Test Case Prioritization

Xun Yuan
CMSC 838M
11/16/04

Goal of Test Case Prioritization

Test case prioritization schedule test cases in order to increase their ability to meet some performance goal:

- Rate of fault detection
- Rate of code coverage
- Rate of increase of confidence in reliability

Test Case Prioritization

Given: T, a test suite;
PT, the set of permutations of T;
(all possible prioritizations of T)
f, a function from PT to a real number (award value)
Problem: find T'∈PT such that ∀T''∈PT,
T'? T'', f(T')≥f(T'')

Outline

- Measuring Effectiveness
  - Award function
- Prioritization Technique
- Evaluation

Incorporating Varying Test Costs and Faults Severities into Test Case Prioritization

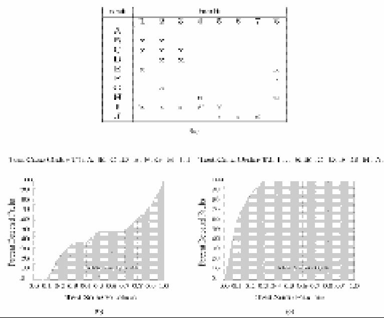
Sebastian Elbaum
Alexey G. Malishevsky
Gregg Rothermel
2001

Average of the Percentage of Faults Detected (APFD)

- T: test suite containing n test cases
- F: set of m faults revealed by T
- TFi: the first test case in ordering T' of T which reveals fault i

$$APFD = 1 - \frac{TF1 + TF2 + \ldots + TFm}{nm} + \frac{1}{2n}$$

Average of the Percentage of Faults Detected (APFD)



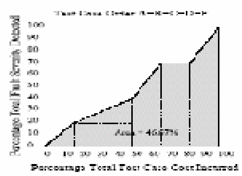
"Cost-cognizant" APFDc

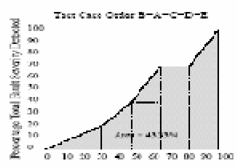
- No assumption of equal severity of faults
- No assumption of equal costs of test cases
- Reward test case orders proportionally to their rate of unit of fault severity detected per unit test cost

APFDc

- A-B-C-D-E vs.
- B-A-C-D-E

test	fault 1	2	3	4	5	6	7	8	9	10
A	x				x					
B						x	x			
C	x	x	x	x	x	x	x			
D					x					
E								x	x	x



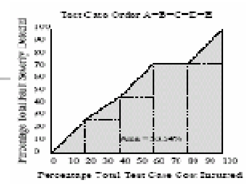


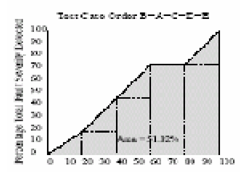
APFDc

- A-B-C-D-E vs.
- B-A-C-D-E

(Assume fault 2-10 have severity k, fault 1 has severity 2k)

test	fault 1	2	3	4	5	6	7	8	9	10
A	x				x					
B						x	x			
C	x	x	x	x	x	x	x			
D					x					
E								x	x	x





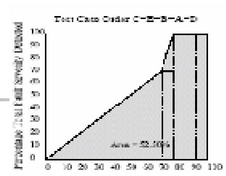
APFDc

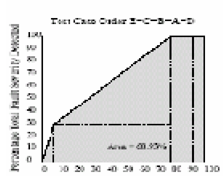
- C-E-B-A-D vs.

  E-C-B-A-D

  (Assume test case C is high-cost)

test	fault 1	2	3	4	5	6	7	8	9	10
A	x				x					
B						x	x			
C	x	x	x	x	x	x	x			
D					x					
E								x	x	x





Conclusion

- APFDc incorporates varying test case costs and fault severity
- APFDc better assesses the rate of fault detection of prioritized test cases

Effectively Prioritizing Tests in Development Environment

Amitabh Srivastava
Jay Thiagarajan

PPRC, Microsoft Research

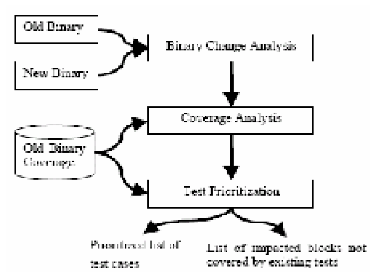
Effectiveness of various prioritizing techniques

