An Introduction to Regression Testing

Prof. Lionel Briand
Ph.D., IEEE Fellow

Objectives

- Problem definition
- Outline types of solutions
- Discuss situation in SOFIE
- References

Definition

Version 1

1. Develop P
2. Test P with test set T
3. Release P

Version 2

1. Modify P into P'
2. Test P' for new functionalities
3. Perform regression testing on P' to ensure that the code carried over from P behaves correctly
   - Main principle: reusing tests derived for P!
   - Identifying T', a subset of T
4. Release P'

Terminology

- Regress = returning to a previous, usually worse, state
- Regression testing
  - Portion of the testing life cycle
  - During which program P has been modified into program P' (maintenance)
  - P' needs specific testing attention to ensure that
    - newly added or modified code behaves correctly
    - code carried over unchanged, continues to behave correctly
- Important problem:
  - Regression testing constitute the vast majority of testing effort in many software development environments.

Regression Testing vs. Testing

Regression Testing is not a simple extension of testing

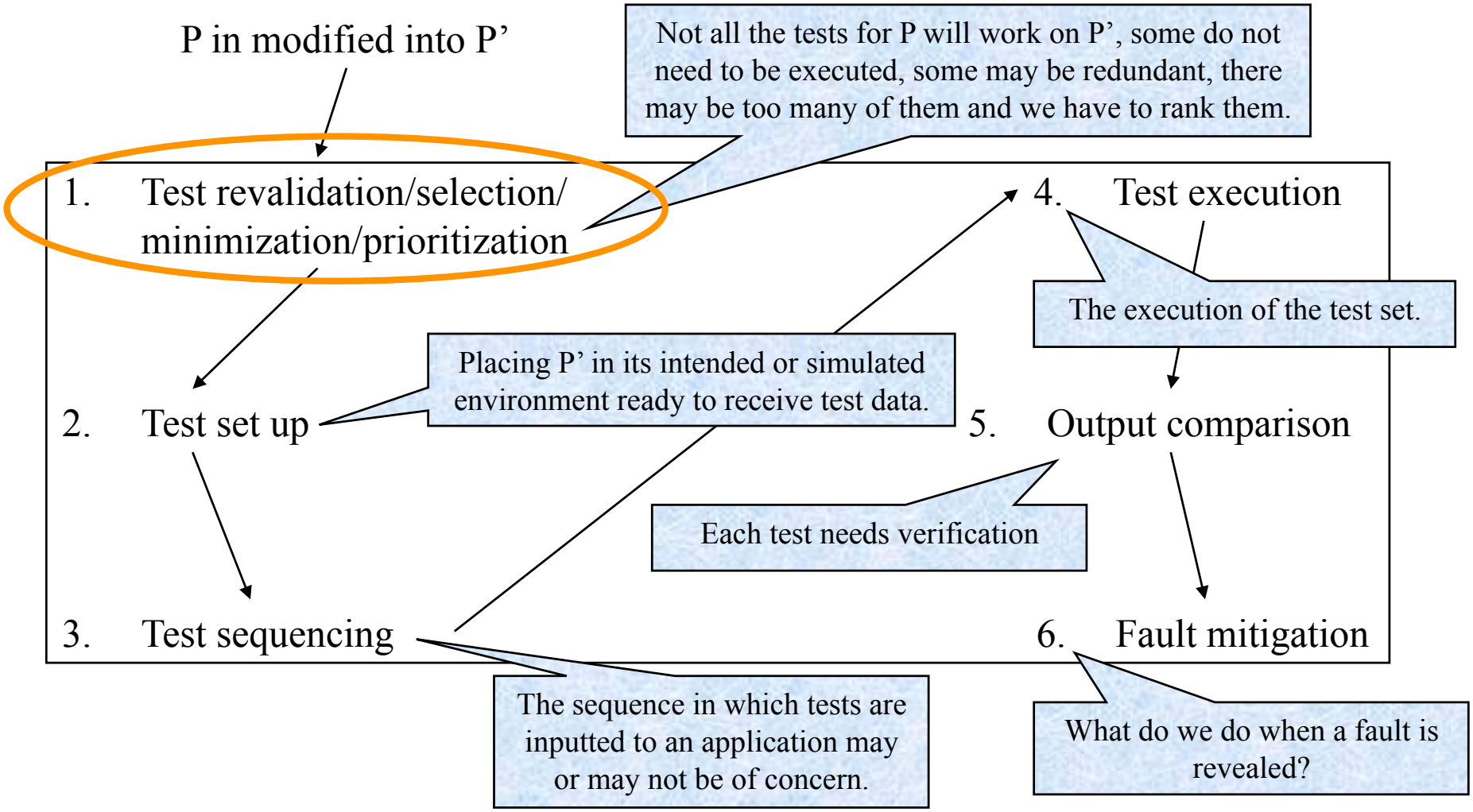
Main differences are:

1. Availability of test plan
   - Testing starts with a specification, an implementation of the specification and a test plan (black-box and/or white-box test cases). Everything is new
   - Regression testing starts with a (possibly modified) specification, a modified program, and an old test plan (which requires updating)
2. Scope of test
   - Testing aims to check the correctness of the whole program
   - Regression testing aims to check (modified) parts of the program
3. Time allocation
   - Testing time is normally budgeted before the development of a product (part of development costs)
   - Regression testing time should be accounted for in the planning of a new release of a product

Regression Testing vs. Testing (cont.)

4. Development information
   - During testing, knowledge about the development process is available (it is part of it)
   - During regression testing, the testers may not be the ones who developed/tested the previous versions!
5. Completion time
   - Completion time for regression testing should normally be much less than that for exhaustive testing (only parts are tested)
6. Frequency
   - Testing occurs frequently during code production
   - Regression testing occurs many times throughout the life of a product, possibly once after every modification is made to it

Regression Testing Process



Regression Testing Process (cont.)

- Test Revalidation
  - Checking which tests for P remain valid for P'.
  - To ensure that only test that are applicable to P' are used during regression testing.
  - How can we automatically identify Obsolete test cases?
- Test selection
  - Should we select and execute all tests in T ?
  - Test only modification-traversing tests?
  - Classification of tests:
    - Re-testable: should be re-run
    - Reusable: may not need to be re-run.

➢ Tests in T are either Obsolete (T_O), Re-testable (T_{RT}), or Reusable (T_{RU}).

Regression Testing Process (cont.)

- Test minimization
  - Discards tests seemingly redundant with respect to some criteria.
  - E.g., if two tests t_1 and t_2 execute function f, one might keep only t_2.
  - Purpose: reducing the number of tests to execute for regression testing.
- Test prioritization
  - Prioritizing tests based on some criteria
  - Context: after revalidation, selection, minimization, one may still have too many test cases (cannot afford to execute them all).
- Comments:
  - Minimization and prioritization can be considered selection techniques.
  - Minimization and prioritization techniques can be used during testing P (not necessarily during regression testing).

Regression Testing Process

- Test Revalidation
- (Regression) Test Selection
- Test Minimization
- Test Prioritization

Test Revalidation

- Three ways that a test case may become obsolete:
  - Program modification leads to (now) incorrect input /output relation
  - If P has been modified, some test cases may correctly specify the input/output relation, but may not be testing the same construct.
    - For example, the modifications from P to P' change some equivalence classes. t∈T used to exercise a boundary, which is no longer a boundary. t∈T is obsolete as it does no longer test a construct (boundary) of interest.
  - A structural test case t∈T may no longer contribute to the structural coverage of the program.

Regression Testing Process

- Test Revalidation
- (Regression) Test Selection
- Test Minimization
- Test Prioritization

