Technical Report 99-60-01, Computer Science Department, Oregon State University, January 1999

Regression Test Selection for C++ Software

Authors:
Gregg Rothermel
Mary Jean Harrold
Jeinay Dedhia

Sanjay Kulhari
PhD Student, Computer Science Department
UC Riverside

Regression test selection

- Given:
  - P : A method, class or a program.
  - T: Test suite to test P.
  - P’: Modified version of P

- Problem definition: Given P, T and P’, choose an appropriate subset of T that executes the new or modified code and tests the formerly executed code that has now been deleted.

Motivation

- Modified code should behave as expected and should not break the behavior of unmodified code.
- Time spent on test selection should be minimal and combined time of selection and execution should not exceed time for testing all the existing tests for previous version.
- Regression testing can be expensive in object oriented paradigm due to code reuse, so efficient test selection can be very beneficial.

Outline

- Background
  - Regression testing/ Regression testing in object oriented software.
  - CFG/ICFG/Code Instrumentation
- Regression test selection technique for
  - modified application programs
  - modified and derived classes
- OOP features handled by the test selection technique.
- Experimental results
- Related work
- Conclusion and Future work

Regression testing

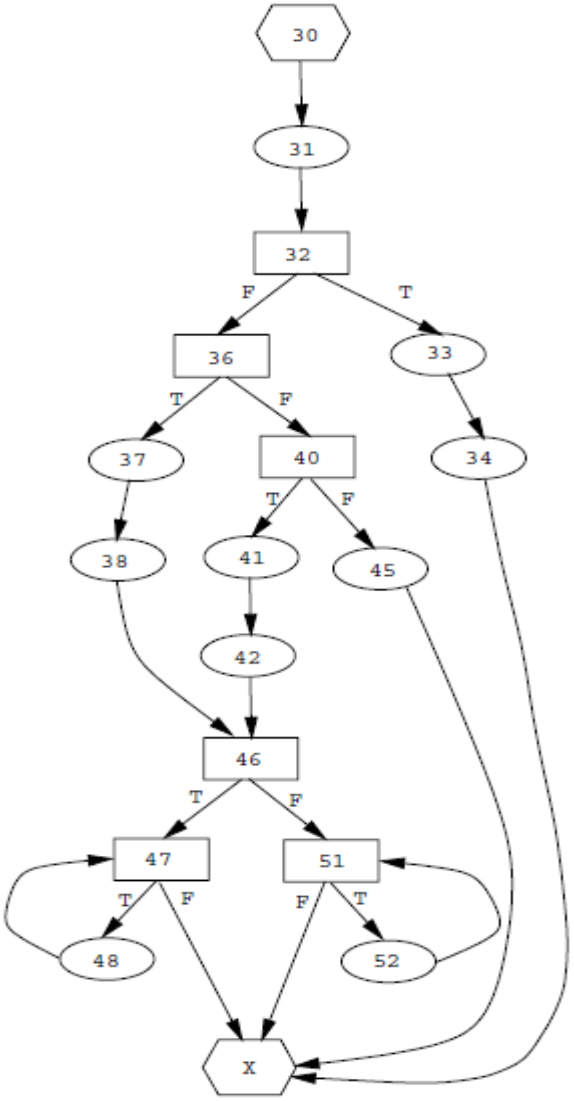
- Regression test selection
  - Select a subset of existing test cases.
- Coverage identification
  - Create additional tests to cover new functionality.
- Test suite execution
  - Execute tests to establish correctness
- Test suite maintenance
  - Create the new test suite and test history.

Regression testing in OO software

- Testing modified class
  - Test driver invokes sequence of methods and verifies that objects have attained proper states.
- Testing dependent application programs
  - Test application programs that use the modified class.
- Testing derived classes
  - Test classes derived from the modified class.

