Improving Test Suites via Operational Abstraction

Michael Ernst

MIT Lab for Computer Science

http://pag.lcs.mit.edu/~mernst/

Joint work with
Michael Harder, Jeff Mellen, and Benjamin Morse

Creating test suites

Goal: small test suites that detect faults well

Larger test suites are usually more effective

- Evaluation must account for size

Fault detection cannot be predicted

- Use proxies, such as code coverage

Test case selection

Example: creating a regression test suite

Assumes a source of test cases

- Created by a human
- Generated at random or from a grammar
- Generated from a specification
- Extracted from observed usage

Contributions

Operational difference technique for selecting test cases, based on observed behavior

- Outperforms (and complements) other techniques (see paper for details)
- No oracle, static analysis, or specification

Stacking and area techniques for comparing test suites

- Corrects for size, permitting fair comparison

Outline

Operational difference technique for selecting test cases

Generating operational abstractions

Stacking and area techniques for comparing test suites

Evaluation of operational difference technique

Conclusion

Operational difference technique

Idea: Add a test case *c* to a test suite *S* if *c* exercises behavior that *S* does not

Code coverage does this in the textual domain

We extend this to the semantic domain

Need to compare run-time program behaviors

- Operational abstraction: program properties
- x > y
- a[] is sorted

Test suite generation or augmentation

Idea: Compare operational abstractions induced by different test suites

Given: a source of test cases; an initial test suite

Loop:

- Add a candidate test case
- If operational abstraction changes, retain the case
- Stopping condition: failure of a few candidates

The operational difference technique is effective

Operational difference suites

- are smaller
- have better fault detection

than branch coverage suites

(in our evaluation; see paper for details)

Example of test suite generation

Program under test: **`abs`** (absolute value)

Test cases: 5, 1, 4, -1, 6, -3, 0, 7, -8, 3, …

Suppose an operational abstraction contains:

- var = constant
- var ≥ constant
- var ≤ constant
- var = var
- property ⇒ property

Considering test case 5

Initial test suite: { }

Initial operational abstraction for { }: ∅

Candidate test case: 5

New operational abstraction for { 5 }:

- Precondition: arg = 5
- Postconditions: arg = return

New operational abstraction is different,
so retain the test case

Considering test case 1

Operational abstraction for { 5 }:

- Pre: $arg = 5$
- Post: $arg = return$

Candidate test case: 1

New operational abstraction for { 5, 1 }:

- Pre: $arg \geq 1$
- Post: $arg = return$

Retain the test case

Considering test case 4

Operational abstraction for { 5, 1 }:

- Pre: $arg \geq 1$
- Post: $arg = return$

Candidate test case: 4

New operational abstraction for { 5, 1, 4 }:

- Pre: $arg \geq 1$
- Post: $arg = return$

Discard the test case

Considering test case -1

Operational abstraction for { 5, 1 }:

- Pre: arg $\geq$ 1
- Post: arg = return

Candidate test case: -1

New operational abstraction for { 5, 1, -1 }:

- Pre: arg $\geq$ -1
- Post: arg $\geq$ 1 $\Rightarrow$ (arg = return)
  arg = -1 $\Rightarrow$ (arg = -return)
  return $\geq$ 1

Retain the test case

Considering test case -6

Operational abstraction for { 5, 1, -1 }:

- Pre: $arg \geq -1$
- Post: $arg \geq 1 \Rightarrow (arg = return)$
  $arg = -1 \Rightarrow (arg = -return)$
  $return \geq 1$

Candidate test case: -6

New operational abstraction for { 5, 1, -1, -6 }:

- Pre: $\emptyset$
- Post: $arg \geq 1 \Rightarrow (arg = return)$
  $arg \leq -1 \Rightarrow (arg = -return)$
  $return \geq 1$

Retain the test case

Considering test case -3

Operational abstraction for { 5, 1, -1, -6 }:

- Post: $\text{arg} \geq 1 \Rightarrow (\text{arg} = \text{return})$
  $\text{arg} \leq -1 \Rightarrow (\text{arg} = -\text{return})$
  $\text{return} \geq 1$

Test case: -3

New operational abstraction for { 5, 1, -1, 6, -3 }:

