EnforceMOP:
A Runtime Property Enforcement System for Multithreaded Programs

Qingzhou Luo, Grigore Rosu
JavaMOP

- Runtime verification system
- Monitoring-oriented programming (MOP)
- Specify properties which should always hold in a Java program
- Properties defined separately from source code
- JavaMOP warns you when properties are broken
- Logic-independent architecture
- Monitors monitoring objects
EnforceMOP

- Instead of warning when a property is violated, EnforceMOP blocks thread before property is violated until thread can continue without violating property.
- If all threads are blocked by EnforceMOP, i.e. deadlock, user-specified code runs.
- Users can specify code to run when a thread is blocked.
Use cases

1. Enforce properties in a program to avoid concurrency bugs, as an alternative to manual synchronization

2. Enforce scheduling decisions in unit tests, to be able to reliably test different scheduling possibilities
Use cases

- Enforce properties in a program to avoid concurrency bugs
- Less error-prone than manual synchronization
- More modular: Separated from source code
- Possibly faster: Avoids over-synchronization
Example (1)

Concurrent Modification of ArrayList

```java
1  enforce SafeListIteration(Collection c, Iterator i) {
2      creation event create after(Collection c) returning(Iterator i) :
3          call(Iterator iterable()) && target(c) {}
4
5      event modify before(Collection c) :
6          {
7              call(+ Collection+.add(...)) ||
8              call(+ Collection+.clear(...)) ||
9              call(+ Collection+.offer(...)) ||
10             call(+ Collection+.pop(...)) ||
11             call(+ Collection+.push(...)) ||
12             call(+ Collection+.remove(...)) ||
13             call(+ Collection+.retain(...))
14       } && target(c) {}
15
16      event next before(Iterator i) :
17          call(+ Iterator.next(...)) && target(i) {}
18
19      event hasNext after(Iterator i) returning(boolean b) :
20          call(+ Iterator.hasNext()) && target(i) && condition(!!b) {}
21
22  fsm :
23      na [ create -> init
24          ]
25      init [ next -> unsafe
26                   hasNextfalse -> safe
27                     ]
28      unsafe [ next -> unsafe
29                    hasNextfalse -> safe
30                       ]
31      safe [ modify -> safe
32                     hasNextfalse -> safe
33                        next -> safe
34                       ]
35
36  @nonfail {}
37
38  @deadlock { System.out.println("Deadlock detected!"); } }
```
Use cases

- Enforce scheduling decisions in unit tests
- Faster and more reliable than alternatives
- More modular: same source code can be run with different properties to get different schedules
Example (2)

```java
@Test
public void testPutWithTake() throws InterruptedException {
    final SynchronousQueue q = new SynchronousQueue();
    Thread t = new Thread(new CheckedRunnable() {
        @Override
        public void realRun() throws InterruptedException {
            int added = 0;
            try {
                while (true) {
                    q.put(added);
                    ++added;
                }
            } catch (InterruptedException success) {
                assertMessage("PutWithTake", 1, added);
            }
        }
    });
    t.start();
    Thread.sleep(SHORT_DELAY_MS);
    assertEquals("PutWithTake", 0, q.take());
    Thread.sleep(SHORT_DELAY_MS);
    t.interrupt();
    t.join();
}
```

```java
enforce SynchronousQueueTest::testPutWithTake() {
    String putThread = "";
    event beforeInterrupt before() {
        call(* Thread+::interrupt()) && threadBlocked(putThread);
    }
    event beforeTake before() {
        call(* SynchronousQueue+::take()) && threadBlocked(putThread);
    }
    event beforePut before() {
        call(* SynchronousQueue+::put()) {
            if (putThread.equals("")) {
                putThread = Thread.currentThread().getName();
            }
        }
    }
    ere: beforePut+ beforeTake beforePut+ beforeInterrupt
    @Nonfail {} 
    @deadlock {System.out.println("Deadlock detected!");}
}
```
Logic plugins

- Properties can be expressed in different logic formalisms
- Different formalisms work well for different problems
- Currently supported by EnforceMOP: FSM, ERE, LTL, PTLTL, CFG, SRS
Implementation

- Specification file is compiled together with Java source file by EnforceMOP compiler to create Java bytecode.
- Before each event, the monitor is cloned and the event is executed. If a condition fails, the original monitor blocks.
- If a new event is generated on any thread, redo the above on all monitors
- Drawback: One step lookahead might not be enough for some logic formalisms
Evaluation

- Can be used to solve difficult synchronization bugs in a simple and straightforward fashion

- Can be used to increase performance by avoiding over-synchronization
Related work

- Most other runtime verification systems have hardwired specification languages.
- Other existing runtime verification systems *monitor*, rather than *enforce* properties.
- As a scheduling framework for testing, EnforceMOP is more powerful and usually faster than alternatives.
Conclusions

- Very powerful framework
- Somewhat complicated
- Might lead to new innovations in programming languages
Thank you for listening!