Error Seeding and Mutation Testing

Reading assignment

- L. A. Clarke, A. Podgurski, D. J. Richardson and Steven J. Zeil, "A Formal Evaluation of Data Flow Path Selection Criteria," IEEE Transactions on Software Engineering, 15 (11), November 1989, pp. 1318-1332.
- Background reading
  - S. Rapps and E. J. Weyuker, "Data Flow Analysis Techniques for Test Data Selection," Proceedings of the Sixth International Conference of Software Engineering, Tokyo, Japan, September 1982, pp. 272-277.
  - S. C. Natofos, "On Testing With Required Elements," Proceedings of COMPSAC '81, IEEE Computer Society, November 1981, pp. 132-139.
  - J. W. Laski and B. Korel, "A Data Flow Oriented Program Testing Strategy," IEEE Transactions on Software Engineering, 9 (3), May 1983, pp. 347-354.

Some terminology

- Test case: input data for the single execution of a program
  - A test case may consist of a number of values
- Test set or test suite: set of test cases

Random Testing

- Based on a description of the legal inputs, generates test cases randomly over the program domain
- Benefits
  - Easy to generate test cases
  - Serves as a baseline for comparison
    - Using the same number of test cases, does testing criteria X do as well as random testing at detecting faults/finding failure?
- Drawbacks
  - Need to have an oracle for each test case
  - May not match the operational profile

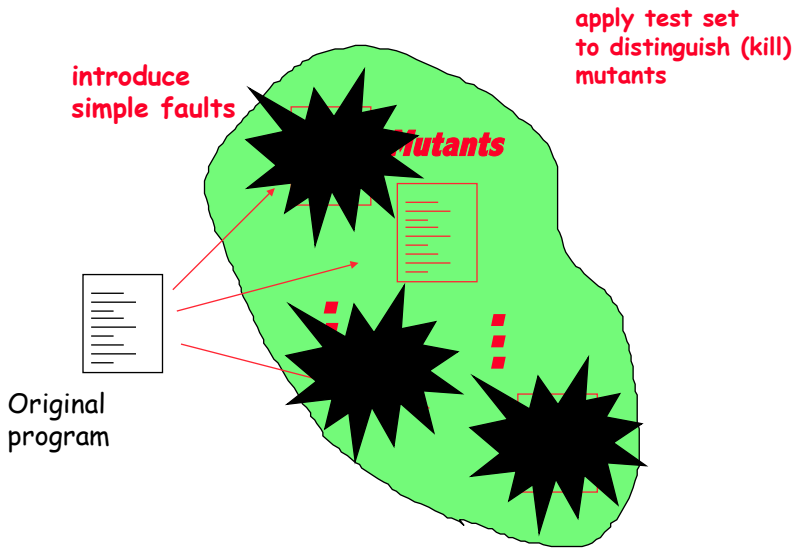
Error Seeding

- Insert "typical" faults into a system
- Determine how many of the inserted faults are found
  - If K of the N faults found, then assume that K/N of actual faults found as well
- Motivates developers/testers
  - Know there is something to find
  - Not looking for their own faults, so more motivated
- Drawback
  - Assumption about percentage of remaining faults not valid unless the seeded faults are "representative"

Mutation Testing

- Systematic method of error seeding
  - originally proposed by Budd, Lipton, DeMillo, and Sayward in the mid 1970s
- Approach: considers all simple (atomic) faults that could occur
  - introduces single faults one at a time to create "mutants" of original program
  - apply test set to each mutant program
  - "test adequacy" is measured by % "mutants killed"

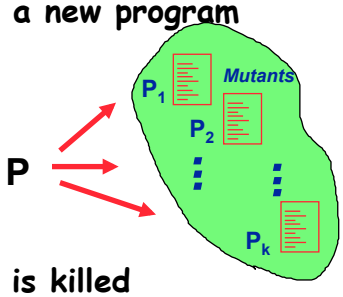
Mutation Testing



Mutation testing process

- Execute program P on test set T
  - P is considered the "correct" program
  - save results R to serve as an oracle

- Each inserted fault results in a new program
  - Mutant programs = $P_1,\ldots,P_k$



- If $P_i(T) \neq P(T)$ then mutant P_i is killed

(If $\exists t \in T \mid P_i(t) \neq P(t)$, then mutant P_i is killed)

