Regression Testing

- Developed first version of software
- Adequately tested the first version
- Modified the software; version 2 now needs to be tested
- How to test version 2?
- Approaches
  - Retest entire software from scratch
  - Only test the changed parts, ignoring unchanged parts since they have already been tested
  - Could modifications have adversely affected unchanged parts of the software?

Regression Testing

- "Software maintenance task performed on a modified program to instill confidence that changes are correct and have not adversely affected unchanged portions of the program."

Regression Testing vs. Development Testing

- During regression testing, an established test set may be available for reuse
- Approaches
  - Retest all
  - Selective retest (selective regression testing) ← Main focus of research

Formal Definition

- Given a program P,
- its modified version P', and
- a test set T
  - used previously to test P
- find a way, making use of T to gain sufficient confidence in the correctness of P'

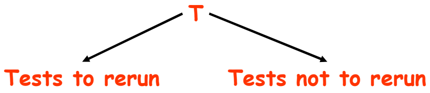
Regression Testing Steps

1. Identify the modifications that were made to P
   - Either assume availability of a list of modifications, or
   - Mapping of code segments of P to their corresponding segments in P'
2. Select T' ⊆ T, the set of tests to re-execute on P'
   - May need results of step 1 above
   - May need test history information, i.e., the <u>input</u>, <u>output</u>, and <u>execution history</u> for each test

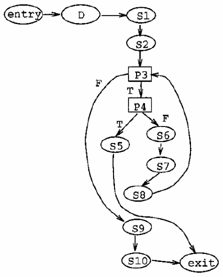
Regression Testing Steps

3. Retest P' with T'
   - Use expected output of P, if same
4. Create new tests for P', if needed
   - Examine whether coverage criterion is achieved
5. Create T''
   - The new test suite, consisting of tests from steps 2 and 4, and old tests that were not selected

Selective Retesting



- **Tests to rerun**
  - Select those tests that will produce different output when run on P'
    - Modification-revealing test cases
    - It is impossible to always find the set of modification-revealing test cases – (we cannot predict when P' will halt for a test)
  - Select modification-traversing test cases
    - If it executes a new or modified statement in P' or misses a statement in P' that it executed in P



```
        Procedure avg

S1.     count = 0
S2.     fread(fileptr,n)
P3.    while (not EOF) do
P4.       if (n<0)
S5.          return(error)
          else
S6.          numarray[count] = n
S7.          count++
          endif
S8.       fread(fileptr,n)
       endwhile
S9.    avg = calcavg(numarray,count)
S10.   return(avg)
```

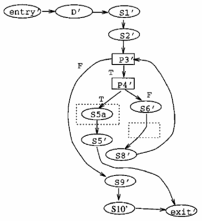
Fig. 1. Procedure avg and its CFG.

Table I. Test Information and Test History for Procedure avg

Test Information			
Test	Type	Output	Edges Traversed
t1	Empty File	0	(entry, D), (D, S1), (S1, S2) (S2, P3) (P3, S9), (S9, S10), (S10, exit)
t2	−1	Error	(entry, D) (D, S1), (S1, S2), (S2, P3), (P3, P4), (P4, S5), (S5, exit)
t3	1 2 3	2	(entry, D) (D, S1), (S1, S2), (S2, P3), (P3, P4), (P4, S6), (S6, S7), (S7, S8), (S8, P3), (P3, S9), (S9, S10), (S10, exit)

Test History

Edge	TestsOnEdge(edge)
(entry, D)	111
(D, S1)	111
(S1, S2)	111
(S2, P3)	111
(P3, P4)	011
(P3, S9)	101
(P4, S5)	010
(P4, S6)	001
(S5, exit)	010
(S6, S7)	001
(S7, S8)	001
(S8, P3)	001
(S9, S10)	101
(S10, exit)	101