Control Flow Graph

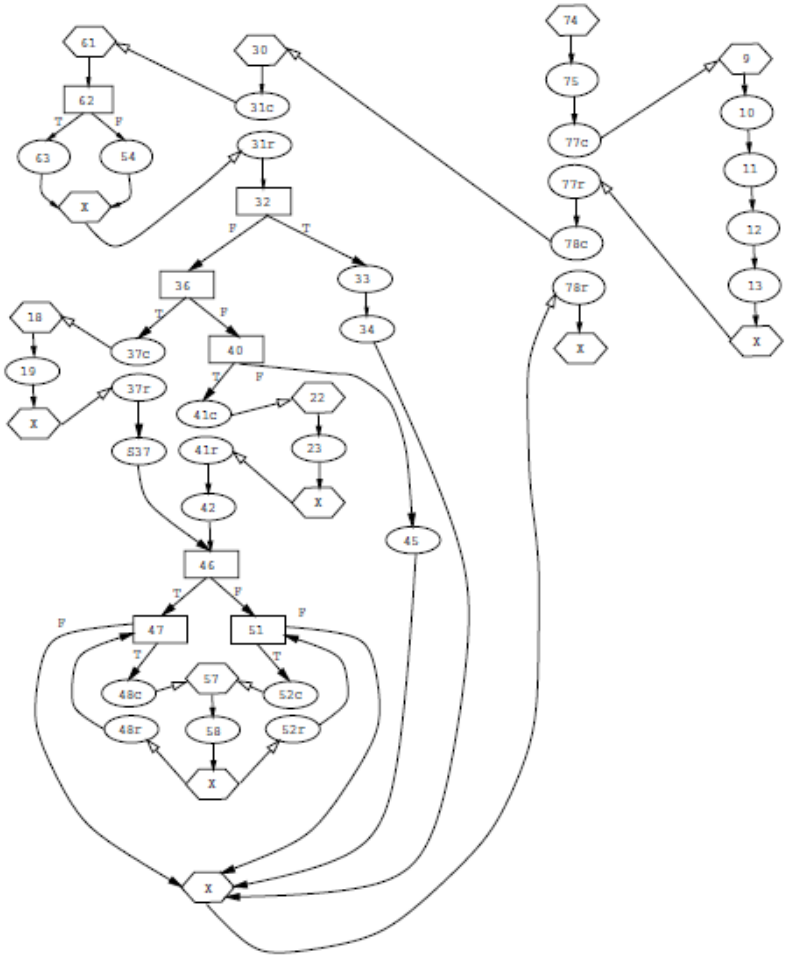
```
30   virtual void go (int floor) {
31     int valid = valid_floor(floor);
32     if (!valid_floor(floor)) {
33       cout << "Invalid floor request\n";
34       return;
35     }
36     if (floor > current_floor) {
37       up();
38       cout << "Elevator is going up";
39     }
40     else if (floor < current_floor) {
41       down();
42       cout << "Elevator is going down";
43     }
44     else
45       return;
46     if (current_direction == UP) {
47       while ((current_floor != floor)
              && (current_floor <= top_floor))
48         add(current_floor, 1);
49     }
50     else {
51       while ((current_floor != floor)
              && (current_floor <= bottom_floor))
52         add(current_floor, -1);
53     }
54   };
```



Interprocedural Control Flow Graph

```
1  #include <iostream.h>
2  #include <stdlib.h>
3  #define UP 1
4  #define DOWN 2
5  typedef int Direction;
6
7  class Elevator {
8  public:
9     Elevator (int l_top_floor)  {
10      current_floor = 1;
11      current_direction = UP;
12      top_floor = l_top_floor;
13      bottom_floor = 1;
14    }
15
16    virtual ~Elevator() {}
17
18    void up() {
19       current_direction = UP;
20    }
21
22    void down() {
23       current_direction = DOWN;
24    }
25
26    Direction direction() {
27       return current_direction;
28    }
29
```

```
55
56 private:
57    add(int &a, const int &b) {
58       a = a+b;
59    }
60
61    int valid_floor(int floor) {
62       if ((floor > top_floor) ||
                (floor < bottom_floor)
63          return 0;
64       return 1;
65    };
66
67 protected:
68    int current_floor;
69    Direction current_direction;
70    int top_floor;
71    int bottom_floor;
72 };
73
74 void main (int argc, char **argv) {
75    Elevator *e_ptr;
76
77    e_ptr = new Elevator(10);
78    e_ptr->go(2);
79 }
```



UNIVERSITY OF CALIFORNIA UCRIVERSIDE

Code instrumentation

- Branch trace
  - Given a program *P* with ICFG *G*, execution of instrumented version of P with test *t* gives branches taken during execution.