Mutation Testing Assumptions

- **Competent Programmer Hypothesis**
  - programmers write programs that are reasonably close to the desired program
    - e.g., sort program is not written as a hash table
- **Coupling Effect**
  - detecting simple atomic faults will lead to the detection of more complex faults

Atomic faults: Operand mutations

- Constant replacement

  e.g., `x := x + 5;` would replace 5 with each constant of the appropriate type that appears in the program
- Scalar variable replacement

  e.g., `y := x + 5;` would replace x with each scalar variable of the appropriate type that appears in the program

More operand mutations

- scalar variable for constant replacement
- constant for scalar variable replacement
- array reference for constant replacement
- array reference for scalar variable replacement
- constant for array reference replacement

More operand mutations

- scalar variable for array reference replacement
- array reference for array reference replacement
- array index replacement for array index replacement
- data statement alteration

Operator mutations

- arithmetic operator replacement
  - e.g., x := x + 5;
  - would replace + with -, *, /, and **
- relational operator replacement
  - e.g., a > b;
  - would replace > with >=, <, <=, =, and /=

More operator mutations

- logical connector replacement
- absolute value insertion
- unary operator insertion
- statement deletion
- return statement replacement
- GOTO label replacement
- DO statement end replacement

Example

- consider the assignment:
  A : = X + 1;
- assume:
  - 2 is the only other constant in the program
  - Y is the only scalar variable of the same type as X and A
  - C[I] is the only array with the same type as X and A

Mutating one statement: A : = X + 1

operand mutations:

A : = X + 2
A : = X + Y
A : = X + A
A : = X + C[I]
A : = Y + 1
A : = A + 1
A : = C[I] + 1
A : = 1 + 1
A : = 2 + 1
X : = X + 1
Y : = X + 1
C[I] : = X + 1

binary operator replacement:

A : = X - 1
A : = X * 1
A : = X / 1
A : = X ** 1

unary operator insertion:

A : = -X + 1
A : = X + (-1)
A : = - (X + 1)

absolute value insertion:

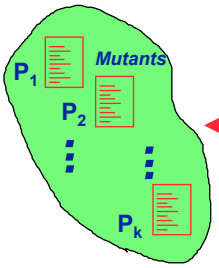
A : = abs (X) + 1
A : = abs (X + 1)

statement replacement:

continue
return
go to 100 (if there is a label 100)

Mutation testing process

- **execute each mutant P_i on T and compare results**
  - If $P_i(T) \neq P(T)$ *then mutant is killed*
    - Has to fail for at least one $t \in T$
  - if $P_i(T) = P(T)$ *then either*
    $P_i = P$, *thus it is an equivalent mutant or*
    the test cases do not reveal
    the error and need to find a new
    test case that does



- apply test data and compare output with oracle
- "killed" distinguished mutants

Equivalent Mutants

- **P = P_i**

P

```
...
A : = X + 1
...
```

P_i

```
...
A : = abs ( X) + 1
...
```

If, at this point in the code,
X >= 0 is always true, then
X = abs(X)

Mutation System

- **Automates the mutation process**
  - uses the initial execution to determine the oracle
  - creates the mutants
  - lists the seeded errors that have not been detected

  user states (interactively) if the mutant is equivalent to the original program or tries to find a test case to kill the mutant

- **Mutation system is a test set evaluation system**

Optimization: Compare program state after executing the mutated statement

- Report mutant not killed -- no need to continue since all outputs will be the same for both programs
- This test case does not kill the mutant
- Only needed to execute up to and including the mutated statement

Another Example:

Original Program

…

X := X + 1

…

State <x,y,z>

<2, 3,0>

<3, 3,0>

Mutated Program

…

X' := X + Y

<5,3,0>

…

Can we report that the mutant was killed?

