White-Box Testing and Code Coverage

Milestone 3

George Dimitrov, Yasir al-Bender, Cory Ebner & Will Nguyen

ABSTRACT

The selection of inputs for test cases is critical. Black-Box testing may provide an initial starting point but Code coverage and White-Box testing may reveal more inputs that can be used towards testing. The tests used in Milestone 2 will undergo this process and the results and recommendations are documented in this paper.

Table of Contents

Table of Figures

1. Initial Code Coverage Results

Our test team consists of four members: George, Yasir, Cory and Will. Each member performed Black-Box testing on four functions of their choosing. The code coverage tool used was **EclEMMA**, a coverage measuring tookit made for Java.

The following tables are a summary of the results of automated code coverage measurements done by each respective member. For each table, there are corresponding collections of screenshots taken of raw data located in Appendix A.

1.1 George's Initial Results

Class Name	**Code Coverage**
Cube	100%
Square	100%
Inverse	100%
Factorial	100%

1.2 Yasir's Initial Results

Class Name	**Code Coverage**
Inverse Cosine	100%
Cosine	100%
Sine	100%
Tangent	100%

1.3 Cory's Initial Results

Class Name	**Code Coverage**
Combination	100%
Cube Root	100%
Logarithmic (Base 10)	100%
Addition	100%

1.4 Will's Initial Results

Class Name	**Code Coverage**
Natural Logarithm	100%
Inverse Logarithm	100%
AND	93%
XOR	90%

2. Manual Coverage Results

To ensure that the code coverage results from the programs used are accurate, each member of the test team will select a single unit and perform manual code coverage calculations. The differences between the automated and manual code coverage calculations will be noted and discussed below:

2.1 George's Manual Results

The following calculations are for the `function(double x)` method in `factorial.java`

```
public double function( double x ){
        if(  x < 0 || Math.round( x ) - x != 0 ) ← 4 branches
            throw new ArithmeticException( "Factorial error" );
        else if( x == 0 ) ← 2 Branches
            return 1;
        else
            return x * function( x - 1 );
}
```

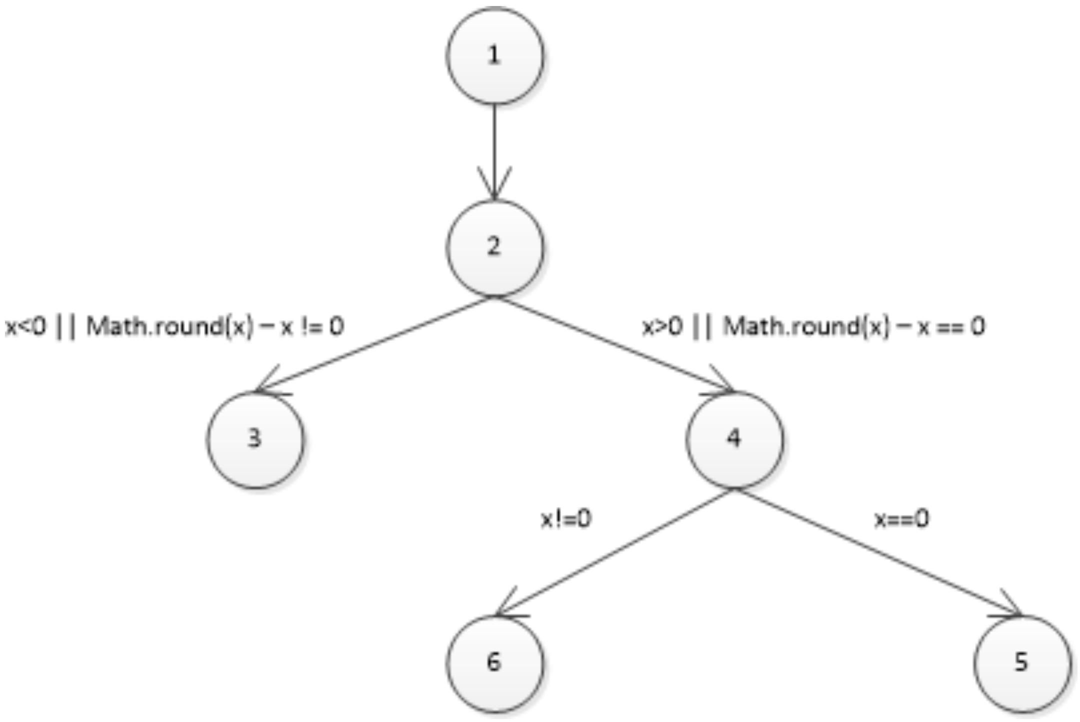


Figure 1: Factorial Control Flow Graph

Condition Coverage:

In the first line:

if(x < 0 || Math.round(x) - x != 0)

we have:

x<0 which has 2 outcomes, x<0 or x>0
we also have:
Math.round(x) - x != 0 or Math.round(x) - x == 0, this is another 2 outcomes for a total of 4 possibilities in the first line.
Next branch is:
if(x==0)
So either x==0 or x!=0, that's +2 outcomes for total branches.

In the end we have:
6 lines of code
2+2+2 = 6 branches

Statement and Branch Coverage:

- Test case 1 covers all branches, part 1(Testing with 5) covers nodes 1-2-4-6-5
- Part 2 (Testing with 0) covers nodes 1-2-4-5
- Part 4 (Testing 2.5) covers nodes 1-2-3, but this was a design fault by developer.
- Test case 2 covers nodes 1-2-3 by testing -5.

Path Coverage:
Those same tests used in statement coverage also accomplish path coverage for the entire function.

