# Regression Testing

Presented by Dennis Jeffrey

September 18, 2006

CSc 620

Neelam Gupta and Rajiv Gupta, Instructors

# Outline of Talk

- Overview of Regression Testing
- "Incremental Regression Testing" (H. Agrawal, J. R. Horgan, E.W. Krauser, and S. London, 1993)
- "Prioritizing Test Cases Using Relevant Slices" (D. Jeffrey and N. Gupta, 2006)

# Regression Testing: Definition

- Regression testing: *testing modified software* to ensure that changes are correct and do not adversely affect other parts of the software
  - Make use of existing test cases developed for previous versions of the software
  - May have to create new test cases as well

# Why is Regression Testing Important?

- Software is buggy (as we all know ☺)
- Software is modified over time
  - Adding new functionality
  - Improving performance
- Protect against regressions: Modifications to software may break functionality that *used to work correctly!*

# Regression Example

```
read(x, y);
if (y < 0) then
   power = -y;
else
   power = y;
endif
z = 1;
while (power != 0) do
   z = z - y;
   power = power – 1;
endwhile
if (y < 0) then
   z = 1 / z;
endif
result = z;
write(result);
```

Program takes integer input (x,y) and should output $x^y$

These 2 tests succeed and cover all branches:

(x=2, y=-1)

(x=1, y=0)

Now try a third test case:

(x=3, y=3)

# Before Program Modification:

```
   read(x, y);
   if (y < 0) then
      power = -y;
   else
      power = y;
   endif
   z = 1;
   while (power != 0) do
*     z = z - y;
      power = power – 1;
   endwhile
   if (y < 0) then
      z = 1 / z;
   endif
   result = z;
   write(result);
```

| Test Case (x, y) | Expected Output | Actual Output |
|---|---|---|
| (2, -1) | 1/2 | 1/2 |
| (1, 0) | 1 | 1 |
| (3, 3) | 27 | -8 |

Test case (3, 3) fails!

Possible fix: when raising to a power, values should be multiplied, not subtracted!

Change line * to be:

```
z = z * y;
```

# After Program Modification:

```
   read(x, y);
   if (y < 0) then
      power = -y;
   else
      power = y;
   endif
   z = 1;
   while (power != 0) do
*     z = z * y;
      power = power - 1;
   endwhile
   if (y < 0) then
      z = 1 / z;
   endif
   result = z;
   write(result);
```

| Test Case (x, y) | Expected Output | Actual Output |
|---|---|---|
| (2, -1) | 1/2 | -1 |
| (1, 0) | 1 | 1 |
| (3, 3) | 27 | 27 |

Test case (3, 3) now succeeds!

Uh oh – test case (2, -1) fails

Regression testing should reveal this (correct fix: line * should be: **z = z * x**)

# Selecting Existing Test Cases

- Which existing test cases should be used for regression testing?
- Option 1: Re-run every existing test case
  - Assuming unlimited time and resources, this is best
  - Problem: we don't have unlimited time and resources!
- Option 2: Re-run a subset of existing test cases
  - But *which* existing test cases are likely to expose regressions in the software?
  - This presentation will describe some approaches to identify a subset of existing test cases to use for regression testing

# Incremental Regression Testing

H. Agrawal, J. R. Horgan, E.W. Krauser, and S. London
*International Conference on Software Maintenance*, 1993

# Incremental Regression Testing

- Focus of earlier regression testing research
  - Achieve required *coverage* of the modified program with minimal re-work
- Focus of this research paper
  - Verify that behavior of modified program is unchanged, except where required by the modifications
  - Approach: Identify test cases in the existing test suite on which the original and modified programs may produce different outputs

# Modified Program: Assumptions

- Assumption 1: No statements are added to or deleted from the program
- Assumption 2: No changes are made to the left-hand side of assignment statements
- These two assumptions will be relaxed later

# Two Observations

- If a statement is not executed by a test case, it cannot affect the program output of that test case

