Mutation testing

- Mutation testing is an approach to evaluate — and to improve — test suites
- Basic idea
  - Create small variants of the program under test
  - If the tests don’t exhibit different behavior on the variants then the test suite is not sufficient
- The material on the following slides is due heavily to Pezzè and Young on fault-based testing

Estimation

- Given a big bowl of marbles, how can we estimate how many?
- Can’t count every marble individually

What if I also…

- … have a bag of 100 other marbles of the same size, but a different color (say, black) and mix them in?
- Draw out 100 marbles at random and find 20 of them are black
- How many marbles did we start with?
Estimating test suite quality

• Now take a program with bugs and create 100 variations each with a new and distinct bug
  – Assume the new bugs are exactly like real bugs in every way
• Run the test suite on all 100 new variants
  – ... and the tests reveal 20 of the bugs
  – … and the other 80 program copies do not fail
• What does this tell us about the test suite?

Basic Assumptions

• The idea is to judge effectiveness of a test suite in finding real faults by measuring how well it finds seeded fake faults
• Valid to the extent that the seeded bugs are representative of real bugs: not necessarily identical but the differences should not affect the selection

Mutation testing

• A mutant is a copy of a program with a mutation: a syntactic change that represents a seeded bug
  – Ex: change \( i < 0 \) to \( i \leq 0 \)
• Run the test suite on all the mutant programs
• A mutant is killed if it fails on at least one test case
  – That is, the mutant is distinguishable from the original program by the test suite, which adds confidence about the quality of the test suite
• If many mutants are killed, infer that the test suite is also effective at finding real bugs

Mutation testing assumptions

• Competent programmer hypothesis: programs are nearly correct
  – Real faults are small variations from the correct program and thus mutants are reasonable models of real buggy programs
• Coupling effect hypothesis: tests that find simple faults also find more complex faults
  – Even if mutants are not perfect representatives of real faults, a test suite that kills mutants is good at finding real faults, too
Mutation Operators

- Syntactic change from legal program to legal program and are thus specific to each programming language
- Ex: constant for constant replacement
  - from \((x < 5)\) to \((x < 12)\)
  - Maybe select from constants found elsewhere in program text
- Ex: relational operator replacement
  - from \((x <= 5)\) to \((x < 5)\)
- Ex: variable initialization elimination
  - from \(\text{int } x = 5;\) to \(\text{int } x;\)

Live mutants scenario

- Create 100 mutants from a program
  - Run the test suite on all 100 mutants, plus the original program
  - The original program passes all tests
  - 94 mutant programs are killed (fail at least one test)
  - 6 mutants remain alive
- What can we learn from the living mutants?

How mutants survive

- A mutant may be equivalent to the original program
  - Maybe changing \((x < 0)\) to \((x <= 0)\) didn’t change the output at all!
  - The seeded “fault” is not really a “fault” – determining this may be easy or hard or in the worst case undecideable
- Or the test suite could be inadequate
  - If the mutant could have been killed, but was not, it indicates a weakness in the test suite
  - But adding a test case for just this mutant is a bad idea – why?

Weak mutation: a variation

- There are lots of mutants – the number of mutants grows with the square of program size
- Running each test case to completion on every mutant is expensive
- Instead execute a “meta-mutant” that has many of the seeded faults in addition to executing the original program
  - Mark a seeded fault as “killed” as soon as a difference in an intermediate state is found – don’t wait for program completion
  - Restart with new mutant selection after each “kill”
Statistical Mutation: another variation

- Running each test case on every mutant is expensive, even if we don’t run each test case separately to completion
- Approach: Create a random sample of mutants
  - May be just as good for assessing a test suite
  - Doesn’t work if test cases are designed to kill particular mutants

In real life ...

- Fault-based testing is a widely used in semiconductor manufacturing
  - With good fault models of typical manufacturing faults, e.g., “stuck-at-one” for a transistor
  - But fault-based testing for design errors – as in software – is more challenging
- Mutation testing is not widely used in industry
  - But plays a role in software testing research, to compare effectiveness of testing techniques
- Some use of fault models to design test cases is important and widely practiced

Summary

- If bugs were marbles ...
  - We could get some nice black marbles to judge the quality of test suites
- Since bugs aren’t marbles ...
  - Mutation testing rests on some troubling assumptions about seeded faults, which may not be statistically representative of real faults
- Nonetheless ...
  - A model of typical or important faults is invaluable information for designing and assessing test suites
Symbolic execution

\[ x > \beta \]
\[ x = x + y \]
\[ y = x - y \]
\[ x = x - y \]
\[ x > y \]
\[ \text{assert(false)} \]
\[ \text{end} \]

What's really going on?

- Create a symbolic execution tree
- Explicitly track path conditions
- Solve path conditions – “how do you get to this point in the execution tree?” — to defines test inputs
- Goal: define test inputs that reach all reachable statements

Another example (Sen and Agha)

```
int double (int v){
    return 2*v;
}
void testme (int x, int y){
    x = double (y);
    if (x == x+10) {
        if (x > y+10) {
            ERROR;
        }
    }
}
```

Error: possible by solving equations

\[ 2 \alpha = \alpha \& \alpha > \beta + 10 \]
\[ = 2 \alpha > \beta + 10 \]
\[ = \beta > 10 \]
\[ = [\beta > 10 \& 2 \alpha = \alpha ] \]
Way cool – we’re done!

