Runtime Verification using JavaMOP

Venkatesh Vinayakarao, IIIT Delhi.

Monitor Oriented Programming (MOP)[1] is a framework for runtime verification using the concept of monitors. JavaMop[2] is a Java implementation of MOP. In this paper, we show how to use JavaMop to monitor the properties and usage of specific API in small Java programs.

Additional Key Words and Phrases: Dynamic Analysis, JavaMOP, MOP, ERE.

1. INTRODUCTION

As the number of reused software libraries grow, the vulnerability of software to unforeseen bugs also grows. API developers often make assumptions and the API users do not pay attention to them. For example, the Javadoc for HashSet[3] mentions the following:

> Note that the fail-fast behavior of an iterator cannot be guaranteed as it is, generally speaking, impossible to make any hard guarantees in the presence of unsynchronized concurrent modification. Fail-fast iterators throw ConcurrentModificationException on a best-effort basis. Therefore, it would be wrong to write a program that depended on this exception for its correctness: the fail-fast behavior of iterators should be used only to detect bugs.

Such design and implementation are very common. This demands to the need for runtime monitoring. In other words, we need tools, libraries or frameworks to spot such error states, and safely continue with the program, or terminate the program. MOP framework allows us to precisely do the same. In this paper, we take the example of HashSets and Iterator and demonstrate how to use a Java implementation of MOP (JavaMOP) to monitor and report interesting events in a program during runtime.

2. HASHSET AND ITS PROPERTIES

2.1. HashSet and Iterator

In Java, HashSet implements the Set interface and is one of the well known Collections. Figure 1 shows the class hierarchy of HashSet as mentioned in its Javadoc page. Listing 1 shows a simple Java method which adds 100 random numbers to a set. It uses Iterator to navigate through the HashSet and sum up the result. This sum is finally printed.

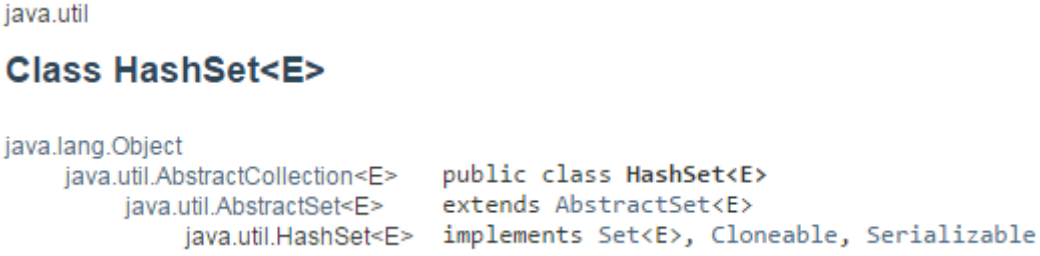

Fig. 1. Class Hierarchy of HashSet. Note that HashSet belongs to Collections and implements Set interface.

[3]http://docs.oracle.com/javase/6/docs/api/java/util/HashSet.html

```
private static void safe_HS() {
  HashSet set = new HashSet();
  for (int i=0; i<100; i++) {
    int rnd = (int) (Math.random() * 10000);
    set.add(new Integer(rnd));
  }

  Iterator iter = set.iterator();
  int sum = 0;
  while (iter.hasNext()) {
    sum += (Integer)iter.next();
  }
  System.out.println("sum: " + sum);
}
```

Listing 1. A sample usage of HashSet. Hundred random numbers are added to a HashSet and their sum is printed. Note that we use an iterator to navigate through the collection.

Iterator is more than just a handy utility for collections. Java has several collections and each collection is unique in its purpose, design and implementation. To iterate over all these collections, we need some well defined semantics. For instance, we can only navigate forward with an Iterator. Any object in java can implement Iterable and thus allow its user to use Iterator to navigate it. This makes the code elegant. Iterator is also fail-safe. We will discuss this property in detail in the next section.

