1. **Background knowledge**
2. **What is mutation testing and why we need mutation testing?**

There are some emerged questions when a programmer or manager deals with a test suite for a complex program such as:

* How do I safety refactor my tests?
* How do I know that I can trust a test suite that I inherited?
* How do I ensure my team is writing effective tests?

These questions can be combined into single one question: How can I assess the quality of test suite?

Traditionally, we have some ways to increase and ensure the quality of test suite such as: code coverage, code review, test driven development (TDD)…However, each of them is not sufficient enough to ensure the quality of test suite. With code review, we can catch problems inconsistently. The result depends on experience of conductor of code review session. Event test cases are developed based on TDD, we are not sure 100% about the quality of our created test cases as well as inherited test cases. Using code coverage to measure the quality of test case is not bad idea. However, test coverage (e.g line, statement, branch…) measures only with code are executed by your test cases. It does not ensure that your test cases are actually able to detect faults in the executed code. In other words, test coverage only identifies code that is not definitely not tested.

Mutation testing is conceptually quite simple. Faults (or mutations) are automatically inserted (seeded) in your code, and then your tests are run. If your tests fail then the mutation is killed, if your tests pass then the mutation lived. In other words, your unit tests are run with modified versions of your application code. When the application code changes, it should generate different results and cause unit tests to be failed. If a unit test does not fail, it may indicate an issue with the test suite.

In conclusion, normal code coverage highlights code that is definitely not tested, while mutation testing highlights code that definitely is tested.

1. **The process of mutation testing**

In mutation testing, the original program is called p, a set of intended faulty programs p’, called mutants which are generated by a few single syntactic changes to the original program p. For example:

**Program p**

If (a > 0 && b >0) {

Return 1;

}

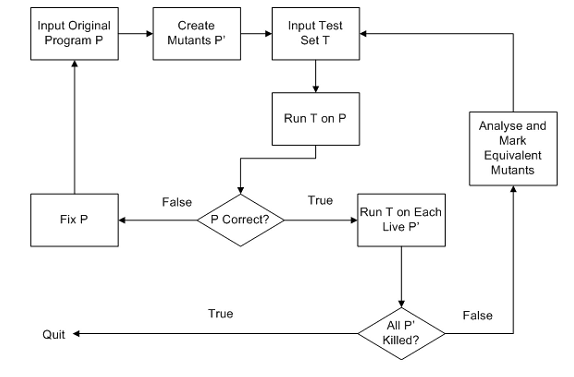
**Mutant p’:**

If (a> 0 || b >0) {

Return 1;}

A transformation rule which is used to generate a modified program (mutant) is called mutation operator (mutators). The bellowing table lists set of standard mutation operators for Fortran programming language

|  |  |
| --- | --- |
| **Mutator** | **Description** |
| LCR | Logical connector replacement |
| AOR | Arithmetic operator replacement |
| CRP | Constant replacement |
| ROR | Relational operator replacement |
| RSR | Return statement replacement |
| SDL | Statement deletion |
| …. | …. |



Mutation testing process contains following processes:

* Step 1: Input supplying:
  + Original program p
  + Mutant p is created by applying different mutation operators.
  + A test set T is supplied to the system.
* Step 2: The test set needs to be successfully executed against the original program p to check its correctness for the test case.
* Step 3: If p is incorrect, it needs to be fixed before running other mutants.
* Step 4: If p is correct, each mutant p’ will be run against test set T.
* Step 5: If the result of running p’ is different from result of running p, then mutant p’ is said to be killed, otherwise it is said to have “survived”.

After all test cases are run, there may still be some surviving mutants. To improve test set T, we should define more test case to kill these surviving mutants.

1. **Basic concepts**
   1. **Mutants:** modified version of the original program by applying different transformation rules is called mutants.
   2. **Mutation operators:** set of transformation rule which is used to change the original program is called mutation operators. To increase the flexibility of mutation testing in practice, Jia and Harman introduced a scripting language, the Mutation Operation Constraint Script (MOCS). MOCS provides 2 ways for mutation operators:
      * + **Direct substitution constraint:** allow user to select a specific transformation rule that performs a simple change.
        + **Environmental Condition Constraint:** is used to specify domain for applying mutation operators.
   3. **Equivalent mutants**

There are some mutants which always produce the same result as the original program, although they are syntactically different. Automatically detecting all equivalent mutants is almost impossible. It requires human decision.