- Edge trace
  - Using branch trace determine the edges in G, that were traversed when t was executed.
  - Edge trace for a test t on P is linear in size with number of edges in G.

Code instrumentation

- Test History
  - Gather edge trace information for each test in T such that for each test, a set of traversed edges (n1,n2) is recorded.

test	top_floor	bottom_floor	current_floor	floor	edge trace
t1	10	1	1	-1	(30,31),(31,32),(32,33),(33,34),(34,X)
t2	10	1	1	5	(30,31),(31,32),(32,36),(36,37),(37,38),(38,46),(46,47),(47,48),(48,47),(47,X)
t3	10	1	5	2	(30,31),(31,32),(32,36),(36,45),(45,41),(41,42),(42,46),(46,51),(51,52),(52,51),(51,X)
t4	10	1	2	2	(30,31),(31,32),(32,36),(36,40),(40,45),(45,X)

  - Method **TestOnEdge**(n1,n2) returns the test cases that traverse edge (n1,n2)

Test selection technique

- Approach
  - Traverse ICFGs of original and modified program to look for nodes that are not equivalent (modification traversing)
  - Using test history, select all tests that have reached that point.
  - All tests are considered at once and no separate traversals for each test.
  - Nodes are marked ‘visited’ and algorithm terminates in time proportional to graph size.

Test selection algorithm (SelectTests)

```
algorithm   SelectTests(P,P',T):T'
input       P,P': base and modified versions of a program
            T: a test set used to test P
output      T': the subset of T selected for use in regression testing P'
global      E: a subset of the edges in the ICFG for P

1.  begin
2.      T' = ∅
3.      E = ∅
4.      construct G and G', ICFGs for P and P', with entry nodes e and e'
5.      Compare( e, e' )
6.      for each edge (n1, n2) ∈ E do
7.          T' = T' ∪ TestsOnEdge((n1,n2))
8.      return T'
9.  end

procedure   Compare(N,N')
input       N and N': nodes in G and G'

10. begin
11.     mark N "N'-visited"
12.     if ¬ OutEdgesEquivalent(N,N')
13.         for each successor C of N in G do
14.             E = E ∪ (N,C)
15.         endfor
16.     else
17.         for each successor C of N in G do
18.             L = the label on edge (N,C) or ε if (N,C) is unlabeled
19.             C'=the node in G' such that (N',C') has label L
20.             if C is not marked "C'-visited"
21.                 if ¬ NodesEquivalent( C,C' )
22.                     E = E ∪ (N,C)
23.                 else
24.                     Compare( C, C' )
25.                 endif
26.             endif
27.         endfor
28.     endif
29. end
```

