Regression Testing

What is Regression Testing?

- A testing activity to make sure that:
  - Not only the newly added or modified code behaves correctly,
  - But also code carried over unchanged from the previous version continues to behave correctly.

<table>
<thead>
<tr>
<th>Version 1</th>
<th>Version 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Develop $P$</td>
<td>4. Modify $P$ to $P'$</td>
</tr>
<tr>
<td>2. Test $P$</td>
<td>5. Test $P'$ for new functionality</td>
</tr>
<tr>
<td>3. Release $P$</td>
<td>6. Perform regression testing on $P'$ to ensure that the code carried over from $P$ behaves correctly</td>
</tr>
<tr>
<td>7. Release $P'$</td>
<td></td>
</tr>
</tbody>
</table>

What Tests to Use?

- Idea 1
  - All valid tests from the previous version and new tests created to test added functionality. (TEST-ALL approach)
  - The test-all approach is best when you want to be certain that the new version works on all tests developed for the previous version and any new tests.
  - But what if you have limited resources to run tests and have to meet a deadline?
  - What if running all tests as well as meeting the deadline is simply not possible?

- Idea 2
  - Select a subset $T_r$ of the original test set $T$ such that successful execution of the modified code $P'$ against $T_r$ implies that all the functionality carried over from the original code $P$ to $P'$ is intact.
  - Finding $T_r$ is accomplished by test selection, minimization and test prioritization.
Regression Test Selection Problem

- Given test set \( T \), our goal is:
  - To determine \( T_r \), such that successful execution of \( P' \) against \( T_r \) implies that modified or newly added code in \( P' \) has not broken the code carried over from \( P \).
- Note
  - Some tests might become obsolete when \( P \) is modified to \( P' \).
  - Such tests are not included in the regression subset \( T_r \). The task of identifying such obsolete tests is known as test revalidation.

Test Selection using Execution Trace

- Step 1: Given \( P \) and test set \( T \), record the execution trace of \( P \) for each test in \( T \).
- Step 2: Extract test vectors from the execution traces for each node in the CFG of \( P \).
- Step 3: Construct syntax trees for each node in the CFGs of \( P \) and \( P' \). This step can be executed while constructing the CFGs of \( P \) and \( P' \).
- Step 4: Traverse the CFGs and determine the a subset of \( T \) appropriate for regression testing of \( P' \).

Regression Test Process

We will learn how to select tests for regression testing.

Execution Trace

- Let \( G=(N, E) \) denote the CFG of program \( P \).
- \( N \) is a finite set of nodes and
- \( E \) a finite set of edges connecting the nodes.
- Suppose that nodes in \( N \) are numbered 1, 2, and so on and that Start and End are two special nodes.
- Let \( T_{\text{old}} \) be the set of all valid tests for \( P' \). Thus \( T_{\text{old}} \) contains only tests valid for \( P' \). It is obtained by discarding all tests that have become obsolete for some reason.
Execution Trace (cont.)

Example
- An execution trace of program P for some test t in Tₚₙ is the sequence of nodes in G traversed when P is executed against t.

Example
- An execution trace of program P for some test t in Tₚₙ is the sequence of nodes in G traversed when P is executed against t.

Example
- An execution trace of program P for some test t in Tₚₙ is the sequence of nodes in G traversed when P is executed against t.

Example
- An execution trace of program P for some test t in Tₚₙ is the sequence of nodes in G traversed when P is executed against t.

Syntax Tree
- A syntax tree is constructed for each node of CFG(P) and CFG(P').
  - Recall that each node represents a basic block.

Syntax Tree
- A syntax tree is constructed for each node of CFG(P) and CFG(P').
  - Recall that each node represents a basic block.

Syntax Tree
- A syntax tree is constructed for each node of CFG(P) and CFG(P').
  - Recall that each node represents a basic block.

Syntax Tree
- A syntax tree is constructed for each node of CFG(P) and CFG(P').
  - Recall that each node represents a basic block.

Test Selection
- Given the execution traces and the CFGs for P and P', the following three steps are executed to obtain a subset T' of T for regression testing of P'.

Step 1: Set T' = ∅. Unmark all nodes in G and in its child CFGs.
Step 2: Call procedure SelectTests (G, Start, G', Start'), where G and G' are, respectively, the start nodes in G and G'.
Step 3: T' is the desired test set for regression testing of P'.
Test Selection (cont.)