- Not all statements in the program are executed by each test case

# Execution Slice Technique

- Key Idea: If a test case does not execute any modified statement, it need not be re-run
- Definition: An execution slice is the set of statements executed by a test case in a program
- Approach: Compute the *execution slice* of each test case, then re-run only those test cases whose execution slices contain a modified statement

# Example: Execution Slice Technique

```
S1:  read(a, b, c);
S2:  class := scalene;
S3:  if a==b or b==a
S4:    class := isosceles;
S5:  if a*a == b*b + c*c
S6:    class := right;
S7:  if a==b and b==c
S8:    class := equilateral;
S9:  case class of
S10:   right      : area := b*c / 2;
S11:   equilateral: area := a*2 * sqrt(3)/4;
S12:   otherwise  : s := (a+b+c)/2;
S13:                area := sqrt(s*(s-a)*(s-b)*(s-c));
     end
S14: write(class, area)
```

| Test | Input (a, b, c) | Output class | Output area |
|---|---|---|---|
| T1 | (2, 2, 2) | equilateral | 1.73 |
| T2 | (4, 4, 3) | isosceles | 5.56 |
| T3 | (5, 4, 3) | right | 6.00 |
| T4 | (6, 5, 4) | scalene | 9.92 |
| T5 | (3, 3, 3) | equilateral | 2.60 |
| T6 | (4, 3, 3) | scalene | 4.47 |

- Debugging for failing test T5 reveals that in statement S11, the expression **`a*2`** should instead be **`a*a`**

- Given the above modification, which existing test cases should be re-run? Certainly T5, but what about T1 – T4?

# Execution Slice of T3 (a=5, b=4, c=3):

```
S1:  read(a, b, c);
S2:  class := scalene;
S3:  if a==b or b==a
S4:    class := isosceles;
S5:  if a*a == b*b + c*c
S6:    class := right;
S7:  if a==b and b==c
S8:    class := equilateral;
S9:  case class of
S10:   right      : area := b*c / 2;
S11:   equilateral: area := a*2 * sqrt(3)/4;
S12:   otherwise  : s := (a+b+c)/2;
S13:                area := sqrt(s*(s-a)*(s-b)*(s-c));
     end
S14: write(class, area)
```

- T3 does not need to be re-run (same for T2, T4)
- T1 *does* need to be re-run!

- Now consider failing test case T6
- Debugging reveals that in statement S3, the predicate **`b==a`** should instead be **`b==c`**
- Given the above modification, which of the other test cases need to be re-run (besides T6)?

- Execution slice technique → *all of them*!
- Observation: this modification will *not* affect the program output for any test cases that classify the triangle as “equilateral” or “right”
- Conclusion: The execution slice technique may select some test cases that actually do *not* need to be re-run!

- New Observation: Even if a statement is executed by a test case, it does not necessarily affect the program output for that test case

# Dynamic Slice Technique

- Key Idea: A test case whose output is not affected by a modification, need not be re-run
- Definition: A dynamic slice on the output of a program is the set of statements that are executed by a test case *and* that affect the output of the program for that test case
- Approach: Compute the *dynamic slice* on the output of each test case, then re-run only those test cases whose dynamic slices contain a modified statement

# Example: Dynamic Slice Technique

- Consider the same example program used when discussing execution slices (recall: to fix T6, change predicate **b==a** in S3 to be **b==c**)

- Besides T6, which of test cases T1 – T5 should also be re-run on the modified program?

# Dynamic Slice of T1 (a=2, b=2, c=2):

```
S1:  read(a, b, c);
S2:  class := scalene;
S3:  if a==b or b==a
S4:    class := isosceles;
S5:  if a*a == b*b + c*c
S6:    class := right;
S7:  if a==b and b==c
S8:    class := equilateral;
S9:  case class of
S10:   right      : area := b*c / 2;
S11:   equilateral: area := a*a * sqrt(3)/4;
S12:   otherwise  : s := (a+b+c)/2;
S13:                area := sqrt(s*(s-a)*(s-b)*(s-c));
     end
S14: write (class, area)
```

- T1 does not need to be re-run (same for T3 – T5)
- T2 *does* need to be re-run!

- Let us take a closer look at test case T4, which was identified as *not necessary* to be re-run...

