – Operators

- Flow control messages
  - Remove Exception Handler Operator
  - Replace #ifFalse: receiver with false
  - Replace #ifFalse: receiver with true
  - Replace #ifFalse: with #ifTrue:
  - Replace #ifFalse:IfTrue: receiver with false
  - Replace #ifFalse:IfTrue: receiver with true
  - Replace #ifTrue: receiver with false
  - Replace #ifTrue: receiver with true
  - Replace #ifTrue: with #ifFalse:
  - Replace #ifTrue:ifFalse: receiver with false
  - Replace #ifTrue:ifFalse: receiver with true

Why is not widely used?

Is not new … - History

Begins in 1971, R. Lipton, “Fault Diagnosis of Computer Programs”

Generally accepted in 1978, R. Lipton et al, “Hints on test data selection: Help for the practicing programmer”

Why is not widely used?

Maturity Problem: Because Testing is not widely used YET!

(Although it is increasing)

Why is not widely used?

Integration Problem: Inability to successfully integrate it into the software development process

(TDD plays a key role now)

Why is not widely used?

Technical Problem: It is a Brute Force technique!

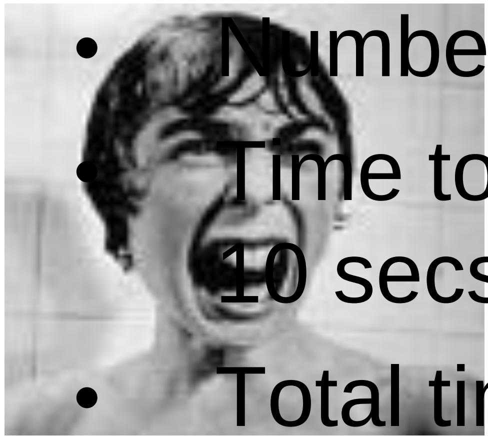
Technical Problems

- Brute force technique

$$N \times M$$

N = number of tests

M = number of mutants



Aconcagua

- Number of Tests: 666
- Number of Mutants: 1005
- Time to create a mutant/compile/link/run: 10 secs. each aprox.?
- Total time:
  - 6693300 seconds
  - 1859 hours, 15 minutes

Another way of doing it…

```
CreditCard>>= anotherCreditCard
   ^(anotherCreditCard isKindOf: self class) and: [ number =
     anotherCreditCard number ]
```

```
CreditCard>>= anotherCreditCard
   MutantId = 12 ifTrue: [ ^(anotherCreditCard isKindOf: self class) or: [
     number = anotherCreditCard number ].
   MutantId = 13 ifTrue: [ ^(anotherCreditCard isKindOf: self class)
     nand: [ number = anotherCreditCard number ].
   MutantId = 14 ifTrue: [ ^(anotherCreditCard isKindOf: self class) eqv: [
     number = anotherCreditCard number ].
```

Aconcagua

- Number of Tests: 666
- Number of Mutants: 1005
- Time to create the metamutant/compile/link: 2 minutes?
- Time to run the tests per mutant: 1 sec
- Total time:
  - 1125 seconds
  - 18 minutes 45 seconds