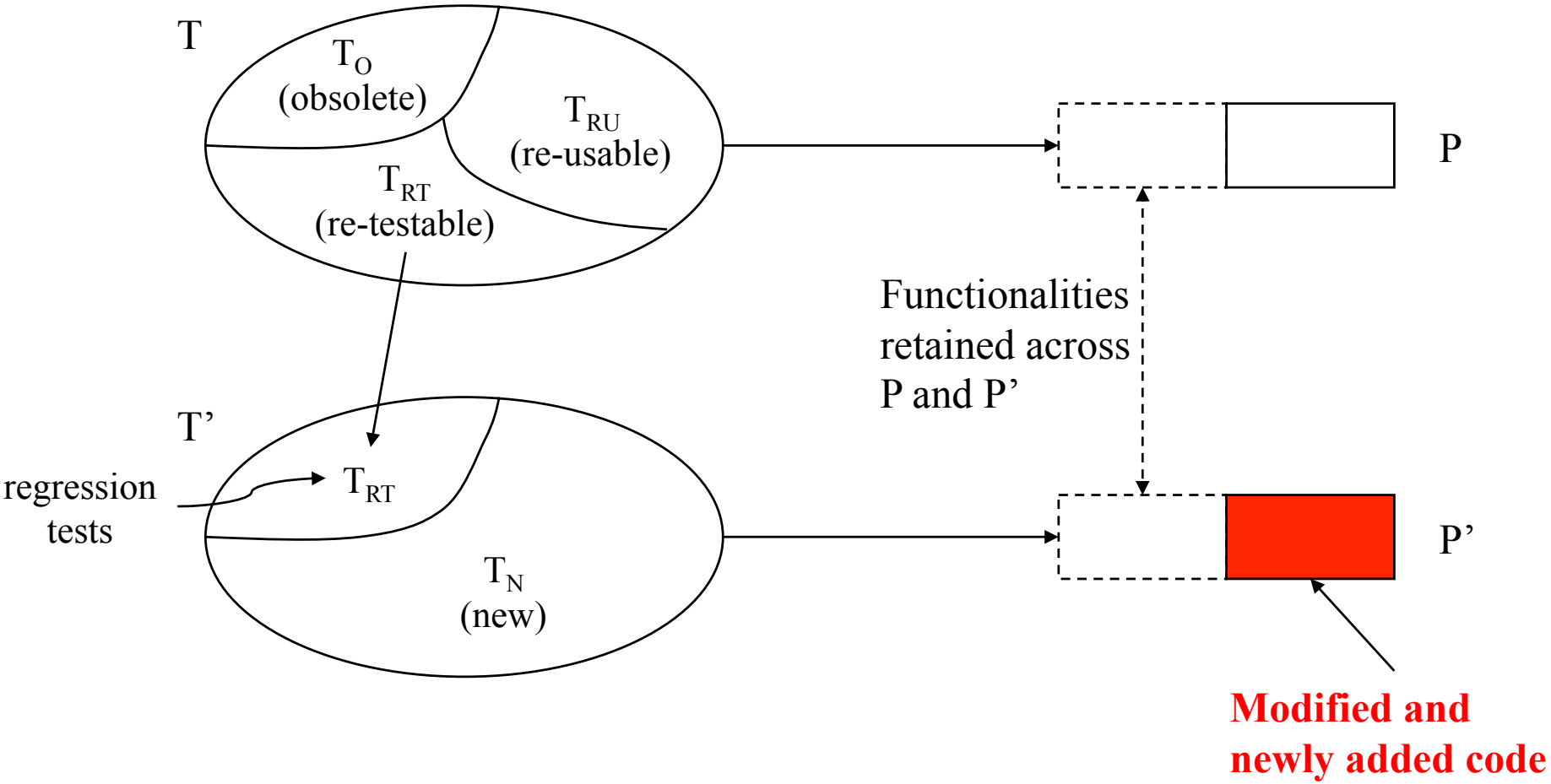
The Regression Test Selection Problem

- Let P denote the program that has been tested
  - Using test set T
  - Against specification S
- Let P' be generated by modifying P
  - P' must conform to S'
    - S and S' could be the same (e.g., corrective maintenance)
    - S⊂S' (e.g., S' contains new features, redefined features)
- Regression Testing Problem
  - Find a test set T_{RT} on which P' is to be tested to ensure that code that implements functionalities carried over from P work correctly.
- In addition P' must be tested to ensure that newly added code behaves correctly.

Regression Test Selection Techniques

- Test All
  - $T - T_O$
  - Often not a practical solution as $T - T_O$ is often too large (too many test cases to run in the amount of time allocated)
- Random Selection
  - Select randomly tests from $T - T_O$
  - The tester decides how many tests to select
  - May turn out to be better than no regression testing at all
  - But may not even select tests that exercise modified parts of the code!
- Selecting Modification Traversing Tests
  - Selecting a subset of $T - T_O$ such that only tests that guarantee the execution of modified code and code that might be impacted by the modified code in P' are selected.
  - A technique that does not discard any test that will traverse a modified or impacted statement is known as a "safe" regression test selection technique.

Overview



Test Selection Using Execution Traces

1. P is (has been) executed and execution traces are (have been) recorded
   - Execution trace = sequence of control flow graph (CFG) nodes hit during one test execution
   - trace(t) = execution trace of test case t.
   - test(n) = set of tests that hit node n at least once.

   We consider execution traces for $T - T_O$.
2. P' is compared with P
   - Build the control flow graphs G and G' of P and P', respectively
     - G and G' account for control flow, as well as function calls and variable declarations
   - Identify nodes (edges) in P and P' that are equivalent or not
     - Different techniques exist (e.g., syntax trees)
3. Selection
   - For each node $n \in G$ that does not have an equivalent node in G'
   - $T_{RT} = T_{RT} \cup test(n)$

Example

An execution trace of program P for some test t in T is the sequence of nodes in G traversed when P is executed against t. As an example, consider the following program.

