Mutation Testing:
A New Approach to Automatic Error-Detection

Mutation Testing is a powerful method for finding errors in software programs.
This paper will describe the process of mutation testing and how a new approach to this technology benefits the software industry.

What is Mutation Testing?

Mutation testing has been well-known to computer scientists for decades. It was introduced as a way of measuring the accuracy of test suites. In general, there is no easy way to tell if the test suite thoroughly tests the program or not. If the program passes the test suite, one may only say that program works correctly on all the cases that are included in the test suite. The more cases a test suite contains, the higher the probability that the program will work correctly in the real world. However, there is no mathematical way to measure how accurate the test suite is and the probability that the program will work correctly.

The idea of mutation testing was introduced as an attempt to solve the problem of not being able to measure the accuracy of test suites. In mutation testing, one is in some sense trying to solve this problem by inverting the scenario. The thinking goes as follows: Let's assume that we have a perfect test suite, one that covers all possible cases. Let's also assume that we have a perfect program that passes this test suite. If we change the code of the program (this process is called mutating) and we run the mutated program (mutant) against the test suite, we will have two possible scenarios:

1) The results of the program were affected by the code change and the test suite detects it. We assumed that the test suite is perfect, which means that it must detect the change. If this happens, the mutant is called a killed mutant.

2) The results of the program are not changed and the test suite does not detect the mutation. The mutant is called an equivalent mutant.

If we take the ratio of killed mutants to all the mutants that were created, we get a number that is smaller than 1; this number measures how sensitive the program is to the code changes.

In the normal world, we do not have the perfect program and we do not have the perfect test suite. Thus, we can have one more scenario:

3) The results of the program are different, but the test suite does not detect it because it does not have the right test case.

If we take the ratio of all the killed mutants to all the mutants generated, we get a number smaller than 1 that also contains information about accuracy of the test suite.

In practice, there is no way to separate the effect that is related to test suite inaccuracy and that which is related to equivalent mutants. In the absence of other possibilities, one can accept the ratio of killed mutants to all the mutants as the measure of the test suite accuracy.

The following C code example illustrates the ideas described above (this code sample, and all following code samples, will compile and run on any Linux machine):

```
main(argc, argv)                         /* line 1 */
int argc;                                    /* line 2 */
char *argv[];                            /* line 3 */
{                                        /* line 4 */
  int c=0;                               /* line 5 */
                                         /* line 6 */
    if(atoi(argv[1]) < 3){               /* line 7 */
      printf("Got less then 3\n");       /* line 8 */
      if(atoi(argv[2]) > 5)              /* line 9 */
        c = 2;                               /* line 10 */
    }                                    /* line 11 */
    else                                 /* line 12 */
      printf("Got more then 3\n");           /* line 13 */
   exit(0);                              /* line 14 */
}                                        /* line 15 */
```

Let's call the above program test1.c. The program can be compiled with the following command:

```
cc -o test1 test1.c.
```

The program reads its arguments and prints messages accordingly.

Now let's assume that we have the following test suite that tests the program:

test case 1:
    input 2 4
    output Got less than 3
test case 2:
    input 4 4
    output Got more than 3
test case 3:
    input 4 6
    output Got more than 3
test case 4:
    input 2 6
    output Got less than 3
test case 5:
    input 4
    output Got more than 3

The test suite above is quite representative of the test suites in the industry. It tests positive tests, which means it tests if the program reports correct values for the correct inputs. It completely neglects illegal inputs to the program. The test1 program fully passes the test suite; however, it has serious hidden errors.

Now, let's mutate the program. We can start with the following simple changes:

Mutant 1: change line 9 to the form
```
     if(atoi(argv[2]) <= 5)
```

Mutant 2: change line 7 to the form
```
     if(atoi(argv[1]) >= 3)
```

Mutant 3: change line 5 to the form
```
     int c=3;
```

If we run this modified program against the test suite, we will get the following results:

Mutant 1 and 3 - program will completely pass the test suite
Mutant 2 - program will fail all test cases.

Mutants 1 and 3 do not change the output of the program, and are thus equivalent mutants. The test suite does not detect them. Mutant 2, however, is not an equivalent mutant. Test cases 1-4 will detect it through wrong output from the program. Test case 5 may have different behavior on different machines. It may show up as bad output from the program, but at the same time, it may be visible as a program crash. We will return to this case a little bit later in the paper.

If we calculate the statistics, we see that we created three mutants and only one was killed. This tells us that the number that measures the quality of the test suite is 1/3. As we can see, the number 1/3 is low. It is low because we generated two equivalent mutants. This number should serve as a warning that we are not testing enough. In fact, the program has two serious errors that should be detected by the test suite.

Let's return to Mutant 2 and its run against test case 5. If the program crashes, then the mutation testing that we performed not only measured the quality of the test suite, but also detected a serious error in the code. This is how mutation testing can find errors.