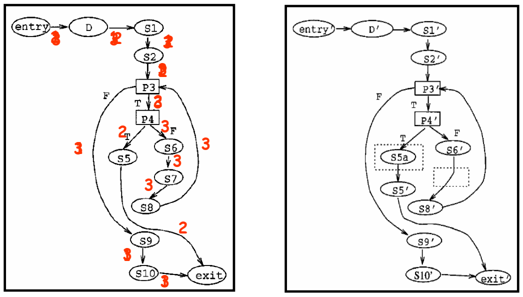


```
Procedure avg2

S1'. count = 0
S2'. fread(fileptr,n)
P3'. while (not EOF) do
P4'.    if (n<0)
S5a.       print("bad input")
S5'.       return(error)
        else
S6'.       numarray[count] = n

        endif
S8'.    fread(fileptr,n)
     endwhile
S9'. avg = calcavg(numarray,count)
S10'.return(avg)
```

Fig. 3. Procedure avg2 and its CFG.

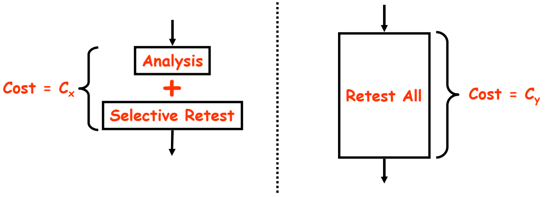
```
     Procedure avg

S1.  count = 0
S2.  fread(fileptr,n)
P3.  while (not EOF) do
P4.     if (n<0)
S5.        return(error)
        else
S6.        numarray[count] = n
S7.        count++
        endif
S8.     fread(fileptr,n)
     endwhile
S9.  avg = calcavg(numarray,count)
S10. return(avg)
```

```
Procedure avg2

S1'. count = 0
S2'. fread(fileptr,n)
P3'. while (not EOF) do
P4'.    if (n<0)
S5a.       print("bad input")
S5'.       return(error)
        else
S6'.       numarray[count] = n

        endif
S8'.    fread(fileptr,n)
     endwhile
S9'. avg = calcavg(numarray,count)
S10'.return(avg)
```



T' = {t2, t3}

Cost of Regression Testing



We want $C_x < C_y$

Key is the test selection algorithm/technique

We want to maintain the same "quality of testing"

Selective-retest Approaches

- Coverage-based approaches
  - Rerun tests that could produce different output than the original program. Use some coverage criterion as a guide
- Minimization approaches
  - Minimal set of tests that must be run to meet some structural coverage criterion
    - E.g., every program statement added to or modified for P' be executed (if possible) by at least one test in T

Selective-retest Approaches

- Safe approaches
  - Select every test that may cause the modified program to produce different output than the original program
    - E.g., every test that when executed on P, executed at least one statement that has been deleted from P, at least one statement that is new in or modified for P'
- Data-flow coverage-based approaches
  - Select tests that exercise data interactions that have been affected by modifications
    - E.g., select every test in T, that when executed on P, executed at least one def-use pair that has been deleted from P', or at least one def-use pair that has been modified for P'

Selective-retest Approaches

- Ad-hoc/random approaches
  - Time constraints
  - No test selection tool available
    - E.g., randomly select n test cases from T

Factors to consider

- Testing costs
- Fault-detection ability
- Test suite size vs. fault-detection ability
- Specific situations where one technique is superior to another

Open Questions

- How do techniques differ in terms of their ability to
  - reduce regression testing costs?
  - detect faults?
- What tradeoffs exist b/w testsuite size reduction and fault detection ability?
- When is one technique more cost-effective than another?
- How do factors such as program design, location, and type of modifications, and test suite design affect the efficiency and effectiveness of test selection techniques?

Experiment

- Hypothesis
  - Non-random techniques are more effective than random techniques but are much more expensive
  - The composition of the original test suite greatly affects the cost and benefits of test selection techniques
  - Safe techniques are more effective and more expensive than minimization techniques
  - Data-flow coverage based techniques are as effective as safe techniques, but can be more expensive
  - Data-flow coverage based techniques are more effective than minimization techniques but are more expensive

Measure

- Costs and benefit of several test selection algorithms
- Developed two models
  - Calculating the cost of using the technique w.r.t. the retest-all technique
  - Calculate the fault detection effectiveness of the resulting test case