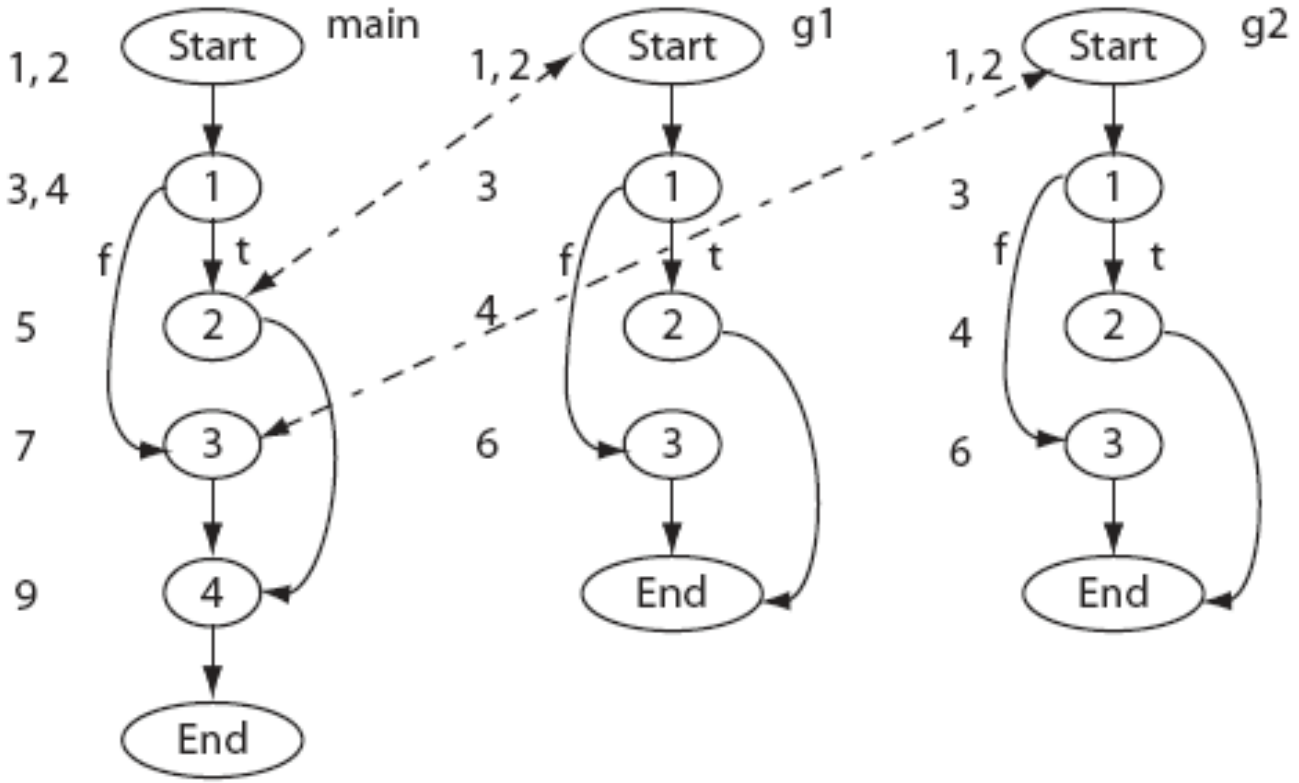
```
 1  main(){
 2  int x,y,p;
 3  input (x,y);
 4  if (x<y)
 5    p=g1(x,y);
 6  else
 7    p=g2(x,y);
 8  endif
 9  output (p);
10  end
11  }
```

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

```
 1  int g2 (int a, b){
 2  int a,b;
 3  if(a==(b+1))
 4    return(b*b);
 5  else
 6    return(a*a);
 7  }
```

Example: Control Flow Graph

Here is the CFG for our example program.



Example: Traces

Now consider the following set of three tests and the corresponding trace.

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

Test (t)	Execution trace ($trace(t)$)
t_1	main.Start, main.1, main.2, g1.Start, g1.1, g1.3, g1.End, main.2, main.4, main.End.
t_2	main.Start, main.1, main.3, g2.Start, g2.1, g2.2, g2.End, main.3, main.4, main.End.
t_3	main.Start, main.1, main.2, g1.Start, g1.1, g1.2, g1.End, main.2, main.4, main.End.