- Post: $\text{arg} \geq 1 \Rightarrow (\text{arg} = \text{return})$
  $\text{arg} \leq -1 \Rightarrow (\text{arg} = -\text{return})$
  $\text{return} \geq 1$

Discard the test case

Considering test case 0

Operational abstraction for { 5, 1, -1, -6 }:

- Post: $arg \geq 1 \Rightarrow (arg = return)$
  $arg \leq -1 \Rightarrow (arg = -return)$
  $return \geq 1$

Test case: 0

New operational abstraction for {5, 1, -1, -6, 0 }:

- Post: $arg \geq 0 \Rightarrow (arg = return)$
  $arg \leq 0 \Rightarrow (arg = -return)$
  $return \geq 0$

Retain the test case

Considering test case 7

Operational abstraction for { 5, 1, -1, -6, 0 }:

- Post: $arg \geq 0 \Rightarrow (arg = return)$
  $arg \leq 0 \Rightarrow (arg = -return)$
  $return \geq 0$

Candidate test case: 7

New operational abstraction for { 5, 1, -1, -6, 0, 7 }:

- Post: $arg \geq 0 \Rightarrow (arg = return)$
  $arg \leq 0 \Rightarrow (arg = -return)$
  $return \geq 0$

Discard the test case

Considering test case -8

Operational abstraction for { 5, 1, -1, -6, 0 }:

- Post: $arg \geq 0 \Rightarrow (arg = return)$

  $arg \leq 0 \Rightarrow (arg = -return)$

  $return \geq 0$

Candidate test case: -8

New operational abstraction for { 5, 1, -1, -6, 0, -8 }:

- Post: $arg \geq 0 \Rightarrow (arg = return)$

  $arg \leq 0 \Rightarrow (arg = -return)$

  $return \geq 0$

Discard the test case

Considering test case 3

Operational abstraction for { 5, 1, -1, -6, 0 }:

- Post: $arg \geq 0 \Rightarrow (arg = return)$
  $arg \leq 0 \Rightarrow (arg = -return)$
  $return \geq 0$

Candidate test case: 3

New operational abstraction for { 5, 1, -1, -6, 0, 3 }:

- Post: $arg \geq 0 \Rightarrow (arg = return)$
  $arg \leq 0 \Rightarrow (arg = -return)$
  $return \geq 0$

Discard the test case; third consecutive failure

Minimizing test suites

Given: a test suite

For each test case in the suite:

- Remove the test case if doing so does not change the operational abstraction

Outline

Operational difference technique for selecting test cases

⇒ Generating operational abstractions

Stacking and area techniques for comparing test suites

Evaluation of operational difference technique

Conclusion

Dynamic invariant detection

Goal: recover invariants from programs

Technique: run the program, examine values

Artifact: Daikon

`http://pag.lcs.mit.edu/daikon`

Experiments demonstrate accuracy, usefulness

Goal: recover invariants

Detect invariants (as in **`assert`**s or specifications)

- **`x > abs(y)`**
- **`x = 16*y + 4*z + 3`**
- array **`a`** contains no duplicates
- for each node **`n`**, **`n = n.child.parent`**
- graph **`g`** is acyclic
- if **`ptr`** $\neq$ **`null`** then **`*ptr`** > **`i`**