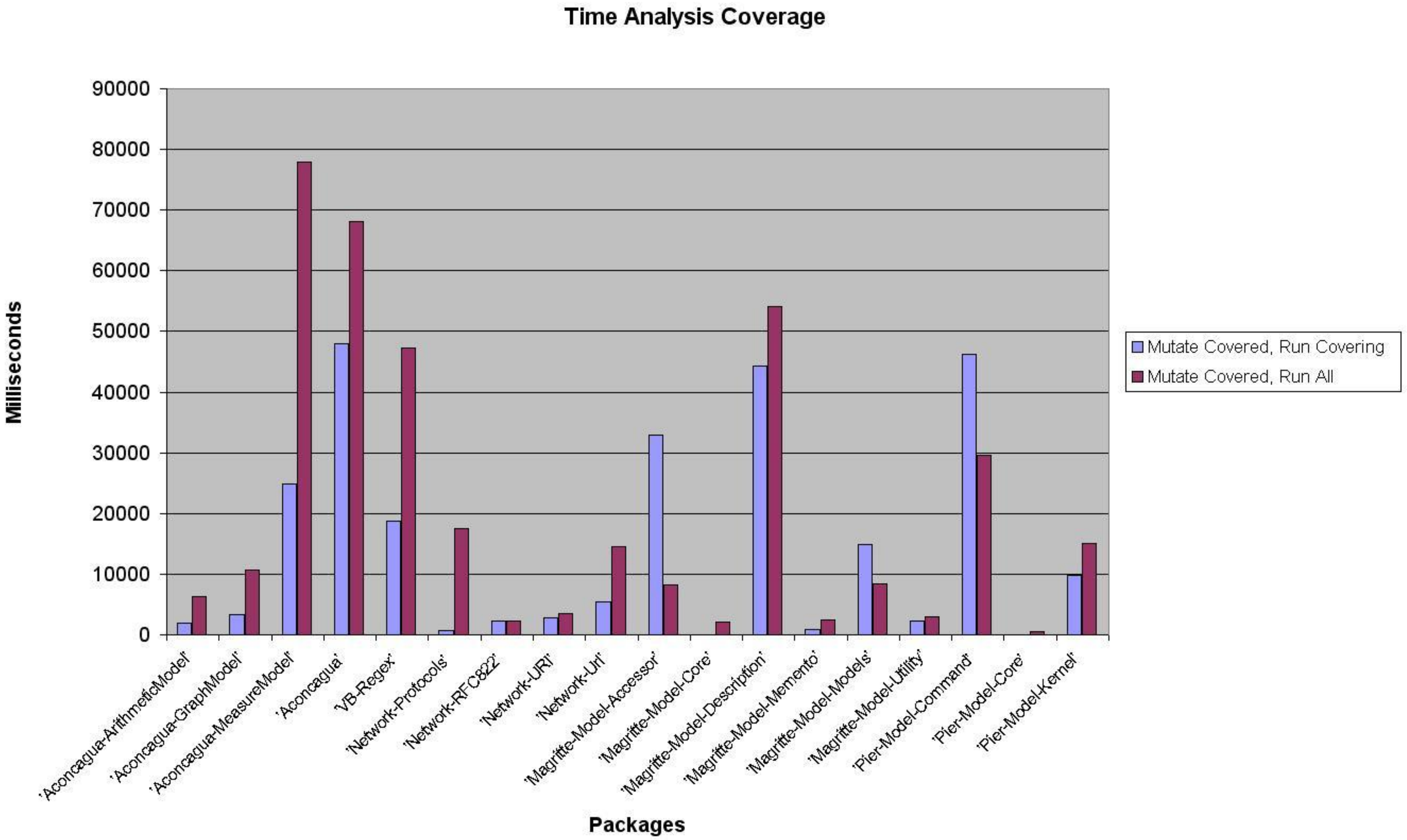
MuTalk Optimizations Running Strategies

Mutate all methods, run all tests per mutant – Create a mutant for each method – Run all the test for each mutant – Disadvantage: Slower strategy	Mutate covered methods, run all tests per mutant – Takes coverage running all tests – Mutate only covered methods – Run all methods per mutant – Relies on coverage
Mutate all methods, run only test that cover mutated method – Run coverage keeping for each method the tests that covered it – Create a mutant for each method – For each mutant, run only the tests that covered the original method	Mutate covered methods, run only test that covered mutated methods – Run coverage keeping for each method the tests that covered it – Create a mutant for only covered methods – For each mutant, run only the tests that covered the original method