- Source code differencing
  - Simple and fast
  - Can be built using commonly available tools
  - Will fail when macro definition changes or renaming
  - Static analysis is needed
- Data and control flow analysis
  - Flow analysis is difficult in languages like C/C++ with pointers, casts and aliasing
  - Interprocedural data flow techniques are extremely expensive and difficult to implement in complex environment

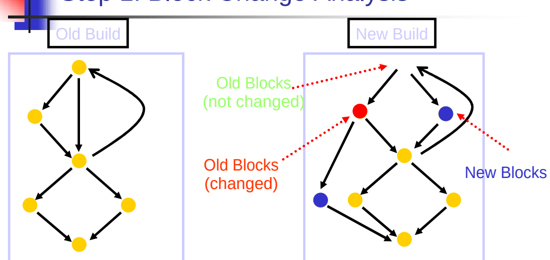
Echelon : Basic Idea

- Focus on change from previous version
  - Determine change at very fine granularity – basic block/instruction
- Operates on binary code
  - Easier to integrate in production environment
  - Scales well to compute results in minutes
- Simple heuristic algorithm to predict which part of code is impacted by the change

Echelon : Test Prioritization



Step 1: Block Change Analysis

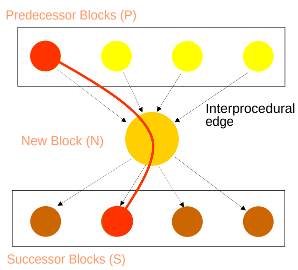


BMAT – Binary Matching [Wang, Pierce and McFarling JILP 2000]

Step 2: Coverage Impact Analysis

- Determine which "Impacted Blocks" (old modified and new blocks) in the new version are likely to be covered by an existing test case.
- Compute the coverage of test cases for the new build
  - Coverage for old (unchanged and modified) blocks are same as the coverage for the old build
  - Coverage for new nodes requires more analysis

Coverage Impact Analysis



- A sequence of test cases may cover a new block N if it covers at least one Predecessor block and at least one Successor Block
- If P or S is a new block, then its Predecessors or successors are used (iterative process)

Coverage Impact Analysis

- Limitations - New node may not be executed
  - If there is a path from predecessor to successor that does not go through the new block
  - If there are changes in control path due to data changes

Step 3: Test Prioritization

- Uses the impacted block set for each test case as a basis for prioritization
- Iteratively finds a short sequence of test cases which cover the maximum amount of impacted blocks

```
Input:
TestList: set of tests
Coverage (t) : set of blocks covered by  test t
ImpactedBlkSet: set of new and old modified blocks

Output: a set of sequences Seq

Algorithm
while (any t in TestList covers any block in ImpactedBlkSet
)
{
   CurrBlkSet = ImpactedBlkSet
   Start a new sequence Seq
   while (any t in TestList covers any block in CurrBlkSet )
   {
      for each t in TestList compute
      {
            Weight(t) = count[CurrBlkSet ∩ Coverage(t)]
      }
      Select test t in TestList with maximum weight
      Add t to current sequence Seq
      Remove t from TestList
      CurrBlkSet = CurrBlkSet − Coverage (t)
   }
}
Put all remaining tests in TestList in a new sequence Seq
```

Empirical Evaluation

- Performance
  - Test sequence characteristics
  - Predicated blocked coverage accuracy
- Effectiveness
  - Early detection of defects when tests run in prescribed, prioritized order

Performance

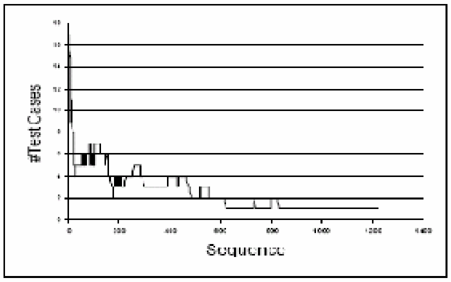
Table 1. Program information

	Version 1	Version 2
Date	December 2000	January 2001
Functions	31,020	31,026
Blocks	668,068	668,274
File Size (bytes)	8,880,128	8,880,128
PDB Size (bytes)	22,602,752	22,651,904
No. of Tests	3128	3128