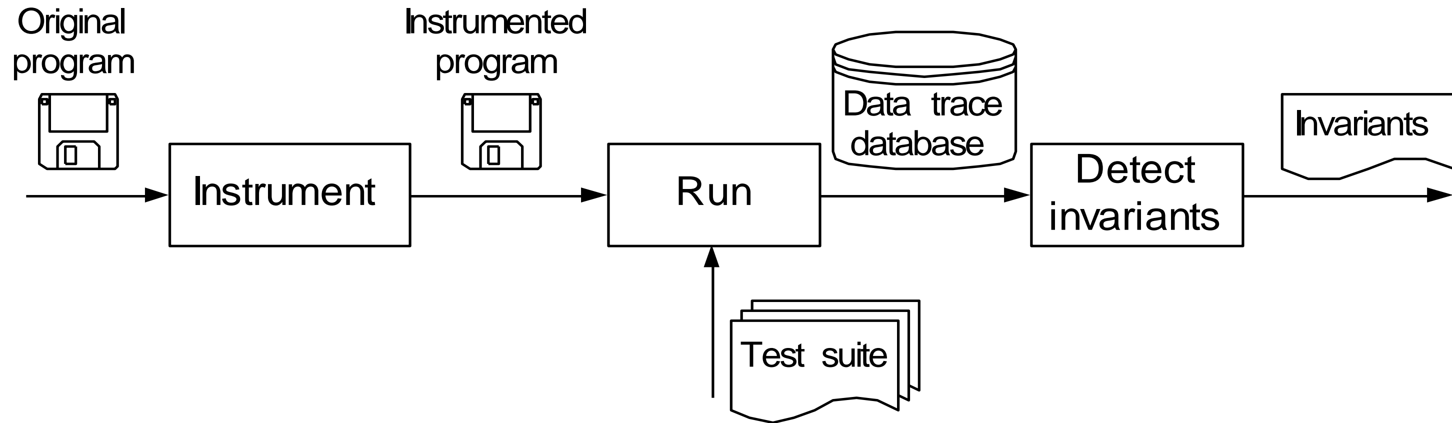
Uses for invariants

- Write better programs [Gries 81, Liskov 86]
- Document code
- Check assumptions: convert to `assert`
- Maintain invariants to avoid introducing bugs
- Locate unusual conditions
- Validate test suite: value coverage
- Provide hints for higher-level profile-directed compilation [Calder 98]
- Bootstrap proofs [Wegbreit 74, Bensalem 96]

Ways to obtain invariants

- Programmer-supplied
- Static analysis: examine the program text [Cousot 77, Gannod 96]
  - properties are guaranteed to be true
  - pointers are intractable in practice
- Dynamic analysis: run the program
  - complementary to static techniques

Dynamic invariant detection



Look for patterns in values the program computes:

- Instrument the program to write data trace files
- Run the program on a test suite
- Invariant engine reads data traces, generates potential invariants, and checks them

Checking invariants

For each potential invariant:

- instantiate
  (determine constants like a and b in $y = ax + b$)
- check for each set of variable values
- stop checking when falsified

This is inexpensive: many invariants, each cheap

Improving invariant detection

Add desired invariants: implicit values, unused polymorphism

Eliminate undesired invariants: unjustified properties, redundant invariants, incomparable variables

Traverse recursive data structures

Conditionals: compute invariants over subsets of data (if $x>0$ then $y \neq z$)