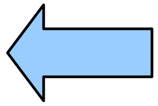
MuTalk - Aconcagua Statistics

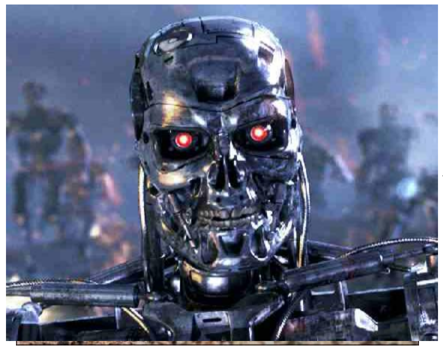
- Mutate All, Run All: 1 minute, 6 seconds
- Mutate Covered, Run Covering: 36 seconds
- Result:
  - 545 Killed
  - 6 Terminated
  - 83 Survived

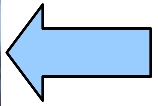
More Statistics

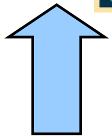


MuTalk Optimizations
Terminated Mutants









Try to kill the Mutant!

The killer has to be "Terminated"

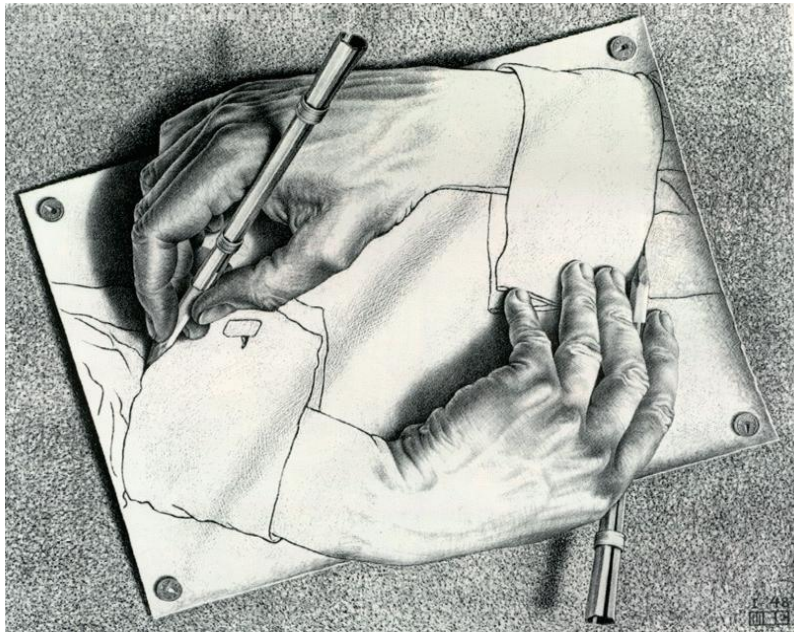
The Test Suite

MuTalk - Terminated Mutants

- Take the time it runs each test the first time
- If the test takes more thant 3 times, terminate it

Let’s redefine MuTalk as…

Mutation Testing Tool for Smalltalk (Pharo and Squeak) **that uses meta-facilities to run faster and provide inmediate feedback**



Work in progress

- Operators Categorization based on how useful they are to detect errors
- Filter Operators on View
- Cancel process

Future work

- Make Operators more “inteligent”
  - a = b ifTrue: […]
    - a = b ifFalse: [] is equivalent to a ~= b ifTrue: []
- Suggest tests using not killed mutants
- Use MuTalk to test MuTalk?

Why does it work?

“Complex faults are coupled to simple faults in such a way that a test data set that detects all simple faults in a program will detect most complex faults” (Coupling effect)