Modeling Cost

- Did not have implementations of all techniques
  - Had to simulate them
- Experiment was run on several machines (185,000 test cases) – results not comparable
- Simplifying assumptions
  - All test cases have uniform costs
  - All sub-costs can be expressed in equivalent units
    - Human effort, equipment cost

Modeling Cost

- Cost of regression test selection
  - Cost = A + E(T')
  - Where A is the cost of analysis
  - And E(T') is the cost of executing and validating tests in T'
  - Note that E(T) is the cost of executing and validating all tests, i.e., the retest-all approach
  - Relative cost of executing and validating = |T'|/|T|

Modeling Fault-detection

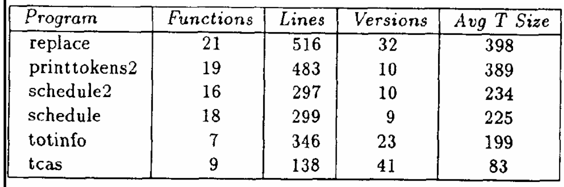
- Per-test basis
  - Given a program P and
  - Its modified version P'
  - Identify those tests that are in T and reveal a fault in P', but that are not in T'
  - Normalize above quantity by the number of fault-revealing tests in T
- Problem
  - Multiple tests may reveal a given fault
  - Penalizes selection techniques that discard these test cases (i.e., those that do not reduce fault-detection effectiveness)

Modeling Fault-detection

- Per-test-suite basis
  - Three options
    - The test suite is inadequate
      - No test in T is fault revealing, and thus, no test in T' is fault revealing
    - Same fault detection ability
      - Some test in both T and T' is fault revealing
    - Test selection compromises fault-detection
      - Some test in T is fault revealing, but no test in T' is fault revealing
- 100 - (Percentage of cases in which T' contains no fault-revealing tests)

Experimental Design

- 6 C programs
- Test suites for the programs
- Several modified versions



Program	*Functions*	*Lines*	*Versions*	*Avg T Size*
replace	21	516	32	398
printtokens2	19	483	10	389
schedule2	16	297	10	234
schedule	18	299	9	225
totinfo	7	346	23	199
tcas	9	138	41	83

Table 1: Experimental Subjects.

Test Suites and Versions

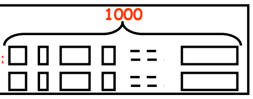
- Given a test pool for each program
  - Black-box test cases
    - Category-partition method
  - Additional white-box test cases
    - Created by hand
    - Each (executable) statement, edge, and def-use pair in the base program was exercised by at least 30 test cases
- Nature of modifications
  - Most cases single modification
  - Some cases, 2-5 modifications

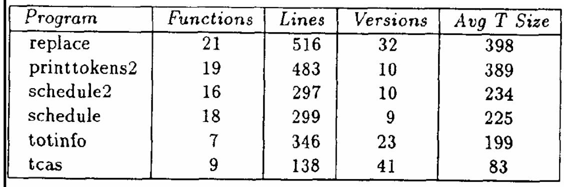
Versions and Test Suites

- Two sets of test suites for each program
  - Edge-coverage based
    - 1000 edge-coverage adequate test suites
    - To obtain test suite T, for program P (from its test pool): for each edge in P's CFG, choose (randomly) from those tests of pool that exercise the edge (no repeats)
  - Non-coverage based
    - 1000 non-coverage-based test suites
    - To obtain the kth non-coverage based test suite, for program P: determine n, the size of the kth coverage-based test suite, and then choose tests randomly from the test pool for P and add them to T, until T contains n test cases

Another look at the subjects

- For each program
- 1000 edge-coverage based test suites:
- 1000 non-coverage based test suites:





Program	*Functions*	*Lines*	*Versions*	*Avg T Size*
replace	21	516	32	398
printtokens2	19	483	10	389
schedule2	16	297	10	234
schedule	18	299	9	225
totinfo	7	346	23	199
tcas	9	138	41	83

Table 1: Experimental Subjects.