# Dynamic Slice of T4 (a=6, b=5, c=4):

```
S1:  read(a, b, c);
S2:  class := scalene;
S3:  if a==b or b==a
S4:    class := isosceles;
S5:  if a*a == b*b + c*c
S6:    class := right;
S7:  if a==b and b==c
S8:    class := equilateral;
S9:  case class of
S10:   right      : area := b*c / 2;
S11:   equilateral: area := a*a * sqrt(3)/4;
S12:   otherwise  : s := (a+b+c)/2;
S13:                area := sqrt(s*(s-a)*(s-b)*(s-c));
     end
S14: write(class, area)
```

- As expected, T4's output is not changed due to this modification
- However, suppose that the modification in statement S3 is made (erroneously) to be **b==b** instead of **b==c**
- Observation: T4's output now *changes* due to this new modification!
- Conclusion: The dynamic slice technique may actually *miss* some relevant test cases that *should* be re-run!

- New Observation: A statement executed by a test case that did not *actually* affect the program output could still *potentially* affect the program output!

- Idea: Besides identifying the statements *actually affecting* program output, we should *also* identify those statements that *could* affect the output *if they evaluate differently*

# Relevant Slice Technique

- Key Idea: A test case whose output is neither affected nor potentially affected by a modification, need not be re-run

- Definition: A relevant slice on the output of a program is the set of statements that are executed by a test case *and* that affect, or could potentially affect, the output of the program for that test case

- Approach: Compute the *relevant slice* on the output of each test case, then re-run only those test cases whose relevant slices contain a modified statement

# Relevant Slices: Details

- Includes all the statements in the dynamic slice, but *also* includes:
  - The set of predicates P on which the statements in the dynamic slice are potentially dependent
  - The closure of the data and potential dependencies of the predicates in P
- Definition: A use of variable V at statement L is potentially-dependent upon a predicate P (executed earlier in the execution trace) if both of the following hold:
  - V is not defined between P and L but there exists another path between P and L that defines V
  - Changing the outcome of P *may* cause that un-traversed path to be traversed

# Relevant Slice Example

```
1: x = …;
2: if(c1) {
3:    if(c2)
4:       x = …;
5:    y = …x…;
   }
6: z = …x…;
```

Suppose c1 is “true” and c2 is “false” (the execution trace is <1,2,3,5,6>)

The use of “x” at line 6 *actually* depends upon the definition at line 1

The use of “x” at line 6 *potentially* depends upon the predicate c2 (but *not* on c1)

# Example: Relevant Slice Technique

- Consider the same example program used when discussing execution/dynamic slices (recall: to fix T6, change predicate **b==a** in S3 to be **b==c**)

- We know (from the previous example) that T2 and T6 need to be re-run

- Does anything else need to be re-run?

# Relevant Slice of T4 (a=6, b=5, c=4):

```
  S1:  read(a, b, c);
  S2:  class := scalene;
* S3:  if a==b or b==a
  S4:    class := isosceles;
* S5:  if a*a == b*b + c*c
  S6:    class := right;
* S7:  if a==b and b==c
  S8:    class := equilateral;
  S9:  case class of
  S10:   right      : area := b*c / 2;
  S11:   equilateral: area := a*a * sqrt(3)/4;
  S12:   otherwise  : s := (a+b+c)/2;
  S13:                area := sqrt(s*(s-a)*(s-b)*(s-c));
       end
  S14: write (class, area)
```

- T4 *does* need to be re-run!