- First example can’t reach `assert(false)`, and it’s easy to reach `end` via both possible paths.
- Second example: can reach `error` and `end` via both possible paths.
- Well, what if we can’t solve the path conditions?
  - Some arithmetic, some recursion, some loops, some pointer expressions, etc.
  - We’ll see an example.
- What if we want specific test cases?

Concolic testing: Sen et al.

- Basically, combine concrete and symbolic execution.
- More precisely…
  - Generate a random concrete input.
  - Execute the program on that input both concretely and symbolically simultaneously.
  - Follow the concrete execution and maintain the path conditions along with the corresponding symbolic execution.
  - Use the path conditions collected by this guided process to constrain the generation of inputs for the next iteration.
  - Repeat until test inputs are produced to exercise all feasible paths.

```c
int double (int v){
    return 2*v;
}
void testme (int x, int y){
    z = double (y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
}}
```

2nd example redux
1st iteration x=22, y=7
- Now solve \( 2 \cdot \beta = \alpha \) to force the other branch.
- \( x = 1; y = 2 \) is one solution.

2nd example
2nd iteration x=1, y=2
- Now solve \( 2 \cdot \beta = \alpha \) & \( \alpha <= \beta + 10 \) to force the other branch.
- \( x = 30; y = 15 \) is one solution.
• Now solve $2^*\beta = \alpha \& \alpha \leq \beta + 10$
to force the other branch

• $x = 30; y = 15$ is one solution

Concolic testing example: P. Sağlam

• Random seed
  - $x = -3; y = 7$

• Concrete
  - $z = 9$

• Symbolic
  - $z = x^3+3x^2+9$

• Take then branch with constraint $x^3+3x^2+9 \neq y$

• Take else branch with constraint $x^3+3x^2+9 = y$

Concolic testing example: P. Sağlam

• Solving is hard for $x^3+3x^2+9 = y$

• So use $z$’s concrete value, which is currently $9$, and continue concretely

• $9 \neq 7$ so then is good

• Symbolically solve $9 = y$ for else clause

• Execute next run with $x = -3; y = 9$

• When symbolic expression becomes unmanageable (e.g., non-linear) replace it by concrete value
Concolic testing example: P. Sağlam

- Random
  - Random memory graph reachable from P
  - Random value for x
  - Probability of reaching abort() is extremely low
- (Why is this a somewhat misleading motivation?)

typedef struct cell {
    int v;
    struct cell *next;
} cell;

int f(int v) {
    return 2*v + 1;
}

int testme(cell *p, int x) {
    if (x > 0)
        if (p != NULL)
            if (f(x) == p->v)
                if (p->next == p)
                    abort();
                return 0;
}

Let's try it

typedef struct cell {
    int v;
    struct cell *next;
} cell;

int f(int v) {
    return 2*v + 1;
}

int testme(cell *p, int x) {
    if (x > 0)
        if (p != NULL)
            if (f(x) == p->v)
                if (p->next == p)
                    abort();
                return 0;
}

p=NULL;
x=236

Let's try it

typedef struct cell {
    int v;
    struct cell *next;
} cell;

int f(int v) {
    return 2*v + 1;
}

int testme(cell *p, int x) {
    if (x > 0)
        if (p != NULL)
            if (f(x) == p->v)
                if (p->next == p)
                    abort();
                return 0;
}

p=[634,NUL];
x=236

Let's try it

typedef struct cell {
    int v;
    struct cell *next;
} cell;

int f(int v) {
    return 2*v + 1;
}

int testme(cell *p, int x) {
    if (x > 0)
        if (p != NULL)
            if (f(x) == p->v)
                if (p->next == p)
                    abort();
                return 0;
}

p=[3,p];
x=1

Let's try it

typedef struct cell {
    int v;
    struct cell *next;
} cell;

int f(int v) {
    return 2*v + 1;
}

int testme(cell *p, int x) {
    if (x > 0)
        if (p != NULL)
            if (f(x) == p->v)
                if (p->next == p)
                    abort();
                return 0;
}

p=[3,p];
x=1
**Let's try it**

```c
typedef struct cell {
    int v;
    struct cell *next;
} cell;

int f(int v) {
    return 2*v + 1;
}

int testme(cell *p, int x) {
    if (x > 0)
        if (p != NULL)
            if (f(x) == p->v)
                if (p->next == p)
                    abort();
    return 0;
}
```

<table>
<thead>
<tr>
<th>Concrete</th>
<th>Symbolic</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>typedef struct cell { int v; struct cell *next; } cell;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>int f(int v) { return 2*v + 1; }</td>
<td></td>
<td></td>
</tr>
<tr>
<td>int testme(cell *p, int x) { if (x &gt; 0) if (p != NULL) if (f(x) == p-&gt;v) if (p-&gt;next == p) abort(); return 0; }</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Questions?**