**Xiaobin Chen**

37746611

**Part 1, Step 1: Mutation analysis**

1. Original:

|  |  |
| --- | --- |
| Live Mutants # | 16 |
| Killed Mutants # | 148 |
| Total Mutants # | 164 |
| Mutant Score | 90.0% |

1. Added tests
2. testTwoSegmentsDiffMoreThan500SecondLarger(),

killed mutant is AOIU\_4.

1. testTwoSegmentsSumMoreThan500(),

killed mutant is AORB\_4

1. testTwoSegmentsDepartureMoreThan14DaysFromNowVeryLongTime(),

killed mutant is AORB\_43

1. testTwoSegmentsDepartureLessThan3DaysFromNowIsFreqFlierV1(),

killed mutants are AORB\_62, AORB\_63, AORB\_65

1. testInvalidPrice1V1(),

killed mutant is COR\_1

1. testInvalidPrice1Price2()

killed mutant is COR\_2

1. testInvalidPrice1AndDepartureTime()

killed mutant is COR\_4

1. testInvalidPrice1AndDuration()

killed mutant is COR\_6

1. testTwoSegmentsDiffEquals500()

killed mutant is ROR\_15

1. testOneSegmentDurationEqualsTo8()

killed mutant is ROR\_22

1. testTwoSegmentsDepartureEqualsTo14DaysFromNow()

killed mutant is ROR\_29

1. Mutants that could not be killed:
2. ROR\_1.

Because original program P first tests if price1 < 0 in

**if** **(**price1 **<** 0 **||** price2 **<** 0 **||** departureTime **<** System**.**currentTimeMillis**()** **||** duration **<** 0**)**

and then tests if price1 > 0 in

**if** **(**price1 **!=** 0 **&&** price2 **!=** 0**)**

Mutant program P’ tests if price1 < 0 in the same code block and then tests if price1 > 0 in

**if** **(**price1 **>** 0 **&&** price2 **!=** 0**) //this is the mutant code**

With other codes being the same, we can see that P and P’ are equivalent mutants, produce the same outputs for all inputs, thus this mutant could not be killed.

1. ROR\_8

ROR\_8 is similar to ROR\_1, the different is just that the affected variable is now price2 (variable price1 and price2 are equivalent). We can easily verify that the mutant program is equivalent to the original one for all inputs, using the same reason as in ROR\_1, so the mutant could not be killed.

1. ROR\_37

Because the original program P tests

**if** **(**price **==** Double**.**POSITIVE\_INFINITY**)**

which is equivalent to mutant program P’ that tests

**if** **(**price **>=** Double**.**POSITIVE\_INFINITY**) //because price could not be greater than Double.POSITIVE\_INFINITY**

So P and P’ are equivalent mutants, produce the same outputs for all inputs, thus this mutant could not be killed.

1. Final score:

|  |  |
| --- | --- |
| Live Mutants # | 3 |
| Killed Mutants # | 161 |
| Total Mutants # | 164 |
| Mutant Score | 98.0% |

**Part 1, Step 2: Mutation testing**

1. Final score:

|  |  |
| --- | --- |
| Live Mutants # | 4 |
| Killed Mutants # | 85 |
| Total Mutants # | 89 |
| Mutant Score | 95.0% |

1. Mutants that could not be killed:
2. AOIU\_5

Because in the original program P, the code is:

planes**[**i**].**location**.**x **+=** planes**[**i**].**velocity **\*** Math**.**cos**(**planes**[**i**].**bearing**/**360.0 **\*** 2 **\*** Math**.**PI**);**

and the mutant program P’ adds an ‘-’ sign in Math.cos(), like this:

planes**[**i**].**location**.**x **+=** planes**[**i**].**velocity **\*** Math**.**cos**(-**planes**[**i**].**bearing**/**360.0 **\*** 2 **\*** Math**.**PI**);**

As we know cos(-x) == cos(x), so they are actually equivalent mutants, produce the same result for all inputs test cases.

1. AOIU\_6

The mutant program P’ adds an ‘-’ sign in Math.cos(), like this:

planes**[**i**].**location**.**x **+=** planes**[**i**].**velocity **\*** Math**.**cos**(**planes**[**i**].**bearing**/**360.0 **\*** 2 **\*** **-**Math**.**PI**);**

From the analysis above for AOIU\_5, we know P and P’ are actually equivalent mutants, produce the same result for all inputs test cases. Thus it could not be killed.

1. COI\_9

The mutant program P’ insert a ‘!’ in forLoop condition, like this:

**for** **(**int i **=** 0**;** **!(**i **<** planes**.**length**);** i**++)**

As a result, the entire for loop is skipped for all planes.length > 0, thus the initial check on safeDistance between all planes is skipped.

However, in the following code block, when checking the distance after moving a step, the program also checks the distance between two original positions of planes in this line of code:

**if** **(**point1**.**distance**(**other1**)** **<** safeDistance**)** **return** **false;**

So actually the check on initial positions of planes is covered by this line of code. So the change of code in P’ do not affect the outputs, so original program P and P’ are equivalent mutants.