2.1 Yasir's Manual Results

The following calculations are for the `function(oobjectx)` method in `acos.java`

```
public OObject function( OObject x ){
             if( x instanceof Complex ){
                 Complex c = (Complex)x;
                 if( scale != 1 && StrictMath.abs( c.imaginary() ) > 1e-6 )
                          throw new RuntimeException( "Error" );
                 if( scale != 1 && StrictMath.abs( c.real() ) > 1 )
                          throw new RuntimeException( "Error" );
                 return c.acos().scale( iscale );
             } else {
                 return x.acos( angleType );
             }
}
```

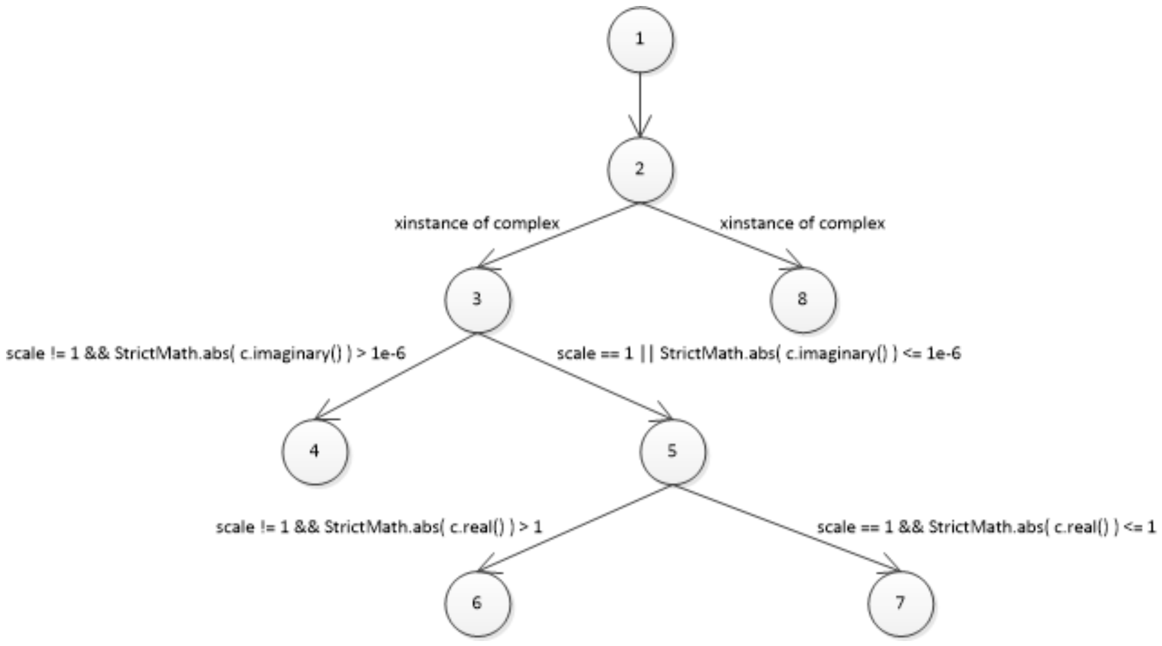

1
2
xinstance of complex
xinstance of complex
3
8
scale != 1 && StrictMath.abs(c.imaginary()) > 1e-6
scale == 1 || StrictMath.abs(c.imaginary()) <= 1e-6
4
5
scale != 1 && StrictMath.abs(c.real()) > 1
scale == 1 && StrictMath.abs(c.real()) <= 1
6
7

Figure 2: Inverse Cosine Control Flow Graph

Condition Coverage:

- x instanceof Complex
- x ! instanceof Complex
- scale != 1 && StrictMath.abs(c.imaginary()) > 1e-6
- scale == 1 || StrictMath.abs(c.imaginary()) <= 1e-6
- scale != 1 && StrictMath.abs(c.real()) > 1
- scale == 1 && StrictMath.abs(c.real()) <= 1

All conditions are covered

Path/Statement and Branch Coverage:

- testOObject() traverse nodes 1-2-8
- testComplexObject() traverses 1-2-3-5-7
- testErrorComplexObjectImagineryGreaterThan() traverses 1-2-3-4
- public void testErrorComplexObjectRealGreaterThan1() traverses 1-2-3-5-6

2.2 Cory's Manual Results

The following calculations are for the `function(double x, double y)` method in `combination.java`

```
public double function( double x, double y ){
             if(  x < 0 || Math.round( x ) - x != 0 )
                    throw new ArithmeticException( "Combination error" );
             if(  y < 0 || y > x || Math.round( y ) - y != 0 )
                    throw new ArithmeticException( "Combination error" );
```

```
                if( y == 0 )
                        return 1;
                else
                        return x  / y * function( x - 1, y - 1 );
}
```

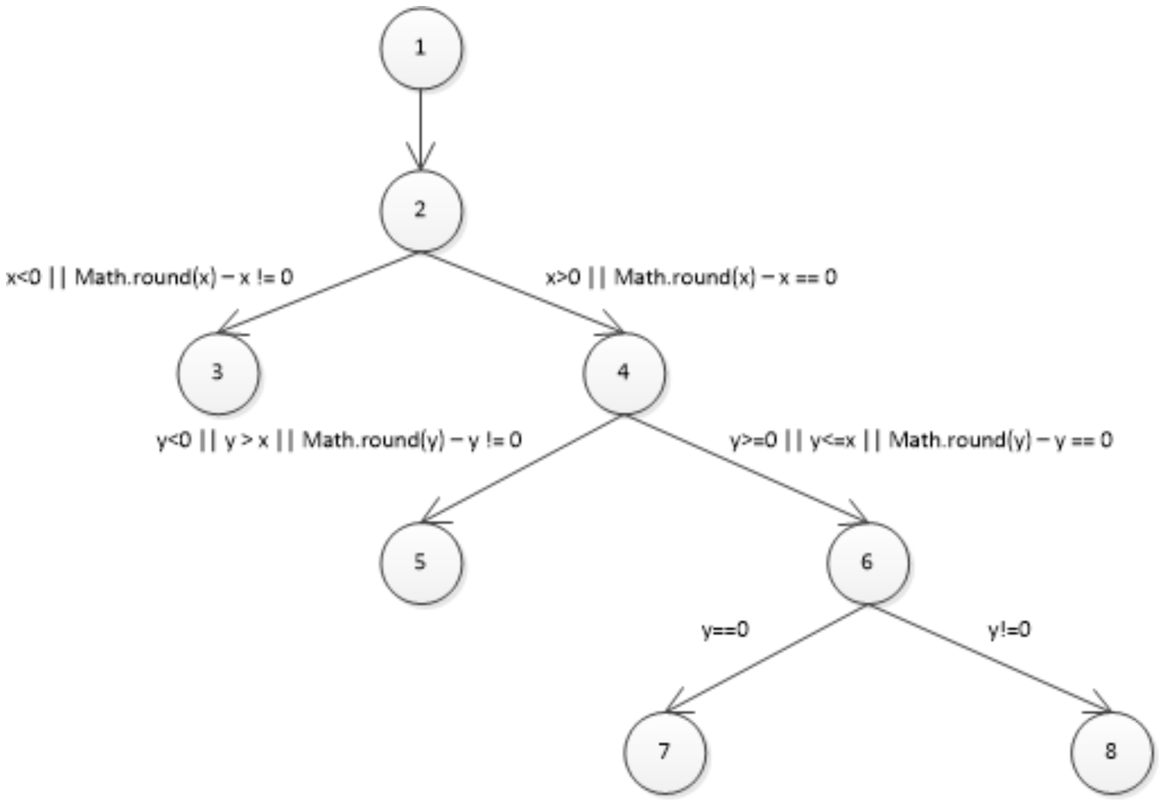

1
2
x<0 || Math.round(x) – x != 0
x>0 || Math.round(x) – x == 0
3
4
y<0 || y > x || Math.round(y) – y != 0
y>=0 || y<=x || Math.round(y) – y == 0
5
6
y==0
y!=0
7
8

