HOMEWORK ASSIGNMENT: CHAPTER 19

Question 19.1:
Give an example of how verifying access is protected with synchronized(l) would be harder than against using lock(l)/unlock(l).

!---19.2,19.3 answered instead of this question--!

Question: 19.2:
Give an example for which static program analysis will not be able to determine whether access at a particular program location is always protected by the same lock, including your explanation.

In java, as each lock is identified by a corresponding object, where execution path/context is determined by some aspect of a shared variable ('x')'s state, the lock secured or released on 'x' may be dependent upon paths in which different objects – of possibly different class types – could be able to determine paths in which differing 'lock' objects are secured or released on 'x'.

Take for instance, classes Cat, Dog represented as:

```
static Cat{
  boolean lock_determinant = true;

   inspectCat(){
     if(lock_determinant)
          unlock(this);
   }
}
```

```
Dog1{
  boolean speak=false;

  if(speak){
     Cat.lock_determinant=true;
     inspectCat();
  }
}
```

```
Dog2{
  boolean speak=true;

  if(speak){
     Cat.lock_determinant=false;
     inspectCat();
  }
}
```

As class Cat is a shared class (and by association its class variables are as well), the presence of any number of objects which may independently influence the state of Cat and the variables upon which securing/releasing of a lock (object) held on it is conditioned will present difficulty in statically determining whether access at a program location beyond which the 'inspectCat()' point of execution has been reached more difficult.

Question 19.3:
How might "revival" of suppressed error reports based on a false alarm outcome determined by symbolic testing, today, be made possible for future runs. Discuss pros/cons associated with your approach to this workaround.

Flags settable as by breakpoints set in a debugging mode (as achievable using Purify by annotating/ instrumenting code), could allow for revising values at key points of a program execution upon which suppressed error reports could be revived, influencing the paths of execution examined by the static analysis tool, after the point of revision is succeeded. Pros to such an approach include ease of use on the program debugging teams. Cons associated include the reliance upon programmers to accommodate such instrumentation by annotating their sourcecode during implementation and using specialized static analysis/debugging tools, which may impose other limitations due to tooling & process.