- **Idea**
  - Traverse the two CFGs from their respective START nodes using a recursive descent procedure.
  - The descent proceeds in parallel and the corresponding nodes are compared.
  - **If two** two nodes $N$ in $\text{CFG}(P)$ and $N'$ in $\text{CFG}(P')$ are found to be syntactically different, all tests in the test vector of the node $N$ – test $(N)$ – are added to $T'$.

Test Selection: Example

- Suppose that function $g_1$ in $P$ is modified as follows.

```java
1 int g1(int a, b) { ← Modified g1.
2 int a, b;
3 if(a==b) ← Predicates modified.
4 return(a*a),
5 else
6 return(b+b).
7 }
```

Try the SelectTests algorithm and check if you get $T' = \{t_1, t_3\}$.

Test Selection: Questions

- Try the SelectTests algorithm, assuming that $g_1$ line 4 is modified to $a*a*a$. $T' =$? Does the algorithm return only the tests related to the modified code?
- What tests will be selected when only, say, one declaration is modified?
- Can you think of a way to select only tests that correspond to variables in the modified declaration?

Test Selection: Declaration Changes

```java
1 main()
2 int x, y, z; ← Replaced by
3     int x, y; float z;
4     z=0;
5     input (x, y);
6     if (x>y)
7       [z=x+y; output(z)];
8     output ((float) (x-y));
9 }
```

$T = \{t_1 : <x = 1, y = 3>, t_2 : <x = 2, y = 1>, t_3 : <x = 3, y = 4>\}$

$T' =$?
Test Selection using Dynamic Slicing

- Execution trace based test selection can lead to including tests that are not really needed.

**Example**

```c
1 main()
2 { q, r } = 1, 3; r = 2;
3 y = 0;
4 input (p, q, r);
5 if (p < q)
6   z = 1; This statement is modified to z;
7 if (x = 1)
8   z = 2;
9 output (z);
10 end
11 }
```

Execution trace based = \( t_1, t_3 \)
But \( t_1 \) does not affect the outcome.

Dynamic Slice: Example

```c
1 main()
2 int p, q, r, z;
3 x = 0;
4 input (p, q, r);
5 if (p < q)
6   z = 1; This statement is modified to z;
7 if (x > 1)
8   z = 2;
9 output (z);
10 end
11 }
```

- Dynamic slice of \( P \) with respect to variable \( z \) at line 9 and test \( t \) consists of statements at lines 4, 5, 7, 8.
- Static slice of \( z \) at line 9 consists of statements at lines 3, 4, 5, 6, 7, 8.

Dynamic Dependence Graph (DDG)

- The DDG is needed to obtain a dynamic slice. Here is how a DDG \( G \) is constructed.
  - Step 1: Initialize \( G \) with a node for each declaration. There are no edges among these nodes.
  - Step 2: Add to \( G \) the first node in trace(\( t \)).
  - Step 3: For each successive statement in trace(\( t \)) a new node \( n \) is added to \( G \). Control and data dependence edges are added from \( n \) to the existing nodes in \( G \).
  - Recall the definitions of control and data dependence edges.
Consider the following program and a test t:
\[ <x=2, y=4> \]
- Assume successive values of \( x \) to be 2, 0 and 5 and for these \( f_1(x) \) is 0, 2, and 3 respectively.
- \( \text{trace}(t) = \{1, 2^1, 3^1, 4^1, 6^1, 7^1, 2^2, 3^2, 5^2, 6^2, 7^2, 2^3, 8\} \)
- Ignore declarations for simplicity.
- Add a node to \( G \) corresponding to statement 1.

**Construction of a DDG: Example (cont.)**

\( \text{trace}(t) = \{1, 2^1, 3^1, 4^1, 6^1, 7^1, 2^2, 3^2, 5^2, 6^2, 7^2, 2^3, 8\} \)
- Add another node for statement 2 in \( \text{trace}(t) \). Also add a data dependence edge from 2 to 1.
- Add yet another node for statement 3 in \( \text{trace}(t) \). Also add a data dependence edge from 3 to 1 and a control edge from node 3 to 2.

**Obtaining Dynamic Slice (DS)**

- Given program \( P \), test \( t \), variable \( v \), and location \( l \),
  - Step 1: Execute \( P \) against test \( t \) and obtain \( \text{trace}(t) \).
  - Step 2: Construct the DDG \( G \) from \( P \) and \( \text{trace}(t) \).
  - Step 3: Identify in \( G \) node \( n \) labeled \( L \) that contains the last assignment to \( v \). If no such node exists then the dynamic slice is empty, otherwise execute Step 4.
  - Step 4: Find in \( G \) the set \( DS(t, v, n) \) of all nodes reachable from \( n \), including \( n \). \( DS(t, v, n) \) is the dynamic slice of \( P \) with respect to \( v \) at location \( L \) and test \( t \).
Obtaining DS: Example

- Suppose we want to compute the dynamic slice of P with respect to variable w at line 8 and test t shown earlier.