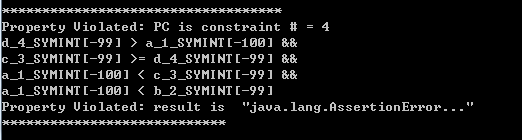
1. AORS\_4

This mutant should be killed by some test cases, but as reported in Piazza, it leads to unexpected behavior, so this mutant is ignored.

**Part 2: Model Checking**

**Step1**

1. JPF output:



Property violated is that the result of min() is not the minimum value of inputs.

Violated assertion:

**assert** **(**r **<=** a **&&** r **<=** b **&&** r **<=** c **&&** r **<=** d**);**

From the output, we can see that some values of *abcd* that causes violation of assertion can be that a(-100), b(-99), c(-99), d(-99).

First three lines from bottom upwards specify the path condition:

*a < b && a < c && d>=c*

we first have *a < b*, then *a < c*, then !(d<c). Then we have the violation that *d(-99)* is not less or equal to *a(-100)* as expected on this path.

So as we follow the path described above, we can find that bug. The bug is that when *a < b* and *a < c* and *c >= d*, the original program does not compare *a* and *d.* So add a comparison between *a* and *d* to find the minimum.

**if** **(**a **<** b**)** **{**

**if** **(**a **<** c**)** **{**

**if** **(**c **<** d**)** r **=** a**;**

**else** **{**

**if** **(**a **<** d**)** r **=** a**; //comparison between a and d**

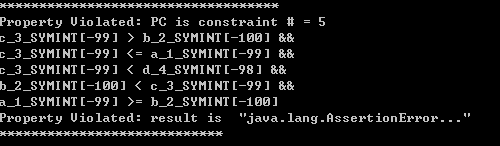
**else** r **=** d**;**

**}**

**}  
 …**

**}**

1. After fixing this bug, next bug appears:



Property violated is that the result of min() is not the minimum value of inputs.

Violated assertion:

**assert** **(**r **<=** a **&&** r **<=** b **&&** r **<=** c **&&** r **<=** d**);**

We can find the path condition is *a >= b*, then *b < c* then *c < d*:

*a >= b && b < c && c < d*

*c(-99)* satisfies *c <= a(-99),* but causes the violation because of that *c(-99)* is not less or equal to *b(-100)* as expected on this path.

The bug is that when abcd satisfy *a >= b*, then *b < c* then *c < d*, the minimum should be *b* not *c.* so change from

**else** **if** **(**b **<** c**)** **{**

**if** **(**c **<** d**)** r **=** c**;**

…

to

**else** **if** **(**b **<** c**)** **{**

**if** **(**c **<** d**)** r **=** b**;**

…

**Step2**

Assertion added:

// all elements in return array are positive

**for** (**int** i = 0; i < R.length; i++){

**assert** (R[i] > 0);

}

// all elements in new array are in increasing order.

**for** (**int** i = 1; i <R.length; i++){

**assert** (R[i] >= R[i - 1]);

}

// all positive elements are in the return array R[].

**int**[] original = *copyArray*(A);

**int** index = 0;

**for** (**int** i = 0; i < R.length; i++){

**for** (index = 0; index < original.length; index++){

**if** (original[index] == R[i]){

original[index] = -1; //find corresponding num in A[] and remove it

**break**;

}

}

**assert**(index != original.length); //every positive num in R[] has corresponding positive num in A[]

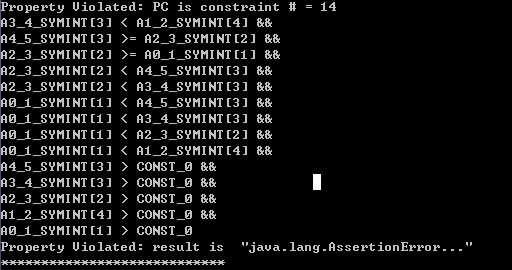
}

**for** (**int** i = 0; i < original.length; i++){

**assert** (original[i] <= 0); //not positive num left in A[]

}

JPF output:



Property violated is that the result of sort() is not in increasing order.

Violated assertion:

**for** **(**int i **=** 1**;** i **<**R**.**length**;** i**++){**

**assert** **(**R**[**i**]** **>=** R**[**i **-** 1**]);**

**}**

The value that caused violation is that A[] contains positive numbers {1, 4, 2, 3, 3}. After sorting, R[] is not in increasing order.

The bug is that in code:

**for** **(**int i **=** 0**;** i **<** R**.**length**;** i**++)** **{**

**for** **(**int j **=** i**;** j **<** R**.**length**-**1**;** j**++)** **{**

**if** **(**R**[**j**]** **<** R**[**j**+**1**])** **{**

…

change it to

**for** **(**int i **=** 0**;** i **<** R**.**length**;** i**++)** **{**

**for** **(**int j **=** 0**;** j **<** R**.**length**-**1**-**i**;** j**++)** **{**

**if** **(**R**[**j**]** **>** R**[**j**+**1**])** **{**

…

After the changes, for every i, the program computes the largest value in range [0, R.length -1- i], and put it R[R.length -1 -i], so that the value in R[R.length-1-i] >= all values in range[0, R.length-1-i] in R[], which ensures the increasing order when i == R.length-1.

**Step3**