Figure 3: Combination Control Flow Graph

Condition Coverage:

- x < 0
- x >= 0
- Math.round(x) –x != 0
- Math.round(x) –x == 0
- y < 0
- y >= 0
- y > X
- y < x
- Math.round(y) –y != 0
- Math.round(y) –y == 0
- y == 0
- y != 0

$$Condition\ Coverage = \frac{\#\ of\ condition\ cases\ covered}{(\#\ of\ all\ conditions - \#\ of\ unreachable)}$$

$$= \frac{12}{(12-0)} = 1$$

Therefore we have 100% condition coverage.

Statement and Branch Coverage:

- Test 1 traverses nodes 1-2-3
- Test 3 traverses nodes 1-2-4-5
- Test 5 traverses nodes 1-2-4-6-7
- Test 4 traverses nodes 1-2-4-6-8

All nodes and edges are covered; therefore we have 100% statement and branch coverage.

Path Coverage:

- Test 1 traverses nodes 1-2-3
- Test 3 traverses nodes 1-2-4-5
- Test 5 traverses nodes 1-2-4-6-7
- Test 4 traverses nodes 1-2-4-6-8

All paths leading from the initial to the final node are covered; therefore we have 100% path coverage.

2.3 Will's Manual Results

The following calculations are for the `function(double x, double y)` method in `xor.java`

```
public double function( double x, double y ){
             if( Double.isNaN( x ) || Double.isNaN( y )
                 || Double.isInfinite( x )
                 || Double.isInfinite( y ) )
                 throw new RuntimeException( "Boolean Error" );
             if( Math.abs( y ) > Math.abs( x ) ){
                 double tmp = x;
                 x = y;
                 y = tmp;
             }
             long x_bits = Double.doubleToLongBits( x );
             boolean x_sign = (x_bits >> 63) == 0;
             int x_exponent = (int)((x_bits >> 52) & 0x7FFL);
             long x_significand = x_exponent == 0 ? (x_bits &
0xFFFFFFFFFFFFFL) << 1
```

```
                    : (x_bits & 0xFFFFFFFFFFFFFL) | 0x10000000000000L;
                long y_bits = Double.doubleToLongBits( y );
                boolean y_sign = (y_bits >> 63) == 0;
                int y_exponent = (int)((y_bits>>52) & 0x7FFL);
                long y_significand = y_exponent == 0 ? (y_bits &
0xFFFFFFFFFFFFFL) << 1
                    : (y_bits & 0xFFFFFFFFFFFFFL) | 0x10000000000000L;
                y_significand >>= (x_exponent - y_exponent);

                // actually carry out the operation
                x_significand ^= y_significand;

                // now reconstruct result
                if( x_exponent == 0 )
                    x_significand >>= 1;
                else {
                    if( x_significand == 0 ) return 0;
                    while( (x_significand & 0x10000000000000L) == 0 ){
                       x_significand <<= 1;
                       --x_exponent;
                       if( x_exponent == 0 ){
                           x_significand >>= 1;
                           break;
                       }
                    }
                    x_significand &= 0xFFFFFFFFFFFFFL;
                }

                x_bits = ((long)x_exponent) << 52;
                x_bits |= x_significand;

                double result = Double.longBitsToDouble( x_bits );

                // deal with signs
                if( x_sign ^ y_sign )
                    result =- result;
                return result;
}
```

