## 1.OBL: Method may fail to clean up stream or resource (OBL_UNSATISFIED_OBLIGATION)

1. This is always due to a programming error: someone somewhere forgot to release. I would say it is easy to implement using ASM library. Putting cleanup code in a finally block is always a good practice. We can use ASM byte code to look for open streams, and putting cleanup code in a finally
2. The garbage collector will eventually close streams that have native resources. However, most operating systems limit the number of open files. If we fail to close our streams, the garbage collector may take a very long time to close them and may be that we run out of system file descriptors (file handles) and our code fails to open one more file. We also must make sure to release all references to the object. If we don't, we can end up with a memory leak as we execute the code multiple times. Therefore, I would say it is important to check for this bug.
3. I think testing wouldn't expose this bug unless we have many open streams (resources). The reason is because if we forget to close few number of streams (resources), the garbage collector will eventually close it. Therefore, the test won't be able to detect. The bug is a bad programming practice, and that we need to make it a good habit.
4. I don't think it is sound. Even SpotBugs doesn't output this warning, there might be still an open stream. This is one of the experimental and not fully vetted bug patterns in SpotBugs until may 2020 as the documentation shows [1].
5. I think it is complete. SpotBugs will give warning when a method may fail to clean up (close, dispose of) a stream, database object, or other resource requiring an explicit cleanup operation. SpotBugs is better than OpenJml
6. No, there is no assertion that could detect such performance related bug using OpenJML (ESC MODE). OpenJML is not better than SpotBugs.
7. I think OpenJml(RAC mode) can't be used to detect runtime occurance because garbage collector will take of the issue during runtime.

## 2. IS: Inconsistent synchronization (IS2_INCONSISTENT_SYNC)

1. This can be implemented using ASM library and I think it is moderately hard to implement. We need to scan the bytecode of the analyzed class to search for any synchronized block around the read and the write to check if it contains a mix of locked and unlocked accesses. We can track the findings by storing in hash table and look to see if used by another method. If so, then the bug pattern is detected and a warning message is generated.

2. This bug is important to be detected in SpotBugs and fixed because it may cause a lock issue. Multiple thread might reach same code between the read and the write. These could execute simultaneously and will cause the thread unknown state result.

3. Bug might not be exposed with good testing practice because two or more threads may never reach same code block at same time, and therefore, the test will never detect it. However, a good coding practice will help to avoid the issue at the first place.

4. No. This is a simple bug that can be detected by SpotBugs so we should not see false positive for this bug. SpotBugs needs to simply find synchronized statement and make sure it is used in the class consistently and one of the below points are detected.

- The class contains a mix of locked and unlocked accesses,
- The class is not annotated as javax.annotation.concurrent.NotThreadSafe,
- At least one locked access was performed by one of the class's own methods, and
- The number of unsynchronized field accesses (reads and writes) was no more than one third of all accesses, with writes being weighed twice as high as reads

5. Yes, the SpotBugs detection of the bug must be complete, because the pattern of bug is very easy to be detected.

6. I would say yes. ESC/Java provides support for checking that multi-threaded programs respect a locking discipline that prevents race conditions and simple deadlocks. The Developer need to declared locking order in a partial order. ESC/Java will then check that each thread does in fact acquire locks in the given order. Also, ESC/java is better than SpotBugs for such issues.

7. Yes, we can check concurrency structure of the class with threads and atomicity violations by calling methods in existing classes at concurrent time.

## 3a. SBSC: Method concatenates strings using + in a loop (SBSC_USE_STRINGBUFFER_CONCATENATION)

1. Easy: Using ASM Library, we can look for String concatenation in loops using “+” then check how big is the loop size. This could be analyzed to check if such implementation would have high impact on performance. In case of any performance degradation, a warning message can be sent out.