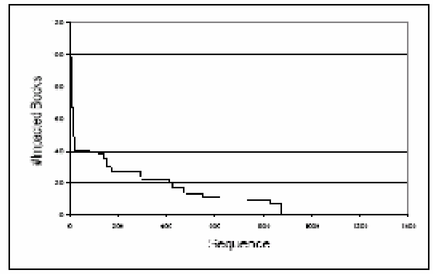
Table 2. Program change information

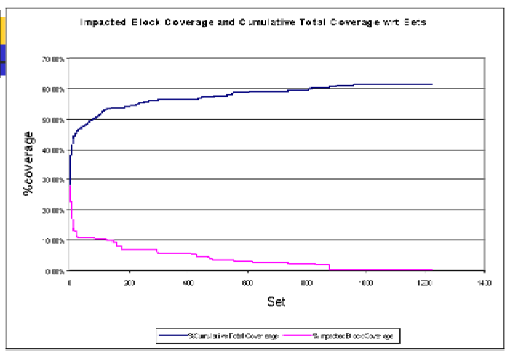
Impacted Blocks	378 (220 New, 158 old)
Impacted Blocks covered	176 Blocks
Tests in first sequence	16 Tests
Number of sequences	1,225
Time taken by Echelon	210 seconds

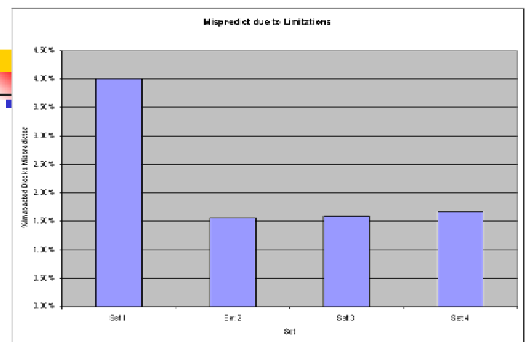
Number of Test Cases in each Sequence



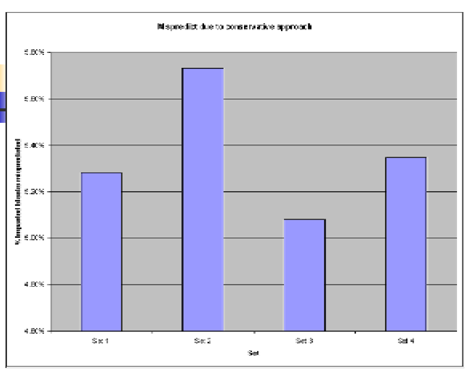
Number of impacted blocks in each sequence







Blocks predicted hit that were not hit

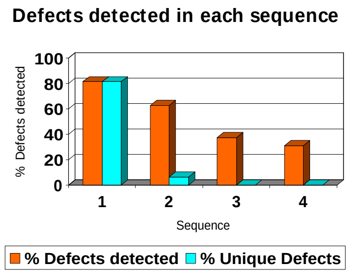


Blocks predicted not hit that were actually hit
(Blocks were target of indirect calls are being predicted as not hit)

Effectiveness of Echelon

- Important Measure of effectiveness is early fault detection
- Measured % of faults vs. % of unique faults in each sequence
- Unique faults are faults not detected by the previous sequence

Effectiveness of Echelon



Summary

- Binary based test prioritization approach can effectively prioritize tests in large scale development environment
- Simple heuristic with program change in fine granularity works well in practice
- Currently integrated into Microsoft Development process

Test Case Prioritization: A Family of Empirical Studies

Sebastian Elbaum
Alexey G. Malishevsky
Gregg Rothermel

2001

Test Case Prioritization Techniques

- Comparator(2)
- Statement Level (4)
- Function level (12)

Comparator Techniques(T1-T2)

- T1:Random ordering
  Lower bound on the performance
- T2:Optimal ordering
  Ordered to optimize rate of fault detection
  Upper bound on the performance

Statement Level Techniques(T3-T6)

Total
- Ordering independent of execution
- Prioritize on total coverage

Additional
- Ordering based on feedback
- Prioritize on coverage not yet covered

- T3:Total statement coverage
- T4:Additional statement coverage

Statement Level Techniques (Cont.)

