System Validation Assignment 1

Karan Rawat - S1748211 M-EMSYS
Anirudh Gottimukkala - S1632469 M-EMSYS

2 JML Annotations

2.1

All sensor values have to stay within the prescribed range.

```
class SensorValue {

    int value;
    final int failSafe;
    final int minValue;
    final int maxValue;

    // INVARIANT(S)
    //@ invariant minValue < maxValue;
    //@ invariant value >= minValue && value <= maxValue;
    //@ invariant failSafe >= minValue && failSafe <= maxValue;
    //@ assignable value;
```

In the class 'SensorValue', the read-in sensor value is stored in value. The read-in sensor value should stay within the range between minValue and maxVlaue. In addition, the failsafe value should be also staying within the range between minValue and maxValue.

```
class LookupScale {

    /** Stores the scale (so called) break points.
     * Scales are required to be strictly monotone,
     * with raising values. */
    int[] values;
    int min_size = 2;

    // INVARIANT(S)

    // invariant values.length >= min_size;
    // invariant (\forall int i; i > 0 && i < values.length; values[i-1] < values[i]);
```

In the class 'LookupScale', the reference field 'values' refers to an integer array which stores the break points in the scale. There are at least two break points in the scale. One is storing the minimum value and the other is storing the maximum value. The length of this array should be greater than or equal to 2.

```
class LookupTable1d {

    /** The only (one dimension, x) scale for this lookup table. */
    //@ invariant scaleX != null;
    LookupScale scaleX;

    /** The lookup values of this table. Contrary to the scales
     *  there are no sortedness requirements of any kind.
     */
    int[] lookupValues;

    // INVARIANT
    //@ invariant scaleX.values.length == lookupValues.length;
```

In the class ‘LookupTable1d’, the reference field ‘scaleX’ refers to a LookupScale object, in which a strictly monotone scale array is stored. The field ‘lookupValues’ refers to a lookup table array whose length is also greater than or equal to 2 and whose length is equal to the length of lookup scale array referred to by ‘scaleX’. So, every break point including minimum value point and maximum value point in the lookup scale array referred to by scaleX is corresponding to each lookup value element in the lookup table array referred to by ‘lookupValues’.

```java
class ScaleIndex {

    /** Integral part. */
    int intPart;

    /** Fractional part. */
    int fracPart;

    /** The size of the corresponding scale this index refers to. */
    int size;

    // INVARIANT(S)
    //@ invariant intPart >= 0 && intPart <size;
    //@ invariant fracPart >= 0 && fracPart < 100;
```

In the class ‘ScaleIndex’, the integral part is storing scale index value which the read-in sensor value corresponds to. The size is equal to both the length of lookup scale array and the length of lookup table array. Scale index has to vary from 0 to the maximum index value. The fractional part is storing a ratio in which the read-in sensor value resides between two associated break points in the scale array. So, the fractional part which indicates a percent ratio should range from 0 to 99.

```java
//@ requires sv != null;
//@ requires sv.minValue <= sv.value && sv.value <= sv.maxValue;
//@ pure;
int getValue(SensorValue sv) {
    ScaleIndex si = scaleX.lookupValue(sv);
    int i = si.getIntPart();
    int f = si.getFracPart();
    int v = lookupValues[i];
    if(i<lookupValues.length-1) {
        int vn = lookupValues[i+1];
        v = v + (vn-v) * f / 100;
    }
    // ASSERTION(S)
    // (note, what you want to check here would normally
    //  be part of the postcondition, but would produce a very
    //  elaborate specification).
    if(i == lookupValues.length) {
        //@ assert v == lookupValues[i];
    }else {
        //@ assert v == lookupValues[i] + (f * (lookupValues[i+1] - lookupValues[i])) / 100;
    }


    return v;
}
```