Outline

Operational difference technique for selecting test cases

Generating operational abstractions

⇒ Stacking and area techniques for comparing test suites

Evaluation of operational difference technique

Conclusion

Comparing test suites

Key metric: fault detection

- percentage of faults detected by a test suite

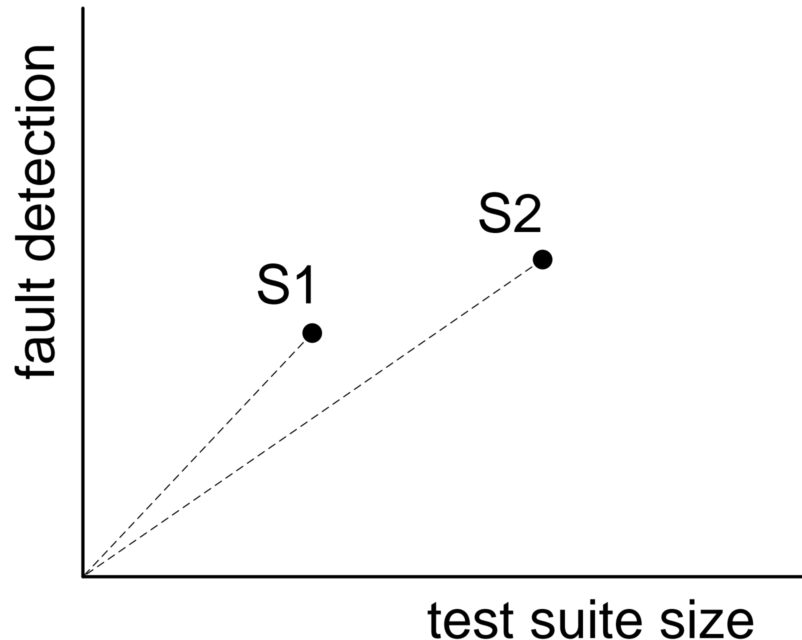
Correlated metric: test suite size

- number of test cases
- run time

Test suite comparisons must control for size

Test suite efficiency

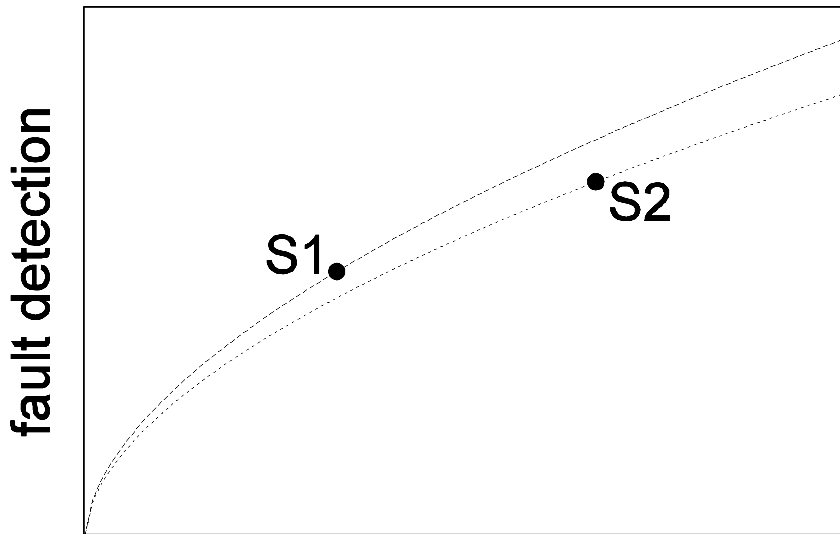
Efficiency = (fault detection)/(test suite size)

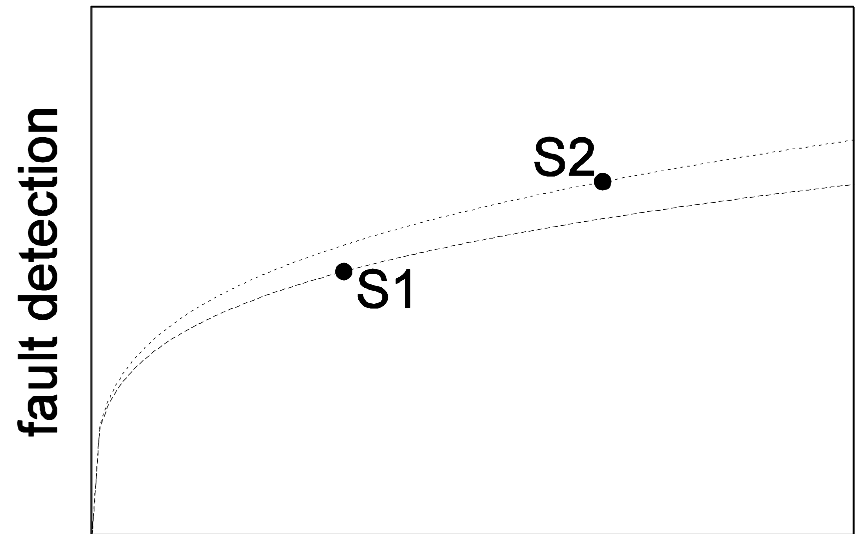


Which test suite *generation* technique is better?

Different size suites are incomparable

A technique induces a curve:





How can we tell which is the true curve?

Comparing test suite generation techiques

Each technique induces a curve

Compare the curves, not specific points

Approach: compare area under the curve

- Compares the techniques at many sizes
- Cannot predict the size users will want

Approximating the curves ("stacking")

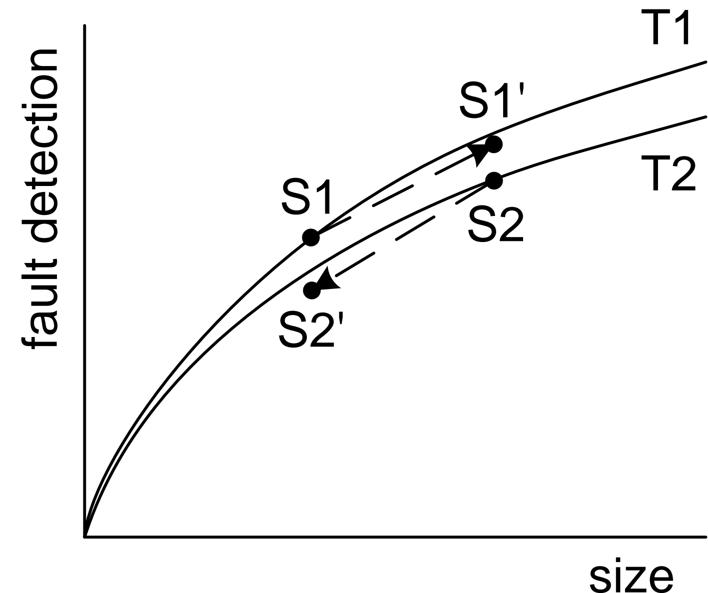
Given a test budget (in suite execution time), generate a suite that runs for that long