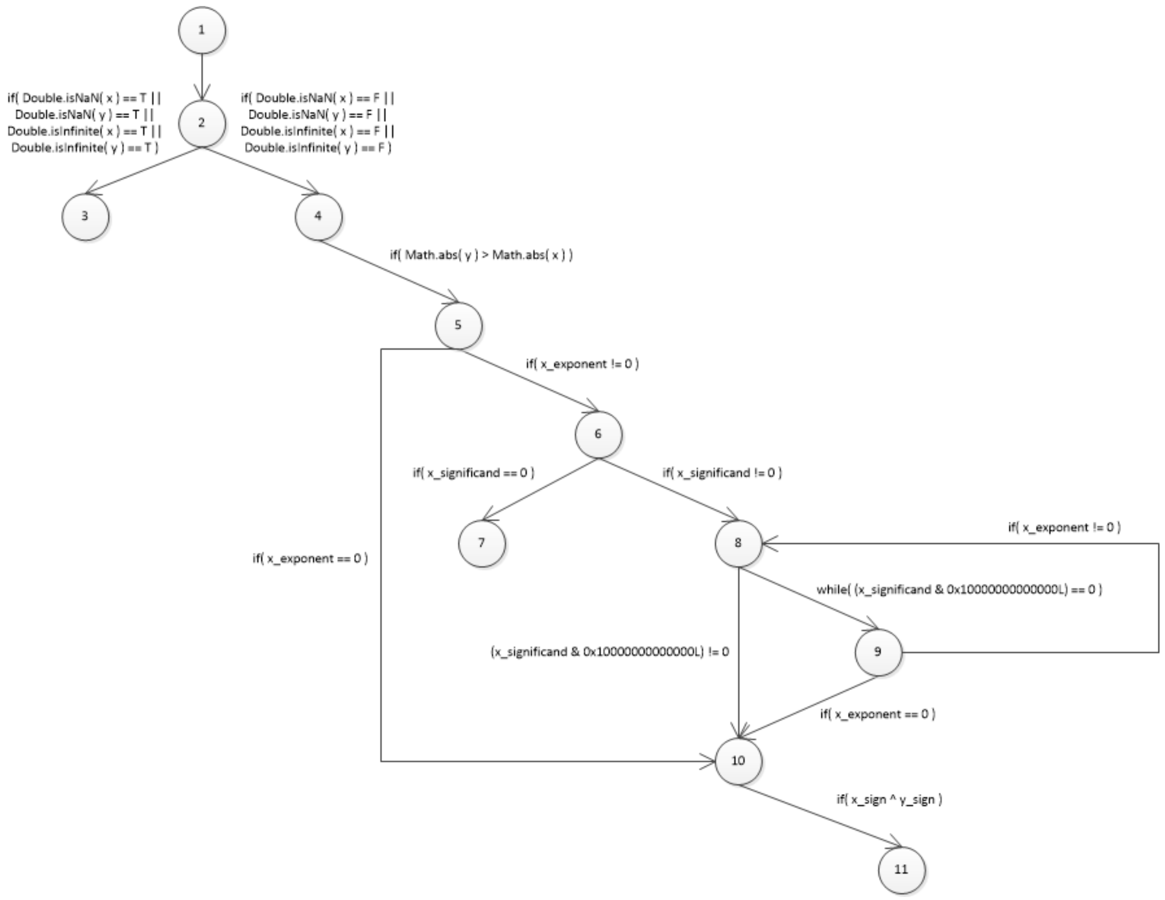

1
if(Double.isNaN(x) == T || Double.isNaN(y) == T || Double.isInfinite(x) == T || Double.isInfinite(y) == T)
2
if(Double.isNaN(x) == F || Double.isNaN(y) == F || Double.isInfinite(x) == F || Double.isInfinite(y) == F)
3
4
if(Math.abs(y) > Math.abs(x))
5
if(x_exponent != 0)
6
if(x_significand == 0)
if(x_significand != 0)
7
8
if(x_exponent != 0)
if(x_exponent == 0)
while((x_significand & 0x10000000000000L) == 0)
(x_significand & 0x10000000000000L) != 0
9
if(x_exponent == 0)
10
if(x_sign ^ y_sign)
11

Figure 4: XOR Control Flow Graph

Condition Coverage:

- Double.isNaN(x) || Double.isNaN(y) || Double.isInfinite(x) || Double.isInfinite(y) == T
- Double.isNaN(x) || Double.isNaN(y) || Double.isInfinite(x) || Double.isInfinite(y) == F
- Math.abs(y) > Math.abs(x)
- x_exponent != 0
- x_exponent == 0
- x_significand == 0
- x_significand != 0
- x_significand & 0x10000000000000L) == 0
- x_significand & 0x10000000000000L) != 0
- x_exponent != 0
- x_exponent == 0
- x_sign ^ y_sign

$$Condition\ Coverage = \frac{\#\ of\ condition\ cases\ covered}{(\#\ of\ all\ conditions - \#\ of\ unreachable)}$$

$$= \frac{11}{(12-1)} = 1$$

The second instance of `x_exponent != 0` is infeasible due to it being highly unlikely and unreachable. This second statement (located on the path between node 9 to 8) should be obsolete because of the previous check for `x_exponent != 0` (located in the paths before nodes 7 and 8). Therefore we have 100% condition coverage.

Statement and Branch Coverage:

- Test 1 traverses nodes 1-2-4-5-10-11
- Test 3 traverses nodes 1-2-4-5-6-8-10-11
- Test 4 traverses nodes 1-2-4-5-6-7
- Test 10 traverses nodes 1-2-3
- Test 14 traverses nodes 1-2-4-5-6-8-9-10-11 **(This test was added to reach 100%, more about it in the next section)**

All nodes and edges are covered; therefore we have 100% statement and branch coverage.

Path Coverage:
Statement and Branch Coverage calculations showed that all except for one path was traverse. As mentioned in Condition Coverage, the path between node 9 and node 8 is infeasible and therefore should be removed. Without the infeasible path, we can conclude that we have 100% coverage.

3. Improved Code Coverage Attempts

Test member George and Cory did not have to perform any improvements with their test cases. Their results in Section 1 indicated that the selection of inputs reveals all possible errors with respect to the Test Selection Problem.

Code coverage and white-box testing showed Yasir, he had to add some extra test to test for methods inside the class that were previously not tested for. Some methods in the class were not being run and with the help of EclEMMA, these methods were targeted and special tests were created.

Tests added:

- public void testNameArray()
- public void testOObject()
- public void testComplexObject()
- public void testErrorComplexObject()
- public void testMain()

These tested the methods in the class that were previously not executed, and completed the 100% code coverage.

Results in Section 1 indicated that Will's set of inputs was incomplete for both ADD and XOR functions. After using Code Coverage tools, he was able to locate where in the code, his tests did not account for. Using this knowledge, he developed 3 new tests and was able to raise his coverage to 96.9% from 90%, in regards to the XOR function. This is presumably the highest coverage achievable due to an infeasible condition/path at line 79.

To improve the ADD function code coverage, the same process was used. Will developed 6 new tests and was about to raise his coverage to 97.1% from 93%. This is presumably the highest coverage achievable due to an infeasible condition/path; the ADD and XOR functions contain very similar code structure and this unit of the code was identical to the infeasible path in XOR.

3.1 Will's Improved Results

Class Name	**Code Coverage**
Natural Logarithm	100%
Inverse Logarithm	100%
AND	97.1% (+4.1%)
XOR	96.9% (+6.9%)

4. Conclusions and Recommendations

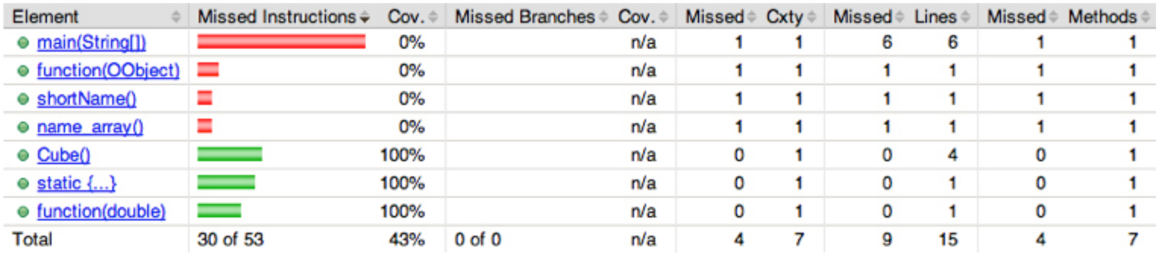
It is in the opinion of the test team that testing process is easier if Black-Box testing is done initially, then Code coverage and finally White-Box testing. Black-Box testing should be done first to find all obvious bugs; this will give testers a good initial guess of what the input selection should be with regards to the Test Selection problem. Code coverage should be done after to find any areas that Black-Box testing may have missed. Afterwards White-Box testing should be performed to eliminate those areas missed; the second and third steps should be repeated as much as possible in order to raise code coverage. The entire process should give the team a near complete selection of possible inputs to test with.