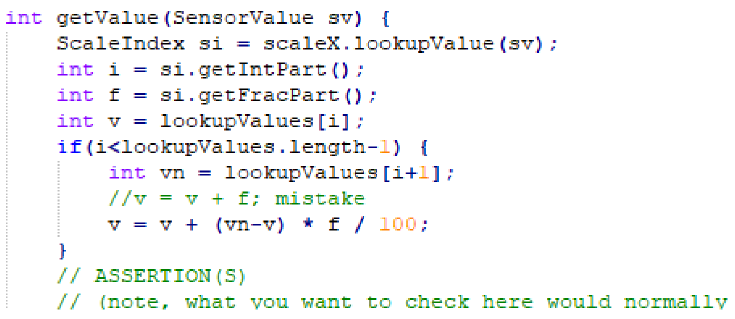
The method ‘getValue’ in the class ‘LookupTable1d’ accepts a SensorValue object and finds its corresponding scale index value in the scale array. Using this found scale index ‘i’, the method ‘getValue’ compute a lookup value ‘v’, which fits in between two associated values in lookup table, that is, lookupValues[i] and lookupValues[i+1].

2.2

After the JML type checking is performed, no error is found in the specifications.

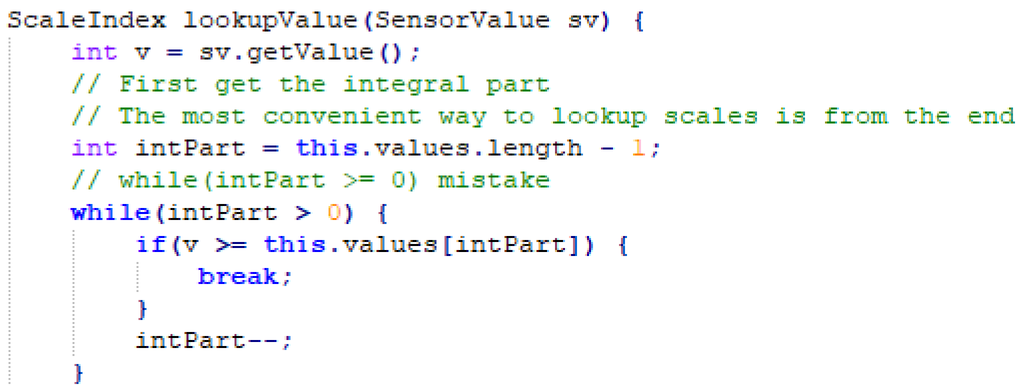
2.3

We can find two serious mistakes in the source code.



```
int getValue(SensorValue sv) {
    ScaleIndex si = scaleX.lookupValue(sv);
    int i = si.getIntPart();
    int f = si.getFracPart();
    int v = lookupValues[i];
    if(i<lookupValues.length-1) {
        int vn = lookupValues[i+1];
        //v = v + f; mistake
        v = v + (vn-v) * f / 100;
    }
    // ASSERTION(S)
    // (note, what you want to check here would normally
```

One mistake is found in the method ‘getValue’ in the class ‘Lookuptable1d’. After running the ‘IgnitionTest’ class and observing the original ignition lookup value output, we have found that the set of output ignition values is irregular. When comparing the ignition lookup table array in the class ‘IgnitionModule’ with output ignition value, we have detected that some of the output ignition values are far exceeding the maximum value in the lookup table. Since the elements in lookup table are decreasing first and then increasing, if the revolution speed is varying from low to high, its corresponding ignition value should be decreasing first and increasing next and its final value should stay within the maximum element value. The mistake is located in the method ‘getValue’ in the class ‘Lookuptable1d’. In the original source code, variable ‘vn’ is unused and it is an obvious error for f (a percent ratio) to be directly added with a corresponding point value ‘v’ from the lookup table. We think we have examined all annotations because all annotations in the ‘assert’ statements meet the specifications we want. The returned lookup value ‘v’ falls in between two adjacent elements in lookup table array.