- T1, T3, and T5 do not need to be re-run

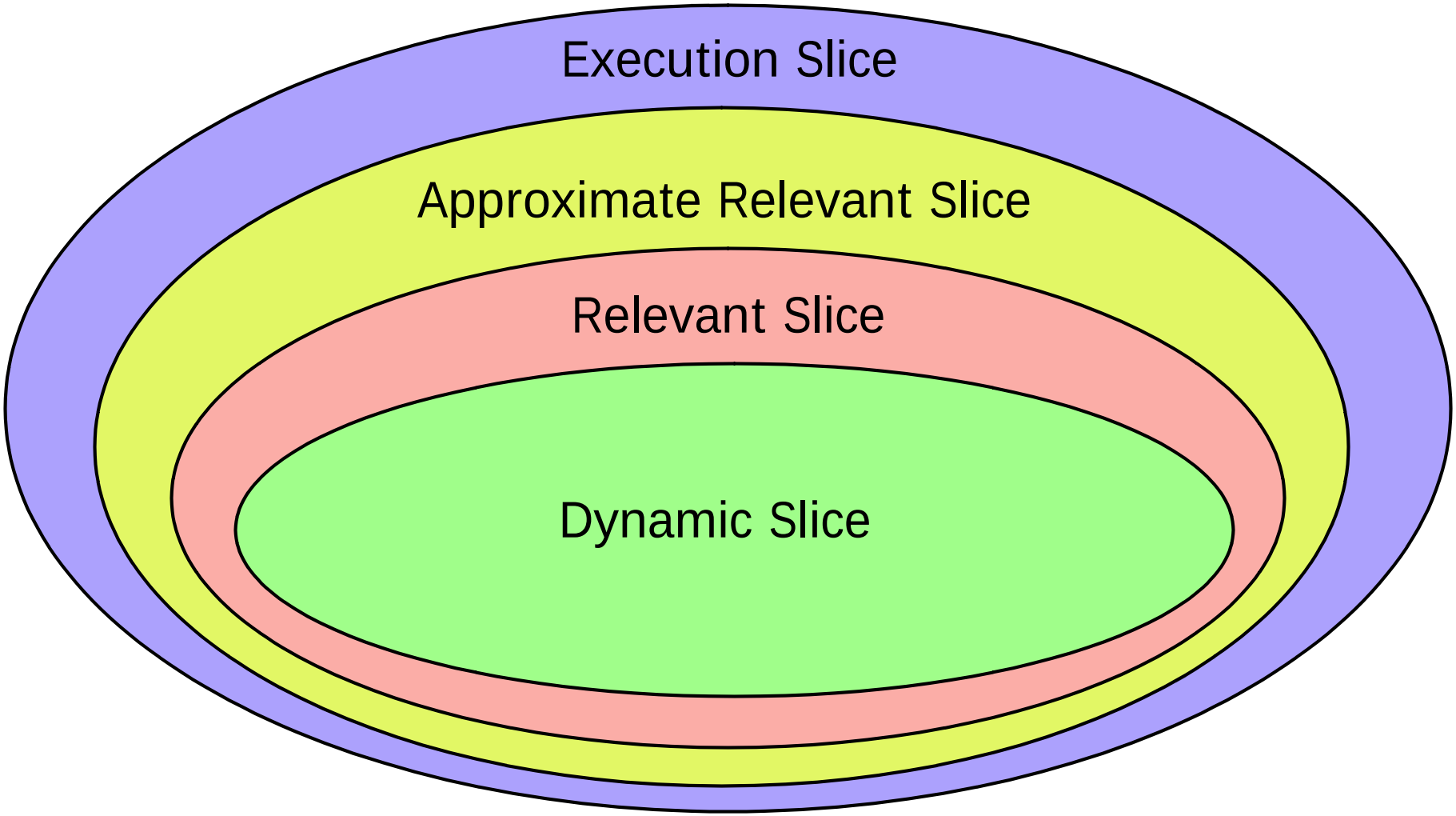
- In this example, only test cases T2, T4, and T6 may have their output affected by a modification in statement S3 (relevant slice technique identifies exactly this subset)

# Approximate Relevant Slice

- Observation: Relevant slices involve computing *static* data dependencies → can be imprecise

- Approximate Relevant Slice: besides statements that actually affected program output, also include *all* the additional executed predicates and their data dependencies
  - Simpler to compute but more conservative than a relevant slice

# Relationships Among Slices



# Relaxing the Assumptions

- Recall: assumed no statements were added/deleted, and there were no changes to LHS of assignments (we now relax these assumptions for the relevant slice technique)

- Deleting an assignment S: "x = ..."
  - Effect: right-hand side of S changed to "x"
  - Re-run any test case whose relevant slice contains S