Z := X * (Y - 3)

Z := X * (Y - 3)

- To save on computation time, estimates the result
  - Will be wrong some of the time
  - Reporting false negatives

Optimization techniques to reduce execution cost

- **Don't actually create and compile all the mutants**
- **For each test case:**
  - **keep track of the internal state during execution of the original program & start with the statement preceding a mutated statement**
  - **stop execution if the values computed by the mutant ever become the same as the value computed by the correct program (*report that the mutant was not killed*)**
- **An alternative approach**
  - **stop execution if the values computed by the mutant are not the same as the original program (report that the mutant was probably killed)**
    - ***called weak mutation testing***

The state at a stmt consists of all the variables that are "live" (will ever be used in the future) at that point in the execution

Examples

X := X + 1; **mutated stmt X' := X + Y;**

Test case:X=2; Y=3

Results: Z= 0

Weak mutation would report that the mutant is killed

Z := X * (Y - 3);

(Strong) mutation would continue to be sure the fault propagates to an output

Stops and reports that the mutant is NOT killed if the state at an intermediate point is the same as the original program

Stops and reports that the mutant is killed if the program produces an output result that differs from the original program's output

Observations about Mutation Testing

- Even with optimization techniques, mutation testing is an expensive way to find faults in a program
  - Many mutant programs need to be executed
- Eliminating equivalent mutants is tedious; killing all mutants is hard
  - first 80% are easy, last 20% are hard
  - Initially, each test case usually kills many mutants

How does mutation testing compare to other testing approaches? Wrt Fault Detection

- Experimental studies showed that it is as effective or almost as effective as other test data selection techniques
  - "Effective" measures ability to detect faults
  - only a few experimental studies done and on limited size/simple programs
- Coupling Hypothesis is not true
  - Killing mutants does not guarantee that all faults will be found
  - Killing mutants does tend to detect faults in addition to the seeded faults
- Mutation testing usually requires significantly more test cases than the other methods
  - Consider test set X and test set Y, where |X| >>|Y|
    - Which detects more faults in a program P?
    - Should normalize the number of test cases
  - More test cases increases the cost
    - Cost of executing the test cases
    - Cost of evaluating the outputs
      - Can reuse the oracle, but must determine the oracle for each test case

How can we compare testing techniques?

- For a program P, compare the faults found using T1 and T2, where C1(T1,P) and C2(T2,P)
  - For any particular program, one criteria might be better than another
    - But for another program, the other criteria might be superior
    - Must consider a large sample of programs
- For the same program P and same criterion C, different test sets, T1 and T2, where C(T1,P) and C(T2,P), might vary greatly on their ability to detect faults
  - Must consider a number of test sets for each program
  - They might vary greatly in their size as well

How can we compare testing techniques?

- **Must do careful experimental studies**
  - **For each program and each criterion, must consider many Test Sets that satisfy that criterion**
  - **Compare the average percentages of faults found for each criterion, using a large set of programs and a large set of test sets for each criterion applied to each program**

For $0 <= i < n_1$, $1 \{T_{11i} | C_1(T_{11i}, P_1)\}$ For $0 <= i < m_1$, $1 \{T_{21i} | C_2(T_{21i}, P_1)\}$

For $0 <= i < n_2$, $1 \{T_{12i} | C_1(T_{12i}, P_2)\}$ For $0 <= i < m_2$, $1 \{T_{22i} | C_2(T_{22i}, P_2)\}$

For $0 <= i < n_s$, $1 \{T_{1ni} | C_1(T_{1ni}, P_n)\}$ For $0 <= i < m_q$, $1 \{T_{2ki} | C_2(T_{2ki}, P_k)\}$

Must also take size into account

For i, $0 <= i < n_{11i} \{T_{11i} | C_1(T_{11i}, P_1)\}$ For $0 <= i < m_{21i} \{T_{21i} | C_2(T_{21i}, P_1)\}$

- Same number of test sets: $n_{ijk} = m_{ijk}$
- Each test case in a test set for a Criterion C1 and a Program Pi has a "buddy" test case of the same size for Criterion C2
  - e.g., size of T_{111} = size of T_{211}

    size of T_{112} = size of T_{212}

How can we compare testing techniques?