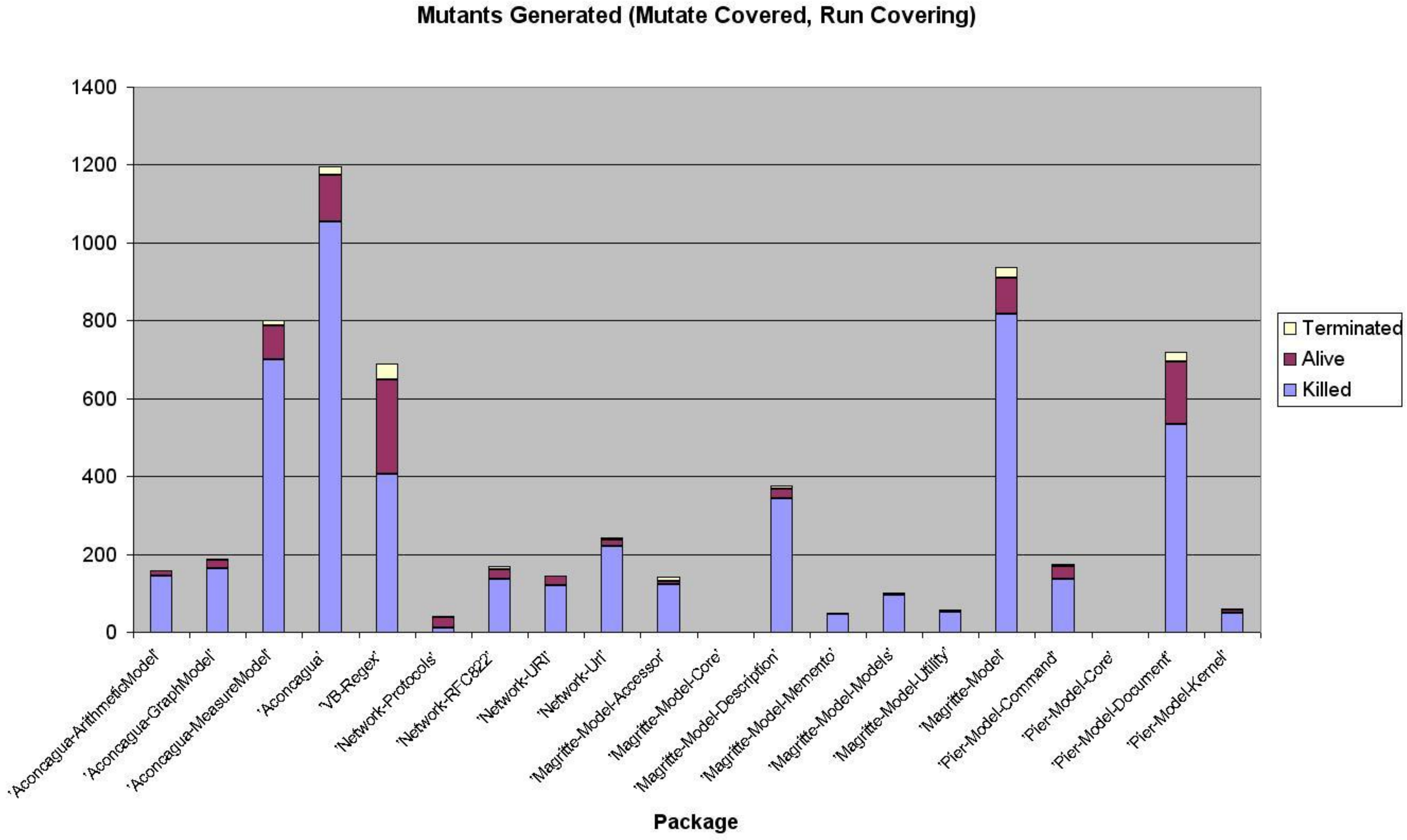
Demonstrated in 1995, K. Wah, “Fault coupling in finite bijective functions”

Why does it work?

“In practice, if the software contains a fault, there will usually be a set of mutants that can only be killed by a test case that also detects that fault”

Geist et al, “Estimation and enhancement of real-time software reliability through mutation analysis”, 1992

More Statistics…



How does it compare to coverage?

- Does not replaces coverage because some methods do not generate mutants
- But:
    - Mutants on not covered methods will survive
    - It provides better insight than coverage
    - Method Coverage fails with long methods/conditions/loops/etc.

Questions?

MuTalk - Mutation Testing for Smalltalk

Hernán Wilkinson
UBA - 10Pines
hernan.wilkinson@gmail.com

Nicolás Chillo
UBA
nchillo@gmail.com

Gabriel Brunstein
UBA
gaboto@gmail.com