- Adding an assignment S: "x = exp"
  - Let $u_1$, $u_2$, ..., $u_n$ be the set of reaching uses of x
  - Effect: uses of "x" in all $u_i$'s are replaced by "exp"
  - Re-run any test case whose relevant slice contains a $u_i$

# Relaxing Assumptions, Continued

- Changing left-hand side of assignment
  S: “x = exp” → “y = exp”
  - Effect: original assignment deleted and new assignment added
  - Re-run any test case that would need to be re-run under either of the two deletion/addition changes

- Deleting a predicate:
  - Effect: predicate is changed to the constant “true”
  - Re-run any test case whose relevant slice contains the deleted predicate

# Relaxing Assumptions, Continued

- Adding a predicate “pred” at location “loc”
  - Let $s_1$, $s_2$, ..., $s_n$ be the set of statements that are control-dependent on “pred”
  - Effect: all $s_i$’s used to be executed when control reached “loc”, but now some or all may be prevented from execution depending upon outcome of “pred”
  - Re-run any test case whose relevant slice contains an $s_i$
- Moving a statement from one location to another
  - Effect: statement deleted from original location and added at the new location
  - Re-run any test case that would need to be re-run under either of the two deletion/addition changes

# Test Case Prioritization Using Relevant Slices

D. Jeffrey and N. Gupta

*Computer Software and Applications Conference,* 1993

# Test Case Prioritization

- Definition: The Test Case Prioritization Problem is to *order* the tests in a test suite so that faults can be revealed as early as possible during testing

- Key Idea: The test cases that are *more likely to reveal faults* should be run *before* test cases that are less likely to reveal faults

# How to Prioritize Test Cases?

- Earlier approaches: order test cases based on their total/additional coverage of testing requirements
  - Does not account for whether the executed testing requirements actually influence the program output
  - A test case may be given higher priority than it should, if it exercises many testing requirements but only few of them can actually affect the output
- Key Observation: modifications that may affect program output for a test case should affect some computation in the *relevant slice* of the program output for that test case
- Key Idea: Take relevant slicing information into account!

# Motivating Example

```
1:  read(a, b, c);
2:  int x := 0, y := 0, z := 0;
3:  x := a + 1;
4:  y := b + 1;
5:  z := c + 1;
6:  int w := 0;
B1: if x > 3 then
B2:   if z > 4 then
7:      w := w + 1;
      endif
    endif
B3: if y > 5 then
8:    w := w + 1;
    endif
9:  write(w);
```

Execution Slice
for test case T:
(a, b, c) = (1, 5, 4)

```
1:  read(a, b, c);
2:  int x := 0, y := 0, z := 0;
* 3:  x := a + 1;
4:  y := b + 1;
5:  z := c + 1;
6:  int w := 0;
* B1: if x > 3 then
B2:    if z > 4 then
7:       w := w + 1;
       endif
     endif
B3: if y > 5 then
8:    w := w + 1;
     endif
9:  write(w);
```

Relevant Slice
for test case T:
(a, b, c) = (1, 5, 4)

- Suppose statement 3 is mistakenly modified to be `x := b + 1` instead of `x := a + 1`
- We expect that test case T will expose the error... and it does!
- Now suppose that statement 3 is mistakenly modified to be `y := a + 1` instead of `x := a + 1`
- This time, test case T does *not* expose the error!
- Conclusion: modifying a statement contained in the relevant slice of a test case *does not guarantee* that the output will be changed for that test case when it is run on the modified program

- Suppose next that statement 5 is mistakenly modified to be `z := b + 1` instead of `z := c + 1`

- We expect that T will *not* expose the error... and indeed it does not

- Now suppose that statement 5 is mistakenly modified to be `y := c + 1` instead of `z := c + 1`

- This time, test case T *does* expose the error!