The team is also in agreement that EclEMMA is highly recommended in code coverage for Java programming. The eclipse plugin offers ease of use and makes it easy to locate where testing is absent.

Appendix A. – Initial Code Coverage Results

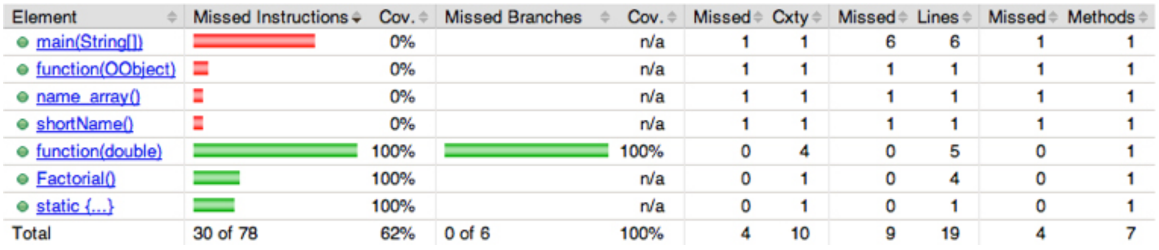
A.1 George’s Initial Code Coverage Results

A.1.1 Cube



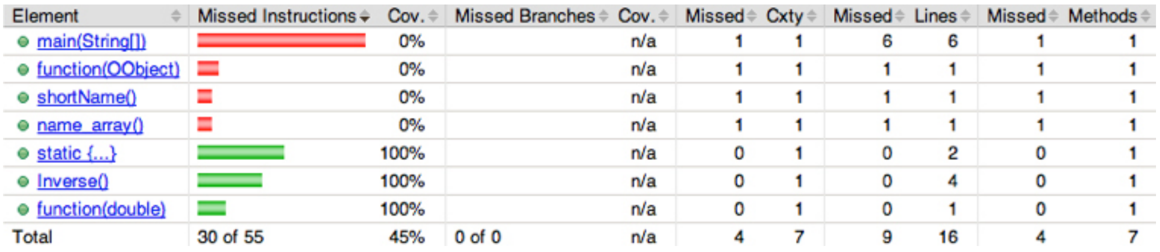
Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
main(String[])	🟥	0%		n/a	1	1	6	6	1	1
function(OObject)	🟥	0%		n/a	1	1	1	1	1	1
shortName()	🟥	0%		n/a	1	1	1	1	1	1
name_array()	🟥	0%		n/a	1	1	1	1	1	1
Cube()	🟩	100%		n/a	0	1	0	4	0	1
static {...}	🟩	100%		n/a	0	1	0	1	0	1
function(double)	🟩	100%		n/a	0	1	0	1	0	1
Total	30 of 53	43%	0 of 0	n/a	4	7	9	15	4	7

A.1.2 Factorial



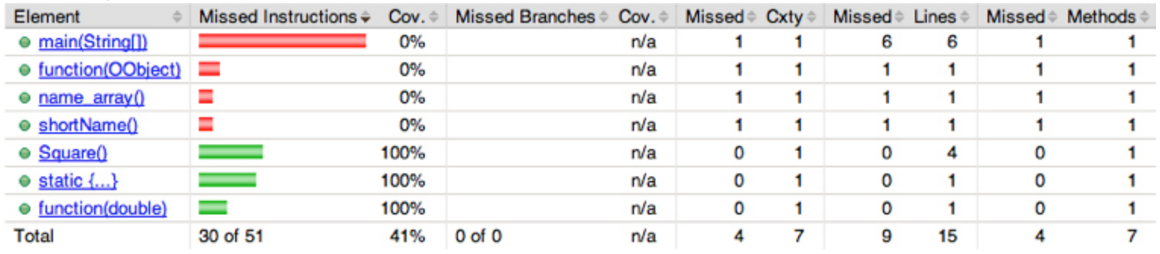
Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
main(String[])	🟥	0%		n/a	1	1	6	6	1	1
function(OObject)	🟥	0%		n/a	1	1	1	1	1	1
name_array()	🟥	0%		n/a	1	1	1	1	1	1
shortName()	🟥	0%		n/a	1	1	1	1	1	1
function(double)	🟩	100%	🟩	100%	0	4	0	5	0	1
Factorial()	🟩	100%		n/a	0	1	0	4	0	1
static {...}	🟩	100%		n/a	0	1	0	1	0	1
Total	30 of 78	62%	0 of 6	100%	4	10	9	19	4	7

A.1.3 Inverse



Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
main(String[])	🟥	0%		n/a	1	1	6	6	1	1
function(OObject)	🟥	0%		n/a	1	1	1	1	1	1
shortName()	🟥	0%		n/a	1	1	1	1	1	1
name_array()	🟥	0%		n/a	1	1	1	1	1	1
static {...}	🟩	100%		n/a	0	1	0	2	0	1
Inverse()	🟩	100%		n/a	0	1	0	4	0	1
function(double)	🟩	100%		n/a	0	1	0	1	0	1
Total	30 of 55	45%	0 of 0	n/a	4	7	9	16	4	7

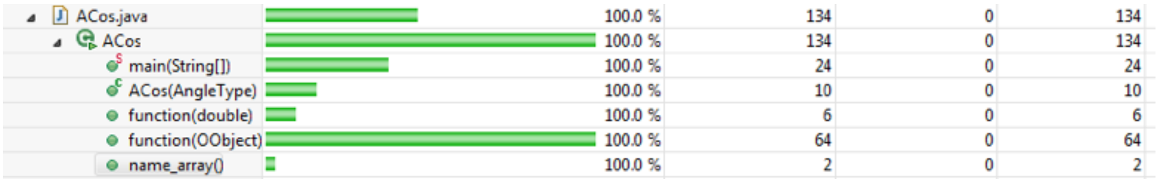
A.1.4 Square



Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
main(String[])	🟥	0%		n/a	1	1	6	6	1	1
function(OObject)	🟥	0%		n/a	1	1	1	1	1	1
name_array()	🟥	0%		n/a	1	1	1	1	1	1
shortName()	🟥	0%		n/a	1	1	1	1	1	1
Square()	🟩	100%		n/a	0	1	0	4	0	1
static {...}	🟩	100%		n/a	0	1	0	1	0	1
function(double)	🟩	100%		n/a	0	1	0	1	0	1
Total	30 of 51	41%	0 of 0	n/a	4	7	9	15	4	7