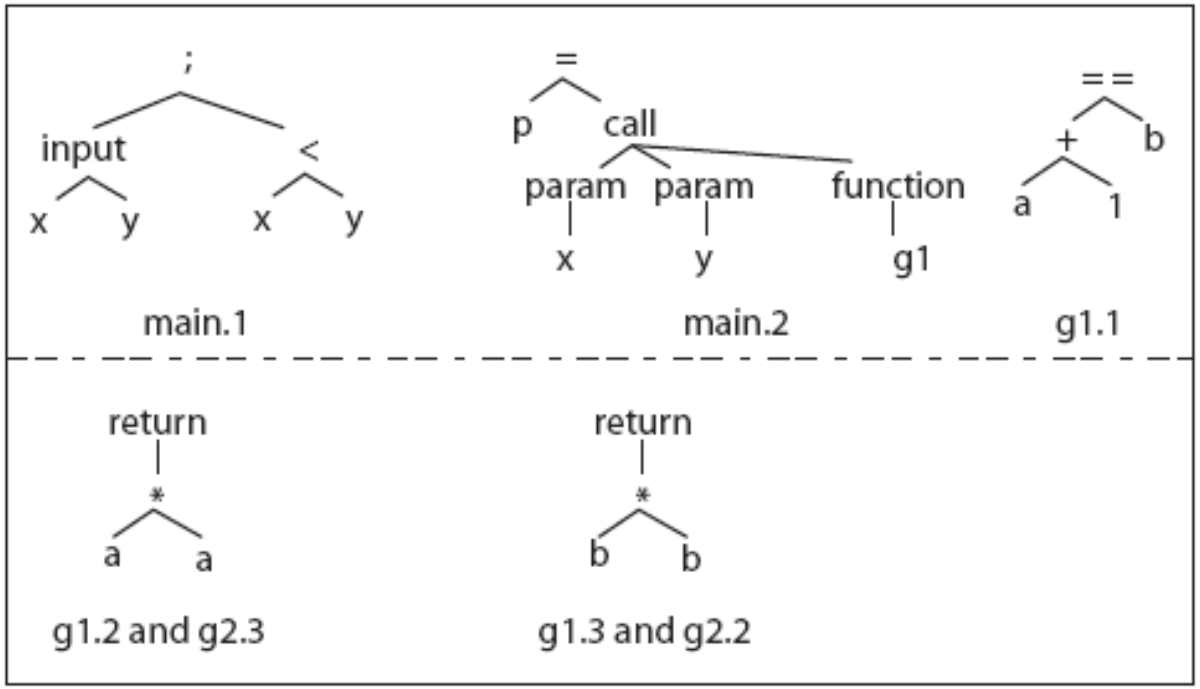
Example: Test vector

A test vector for node n, denoted by test(n), is the set of tests that traverse node n in the CFG. For program P we obtain the following test vectors.

	Test vector (*test*(*n*)) for node *n*			
Function	**1**	**2**	**3**	**4**
main	t_1, t_2, t_3	t_1, t_3	t_2	$t_1, t_2\ t_3$
g1	t_1, t_3	t_3	t_1	–
g2	t_2	t_2	None	–

Example: Syntax trees

A syntax tree is constructed for each node of CFG(P) and CFG(P'). This is used to identify differences in nodes (i.e., compare trees). Recall that each node represents a basic block. Here are sample syntax trees for the example program.



Test selection example

Suppose that function g1 in P is modified as follows.