- Conclusion: in the case of a modification that involves changing the left-hand side of an assignment, an exercised modification that is *outside* the relevant slice of a test case may *still* affect the output

# Prioritizing Tests using Relevant Slices

- Should consider 2 factors
  - # of req. in the relevant slice (a modification affecting the output should necessarily affect some computation in the relevant slice)
  - # of exercised req. that are *not* in the relevant slice (modifying a variable in the LHS of an assignment not in the relevant slice *could affect* a computation *in* the relevant slice)

- Approach: order test cases in decreasing order of weight, where:

  weight := # of requirements in the relevant slice +
  total # of exercised requirements

# Experimental Study

- Used the Siemens suite of programs

| Program | LOC | # Faulty Ver. | Test Pool Size | Description |
|---|---|---|---|---|
| tcas | 138 | 41 | 1608 | altitude separation |
| totinfo | 346 | 23 | 1052 | info accumulator |
| sched | 299 | 8 | 2650 | priority scheduler |
| sched2 | 297 | 10 | 2710 | priority scheduler |
| ptok | 402 | 7 | 4130 | lexical analyzer |
| ptok2 | 483 | 10 | 4115 | lexical analyzer |
| replace | 516 | 32 | 5542 | pattern substitution |

- Generated and prioritized 1000 test suites for each program using 2 approaches:
  - REG+OI+POI: Our weighting approach that uses relevant slicing info
  - REG: An approach that simply orders test cases in decreasing order of total requirement coverage

# Evaluation of Prioritized Test Cases

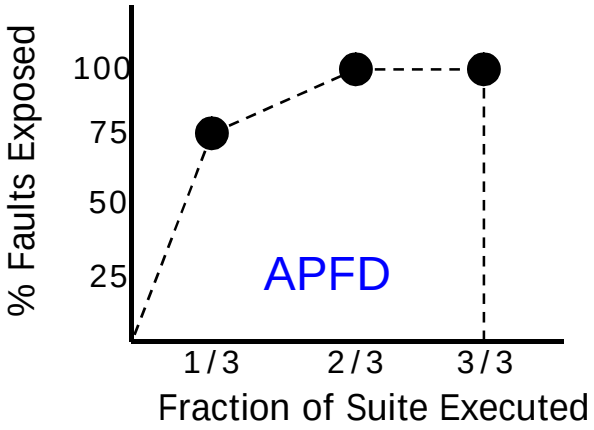
- APFD value ("average percentage of faults detected"): a measure of the rate of fault detection of a test suite
  - Imagine plotting the fraction of test suite executed (x-axis) versus the percentage of faults exposed (y-axis); the area under the curve is the APFD value
  - Higher APFD value is better, because it indicates more faults revealed when a smaller fraction of the test suite has been exercised

# Example of APFD

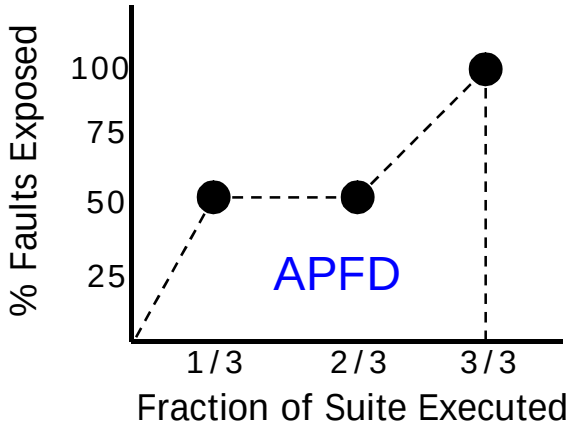
Suppose there are 3 test cases (T1 – T3) and 4 faults (F1 – F4):

| Test Case | Faults Exposed |
|---|---|
| T1 | F1, F3, F4 |
| T2 | <none> |
| T3 | F2, F4 |

Prioritization A: (T1, T3, T2)