2.2. FailSafe Property

Iterator is failsafe. So, while iterating, we can remove elements from the underlying collection through the Iterator API. However, if the underlying data structure (HashTable in the case of HashSet) is changed directly without the use of Iterator, ConcurrentModificaitonException is thrown. Listing 2 shows an example snippet to illustrate this behavior.

```
public static void badIterator(){
        HashSet<Integer> set = new HashSet<Integer>();
        for(int i = 0; i < 100; ++i){
            set.add(new Integer(i));
        }
        Iterator i = set.iterator();

        int sum = 0;
        for(int j = 0; j < 100; ++j){
          if (i.hasNext()) {
              sum += (Integer)i.next();
              set.add(new Integer(j));
            }
        }
        System.out.println(sum);
}
```

Listing 2. Modifying the set directly while iterating causes ConcurrentModificationException

2.3. Performance Guarantees

Javadoc for HashSet mentions the following:

> This class offers constant time performance for the basic operations (add, remove, contains and size), assuming the hash function disperses the elements properly among the buckets. Iterating over this set requires time proportional to the sum of the HashSet instance's size (the number of ele-

ments) plus the ”capacity” of the backing HashMap instance (the number of buckets).

Such performance guarantees are often advertised for the benefit of developers. They even form an important factor in the selection of suitable data structures.

3. RUNTIME VERIFICATION

Now that we have seen HashSet, Iterator and their properties, it is time to wonder if there is a way to monitor and verify them. Can we inspect the code at runtime and report if a HashSet is updated directly when an iterator is iterating? Can we check if the *next()* method is called without calling *hasNext()* method? Can we measure time taken by *add()* method over several runs and ensure that it is operating in constant time? To answer such questions, it will be nice to have a framework to weave instrumentation and monitoring code and inspect the code at runtime. AspectJ is a framework that allows such inclusion of cross cutting concerns without having to modify our original code. MOP and its implementation JavaMop allow us to easily create and manage monitors to reason about states and properties of APIs. We get started with the requisite installation and then discuss the mechanics of verification.

3.1. Framework/Tools Installation

Our objective is three fold:

— Observe and report if *next()* is called twice without calling *hasNext()* before every *next()* call.
— Observe and report set update while iterating.
— Report the performance of one of the update methods(*add()* is profiled in this paper).

Towards this, we use AspectJ and JavaMop. We assume JDK is already installed.

AspectJ. AspectJ can be installed from eclipse site[4]. We used aspectj-1.8.3.jar for this work. This is a executable jar. Just download and run it to complete the installation. Add the aspectjrt.jar to your CLASSPATH.

JavaMOP. Download the latest version of JavaMOP from UIUC website[5]. We used 3.0.0 for this work. Execute the downloaded jar and add bin folder from the installation path to your PATH environment variable and add javamoprt.jar to your CLASSPATH.

To verify the installation, you may type *javamop* at command line. You should see the usage options.

3.2. Test Case

Let's first start with a test case. This test case *Test.java* as shown in Appendix as Listing 7 has some methods discussed before and two safe methods. Figure 2 gives the structure of this class.

Note that *unsafe_HS()* calls *next()* without calling *hasNext()*. The method *badIterator()* is already discussed. It updates collection directly while iterating. We should be able to catch these behavior through runtime verification. We are also interested in profiling and understanding the behavior of HashSet's *add()* method. So, we have added *safe_HS()* and *safe_distributed_HS()* methods. Our intuition is that the performance of *add()* depends on collision of data stored in the underlying data structure. *safe_HS()* puts hundred random numbers into the HashSet while *safe_distributed_HS()* puts 1 to 100 in the HashSet. Thus, *safe_distributed_HS()* avoids collision.