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

T_{RT}={t1, t3}. No different CFG node is exercised with t2

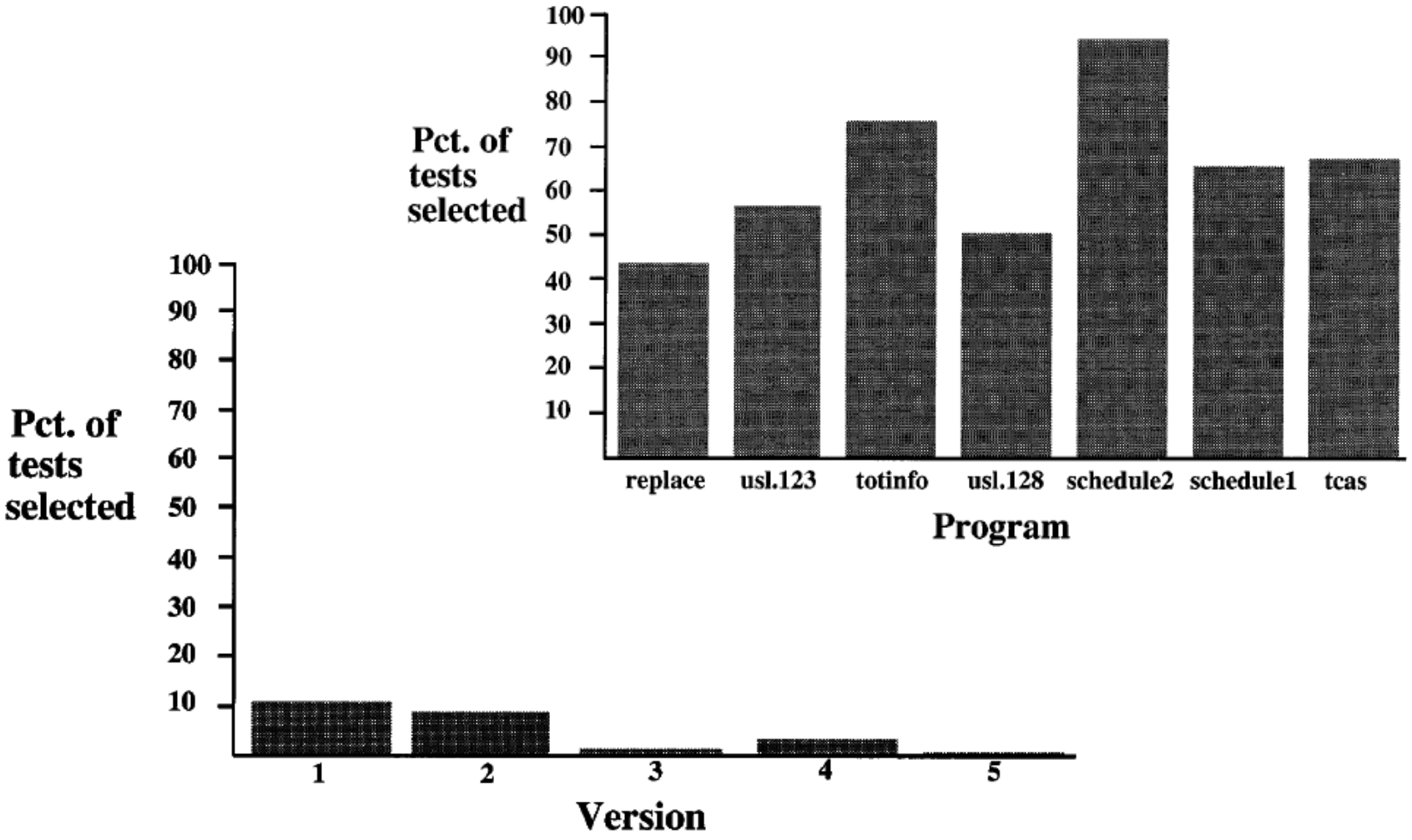
Test Selection Using Dynamic Slicing

Definition: dynamic slice

- Let P be the program under test
- Let t be a test case against which P has been executed
- Let l be a location in P where variable v is used.
- The dynamic slice of P with respect to t and v is the set of statements/nodes (CFG) in P that lie in trace(t) and did *affect* the value of v at l.

1. Computing dynamic slices
   - Many different algorithms, with various complexities and accuracies
2. P' is compared with P
3. Selection
   - Let DS(t) denote a dynamic slice of P with respect to t ($t \in T$) and an output variable of P.
   - Let n be a node in P (CFG) modified to generate P'
   - For all output variables, if $n \in DS(t)$ then re-test t ($T_{RT} = T_{RT} \cup \{t\}$)

Rothermel and Harrold Study



Regression Testing Process

- Test Revalidation
- (Regression) Test Selection
- Test Minimization
- Test Prioritization

Test Minimization

- Regression test selection finds a subset T_{RT} of T.
- Suppose P contains n testable entities
  - Functions, basic blocks, conditions, …
- Suppose that tests in T_{RT} cover m<n of the testable entities
  - There is likely an overlap amongst the entities covered by two tests in T_{RT}
- Is it possible (and beneficial) to reduce T_{RT} to T_{MIN}?
  - such that $|T_{MIN}| << |T_{RT}|$ and
  - each of the m entities covered by tests in T_{RT} are also covered by tests in T_{MIN}
- Question
  - Will T_{MIN} have the same fault detection effectiveness as T_{RT}?
- Answer
  - It depends on the modifications (from P to P'), the faults, the entities used.