- Must do careful experimental studies
  - For each program and each criterion, must consider many Test Sets that satisfy that criterion
  - Compare the average percentages of faults found for each criterion, using a large set of programs and a large set of test sets for each criterion applied to each program
- Can sometimes do analytical evaluations
  - Can sometimes show that one approach subsumes another
  - C1(T,P) => C2 (T,P)
  - Branch Coverage => Statement Coverage
    - If a test set provides branch coverage then that same test set on the same program will provide statement coverage

How does mutation testing compare to other testing approaches? Wrt Subsumption

- Analytic evaluation shows that it "subsumes" some other approaches
  - E.g., Subsumes statement and branch coverage
  - Test cases that satisfy mutation testing will satisfy branch coverage
    - MutationTesting(T,P) => BranchCoverage (T,P)
  - But, there is a huge size discrepancy
    - There exists $T' \subseteq T$, such that MutationTesting(T,P) and BranchCoverage (T',P)
    - But usually $|T'| << |T|$

How does mutation testing compare to other testing approaches?

- For the amount of effort, how does mutation testing compare to random testing?
  - For MutationTesting(T,P), generate a random test set T1, so that |T1|=|T|
  - Random Testing Effort:
    - Easy to generate test cases
    - May be hard to evaluate results
  - Mutation Testing Effort:
    - Hard to generate test cases
    - Easy to evaluate results?
      - original program is oracle,but needed to evaluate original test cases
- Which approach finds more faults?
- Is mutation testing effective at finding real faults in real programs?

Another Mutation-Based Technique:

- Mutating Test Data
  - Instead of mutating program, mutate input
- Bart Miller did an experiment where he demonstrated that arbitrary strings caused UNIX to consistently fail
  - Wanted to understand why storms caused his connection to go down
- Fuzz testing is a set of techniques that "mutate" test cases

Blackbox Fuzz Testing

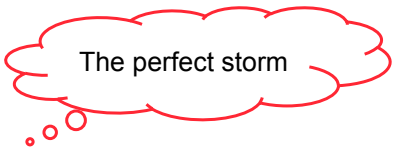
- Main techniques used by "black hats":
  - Code inspection (of binaries) and
  - Blackbox fuzz testing
- Blackbox fuzz testing:
  - Randomly fuzz (modify) a well-formed input
  - Grammar-based fuzzing: rules that encode "well-formed"ness + heuristics about how to fuzz (e.g., using probabilistic weights)
- ·Heavily used in security testing
  - Simple yet effective: many bugs found this way
  - At Microsoft, fuzzing is mandated
    - As reported by Patrice Godfroid at ISSTA 2010
- Whitebox fuzz testing uses static analysis techniques, so will be discussed later in course

N version programming

- Create N versions of a program
  - Run the programs on the same data
  - When the programs disagree, assume the majority answer is correct
- Widely used at Nasa for hardware systems
  - Hardware components can fail independently
  - Considered this approach for software
    - Systems needed to be developed independently
  - Study by Leveson and Knight showed that "majority voting" is not a good predictor of the correct value for software
    - For difficult cases, the majority was often wrong

N version programming + Fuzz testing

- N version programming is expensive
  - Usually costly to create one version of a system
- What about situations where there are N versions of a system
  - E.g., compilers
  - Multiple compilers produced by different vendors
  - Inputs are described by grammars, so easy to apply fuzz testing



C Compiler N-version Testing Experiment

- Conducted by Bill McKeeman at DEC
- Using the language grammar, generated legal C programs
- Ran the generated C programs on n different C compilers and compared the results
- 20% of the time at least one of the compilers generated incorrect code

C compiler experiment-version 2

- Weighted the grammar productions to avoid "hard cases"
- 1% of the time at least one of the compilers generated incorrect code
- Compiler maintainers still gave a low priority to fixing those errors
  - often the generated strings did not correspond to typical cases

    e.g., 007.5
  - thus, not the kind of errors most programmers will report

Comparison

- Grammar based test data generation is a form of "random" testing
- Mutated test data is a form of fuzz testing
  - Tests for system robustness
  - Used in security testing, since the strange corner cases are often where vulnerabilities are found
  - Easy to create blackbox fuzz testers/test cases
  - May be hard to determine the oracle
- Mutation testing mutates the program
  - Used to evaluate the test cases
- Mutation testing is most often used to create test beds of buggy programs
  - Helps evaluate testing criteria

Summary of Mutation testing

- Mutation testing takes error seeding to the absurd,
  but it did stimulate some useful research and insight
- Optimization approaches need to be used
- Mutation testing very useful for generating test beds of buggy programs
  - Only one (known) bug per program
  - May or may not be typical bugs
  - Now widely used to evaluate different testing approaches