[4]http://www.eclipse.org/aspectj/downloads.php
[5]http://fsl.cs.illinois.edu/index.php/All_JavaMOP_Versions

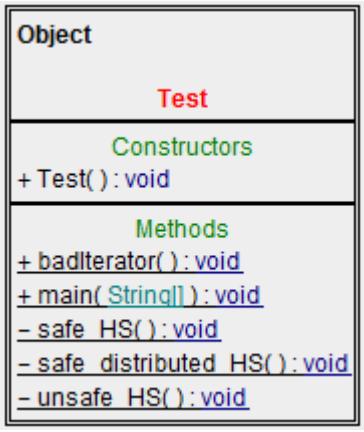

Fig. 2. Class diagram of our Test class. Full code is shown in Appendix.

3.3. Creating MOP Specification

Now we are all set to create our first MOP specification. We create three monitor specifications. Each one is a *.mop* file as shown in listings 3,4 and 5.

```
import java.io.*;
import java.util.*;

suffix HasNext(Iterator i) {
        event hasnext
              before(Iterator i) :
              call(* Iterator.hasNext())
              && target(i) {}
        event next before(Iterator i) :
              call(* Iterator.next())
              && target(i) {}

        ere : next next

        @match {
           System.out.println(
             "@:" + __LOC + " ! hasNext not called before next");
           __RESET;
        }
}
```

Listing 3. This is taken from JavaMOP's example set. It finds the occurrence of two calls to next methods in succession without the call to hasNext.

```
import java.io.*;
import java.util.*;

BadIterator(Collection c, Iterator i) {

        event create after(Collection c)
              returning(Iterator i) :
              call(Iterator Collection+.iterator()) && target(c) {}
        event updatesource after(Collection c) :
              (call(* Collection+.remove*(..))
              || call(* Collection+.add*(..)) ) && target(c) {}
        event next before(Iterator i) :
              call(* Iterator.next()) && target(i) {}
```

```
14
        ere : create next* updatesource updatesource* next
16
        @match {
18          System.out.println("improper iterator usage");
        }
20  }
```

Listing 4. This is taken from JavaMOP's examples. It finds the event where underlying collection is modified while iterating.

```
import java.util.ArrayList;
2  import java.util.HashSet;
   import java.util.Vector;
4
   CollectionAdd(HashSet c, Object o) {
6    long timeBefore = 0;
     long timeAfter = 0;
8    static ArrayList<Integer> durations = new ArrayList<Integer>();

10   event addBefore
       before(HashSet c, Object o) :
12     (call(* HashSet+.add(Object)))&& target(c) && args(o)
       {
14       timeBefore = System.nanoTime();
       }
16
     event addAfter
18     after(HashSet c, Object o) :
       (call(* HashSet+.add(Object))) && target(c) && args(o)
20     {
         timeAfter = System.nanoTime();
22       durations.add( (int) (timeAfter - timeBefore));
       }
24
           event sysexitAfter
26     after() returning:  execution(void main(..)) {


28
     }
30
          ere : sysexitAfter
32
     @match {
34       int avg = 0;
         int min = this.durations.get(0);
36       int max = this.durations.get(0);

38       for(int item: this.durations) {
           avg = avg + item;
40         if (item < min) min = item;
           if (item > max) max = item;
42       }
         avg = avg / (this.durations.size());
44       System.out.println("********HashSet.add(..) performance statistics********")
       ;
                 System.out.println("Mean = " + avg + " seconds.");
46       long sd = 0;
         int count = 0;
48       for(int item: this.durations) {
           sd = sd + (item - avg)*(item-avg);
50         System.out.println(++count + "," + item);
         }
```

```
52
      System.out.println("Standard Deviation = " + Math.sqrt(sd/this.durations.
    size()));
54    System.out.println("Min = " + min);
      System.out.println("Max = " + max);
56  }
}
```

Listing 5. Profiling the add method. We capture the time in nanoseconds before and after the method call and calculate the difference.