Consider a different equivalent mutant (Mutant 4):

```
main(argc, argv)                        /* line 1 */
int argc;                                   /* line 2 */
char *argv[];                           /* line 3 */
{                                       /* line 4 */
  int c=0;                              /* line 5 */
  int a, b;                             /* line 6 */
```

```
                                            /* line 7 */
  a = atoi(argv[1]);                        /* line 8 */
  b = atoi(argv[2]);                        /* line 9 */
  if(a < 3){                                /* line 10 */
    printf("Got less then 3\n");            /* line 12 */
    if(b > 5)                               /* line 13 */
      c = 2;                                /* line 14 */
  }                                         /* line 15 */
  else                                      /* line 16 */
    printf("Got more then 3\n");            /* line 17 */
 exit(0);                                        /* line 18 */
}                                           /* line 19 */
```

The difference between Mutant 4 and previous mutants is that Mutant 4 was created in an attempt to make an equivalent mutant. This means that when it was constructed, an effort was made to build a program that should execute exactly the same as the original program. If we run Mutant 4 against the test suite, test case 5 will probably fail-- the program will crash. As we can see, by creating an equivalent mutant, we in fact increased the detection power of the test suite. The conclusion that we can draw here is that we can increase the accuracy of the test suite in two ways:

1) increase the number of test cases in the test suite
2) run equivalent mutants against the test suite.

These conclusions are very important; the second conclusion is especially important because it tells us that mutants help us test more effectively. In our examples, we created each mutant by manually making a single change to a program. However, the process of generating mutants is difficult and time consuming. A tool that can automatically mutate programs is necessary if techniques that require mutants are going to be used in the real world. It is very difficult to build a tool that creates meaningful non-equivalent mutants; however, if we are focusing on error detection, we need to generate equivalent mutants, not non-equivalent mutants. Fortunately, it is possible to generate equivalent mutants automatically. In fact, many people are already doing so, but do not know it. The following program helps illustrate how this is done:

```
int doublew(x)
int x;
{ return x*2; }

int triple( y)
int y;
{ return y*3; }

main() {
  int i = 2;
  printf("Got %d \n", doublew(i++)+ triple(i++));
}
```

This program has no input and only one output. In principle it only needs one test case:

Test case 1:
  input none
  output 12

The interesting thing about this program is that it can give the answer 13 or 12 depending on the behavior of the compiler. We are not going to explain why in this paper; we will leave figuring out why this occurs as an exercise for the reader. Suppose that the programmer was given the task of creating this program and making sure that it runs on two different platforms. If the platforms have compilers that exhibit different behavior, the programmer will discover the difference when running the program, and this will trigger the question, “What is wrong?” This will probably result in the program’s problem being fixed.

The scenario described above is very common. People who port programs from one platform to another discover that a lot of errors are flushed during porting. What porting really is then, is a form of mutation testing. Each compiler generates equivalent mutants of the program. This time however, the mutant is in the form of object code, not source code. Our company discovered this benefit a long time ago, completely by accident. We started to wonder why we find certain bugs when we port software from one platform to another. Finally we figured it out: mutation testing is responsible. Since then, it has been our standard policy to develop a product on at least two platforms. This way we significantly reduce the amount of errors in the code.

An especially large number of errors is discovered when code is ported from a platform of one architecture to another. For example, porting code from the 32 bit platform to the 64 bit platform will flush all the errors related to being sloppy with pointers and integers.

Now let’s return to the previous example. Suppose that we create following equivalent mutant:

```
int doublew(x)
int x;
{ return x*2; }

int triple( y)
int y;
{ return y*3; }

main() {
  int i = 2;
  int a, b;

  a = doublew(i++);
  b = triple(i++);
  printf("Got %d \n", a+b);
}
```

The result of this program does not depend on the compiler and is, in fact, exactly predictable - it is 13. If we run the mutant against the test suite, we will discover the error.

We have also discovered that a good way to create equivalent mutants is by using the method of source code instrumentation. A tool that uses this method for runtime error detection or coverage analysis or profiling will have the ability to discover very complicated errors in the code. This is what we have learned from our customers who use Insure++ for error detection. The tool reads the source code and, by design, writes out new source code that is equivalent to the original code, but which also contains code that performs error checking. In fact, the transformation of the previous program was taken from Insure++ intermediate code.

It is also important to mention here that instrumentation of the code at the object level is not effective. This instrumentation was not designed to modify code generated by the compiler, so it is not useful at all for mutation testing.

Creation of equivalent mutants can uncover a lot of very strange errors. This method is particularly important in C++. In this language, there is a lot of opportunity for making errors. This is demonstrated by the following list of errors that can be detected:

1) Lack of copy constructors or bad copy constructors
2) Missing or incorrect constructors
3) Wrong order of initialization of code
4) Problems with operations of pointers

The most amazing thing about mutation testing is that it can discover errors that normally are almost impossible to detect. Frequently, when these errors are uncovered, they manifest themselves as the program crashing. Very often, programmers do not understand that. The equivalent mutant is an opportunity to discover errors, not a headache. Typically, programmers expect equivalent mutants to behave the same as the original program. If this were true all the time, mutation testing would be completely useless.

Benefits of Mutation Testing

This new approach to automatic mutation testing brings a new level of error-detection to the software developer. Tools that automate mutation testing are able to uncover ambiguities in code previously thought impossible to detect automatically. By using tools that incorporate mutation testing into state-of-the-art error-detection technology, developers are able to flush out more faults than with any other technology.

Software developers and testers using tools that incorporate this approach to mutation testing will benefit enormously, as such tools automatically uncover more bugs than any other technology. The customer also benefits from mutation testing, as the program a user receives is less buggy and more reliable. This increased confidence will in turn benefit your company where it matters most- the bottom line.