![DDG Diagram]

- Identify the last definition of w in the DDG. This occurs at line 7 (see fig.)
- Traverse the DDG backwards from node 7 and collect all reachable nodes.
- DS: \{1, 2, 3, 5, 6, 7, 8\}.

Test Selection using Dynamic Slice

- Let:
  - T be the test set used to test P.
  - P' is the modified program.
  - \(n_1, n_2, \ldots, n_k\) be the nodes in the CFG of P modified to obtain P'.

- Which tests from T should be used to obtain a regression test T' for P'? 
  - Find DS(t) for P.
  - If any of the modified nodes is in DS(t) then add t to T'.

Test Selection using DS: Exercise

- Suppose line 4 in the program P is modified to obtain P'.
  - Should t be included in T'?
  - Will t be included in T' if we were to use the execution slice instead of the dynamic slice to make our decision?

![Exercise Diagram]

Test Minimization

- Test minimization is yet another method for selecting tests for regression testing.

- To illustrate test minimization,
  - Suppose that P contains two functions, main and f.
  - Now suppose that P is tested using test cases t1 and t2.
  - During testing it was observed that t1 causes the execution of main but not of f and t2 does cause the execution of both main and f.

- Now suppose that P' is obtained from P by making some modification to f.
  - Which of the two test cases should be included in the regression test suite?
Test Minimization (cont.)

- Test minimization is based on the coverage of testable entities in P.
  - Testable entities include, for example, program statements, decisions, def-use chains, and mutants.
- Procedure to minimize a test set
  - Step 1: Identify the type of testable entity to be used for test minimization. Let \( e_1, e_2, \ldots, e_k \) be the \( k \) testable entities of type TE present in P.
  - Step 2: Execute P against all elements of test set T and for each test \( t \) in T determine which of the \( k \) testable entities is covered.
  - Step 3: Find a minimal subset \( T' \) of T such that each testable entity is covered by at least one test in \( T' \).

Test Minimization: Example

- Step 1: Let the basic block be the TE of interest. The basic blocks for both main and function f1 are given.
- Step 2: Suppose the coverage of the basic blocks when executed against three tests is as follows:
  - t1: main: 1, 2, 3. f1: 1, 3
  - t2: main: 1, 3. f1: 1, 3
  - t1: main: 1, 3. f1: 1, 2, 3
- Step 3: A minimal test set for regression testing is \{t1, t3\}.

Test Prioritization

- Test minimization will likely discard test cases. It is possible that if P’ were executed against a discarded test case it would reveal an error in the modification made.
- When very high quality software is desired, it might not be wise to discard test cases. Instead, one can use test prioritization.
- Tests are prioritized based on some criteria.
  - For example, tests that cover the maximum number of a selected testable entity could be given the highest priority.

A Procedure for Test Prioritization

- Step 1: Identify the type of testable entity to be used for test minimization.
  - Let \( e_1, e_2, \ldots, e_k \) be the \( k \) testable entities of type TE present in P. In our previous example TE is function.
- Step 2: Execute P against all elements of test set T and for each test \( t \) in T.
  - For each \( t \) in T compute the number of distinct testable entities covered.
- Step 3: Arrange the tests in T in the order of their respective coverage.
  - Test with the maximum coverage gets the highest priority and so on.
Tools for Regression Testing

- Methods for test selection discussed require the use of an automated tool for all but trivial programs.
- xSuds from Telcordia Technologies can be used for C programs to minimize and prioritize tests.
- Many commercial tools for regression testing simply run the tests automatically
  - They do not use any of the algorithms described here for test selection.
  - Instead, they rely on the tester for test selection. Such tools are especially useful when all tests are to be rerun.

Summary

- Regression testing is an essential phase of software product development.
- In a situation where test resources are limited and deadlines are to be met, execution of all tests might not be feasible.
- In such situations one can make use of sophisticated technique for selecting a subset of all tests and hence reduce the time for regression testing.

Summary

- Test selection for regression testing can be done using any of the following methods:
  - Select only the modification traversing tests [based on CFGs].
  - Select tests using execution slices [based on execution traces].
  - Select tests using dynamic slices [based on execution traces and dynamic slices].
  - Select tests using code coverage [based on the coverage of testable entities].
  - Select tests using a combination of code coverage and human judgment [based on amount of the coverage of testable entities].