2. Strings are immutable. When string concatenation is performed using the "+" operator, the compiler translates this operation into constructing several intermediate strings. Building up a string one piece at a time in a loop requires a new string on every iteration, repeatedly copying longer and longer prefixes to fresh string objects. As a result, performance can be severely degraded. Therefore, this is bug can simply occur from bad coding practice and it needs be detected by SpotBugs as it will put huge performance impact in real world software development.

3. This is bad coding practice so it might be hard to be detected despite a good testing practice. However, we can run JUnitBenchmark performance testing in terms of time complexity to see if there is any performance degrade.

4. This bug pattern should be detected easily. SpotBugs should not incorrectly warn of the bug in cases where it is not really in existence. SpotBugs needs to simply look for String concatenation with in loops using “+”.

5. Yes, the SpotBugs will most likely detect this bug because the pattern of bug is easy to identify. So it is complete.

6. No, there is no assertion that could detect such performance related bug using OpenJML. OpenJML is not better than SpotBugs.

7. No, it couldn’t be used.

## 4. RANGE: Array index is out of bounds (RANGE_ARRAY_INDEX)

1. I would say it is difficult. ArrayIndexOutOfBound is an error that occurs in run time. The length of some java objects arrays like a List, hashtable, and arrays themselves is allocated dynamically during run time, which leaves them with unknown size during compiling time. Thus, writing a code that uses ASM library to find this bug pattern using the size of array would be challenging and less efficient. Moreover, building such code from scratch using ASM bytecode requires performing data flow analysis to each method of a visited class and then computing the control flow graph, which would be analysed to find this bug pattern. This is by itself considered another challenging especially if the size of the bytecode scanned is big.

2. The bug is very important to address and prevent it from happening as it would otherwise cause terminating the software abnormally. However, it is very less likely to see it occurring on real projects and therefore not worth it to check for in SpotBugs. Mostly we often use .size() or .length() to figure the array boundary of dynamic array(e.gLinkedList,HashMap,ArrayList...). So, if we happened wrote a code that access an

element beyond Array.size() (e.g. I <= array.size() ) , a simple unit test would soon throw an ArrayIndexOutOfBounds exception and the bug gets fixed after (e.g I < array.size()).

3. Yes, a good testing practices would expose this bug but up to some extent. An example of test would be a regression test that make sure to test every method of a class that access the array. Java would then throw ArrayIndexOutOfBounds exception if there was one. However, at test will not completely detect all varies of the bug. The test would probably miss detecting the ones that occur at runtime in multithreaded environment. Such environment is always non-deterministic, and the exception could or could not be triggered.

4. It is not sound. In general, SpotBugs could itself have some bugs that causing some incorrectly warn of the bug. Also, because of pattern of ArrayIndexOutOfBoundsException bug is vague and not clear, SpotBugs could produce false reports.

5. No, the SpotBugs detection of this bug is not complete. It would likely miss detecting ArrayIndexOutOfBounds exceptions that occur at runtime. For example, a poor coding in a multi-threaded environment might cause a thread to operate on an array's element that has already been removed by another thread.

6. I would say yes. ESC/Java uses a pre and postcondition “ensure and invariant” to detect any potential occurrence of this pattern in the code. If so, it throw exception. for example:

```
//@ invariant 0 <= i && i <= a.length;
 //  @ endure 0 <= i && i <= a.length;
```

However, ESC/Java is not sound and complete [2] and it can produce spurious error reports because of inaccurate modeling of the Java semantics or may miss some errors, as it also ignores all iterations of a loop beyond a fixed limit. Thus, it produces more false bug warning. SpotBugs is much better at suppressing false bug warning than ESC/Java.

7. I would say yes. By using runtime assertion checking, JML uses a postcondition “ensures” that checks if the index of the array fill within the range of 0 and its length. If not then throw exception would be occurred.

## Reference

1. https://readthedocs.org/projects/spotbugs/downloads/pdf/latest/
2. https://www.cs.ru.nl/E.Poll/papers/tfm2009.pdf