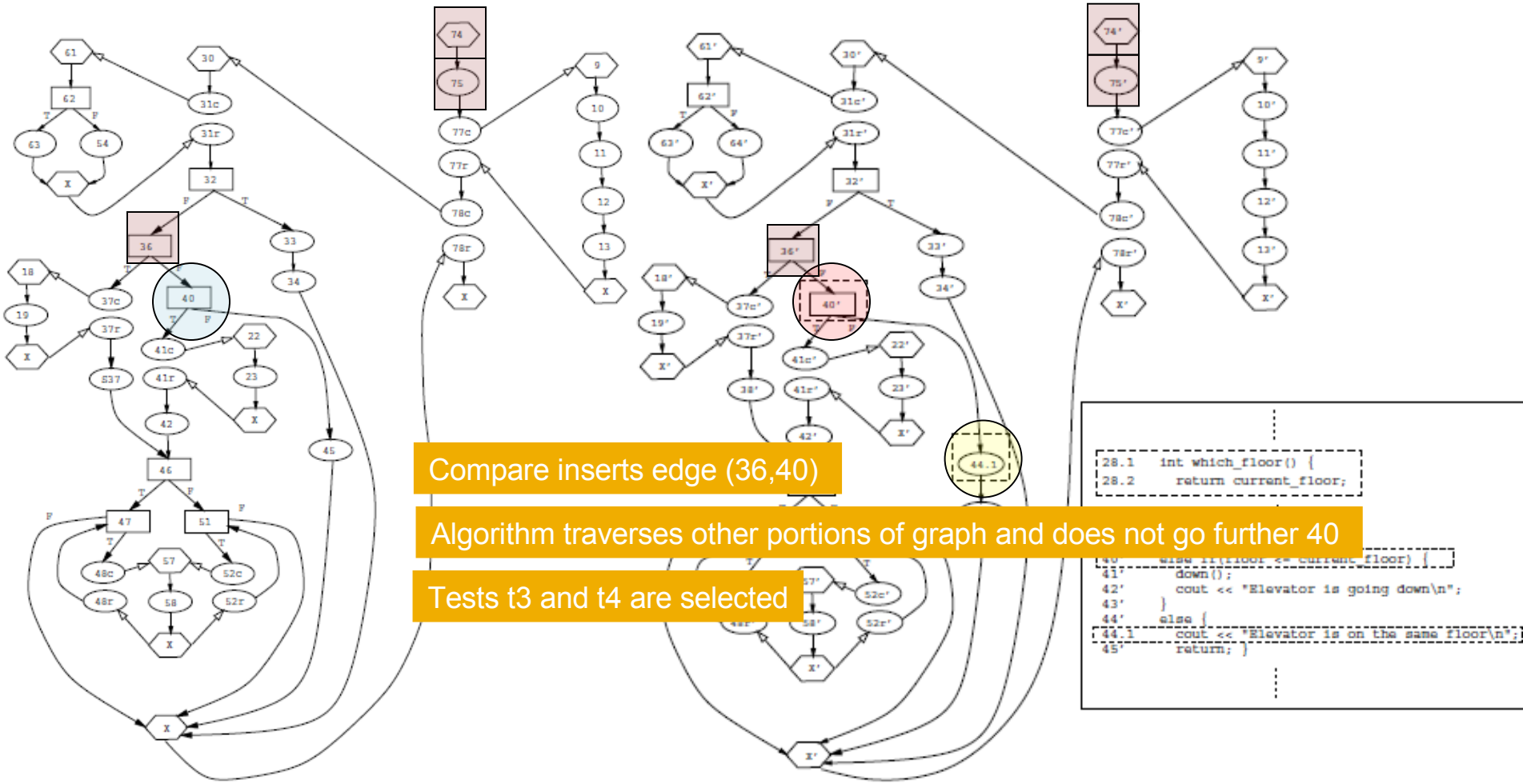
UC RIVERSIDE UNIVERSITY OF CALIFORNIA

Test selection algorithm (SelectTests)

- Input: Program P, modified version P’ and test suite T for P.
- Output: T’ a subset of T that contains tests that are modification traversing for P and P’.

- Processing
  - Constructs ICFGs for P and P’
  - Traverse the graphs recursively using compare method to get edges through which tests are modification traversing.
  - Use TestOnEdge method to retrieve tests from the test history.

Test selection algorithm (SelectTests)



Test selection algorithm (SelectTests)

- Performance
  - Cost(SelectTests) = Cost(ICFG construction for P and P') + Cost(Compare) + Cost(set unions)