Test Selection Tools

- Minimization technique
  - Select a minimal set of tests that cover modified edges
- Safe technique
  - DejaVu
    - we discussed the details earlier in this lecture
- Data-flow coverage based technique
  - Select tests that cover modified def-use pairs
- Random technique
  - Random(n) randomly selects n% of the test cases
- Retest-all

Variables

- The subject program
  - 6 programs, each with a variety of modifications
- The test selection technique
  - Safe, data-flow, minimization, random(25), random(50), random(75), retest-all
- Test suite composition
  - Edge-coverage adequate
  - random

Measured Quantities

- Each run
  - Program P
  - Version P'
  - Selection technique M
  - Test suite T
- Measured
  - The ratio of tests in the selected test suite T' to the tests in the original test suite
  - Whether one or more tests in T' reveals the fault in P'

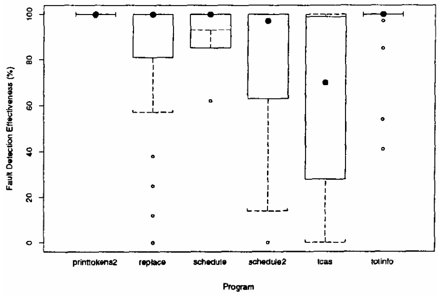
Dependent variables

- Average reduction in test suite size
- Fault detection effectiveness
  - 100-Percentage of test suites in which T' does not reveal a fault in P'

Number of runs

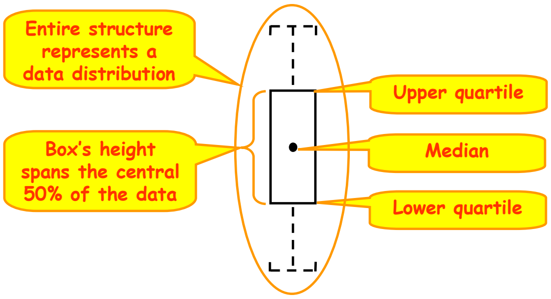
- For each subject program, from the test suite universe
  - Selected 100 edge-coverage adequate
  - And 100 random test suites
- For each test suite
  - Applied each test selection method
  - Evaluated the fault detection capability of the resulting test suite

Fault-detection Effectiveness



100-Percentage of test suites in which T' does not reveal a fault in P'

How to read the graphs



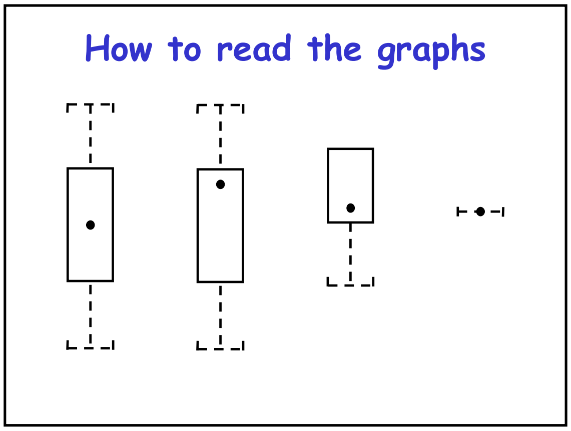
Entire structure represents a data distribution