FEP:Fault Exposing Potential

$$ms(s, t) = \frac{\text{mutants of s exposed by t}}{\text{total mutants of s}}$$

(s: statement; t: test case)

- T5:Total FEP prioritization
- T6:Additional FEP prioritization

Function Level Techniques(T7-T18)

Function Level:
- Worst-case cost analogous to statement level technique
- Less expensive
- Less intrusive

- T7:Total function coverage
- T8:Additional function coverage
- T9:Total FEP (function level) prioritization
- T10:Addition FEP (function level) prioritization

Function Level Techniques (Cont.)

FI: Fault Index
- Each function is assigned a fault index representing the fault proneness based on
  - Function complexity
  - Complexity of changes introduced in the function
- Computational cost is smaller than FEP

- T11:Total FI
- T12:Additional FI
- T13:Total FI with FEP coverage
- T14:Additional FI with FEP coverage

Function Level Techniques (Cont.)

DIFF:
- Cheaper estimation fault proneness
- Syntactic difference:
  - measure the degree of change by adding the number of lines inserted, deleted or changed

- T15:Total DIFF
- T16:Additional DIFF
- T17:Total DIFF with FEP
- T18:Additional DIFF with FEP

Prioritization Techniques Summary

- Granularity: efficiency vs. effectiveness
- Feedback
- Information from the modified program version
  - Techniques based solely on coverage information rely on solely on data gathered on the original version of a program
  - FEP factor in the potential effects of modifications in general and FI utilizes information about modified program version
- Immediate practicality

Empirical Study: Motivation

- RQ1:Can prioritization improve the rate of fault detection?
- RQ2:Fine granularity or coarse granularity?
- RQ3:Can the use of predicators of fault proneness improve the rate of fault detection of prioritization techniques?

Controlled Experiments

- Programs
  - 8 C programs (Siemens & Space)
- Versions
  - First order & higher-order versions
- Test Suites
  - Randomly select test cases from test pool
  - Stopping Criteria: Branch Coverage
  - 50 test suites for each program

Experiment 1: Prioritization

- Statement level techniques (1a) functional level techniques (1b)
- APFD value calculations, with 29 versions 50 test suites per program and all prioritization techniques
- Statistical calculations to determine significance of difference in means

Experiment Results

- 1a

Grouping	Means	Techniques
A	80.733	st-fep-addtl
B	78.867	st-fep-total
B	78.178	st-total
C	76.077	st-addtl

- 1b

Grouping	Means	Techniques
A	77.453	fn-fep-addtl
A	76.957	fn-fep-total
A	76.928	fn-total
B	73.465	fn-addtl

Experiment 2: Granularity Effects

Pair-wise analysis of corresponding pairs

Grouping				Means	Techniques
A				80.733	st-fep-addtl
B				78.867	st-fep-total
B	C			78.178	st-total
	C	D		77.453	fn-fep-addtl
		D	E	76.957	fn-fep-total
		D	E	76.928	fn-total
			E	76.077	st-addtl
F				73.465	fn-addtl

Experiment 3: Adding Prediction of Fault Proneness

Grouping				Means	Techniques
A				79.479	fn-diff-fep-addtl
A				79.450	fn-diff-fep-total
A	B			78.712	fn-fi-fep-total
	B	C		78.167	fn-fi-fep-addtl
		C	D	77.453	fn-fep-addtl
		C	D	77.321	fn-fi-total
		C	D	77.057	fn-diff-total
			D	76.957	fn-fep-total
			D	76.928	fn-total
E				74.596	fn-fi-addtl
E				73.465	fn-addtl
F				67.666	fn-diff-addtl

Case Study: Objects of Study

- Grep & Flex – 5 versions each
  - Test suite: Category Partition Method
  - Faults: Manual seeding as a results of modifications (>20)
- QTB – 6 versions
  - Test execution takes 27 days
  - 139 test cases & 69% function coverage
  - Coverage information only functional

Case Study: Results

- Flex & Grep: Random prioritization performs better than most heuristics
- QTB: Mean APFD values for feedback based techniques is lesser than the mean APEF values of the techniques which do not use feedback
- Techniques using fault proneness did not produce substantial improvements

Conclusion

- Statistically significant improvements
  - Greater variance in case studies
  - Vary across programs
- Function level techniques quite close to statement level techniques
- Statistically significant but small & inconsistent improvements using fault proneness