Program Monitoring

Lecture 7 : Context Free Grammars

Wednesday May 27, 2009

This lecture introduces context free grammars and how they are supported by the JavaMOP system. A property can be regarded as a language defined by a grammar. Traditional parsing techniques can be used to monitor such properties against an execution trace.

Reading

This week we will read the paper:

Efficient Monitoring of Parametric Context-Free Patterns,
Patrick ONeil Meredith, Dongyun Jin, Feng Chen and Grigore Rosu.

Assignment 3

To be submitted to `havelund@gmail.com` before Friday June 5, at 11:59 pm.

Four sub-assignments 3.1, 3.2, 3.3 and 3.4, are presented below. Try to solve assignments 3.1 and 3.4. They should not be be difficult in case you solved assignment 2. Assignments 3.2 and 3.3 are for extra credit. You do not need to solve them.

Assignment 3.1:

Express the two properties in assignment 2 (the URLConnection example) using future time LTL (FTLTL). Create a full JavaMOP spec with pointcuts and try to run it against the test program from assignment 2.

Assignment 3.2:

Express the two properties in assignment 2 (the URLConnection example) using context free grammars (CFG). Create a full JavaMOP spec with pointcuts and try to run it against the test program from assignment 2.

Assignment 3.3:

Consider the following class using Java's reentrant locks:

```
package cfg.locking;

import java.util.concurrent.locks.*;

public class Task extends Thread {
  static int x = 0;
  static Lock l1 = new ReentrantLock();
  static Lock l2 = new ReentrantLock();

  public void run() {
    l1.lock();
    l1.lock();
    System.out.println(x++);
    l1.unlock();
    l1.unlock();
  }

  public static final void main(String[] argv) {
    Task task1 = new Task();
    Task task2 = new Task();
    task1.start();
    task2.start();
  }
}
```

The following specification using CFG specifies that locks should be released within the same method that they are acquired, and that there should be as many releases of a lock as acquisitions.

```
package cfg.locking;

import java.util.*;
import java.util.concurrent.locks.*;

SafeLock(Lock+ l, Thread t) {

  event acq after(Lock+ l, Thread t) :
    call(* Lock+.lock()) && target(l) && thread(t) {}

  event rel after(Lock+ l, Thread t) :
    call(* Lock+.unlock()) && target(l) && thread(t) {}

  event begin before(Thread t) :
    execution(* *.*(..)) && thread(t) {}

  event end after(Thread t) :
    execution(* *.*(..)) && thread(t) {}

  cfg : S -> begin S end | acq S rel | S S | epsilon

  @fail {
    System.out.println("Error in line " + __LOC);
  }
}
```

The specification allows for the same lock to be acquired multiple times (as in the example) before released. Change the specification to require that a lock is not acquired in such a nested manner: once acquired, the lock has to be released again before it can be acquired again. For example, the above example program should be declared illegal since the lock `l1` is acquired twice before released. Note, this restriction goes against the purpose of reentrant locks, but could be regarded as a policy employed in a project.

Assignment 3.4:

Try to evaluate the different specifications you have written. Do you have a favorite notation (FSM, ERE, FTTL, or CFG)? Explain why in that case. How do these notations compare with the AspectJ solution? You are very welcome to critique any of the tools.