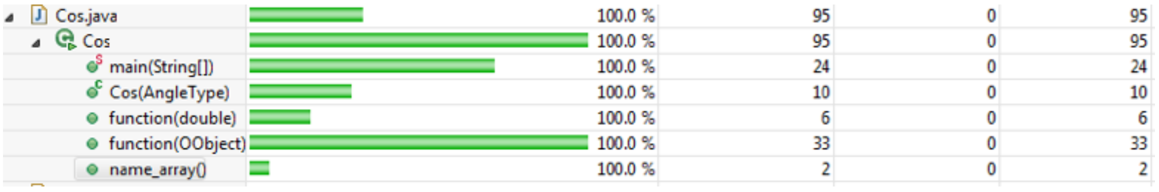
A.2 Yasir's Initial Code Coverage Results

A.2.1 Inverse Cosine



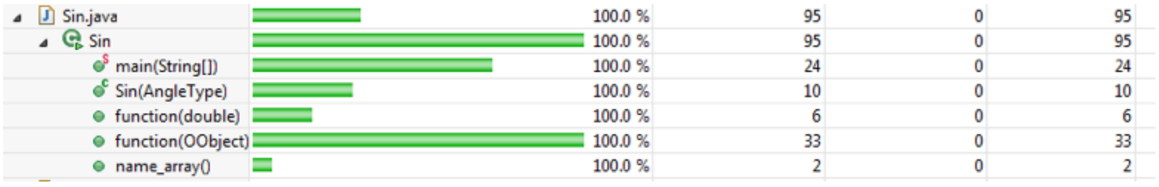
ACos.java	100.0 %	134	0	134
ACos	100.0 %	134	0	134
main(String[])	100.0 %	24	0	24
ACos(AngleType)	100.0 %	10	0	10
function(double)	100.0 %	6	0	6
function(OObject)	100.0 %	64	0	64
name_array()	100.0 %	2	0	2

A.2.2 Cosine



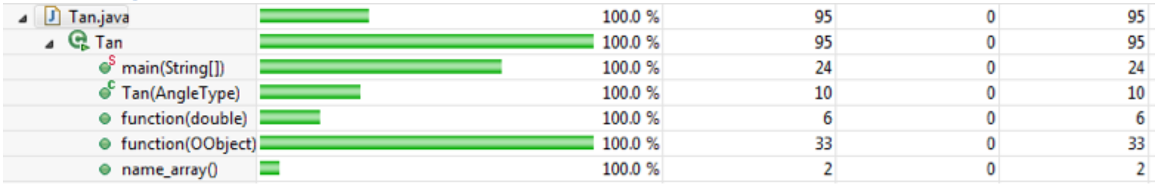
Cos.java	100.0 %	95	0	95
Cos	100.0 %	95	0	95
main(String[])	100.0 %	24	0	24
Cos(AngleType)	100.0 %	10	0	10
function(double)	100.0 %	6	0	6
function(OObject)	100.0 %	33	0	33
name_array()	100.0 %	2	0	2

A.2.3 Sine



Sin.java	100.0 %	95	0	95
Sin	100.0 %	95	0	95
main(String[])	100.0 %	24	0	24
Sin(AngleType)	100.0 %	10	0	10
function(double)	100.0 %	6	0	6
function(OObject)	100.0 %	33	0	33
name_array()	100.0 %	2	0	2

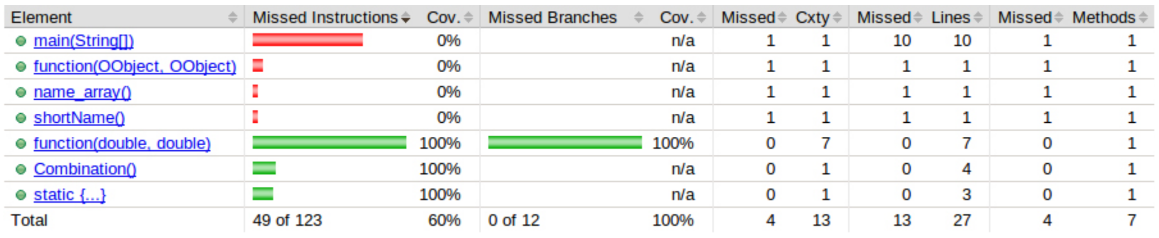
A.2.4 Tangent



Tan.java	100.0 %	95	0	95
Tan	100.0 %	95	0	95
main(String[])	100.0 %	24	0	24
Tan(AngleType)	100.0 %	10	0	10
function(double)	100.0 %	6	0	6
function(OObject)	100.0 %	33	0	33
name_array()	100.0 %	2	0	2

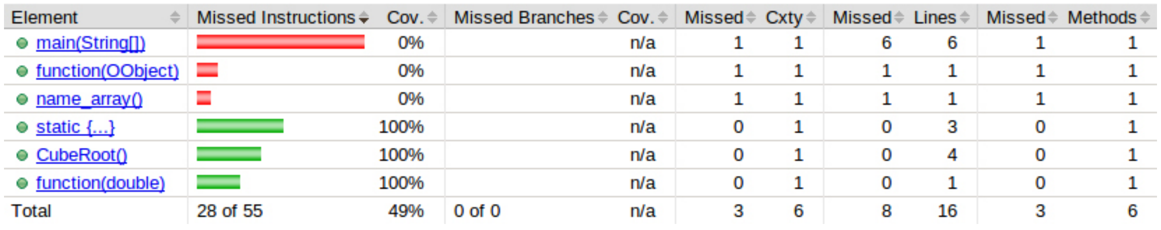
A.3 Cory's Initial Code Coverage Results