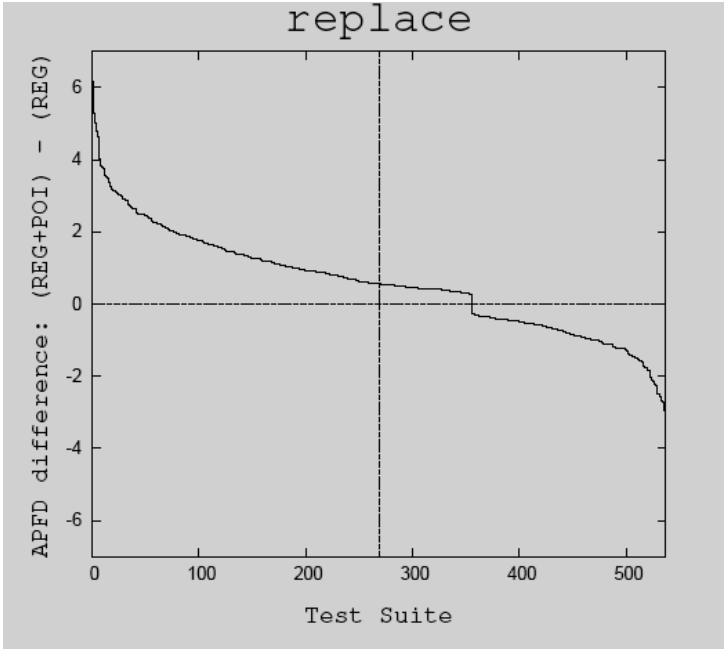

Prioritization B: (T3, T2, T1)



# Experimental Results

- Compare suites improved vs. worsened using REG+OI+POI instead of REG
- For each suite, compute ($APFD_{(REG+OI+POI)} - APFD_{(REG)}$), order them in decreasing order, and plot them

Example plot: program “replace” when prioritizing with respect to statement coverage



# Experimental Results, Continued

- Ratio of the area under the curve *above* line y=0, to the area under the curve *below* y=0

| Program Name | Ratio "area above : area below" | |
|---|---|---|
| | using statements | using branches |
| tcas | 1.51 | 1.88 |
| totinfo | 1.96 | 2.26 |
| sched | 1.42 | 1.55 |
| sched2 | 1.55 | 2.07 |
| ptok | 1.10 | 1.12 |
| ptok2 | 7.77 | 1.28 |
| replace | 2.64 | 1.82 |

# Experimental Results, Continued

- % of test suites with APFD better/worse/same when using REG+OI+POI as opposed to REG

| Program Name | % of Suites Better/Worse/Same using REG+OI+POI | | | | | |
|---|---|---|---|---|---|---|
| | using statements | | | using branches | | |
| | better | worse | same | better | worse | same |
| tcas | 27.3 | 18.8 | 53.9 | 25.6 | 16.3 | 58.1 |
| totinfo | 24.4 | 16.3 | 59.3 | 26.3 | 13.0 | 60.7 |
| sched | 20.1 | 15.1 | 64.8 | 10.7 | 7.5 | 81.8 |
| sched2 | 9.6 | 6.6 | 83.8 | 6.9 | 3.3 | 89.8 |
| ptok | 9.7 | 8.7 | 81.6 | 12.1 | 11.0 | 76.9 |
| ptok2 | 27.5 | 4.4 | 68.1 | 14.2 | 10.7 | 75.1 |
| replace | 35.5 | 18.4 | 46.1 | 28.1 | 16.2 | 55.7 |

# Variation of REG+OI+POI

- Group test cases into two sets, then apply REG+OI+POI to each set individually
  - Set 1: test cases that exercise at least 1 modification
  - Set 2: test cases that do not exercise any modification
- All test cases in set 1 ordered *before* those in set 2
- Significant improvement seen for “ptok” and “ptok2”, due to relatively high # of tests per suite not exercising any modification (for the other programs, almost all test cases exercised at least one modification)

# Final Observations

- Amount of improvement in APFD values relatively small across many test suites
- Likely due to the structure of the Siemens programs and the types of modifications

| Program Name | % Exercised Modifications Leading to Exposed Fault |
|---|---|
| tcas | 5.19 |
| totinfo | 19.44 |
| sched | 9.98 |
| sched2 | 3.96 |
| ptok | 15.14 |
| ptok2 | 26.76 |
| replace | 5.46 |

Intuition that "traversing a modification will likely lead to a different output" does not strongly apply to the Siemens subjects