3.4. Weaving the code

The MOP sepects need to be compiled to create aspect files (*.aj). These aspects are then weaved into the code using aspectj compiler. These two tasks are done using the commands as shown in Listing 6. Note that we need javamoprt.jar and aspectjrt.jar to run our weaved class file. Test.Class now has the weaved code. Running this Test class gives the output as shown in Figure 3.

```
1 C:\tools\javamop3.0\vv>javamop *.mop
-Processing BadIterator.mop
3  BadIteratorMonitorAspect.aj is generated
-Processing CollectionAdd.mop
5  CollectionAddMonitorAspect.aj is generated
-Processing HasNext.mop
7  HasNextMonitorAspect.aj is generated

9 C:\tools\javamop3.0\vv>ajc -1.8 -d . Test.java *.aj

11 C:\tools\javamop3.0\vv>java -cp C:\aspectj1.8\lib\aspectjrt.jar;C:\tools\javamop3
    .0\lib\javamoprt.jar;. Test
```

Listing 6. This is how we weave the code. We create aspects from MOP files and then weave them into the code.

3.5. Results

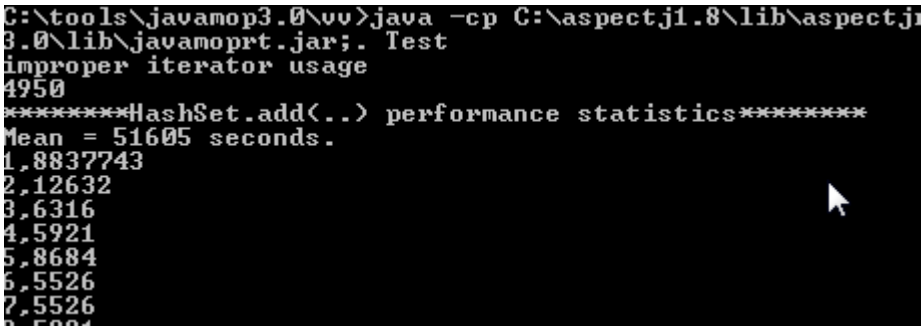

Fig. 3. Weaved code outputs the profiled data.

We collect the profiled data for add method and observe the behavior shown in Figure 4. We simulate two situations:a) We add 100 random numbers to HashSet in the first case and b) We add 1 to 100 in sequence. Note that in case (b), there should not be any collision. We observe if the *add()* method shows any difference in performance. As we expected, there are more stray dots in the second case. We attribute this to retrieval of multiple hash buckets. However, this requires a deeper analysis and study to understand the real reason for such a behavior. The mean and standard deviation