Box's height spans the central 50% of the data

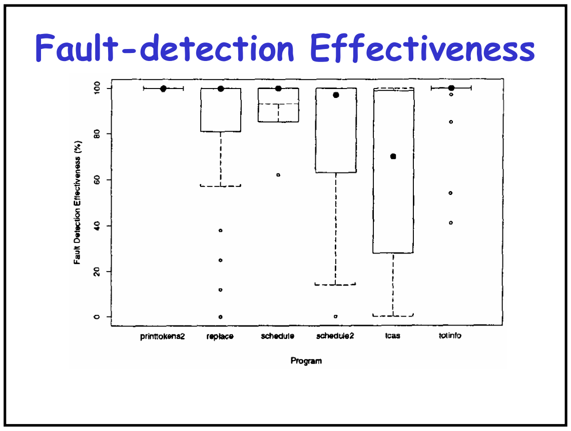
Upper quartile

Median

Lower quartile



How to read the graphs



Fault-detection Effectiveness

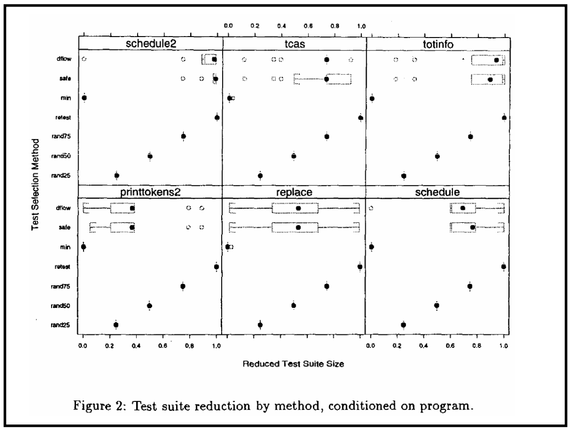


Figure 2: Test suite reduction by method, conditioned on program.

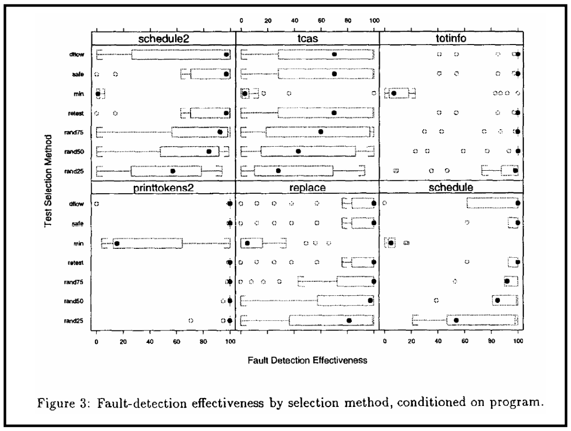


Figure 3: Fault-detection effectiveness by selection method, conditioned on program.

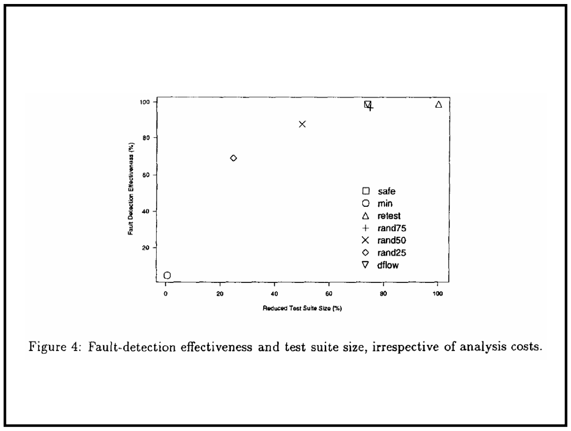
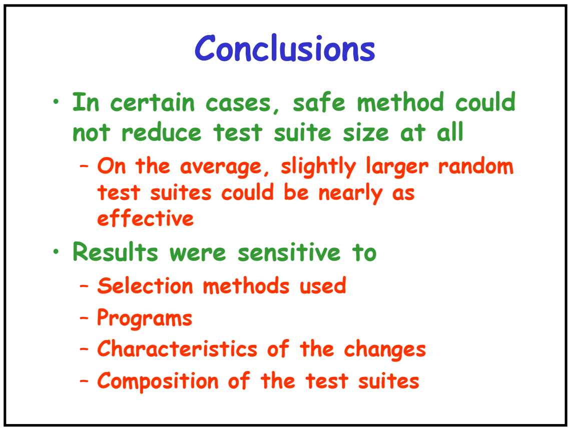


Figure 4: Fault-detection effectiveness and test suite size, irrespective of analysis costs.

Conclusions

- Minimization produces the smallest and the least effective test suites
- Random selection of slightly larger test suites yielded equally good test suites as far as fault-detection is concerned
- Safe and data-flow nearly equivalent average behavior and analysis costs
  - Data-flow may be useful for other aspects of regression testing
- Safe methods found all faults (for which they has fault-revealing tests) while selecting (average) 74% of the test cases



Conclusions

- In certain cases, safe method could not reduce test suite size at all
  - On the average, slightly larger random test suites could be nearly as effective
- Results were sensitive to
  - Selection methods used
  - Programs
  - Characteristics of the changes
  - Composition of the test suites