Algorithms for Test Minimization

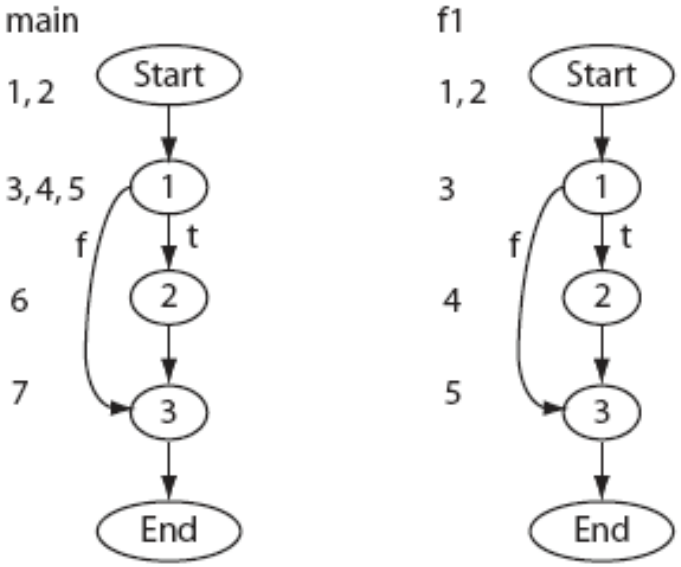
- Naïve algorithm
  - Compute all the subsets of T_{RT} of size 1, size 2, size 3, … and stop when we have found one that covers all the entities covered by T_{RT}
- Greedy algorithm

  (A greedy algorithm follows the heuristic of making the locally optimum choice at each stage with the hope of finding the global optimum.)

  1. Find $t \in T_{RT}$ that covers the maximum number of entities
  2. $T_{MIN} = T_{MIN} \cup \{t\}$
  3. Remove t from T_{RT}, remove the covered entities from consideration
  4. Repeat from step 1
- There exist more sophisticated search algorithms (e.g., genetic algorithms)

Test minimization: Example

Step 1: Let the basic block be the testable entity of interest. The basic blocks for a sample program are shown here for both main and function f1.