A.3.1 Combination



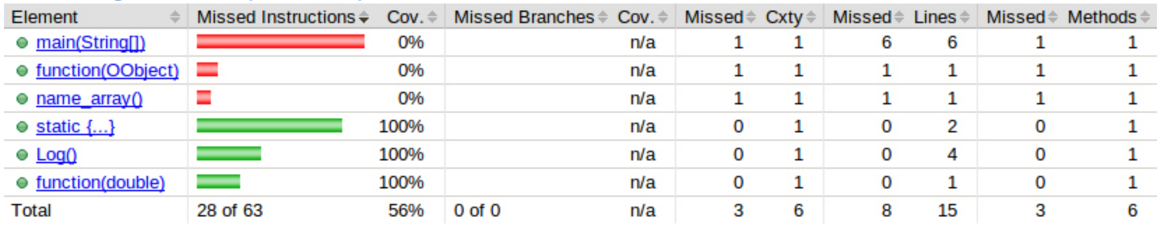
Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
main(String[])	🟥	0%		n/a	1	1	10	10	1	1
function(OObject, OObject)	🟥	0%		n/a	1	1	1	1	1	1
name_array()	🟥	0%		n/a	1	1	1	1	1	1
shortName()	🟥	0%		n/a	1	1	1	1	1	1
function(double, double)	🟩	100%	🟩	100%	0	7	0	7	0	1
Combination()	🟩	100%		n/a	0	1	0	4	0	1
static {...}	🟩	100%		n/a	0	1	0	3	0	1
Total	49 of 123	60%	0 of 12	100%	4	13	13	27	4	7

A.3.2 Cube Root



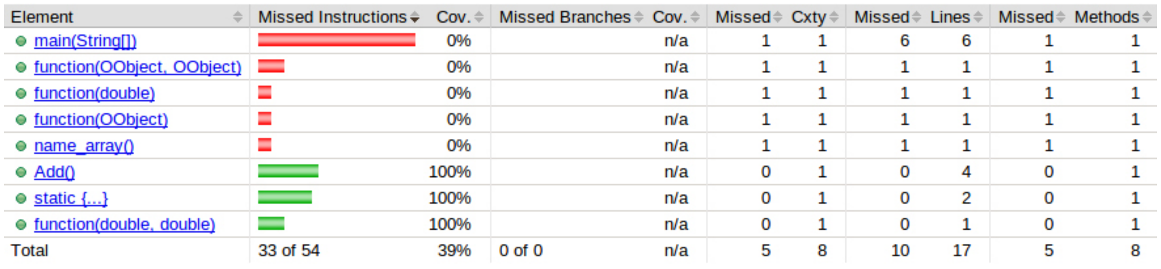
Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
main(String[])	🟥	0%		n/a	1	1	6	6	1	1
function(OObject)	🟥	0%		n/a	1	1	1	1	1	1
name_array()	🟥	0%		n/a	1	1	1	1	1	1
static {...}	🟩	100%		n/a	0	1	0	3	0	1
CubeRoot()	🟩	100%		n/a	0	1	0	4	0	1
function(double)	🟩	100%		n/a	0	1	0	1	0	1
Total	28 of 55	49%	0 of 0	n/a	3	6	8	16	3	6

A.3.3 Logarithmic (Base 10)



Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
main(String[])	🟥	0%		n/a	1	1	6	6	1	1
function(OObject)	🟥	0%		n/a	1	1	1	1	1	1
name_array()	🟥	0%		n/a	1	1	1	1	1	1
static {...}	🟩	100%		n/a	0	1	0	2	0	1
Log()	🟩	100%		n/a	0	1	0	4	0	1
function(double)	🟩	100%		n/a	0	1	0	1	0	1
Total	28 of 63	56%	0 of 0	n/a	3	6	8	15	3	6

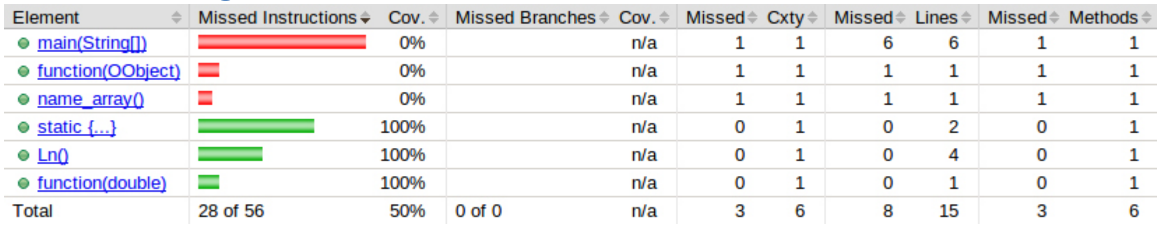
A.3.4 Addition



Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
main(String[])	🟥	0%		n/a	1	1	6	6	1	1
function(OObject, OObject)	🟥	0%		n/a	1	1	1	1	1	1
function(double)	🟥	0%		n/a	1	1	1	1	1	1
function(OObject)	🟥	0%		n/a	1	1	1	1	1	1
name_array()	🟥	0%		n/a	1	1	1	1	1	1
Add()	🟩	100%		n/a	0	1	0	4	0	1
static {...}	🟩	100%		n/a	0	1	0	2	0	1
function(double, double)	🟩	100%		n/a	0	1	0	1	0	1
Total	33 of 54	39%	0 of 0	n/a	5	8	10	17	5	8

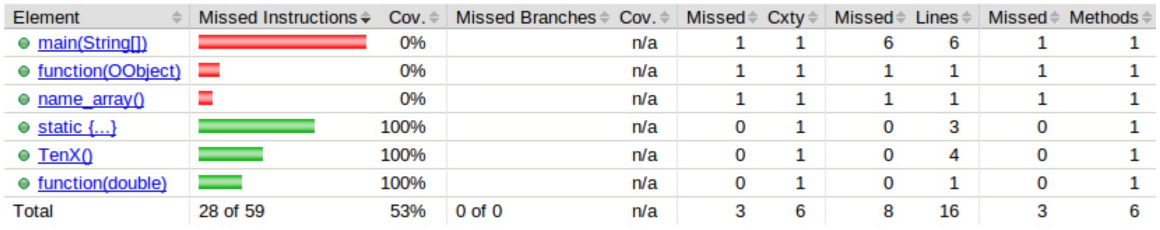
A.4 Will’s Initial Code Coverage Results

