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.
virtual void go (int floor) {
    int valid = valid_floor(floor);
    if (!valid_floor(floor)) {
        cout << "Invalid floor request\n";
        return;
    }
    if (floor > current_floor) {
        up();
        cout << "Elevator is going up";
    } else if (floor < current_floor) {
        down();
        cout << "Elevator is going down";
    } else {
        return;
        if (current_direction == UP) {
            while ((current_floor != floor)
                && (current_floor <= top_floor))
                add(current_floor, 1);
        } else {
            while ((current_floor != floor)
                && (current_floor <= bottom_floor))
                add(current_floor, -1);
        }
    }
}
Interprocedural Control Flow Graph

```cpp
#include <iostream.h>
#include <stdlib.h>
#define UP 1
#define DOWN 2
typedef int Direction;

class Elevator {
public:
    Elevator(int l_top_floor) {
        current_floor = 1;
        current_direction = UP;
        top_floor = l_top_floor;
        bottom_floor = 1;
    }

    virtual ~Elevator() {};

    void up() {
        current_direction = UP;
    }

    void down() {
        current_direction = DOWN;
    }

    Direction direction() {
        return current_direction;
    };

private:
    add(int a, const int b) {
        a = a+b;
    }

    int valid_floor(int floor) {
        if (floor > top_floor ||
            (floor < bottom_floor)
        return 0;
        return 1;
    }

    protected:
        int current_floor;
        Direction current_direction;
        int top_floor;
        int bottom_floor;

    void main (int argc, char **argv) {
        Elevator *e_ptr;
        e_ptr = new Elevator(10);
        e_ptr->go(2);
    }
```
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 $G$. 
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.

- 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 e 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. end
29. end
```
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)

Compare inserts edge (36, 40)
Algorithm traverses other portions of graph and does not go further 40
Tests t3 and t4 are selected
Performance

\[ \text{Cost(SelectTests)} = \text{Cost(ICFG construction for } P \text{ and } P') + \text{Cost(Compare)} + \text{Cost(set unions)} \]

\[ = O(n + n' + nn' + n|T'|) \]
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.
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

Coarse Granularity

<table>
<thead>
<tr>
<th>Versions</th>
<th>Pct. of tests selected</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.01</td>
<td>10</td>
</tr>
<tr>
<td>6.02</td>
<td>20</td>
</tr>
<tr>
<td>6.03</td>
<td>50</td>
</tr>
<tr>
<td>6.04</td>
<td>80</td>
</tr>
<tr>
<td>6.10</td>
<td>100</td>
</tr>
</tbody>
</table>

Fine Granularity

<table>
<thead>
<tr>
<th>Versions</th>
<th>Pct. of tests selected</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.01</td>
<td>10</td>
</tr>
<tr>
<td>6.02</td>
<td>20</td>
</tr>
<tr>
<td>6.03</td>
<td>50</td>
</tr>
<tr>
<td>6.04</td>
<td>80</td>
</tr>
<tr>
<td>6.10</td>
<td>100</td>
</tr>
</tbody>
</table>
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