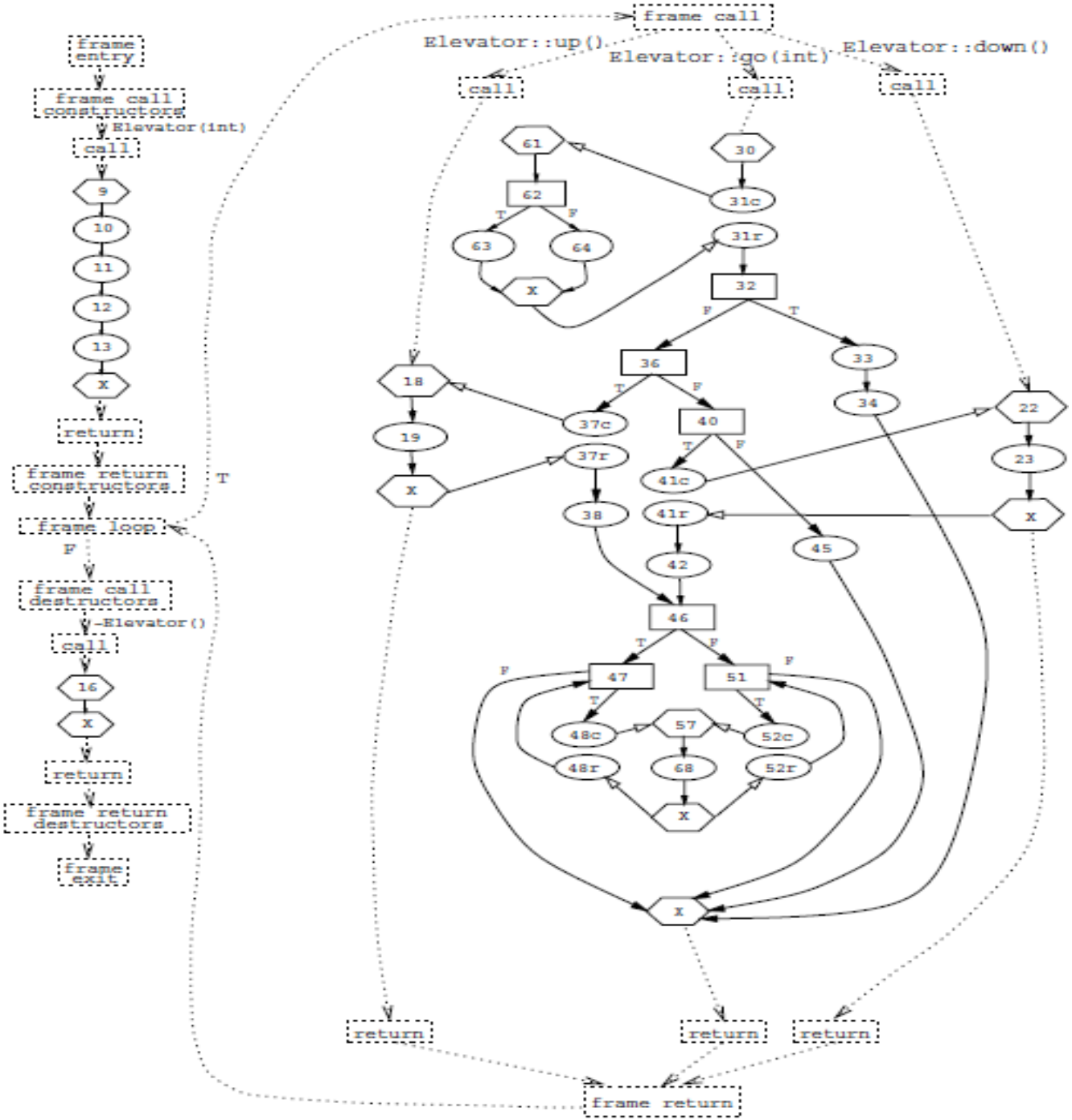
  = $O(n + n' + nn' + n|T'|)$

Regression test selection for modified and derived classes

- Class can have multiple entry points therefore previous technique doesn’t work.
- Naïve approach
  - Create driver programs and use SelectTests algorithm.
  - Disadvantage: Unnecessary construction and traversal of each driver’s ICFG.
- New representation of C++ class
  - Class Control Flow Graph (CCFG)

Class Control Flow Graph (CCFG)

- Collection of individual control flow graphs for the methods in a class.
- Frame
  - Abstraction of a driver program, to simulate arbitrary sequence of calls to public methods.
- Nodes of individual CFGs are connected with frame to give CCFG.


frame entry
frame call constructors
Elevator(int)
call
9
10
11
12
13
X
return
frame return constructors
T
frame loop
F
frame call destructors
~Elevator()
call
16
X
return
frame return destructors
frame exit
frame call
Elevator::up()
Elevator::go(int)
Elevator::down()
call
call
call
61
62
T
F
63
64
X
30
31c
31r
32
F
T
36
33
T
F
18
19
X
37c
37r
38
40
T
F
41c
41r
42
34
22
23
X
45
46
T
F
47
51
F
F
T
T
48c
48r
57
68
X
52c
52r
X
return
return
return
frame return

CCFGs and SelectTests

- SelectTests can be run on CCFGs of modified or derived classes to select regression test.
- SelectTests is invoked on the two versions of CCFGs for the base class when a method is modified.
- When a derived class redefines base class's method SelectTests is invoked on CCFGs of base and derived class.
- If test suite T is available for derived class and the base class is modified, SelectTests is run on CCFGs of the derived classes.