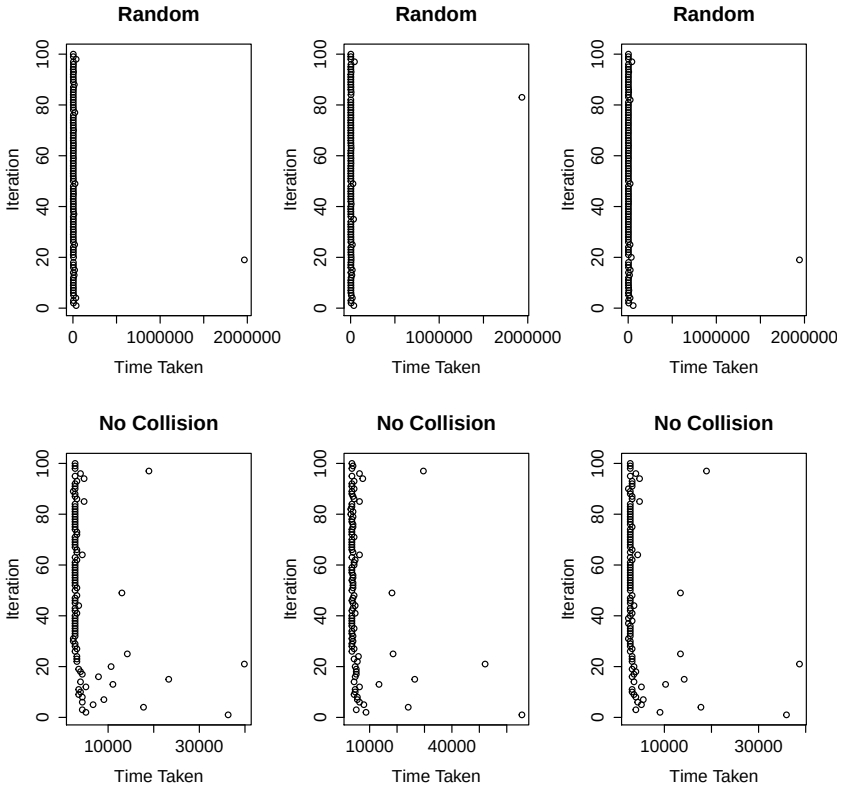

Fig. 4. On most occassions, HashMap's *add()* method displays constant time performance as guaranteed in the Javadoc site. There are few outliers.

for time taken to run the *add()* method indicates that the Random case takes more time. Note that the x-axis is not the same in both the cases (Random Vs. No-Collision).

4. CONCLUSION

In this work, we used JavaMOP and AspectJ to monitor the properties and performance of HashSet. JavaMOP gives a very elegant and clean way to monitor and profile code. We observed that the elegance comes from weaving the code and we did not have to write any code to create these monitors. JavaMOP did all the necessary heavy lifting. All we had to do was to write the mop specificiation. However, as a downside, we felt the learning curve to understand various aspects of MOP syntax was too huge. The MOP syntax requires deep understanding of different logic structures available and the BNF for specification. Moreover, when things don't work the way they should, it was very hard to debug. We need more tools to make this a practically usable solution for dynamic analysis.

Online Appendix to: Runtime Verification using JavaMOP

Venkatesh Vinayakarao, IIIT Delhi.

A. TEST CASE - JAVA CODE

Test case code with methods that use HashSet. The methods in this test case demonstrate safe and unsafe operations on collection and iterator.

```
import java.util.*;
/**
* Demonstrate the use of JavaMop library. Some examples from javamop has been used
     here.
* @author: Venkateshv.
*/
public class Test {

    public static void main(String[] args) {
      badIterator();
        //unsafe_HS();
        //safe_HS();
        //safe_distributed_HS();
    }

     public static void badIterator(){
        HashSet<Integer> set = new HashSet<Integer>();
        for(int i = 0; i < 100; ++i){
            set.add(new Integer(i));
        }
        Iterator i = set.iterator();

        int sum = 0;
        for(int j = 0; j < 100; ++j){
          if (i.hasNext()) {
              sum += (Integer)i.next();
              set.add(new Integer(j));
            }
        }
        System.out.println(sum);
    }

  private static void safe_HS() {
    HashSet set = new HashSet();
        for (int i=0; i<100; i++) {
          int rnd = (int) (Math.random() * 10000);
          set.add(new Integer(rnd));
      }

        Iterator iter = set.iterator();
        int sum = 0;
        while (iter.hasNext()) {
          sum += (Integer)iter.next();
        }
        System.out.println("sum: " + sum);

  }
```


DOI:http://dx.doi.org/10.1145/0000000.0000000

```
47
   private static void safe_distributed_HS() {
49   HashSet set = new HashSet();
       for (int i=0; i<100; i++) {
51       set.add(new Integer(i+1));
     }
53
       Iterator iter = set.iterator();
55     int sum = 0;
       while (iter.hasNext()) {
57       sum += (Integer)iter.next();
       }
59     System.out.println("sum: " + sum);

61 }

63 private static void unsafe_HS() {
   HashSet set = new HashSet();
65     for (int i=0; i<100; i++) {
         int rnd = (int) (Math.random() * 10000);
67       set.add(new Integer(rnd));
     }
69
       Iterator iter = set.iterator();
71     int sum = 0;
       for (int i=0; i<100; i++) {
73       sum += (Integer)iter.next();
       }
75     System.out.println("sum: " + sum);
   }
77 }
```

Listing 7. Test case with few methods that use HashSet.