11 Mutation testing

11.1 Mutation testing in brief

Mutation testing in brief

1. Mutation testing, also known as fault-based testing targets explictly the software faults
2. To evalute the tests, their quality is **NOT** measured in terms of coverage of structural elements
3. Instead, faults are injected in the code and tests are evaluated in terms of how many injected faults are detected

Mutation testing, also known as fault-based testing, was independently introduced by Acree (1980) in his PhD thesis and by DeMillo et al. (1978). The idea is to use the ability to detect seeded artificial faults in the system under test as a means to guide and evaluate testing, rather than relying on structural properties. The overall approach is summarized in Fig. 11.1: the starting point of mutation testing is a program and a set of tests, which all pass on the program. The first step is to generate artificial faults, which are called mutants.

Each mutant differs from the original system in only one small syntactical change. Mutants are generated systematically and automatically using mutation operators, which represent standard types of mutations and apply these changes at all possible locations in the source code. Many different mutation operators have been proposed in the literature for various different programming languages, and a convention is to give these operators three-letter acronyms.

mutation process

Overview of the mutation testing process:

- Mutation operators are applied to the program under test to produce mutants.
- Tests are executed on all mutants; if a test fails on a mutant but passes on the original program, then the mutant is killed.
- If there is no test that kills the mutant, the mutant is alive, and likely reveals a weakness in the test suite

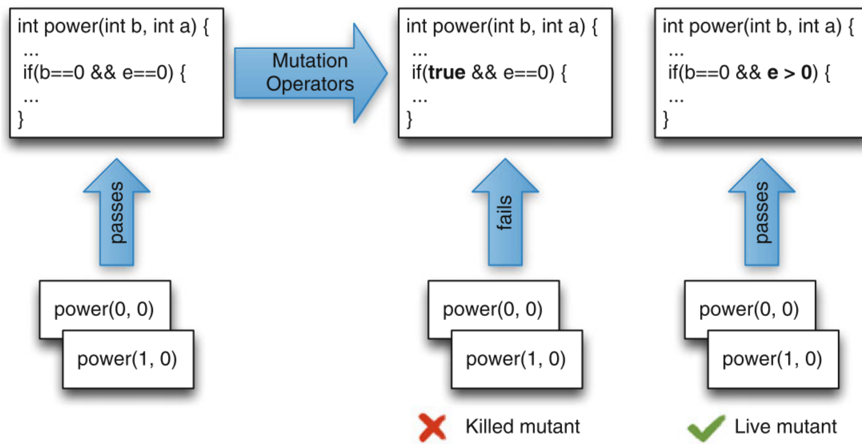

Figure 11.1: Mutation process

```
1  int power(int b, int e){
2   if (e < 0) throw new Exception("Negative exponent");
3   if ((b == 0) && (e == 0)) throw new Exception("Undefined");
4   int r = 1;
5   while (e > 0){
6    r = r * b; e = e - 1;
7   }
8   return r;
9  }
10 @Test
11 public void testPowerOf2() {
12  int result = power(2, 2);
13  assertEquals(4, result);
14 }
```

An example - power method
Figure shows an example of a java code that computes the power.

Figure 7 shows an example mutant for the power function of Fig. 1 in detail; this mutant is the result of applying the COR (conditional operator replacement) mutation operator to line number 4 of the original version of the program. Once a collection of mutants has been produced, every test in the test suite under evaluation (i.e., both tests in Fig. 6) is executed on the original system and on each of the mutants. Since a prerequisite for mutation analysis is that all tests pass on the original program, a mutant is "killed" when the test suite contains at least one test that fails when executed on the mutant. To inform the testing process, the mutation score is computed to indicate how many of the mutants can be detected (read killed) by the test suite.