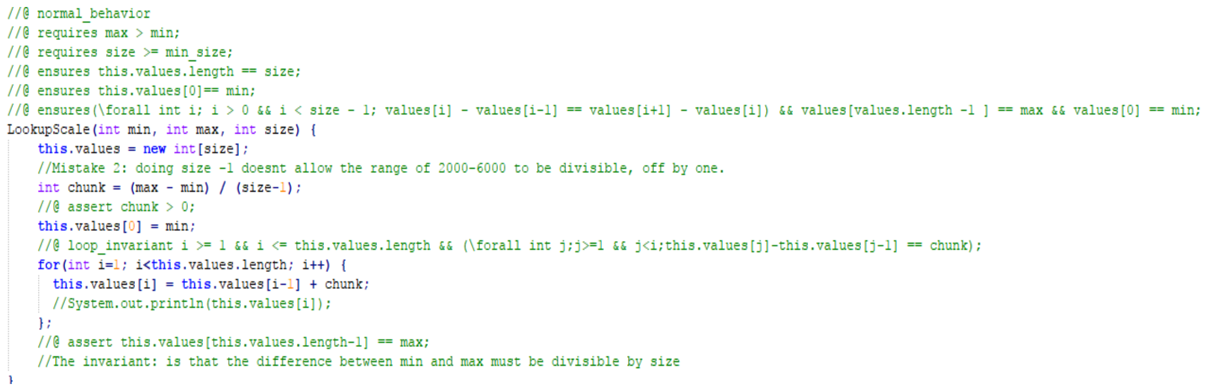


```
ScaleIndex lookupValue(SensorValue sv) {
    int v = sv.getValue();
    // First get the integral part
    // The most convenient way to lookup scales is from the end
    int intPart = this.values.length - 1;
    // while(intPart >= 0) mistake
    while(intPart > 0) {
        if(v >= this.values[intPart]) {
            break;
        }
        intPart--;
    }
```

Another mistake is found in the method ‘lookupValue’ in the class ‘LookupScale’. If the ‘intPart’ is decreased to 0 and the sensor value ‘v’ is still less than ‘values[0]’, then the ‘intPart’ is decremented to -1 and the while loop is stopped. Because the integral part is used as indexing in searching for a lookup value element in lookup table, the ‘intPart’ must be non-negative. Otherwise an out of bound exception may occur. We fixed the mistake by changing the condition in while loop from ‘intPart >= 0’ to ‘intPart > 0’ to avoid a possible case where the ‘intPart’ is decremented to ‘-1’. The annotations we added in the code ensure that the ‘intPart’ is always in between 0 and the length of lookup scale array – 1.

3 Static Checking

3.1

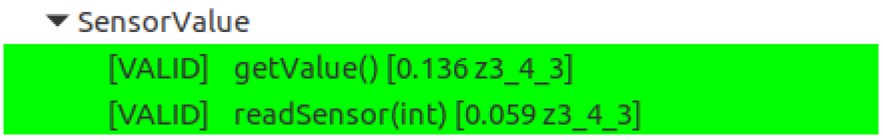


```
//@ normal_behavior
//@ requires max > min;
//@ requires size >= min_size;
//@ ensures this.values.length == size;
//@ ensures this.values[0]== min;
//@ ensures(\forall int i; i > 0 && i < size - 1; values[i] - values[i-1] == values[i+1] - values[i]) && values[values.length -1 ] == max && values[0] == min;
LookupScale(int min, int max, int size) {
    this.values = new int[size];
    //Mistake 2: doing size -1 doesnt allow the range of 2000-6000 to be divisible, off by one.
    int chunk = (max - min) / (size-1);
    //@ assert chunk > 0;
    this.values[0] = min;
    //@ loop_invariant i >= 1 && i <= this.values.length && (\forall int j;j>=1 && j<i;this.values[j]-this.values[j-1] == chunk);
    for(int i=1; i<this.values.length; i++) {
      this.values[i] = this.values[i-1] + chunk;
      //System.out.println(this.values[i]);
    };
    //@ assert this.values[this.values.length-1] == max;
    //The invariant: is that the difference between min and max must be divisible by size
}
```