Other issues

- Interclass and Intraclass testing
  - Test selection for interclass can be done in similar way by including the CFGs of other classes.
- Polymorphism and dynamic binding
  - Build ICFGs that include *polymorphic call nodes* and edges to other possible CFGs
- Objects as parameters
  - Similar to handling polymorphism, build ICFGs that include *polymorphic call nodes* and edges to other possible CFGs

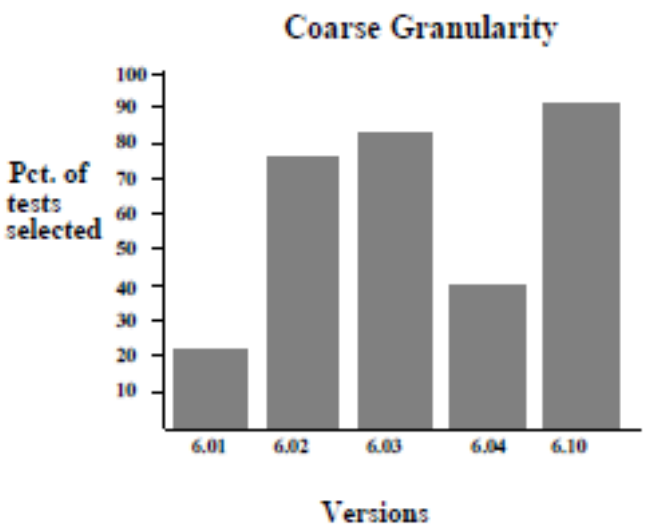
Other issues

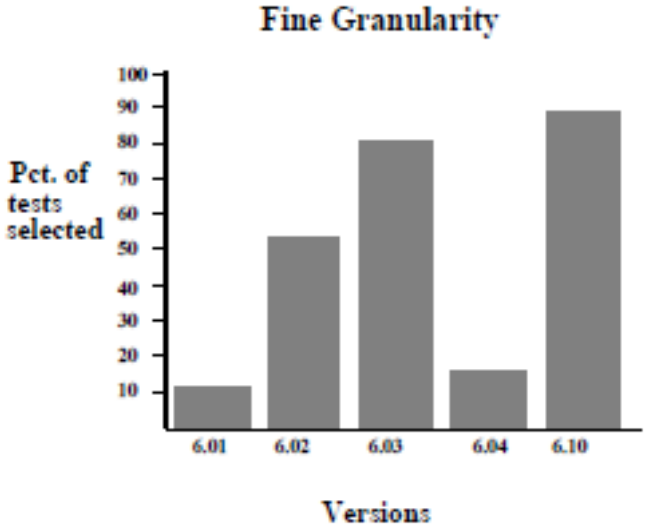
- Handling changes in non executable statements
  - Mark affected statements that refer to variables whose declaration is changed.
- Distinguishing driver, setup and Oracle code from code under test.
  - Test the setup methods independently.
- Specification and code based testing
  - Black box selection technique should be used in conjunction to select test relevant to changed specification.

Experimental results

- Setup
  - Experimented with 6 versions of commercial C++ library.
  - 186 classes, 24849 lines of code.
  - 61 C++ driver programs (test cases)
  - Used simulation technique, because C++ analyzer to develop CFG for the code is not available.

Experimental results





Follow up study

- Categorized modifications as due to
  - Constructors
  - Operators
  - Other

- Collected test selection data for different modifications
  - On two versions constructor and operator changes accounted for 22 – 35 % so in those cases it is better to test them separately.

Related work

- Program dependence graph
  - Construction of CFGs is costly as compared to SelectTests.
- ORD (Object relational Diagram)
  - Describes static relationship among classes.
  - Determines all classes exercised by test cases.
  - Less precise than SelectTests.

Future work

- To obtain empirical data on effects of polymorphism on graph size and algorithm runtime.
- To empirically investigate the approach to handle non executable statements.
- To identify if the changes have made existing test cases inadequate and new test cases are needed.

Questions