* 1. **Mutation score**

Is the ratio of the number of killed mutants over total number of none-equivalent mutants. Mutation testing aims to increase the quality of test suite by raise mutation score up to 1. It means that the test set T is sufficient to detect all faults denoted by mutants.

1. **The problem of mutation testing**

Although mutation testing is an powerful way to assess the quality of test suite, it still has many problems which prevent it becoming a popular testing techniques. The most difficult problem is high computation costs of executing huge number of mutants against a test set. The second problem is related to amount of human effort in detecting equivalent mutants. In other words, if a mutation survives, you can’t tell if your test suite is deficient without checking to see if it is equivalent or not. In the scope of this paper, we will introduce some techniques to overcome the first challenge.

* 1. **Cost reduction techniques**

Examining a small program which contains 68K line of code and 70K line of test code. It takes 10 seconds to compile, 16 seconds to run the unit test. If we use small number of mutation operation (for example 10 mutation operators), the statistical result shows that:

* + - It might generate about 10k mutations.
    - If it takes 1 second to compile each mutation 🡪 2.5 hours to generate the mutants.
    - We will run test suite 10k times (respectively to 10k mutations) 🡪 44.5 hours

Totally, it requires almost 2 days to analyze a small program. Cost reduction techniques for mutation testing are classified into 2 types: reduction of generated mutants and reduction of execution cost.

* + 1. **Mutant reduction techniques**

The solution of mutant reduction is to find a subset of mutant M’ from M where MS(M’) is equivalent to MS(M) (MS: Mutation score). There are 4 techniques to reduce number of generated mutants:

* + - * **Mutant sampling**: is a simple approach that randomly chooses a small subset of mutants from the entire set.
      * **Mutant clustering**: Generating all first order mutants and then classifying the first order mutants into different clusters based on killable test cases. Only a small number of mutants are selected from each cluster to be used in Mutation Testing.
      * **Selective mutation**: Reducing number of mutation operations applied. We aim to choose a small set of mutation operators that generate a subset of all possible mutants without affecting to test effectiveness. Wong and Mathur suggested to use two mutation operators: ABS and ROR. Offutt et al suggested to use 5 operators which are ABS, UOI, LCR, AOR and ROR. These key operators achieved 99.5% mutation score.
    1. **Execution cost reduction techniques**

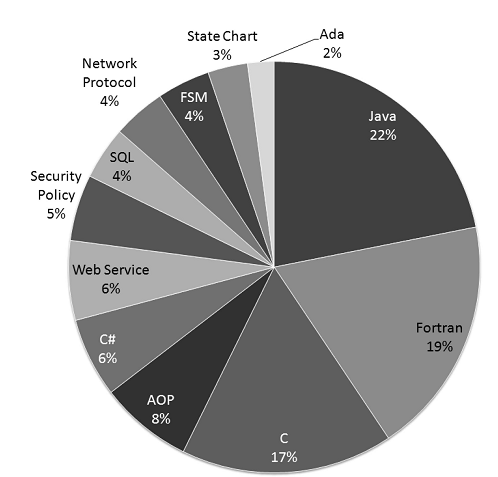
There are 3 types to optimize the execution process. They are:

* + - * Strong mutation: is also known as traditional mutation testing, proposed by DeMillo et al. In a strong mutation, for a given program p, a mutant m is said to be killed when m gives a different output from original program p.
      * Weak mutation (is used by PIT): a program is assumed to contains set of components C = (c1…cm). Mutant m is made by changing component cm. Mutant m is said to be killed if output of execution of component cm is different from mutant m. This means we don’t need to evaluate entire program, the mutants need only to be checked immediately after the execution point of the mutant or mutated component. There may be a concern that different components of original program may cause different result so that weak mutation can be less effective than strong mutation. However, there are many research paper which prove that under certain conditions (such as set of mutation operators), test sets generated by weak mutation can be also expected to be effective as strong mutation.
      * Runtime optimization techniques: There are some techniques which are used to improve runtime performance of mutation testing.
        + Interpreter-based techniques: In this technique, the result of a mutant is interpreted from its source code directly. This technique requires a lot of cost of interpretation.
        + Compiler-based technique: Each mutant is first compiled into an executable program; the compiled mutant is then executed by a number of test cases.
        + Compiler-Integrated Technique: an instrumented compiler is used to generate and compile mutants because there are small differences between mutants. The instrumented compiler will create 2 output from original program: an executable code of original program and set of patches for mutants. Each mutant is created by applying instruction in each patch.
        + Mutant schema generation: Instead of compiling mutants separately, this technique will organize all possible mutant into a super mutant. Therefore, to run each mutant against the test suite, only this super mutant needs to be compiled.
        + ByteCode Translation technique: this is the most recent technique and is applied in PIT program. In this technique, each mutant is generated from the compiled object code of the original program, instead of source code. This means that mutant can run without compilation. However, not all language support medium to manipulate byte code and not all mutation operators can be applied to ByteCode level.

1. **The application of mutation testing**

Mutation testing can be applied in both program source code (Program mutation) and Program Specification (Specification Mutation)

* + - Program mutation: is white box testing when we insert the fault into source code and programmers can see the content of mutants. In program mutation, mutation testing can be applied into unit test and integration test. In program mutation, mutation testing has been already applied to Ada, C, Fortran, Java, C#, SQL
    - Specification mutation: belongs to black box testing where faults are seeded into program specifications. There are some popular mutation testing for specification such as: FSM, State chart
    - Other testing application: mutation testing can be used for regression testing, test data generation. In this paper, we only introduce about regression testing which helps to prioritize test case. Do and Rothermel measured how quickly a test set can detect the mutant. Testing sequences are scheduled based on the rate of mutant killing.



1. **PITEST (PIT)**

* PIT can analyze 10k mutations in about 3 minutes.
  + Compilation cost: Create mutations by manipulating bytecode. 10k mutations generated in < 1 second.
  + Running test:
    - Stop when one fails (Weak mutation)
    - Choosing tests carefully: only a subset of tests could kill each mutant.
  + Incremental analysis
  + Parallel

1. **Evaluation**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Mutation Testing for C#** | **Integrated Tools** | | | | | | **Technology** | |
| **.Net Frame work** | **Visual Studio** | **NUnit** | **NCover** | **SVN** | **Status** | **Mutation reduction Techniques** | **Run time optimization techniques** |
| Nester | 2.0 | 2005 | 2.4.2 | No | No | Not support |  |  |
| Cream (v3.0) | 4.0  (current version: .Net 4.5) | 2008,2010  (Current version: 2017) | 2.5.7  (Current version: 3.10.1) | 3.4.14  (current version: NCover 5.1) | Yes | Still support | Full support (Sampling, Clustering and Selective mutation) | Mutate source code (by NRefactory) |
| Visual Mutator v2.1 | 4.0 | 2013 (Current version: 2017) | NUnit 2.6.3 | No | No | Still support |  | Mutate code after compilation (CIL) |
| Suggested solution | 4.0 and 4.5 | VS 2015 and newer | NUnit 3.10.1 | Integrate with OpenCover, instead of NCover because OpenCover is better (POC) | Maybe |  | Sampling and Selective mutation | Mutate code after compilation (CIL) |

1. Refererences
2. Advanced Mutation Operators Applicable in C# Programs - A. Derezi nska
3. A. Derezi nska, “Quality Assessment of Mutation Operators Dedicated for C# Programs,”Proc. Sixth Int’l Conf. Quality Software,Oct. 2006
4. A. Derezi nska and A. Szustek, “CREAM—A System for ObjectOriented Mutation of C# Programs,” technical report, Warsaw Univ. of Technology, 2007
5. An Analysis and Survey of the Development of Mutation Testing - Yue Jia
6. <http://pitest.org/>
7. https://www.codeproject.com/Articles/362076/Understanding-Common-Intermediate-Language-CIL