3.2

The constructor ‘LookupScale’ constructs a lookup scale table which has size break points, including the minimal value and the maximal value. Since the lookup scale table has at least two elements for the minimal value and the maximal value, the size has to be greater than or equal to 2. The minimal value must be less than the maximal value so that the ‘chunk’ is greater than zero. The newly constructed lookup scale table should be monotone in ascending order. Its first element should be equal to the minimal value while its last element may be less than or equal to the maximal value. Since that chunk = (max - min) / (size -1) may result in zero for ‘chunk’ value when (max - min) < (size -1), we can assume ‘chunk’ to be a positive value we wish. The calculation that chunk = (max - min) / (size -1) loses some precision in integer assignment, the ‘chunk’ is probably smaller than that we require, leading to the fact that the last element in the lookup scale array is less than the maximal value.

3.3

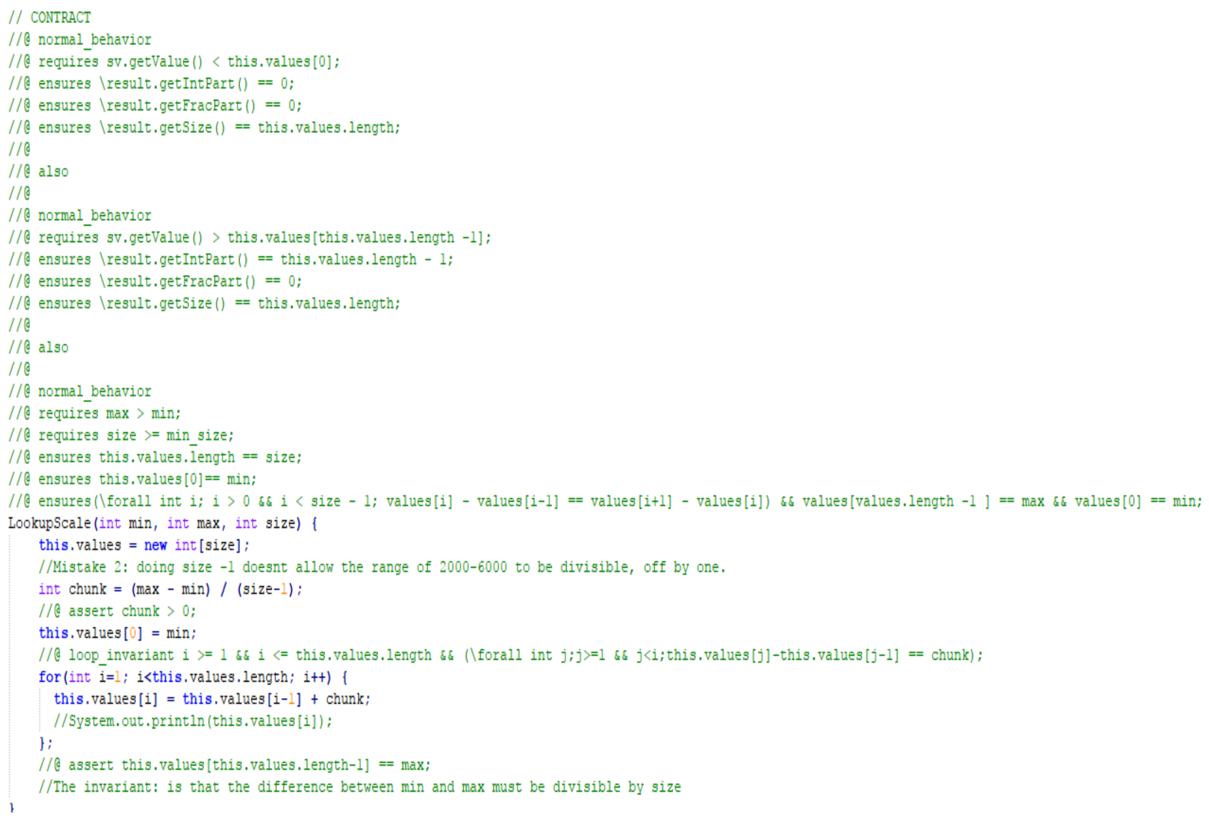


```
▼ SensorValue
    [VALID]   getValue() [0.136 z3_4_3]
    [VALID]   readSensor(int) [0.059 z3_4_3]
```

The ESC verification for the class ‘SensorValue’ is valid. There is no need to further fix the specifications.