A.4.1 Natural Logarithmic



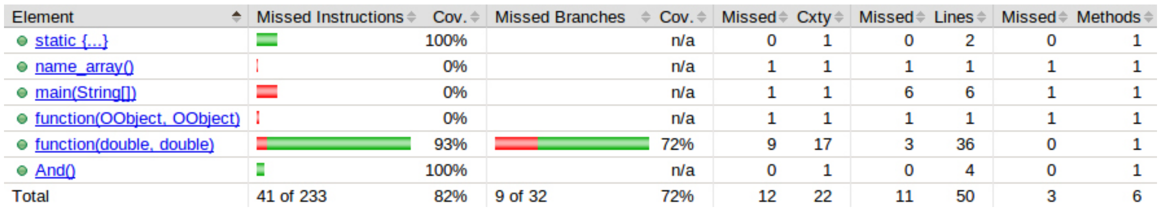
Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
main(String[])	🟥	0%		n/a	1	1	6	6	1	1
function(OObject)	🟥	0%		n/a	1	1	1	1	1	1
name_array()	🟥	0%		n/a	1	1	1	1	1	1
static {...}	🟩	100%		n/a	0	1	0	2	0	1
Ln()	🟩	100%		n/a	0	1	0	4	0	1
function(double)	🟩	100%		n/a	0	1	0	1	0	1
Total	28 of 56	50%	0 of 0	n/a	3	6	8	15	3	6

A.4.2 Inverse Logarithmic



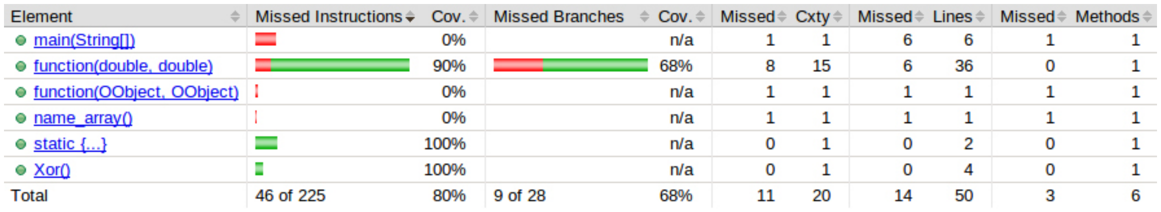
Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
main(String[])	🟥	0%		n/a	1	1	6	6	1	1
function(OObject)	🟥	0%		n/a	1	1	1	1	1	1
name_array()	🟥	0%		n/a	1	1	1	1	1	1
static {...}	🟩	100%		n/a	0	1	0	3	0	1
TenX()	🟩	100%		n/a	0	1	0	4	0	1
function(double)	🟩	100%		n/a	0	1	0	1	0	1
Total	28 of 59	53%	0 of 0	n/a	3	6	8	16	3	6

A.4.3 AND



Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
static {...}	🟩	100%		n/a	0	1	0	2	0	1
name_array()	🟥	0%		n/a	1	1	1	1	1	1
main(String[])	🟥	0%		n/a	1	1	6	6	1	1
function(OObject, OObject)	🟥	0%		n/a	1	1	1	1	1	1
function(double, double)	🟥🟩	93%	🟥🟩	72%	9	17	3	36	0	1
And()	🟩	100%		n/a	0	1	0	4	0	1
Total	41 of 233	82%	9 of 32	72%	12	22	11	50	3	6

A.4.4 XOR

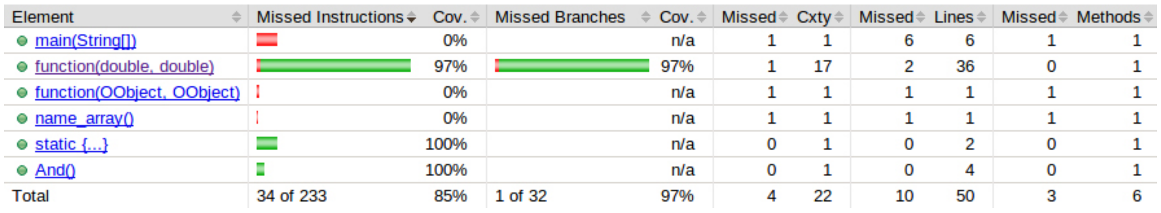


Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
main(String[])	🟥	0%		n/a	1	1	6	6	1	1
function(double, double)	🟥🟩	90%	🟥🟩	68%	8	15	6	36	0	1
function(OObject, OObject)	🟥	0%		n/a	1	1	1	1	1	1
name_array()	🟥	0%		n/a	1	1	1	1	1	1
static {...}	🟩	100%		n/a	0	1	0	2	0	1
Xor()	🟩	100%		n/a	0	1	0	4	0	1
Total	46 of 225	80%	9 of 28	68%	11	20	14	50	3	6

Appendix B. – Post-Improved Code Coverage Results

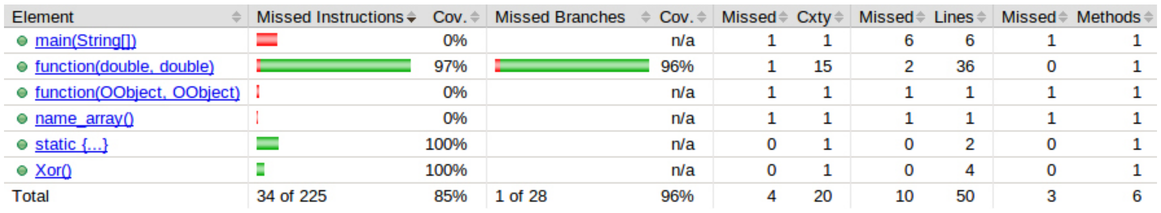
B.1 Will’s Improved Code Coverage Results

B.1.1 AND



Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
main(String[])		0%		n/a	1	1	6	6	1	1
function(double, double)		97%		97%	1	17	2	36	0	1
function(OObject, OObject)		0%		n/a	1	1	1	1	1	1
name_array()		0%		n/a	1	1	1	1	1	1
static {...}		100%		n/a	0	1	0	2	0	1
And()		100%		n/a	0	1	0	4	0	1
Total	34 of 233	85%	1 of 32	97%	4	22	10	50	3	6

B.1.2 XOR



Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
main(String[])		0%		n/a	1	1	6	6	1	1
function(double, double)		97%		96%	1	15	2	36	0	1
function(OObject, OObject)		0%		n/a	1	1	1	1	1	1
name_array()		0%		n/a	1	1	1	1	1	1
static {...}		100%		n/a	0	1	0	2	0	1
Xor()		100%		n/a	0	1	0	4	0	1
Total	34 of 225	85%	1 of 28	96%	4	20	10	50	3	6