To reduce in size:

- select a random subset

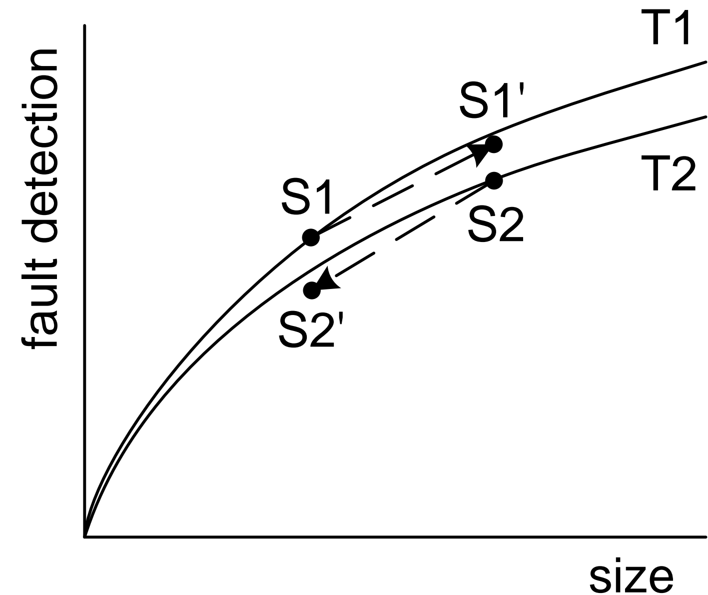
To increase in size:

- combine independent suites



Test suite generation comparison

1. Approximate the curves
2. Report ratios of areas under curves



Outline

Operational difference technique for selecting test cases

Generating operational abstractions

Stacking and area techniques for comparing test suites

⇒ Evaluation of operational difference technique

Conclusion

Evaluation of operational difference technique

- It ought to work: Correlating operational abstractions with fault detection
- It does work: Measurements of fault detection of generated suites

Subject programs

8 C programs

- seven 300-line programs, one 6000-line program

Each program comes with

- pool of test cases (1052 – 13585)
- faulty versions (7 – 34)
- statement, branch, and def-use coverage suites

Improving operational abstractions improves tests

Let the ideal operational abstraction be that generated by all available test cases

Operational coverage = closeness to the ideal

- Operational coverage is correlated with fault detection
  - Holding constant cases, calls, statement coverage, branch coverage
- Same result for 100% statement/branch coverage

Generated suites

Relative fault detection (adjusted by using the stacking technique):

Def-use coverage:	1.73
Branch coverage:	1.66
Operational difference:	1.64
Statement coverage:	1.53
Random:	1.00

Similar results for augmentation, minimization

Augmentation

Relative fault detection (via area technique):

Random:	1.00
Branch coverage:	1.70
Operational difference:	1.72
Branch + operational diff.:	2.16

Operational difference complements structural

	Best technique			Total
	Op. Diff.	equal	Branch	
CFG changes	9	11	9	29
Non-CFG changes	56	54	24	134
Total	65	65	33	163

Outline

Operational difference technique for selecting test cases

Generating operational abstractions

Stacking and area techniques for comparing test suites

Evaluation of operational difference technique

⇒ Conclusion

Future work

How good is the stacking approximation?

How do bugs in the programs affect the operational difference technique?

Contributions

Stacking and area techniques for comparing test suites

- Control for test suite size

Operational difference technique for automatic test case selection

- Based on observed program behavior
- Outperforms statement and branch coverage
- Complementary to structural techniques
- Works even at 100% code coverage
- No oracle, static analysis, or specification required