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

t3: main: 1, 3. f1: 1, 2, 3
```

Step3: A minimal test set for regression testing is `{t1, t3}`.

Risk

- Test minimization is risky
  - Tests removed from T' might be important for finding faults in P'
  - Minimization techniques are not necessarily safe
    - One could discard a test that hit a modified or impacted part of the code.

Regression Testing Process

- Test Revalidation
- (Regression) Test Selection
- Test Minimization
- Test Prioritization

Test Prioritization

- After regression test selection, T_{RT} might be overly large for testing P'
  - Not enough budget to execute all those tests.
- When very high quality software is desired, it might not be wise to discard test cases as in test minimization.
- In such cases use Test prioritization:
  1. Ranking tests (1^{st}, 2^{nd}, …)
  2. Deciding to stop execution of tests after the n^{th} ranked test
- Test prioritization requires a criterion, or criteria for ranking
- Single criterion prioritization
  - Criteria 1: cost (e.g., execution time)
    - Tests with lower costs are ranked first while test with higher costs are ranked last
  - Criteria 2: risk (expected risk of not executing a test)
    - Tests with higher risks are ranked first while test with lower risks are ranked last – How to measure risk?
- One goal of prioritization is to increase the likelihood of revealing faults earlier in the testing process.

A (simplistic) procedure for automated test prioritization

Step 1: Identify the type TE of testable entity to be used for test minimization. Let e1, e2, .. ek be the k testable entities of type TE present in P. (TE = function, basic block …).

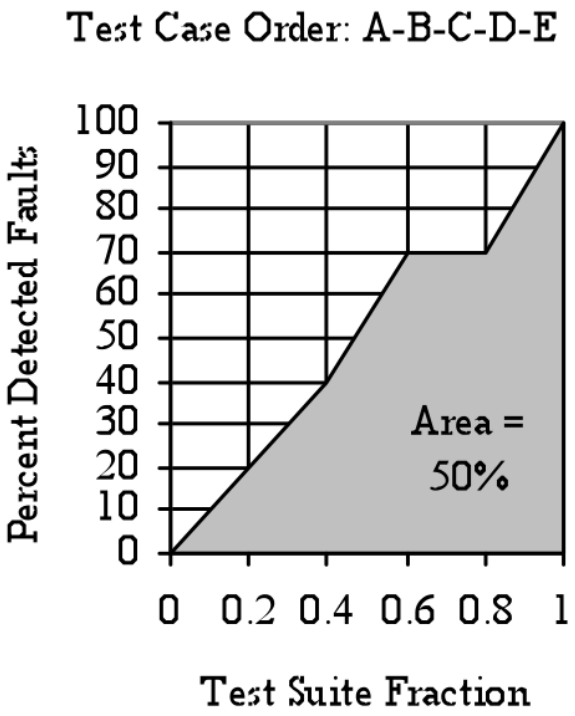
Step 2: Execute P for all tests t in T_{RT} and for each t compute the number of distinct testable entities covered.

Step 3: Arrange the tests in T_{RT} in the order of their respective coverage. Tests with the maximum coverage get the highest priority and so on.

Prioritization Refinements

- Use incremental coverage: for each test t, all tests ordered before it may overlap in coverage, and the incremental coverage of t may be much less
  - Use greedy algorithm: Select highest coverage test case first, then re-compute incremental coverage for other test cases. Select test case with highest incremental test coverage etc.
- Account for sequencing constraints among test cases: test cases may not be independent, some may need to be run before others
- Account for modifications, for example (incremental) coverage of *modified* testable entities
- Prioritization requires more complex algorithms

Evaluation of Prioritization Algorithms



(b)

Let T be a test suite with n test cases and let F represent a set of m faults revealed by T. Let TF_i be the first test case in the ordering T' of T that reveals fault i. The APFD for the test suite T' is given by the following equation:

$$APFD = \left[1 - \frac{TF_1 + TF_2 + \ldots + TF_m}{nm} + \frac{1}{2n}\right] \times 100$$

Prioritization with AHP

- Analytical Hierarchy Process (AHP)
  - Originally designed to prioritize requirements
  - Use expert knowledge
  - Comparing pairs and using information on pairs rather than ranking everything at once
    - Easier to compare pairs than to rank everything

1. Pair-wise comparison of tests, assigning a value to each pair
   - Different dimensions to compare pair (i,j): business value, risk,, cost, frequency of use (by users)

Numerical value	Explanation
1	Two test cases have equal importance.
3	Test case i has a slightly higher importance value than test case j.
5	Test case i has a strongly higher importance value than test case j.
7	Test case i has a very strongly higher importance value than test case j.
9	Test case i has an absolute higher importance value than test case j.
Reciprocals	If test case i has one of the numerical values when it compares with test case j, then test case j has the reciprocal value when compared with i.

Prioritization with AHP (cont.)

2. Build a table
   - Rows and columns are test cases
   - Cells show the evaluation of pairs.
3. Modify table
   i. Sum columns
   ii. Divide cells by column sum
   iii. Compute sum for row
   iv. Divide row sum by number of test cases and obtain a priority

	TC1	TC2	TC3	TC4
TC1	1	1/3	5	7
TC2	3	1	7	9
TC3	1/5	1/7	1	1/3
TC4	1/7	1/9	3	1

	TC1	TC2	TC3	TC4	Sum	Priority
TC1	0.230	0.208	0.3125	0.404	1.1545	0. 29
TC2	0.690	0.629	0.4375	0.519	2.2755	0.57
TC3	0.046	0.088	0.0625	0.019	0.2155	0.05
TC4	0.033	0.069	0.1875	0.058	0.3475	0.09
Sum	4.342	1.587	16	17.33	N/A	N/A

⇨ Prioritization:
TC2, TC1, TC4, TC3

Scalability issue:
Instead of comparing test case pairs, compare the corresponding use cases.

Tools

- Regression testing requires tools (automation)
  - Instrumentation tool to compute traces
  - Tools to build control flow graphs, build traces and slices
  - Tools implementing selection, minimization, prioritization algorithms
- "Un-automated regression testing is equivalent to no regression testing."
- Existing tools:
  - Capture/replay for GUIs
  - DejaVOO (Java): research
  - Telcordia Software Visualization and Analysis Toolsuite (xSuds) for C
  - ??
- A few tools perform white-box regression testing (analysis of source code)
- No tools for black/grey-box regression testing

General References

- Mathur, A.P., "*Foundations of Software Testing*", Pearson, 2008
- Leung, K.N., White, L., "Insights into Regression Testing", *Proc. IEEE International Conference on Software Maintenance (ICSM)*, pp. 60-69, 1989.
- Rothermel, G, Harrold, M.J., "A Safe, Efficient Regression Test Selection Technique," ACM Transactions on Software Engineering and Methodology, 6(2), pp.173-210, 1997
- Briand, L, Labiche, Y, He, S, "Automating regression test selection based on UML designs", Information and Software Technology (Elsevier), 2009

References for DB Applications

- David Willmor and Suzanne M. Embury, "A safe regression test selection technique for database-driven applications", ICSM 2005
- Scott W. Ambler and Pramod J. Sadalage, "Refactoring Databases: Evolutionary Database Design", 2006
- Florian Haftmann, Donald Kossmann, Alexander Kreutz, "Efficient Regression Tests for Database Applications", CIDR Conference, 2005