4 Test Generation

There are three specification cases:



```
// CONTRACT
//@ normal_behavior
//@ requires sv.getValue() < this.values[0];
//@ ensures \result.getIntPart() == 0;
//@ ensures \result.getFracPart() == 0;
//@ ensures \result.getSize() == this.values.length;
//@
//@ also
//@
//@ normal_behavior
//@ requires sv.getValue() > this.values[this.values.length -1];
//@ ensures \result.getIntPart() == this.values.length - 1;
//@ ensures \result.getFracPart() == 0;
//@ ensures \result.getSize() == this.values.length;
//@
//@ also
//@
//@ normal_behavior
//@ requires max > min;
//@ requires size >= min_size;
//@ ensures this.values.length == size;
//@ ensures this.values[0]== min;
//@ ensures(\forall int i; i > 0 && i < size - 1; values[i] - values[i-1] == values[i+1] - values[i]) && values[values.length -1 ] == max && values[0] == min;
LookupScale(int min, int max, int size) {
    this.values = new int[size];
    //Mistake 2: doing size -1 doesnt allow the range of 2000-6000 to be divisible, off by one.
    int chunk = (max - min) / (size-1);
    //@ assert chunk > 0;
    this.values[0] = min;
    //@ loop_invariant i >= 1 && i <= this.values.length && (\forall int j;j>=1 && j<i;this.values[j]-this.values[j-1] == chunk);
    for(int i=1; i<this.values.length; i++) {
      this.values[i] = this.values[i-1] + chunk;
      //System.out.println(this.values[i]);
    };
    //@ assert this.values[this.values.length-1] == max;
    //The invariant: is that the difference between min and max must be divisible by size
}
```

1. When the sensor value is greater than or equal to the last element in the lookup scale array, the returned scale index object has integral part equal to last index (values.length-1) and fractional part of 0.

2. When the sensor value is less than the first element in the lookup scale array, the returned scale index object has both integral part of 0 and fractional part of 0.

3. When the sensor value stays between the first element and the last element of lookup scale array, the returned scale index object has integral part between first index 0 and the last index (values.length-1) and fractional part between 0 to 99.

If the range of revolutions per minute is set between 0 to 8000 rpm, the input rpm (revolutions per minute) for testing should go beyond this scope, such as from -100 to 8100 rpm to cover all possible cases, especially the boundary case.

lookupValue

Tests passed/Failed/Skipped:	0/0/0
Started on:	Fri Oct 20 18:34:42 CEST 2017
Total time:	0 seconds (147 ms)
Included groups:	
Excluded groups:	

(Hover the method name to see the test class name)

PASSED TESTS			
Test method	**Exception**	**Time (seconds)**	**Instance**
test_racEnabled Test class: LookupScale_JML_Test		0	LookupScale_JML_Test@5cbc508c

SKIPPED TESTS			
Test method	**Exception**	**Time (seconds)**	**Instance**

test_lookupValue__SensorValue_sv__0
Test class: LookupScale_JML_Test
Parameters: null, null
	`org.jmlspecs.jmlunitng.testng.PreconditionSkipException: could not construct an object to test`
	`        at LookupScale_JML_Test.test_lookupValue__SensorValue_sv__0(LookupScale_JML_Test.java:112)`
	`... Removed 23 stack frames`
	
	Click to show all stack frames
		0	LookupScale_JML_Test@5cbc508c