11.2 Mutant example

mutant example

```
1  int power(int b, int e){
2   if (e < 0) throw new Exception("Negative exponent");
3   if ((true) && (e == 0)) throw new Exception("Undefined");
4   int r = 1;
5   while (e > 0){
6    r = r * b;
7    e = e - 1;
8   }
9   return r;
10 }
```

- Mutant by applying the COR operator (Conditional Operator Replacement) to line number 4
- The original test case assertEquals(4,power(2, 2)); won't fail - the mutant is NOT killed - the faults is not found

11.2.1 Survived mutants

While a mutant that is killed may increase confidence in the test suite and increase the mutation score, the true value may lie in the live mutants,i.e., those that were not detected by any test: live mutants can guide the developers toward insufficiently tested parts of the system. Depending on the number of mutation operators applied and the program under test, the number of mutants generated can be substantial, making mutation analysis a computationally expensive process. Several different optimizations have been proposed in order to reduce the computational costs, and Offutt and Untch (2001) nicely summarize some of the main ideas. For example, rather than compiling each mutant individually to a distinct binary, meta-mutants (Untch et al. 1993) merge all mutants into a single

program, where individual mutants can be activated/de- activated programmatically; as a result, compilation only needs to be done once. Selective mutation is a further optimization, where the insight that many mutants are killed by the same tests is used to reduce the number of mutants that is considered, either by randomly sampling mutants or by using fewer mutation operators. In particular, work has been done to determine which operators are sufficient, such that if all the resulting mutants are killed, then also (almost) all mutants of the remaining operators are killed (Offutt et al. 1993).

Survived mutants

1. Survived mutants are a sign of weakness of the test suiste (a fault that cannot be found)

2. New tests must be added

3. Note 1: (survived) mutants can be very many ...

mutant example

```
1 int power(int b, int e){
2  //... as before
3  if ((true) && (e == 0))
4   throw new Exception("Undefined");
5  //... as before
6 }
```

- To detect this fault we need a test in which we call power with e = 0 and b != 0. something like:

```
1 @Test public void test0PowerOf2() {
2  int result = power(2, 0);
3  assertEquals(1, result);
4 }
```

- test0PowerOf2 will pass on the original code but it will fail with the mutant -> mutant is killed

11.3 Equivalent mutants

A limitation of mutation testing lies in the existence of equivalent mutants. A mutant is equivalent when, although syntactically different, it is semantically equivalent to the original program. The problem of determining if a mutant is equivalent or not is undecidable in general; hence, human effort is needed to decide when a mutant is equivalent and should not be considered or when a mutant is actually not equivalent and a test should

be created to detect it. It is commonly assumed that equivalent mutants are among the reasons why mutation testing has not been adopted by most practitioners yet. Figure 8 presents an example of equivalent mutant for the power function. The mutation applied is the ROR (relational operator replacement) operator; the condition of the while loop is changed from > to !=. The resulting mutant is equivalent because the evaluations of e>0 and e!=0 are identical, given that the first if condition already handles the case e<0, and therefore e will never be less than 0 when the mutated expression is evaluated. Two theoretical assumptions are the foundation of mutation testing: the cou- pling ef- fect and the competent programmer hypothesis. The coupling effect states that small faults are coupled with more complex ones. The implications of this assumption are that by explicitly testing for simple faults, mutation testing implicitly tackles more complex ones as well. Furthermore, according to the competent programmer hypothesis, software developers tend to write almost-correct programs, i.e., programs whose faults are due to small syntactical mistakes, which can in practice be simulated during mutation testing. There are two influential empirical studies suggesting that mutants are indeed coupled and thus representative of real faults: The first one is by Andrews et al. (2005), and more recently Just et al. (2014) empirically validated the coupling effect by showing that a strong correlation exists between mutant detection and real fault detection. There is now a substantial body of literature on mutation testing, which has been comprehensively surveyed by Jia and Harman (2011).

Equivalent mutants

1. A limitation of mutation testing lies in the existence of equivalent mutants.
2. A mutant is equivalent when, although syntactically different, it is semantically equivalent to the original program.
3. There is NO test that kills an equivalent mutant - they will always survive
4. It is very difficult to say if a mutant has survived beacuase a test is missing or because it is equivalent

equivalent mutants

```
int power(int b, int e){
  if (e < 0)
    throw new Exception("Negative exponent");
  if ((b == 0) && (e == 0))
    throw new Exception("Undefined");
  int r = 1;
  while (e != 0){
    r = r * b; e = e - 1;
  }
  return r;
}
```

- This mutant cannot be killed by any test since it is equivalent.

11.4 Tools for mutation testing

Tools for mutation testing

- There are many tools that perform mutation testing
- DEMO with PIT test: https://pitest.org/

Usare PIT con maven

- vedi https://pitest.org/quickstart/maven/
- aggiungi il jar nel pom e anche plugin
- fornisce due goal