Correct Refactoring of Concurrent Java Code

M. Schäfer, J. Dolby, M. Sridharan,

E. Torlak and F. Tip

ECOOP 2010

Speaker: Zsolt István

Outline

- Motivation
- Contribution
- Theory
- Implementation
- Conclusion

Motivation

- Real-world projects need refactoring
- Serial code
  - Regression testing
- Parallel code
  - Hard to detect modified behavior
  - Use only behavior-preserving refactoring

Motivation II

- Behavior-preserving refactoring:

  all behaviors exhibited by the refactored program can also be exhibited by the original, and vice versa

- Execution model to reason upon:
  - Java Memory Model

Pull Up Members

```
class C1 implements TM {
  static class Super {
    static int x, y;
  }
  static class Sub extends Super {
    static synchronized void m() {
      ++x; ++y;
    }
    static synchronized void n() {
      if (x != y) { print("bug"); }
    }
  }
  public void m1() { Sub.m(); }
  public void m2() { Sub.n(); }
}
```

(a)

```
class C1 implements TM {
  static class Super {
    static int x, y;
    static synchronized void m() {
      ++x; ++y;
    }
  }
  static class Sub extends Super {
    static synchronized void n() {
      if (x != y) { print("bug"); }
    }
  }
  public void m1() { Super.m(); }
  public void m2() { Sub.n(); }
}
```

(b)

Pull Up Members

```java
class C1 implements TM {
  static class Super {
    static int x, y;
  }
  static class Sub extends Super {
    static synchronized void m() {
      ++x; ++y;
    }
    static synchronized void n() {
      if (x != y) { print("bug"); }
    }
  }
  public void m1() { Sub.m(); }
  public void m2() { Sub.n(); }
}
```

(a)

```java
class C1 implements TM {
  static class Super {
    static int x, y;
    static synchronized void m() {
      ++x; ++y;
    }
  }
  static class Sub extends Super {
    static synchronized void n() {
      if (x != y) { print("bug"); }
    }
  }
  public void m1() { Super.m(); }
  public void m2() { Sub.n(); }
}
```

Super.class

Sub.class

(b)

Moving `synchronized`

- Problem: implicit target changed

Moving `synchronized`

- Problem: implicit target changed

- Desugaring: make implicit lock target explicit

```
static synchronized void n() {…}

static void n() {synchronized(A.class) {…} }
```

- Resugaring: reverse operation

Extract Local

```
class C3 implements TM {
  int f;
  public void m1() {
    int g = 0;
    synchronized (this) {
     g =  f;
    }
    if (g % 2 != 0) { print("bug"); }
    synchronized (this) { g = f; }
  }
  public synchronized void m2() {
    ++f; ++f;
  }
}
```

(a)

```
class C3 implements TM {
  int f;
  public void m1() {
    int g = 0;
    int n = f;
    synchronized (this) {
      g =  n;
    }
    if (g % 2 != 0) { print("bug"); }
    synchronized (this) { g = n; }
  }
  public synchronized void m2() {
    ++f; ++f;
  }
}
```

(b)

Extract Local

```
class C3 implements TM {
  int f;
  public void m1() {
    int g = 0;
    synchronized (this) {
     g =  f;
    }
    if (g % 2 != 0) { print("bug"); }
    synchronized (this) { g = f; }
  }
  public synchronized void m2() {
    ++f; ++f;
  }
}
```

(a)

```
class C3 implements TM {
  int f;
  public void m1() {
    int g = 0;
    int n = f;
    synchronized (this) {
      g =  n;
    }
    if (g % 2 != 0) { print("bug"); }
    synchronized (this) { g = n; }
  }
  public synchronized void m2() {
    ++f; ++f;
  }
}
```

(b)

Escaping `synchronized`

- Problem: action not protected anymore by lock
- Alternate problem: unrelated action becomes protected by lock

Escaping `synchronized`

- Problem: action not protected anymore by lock
- Alternate problem: unrelated action becomes protected by lock

- Solution: Dependence edge preservation
  - Actions: ordinary or synch.
  - Ordinary actions can be reordered freely
  - Synch. actions are barriers to reordering

Java Memory Model

- Program = possibly infinite set of threads
- Thread = set of memory traces
- Memory trace = action & value

Java Memory Model

- Program = possibly infinite set of threads
- Thread = set of memory traces
- Memory trace = action & value

- Execution = pick one trace for each thread
- Program is free of data races if all executions are free from data races

During execution

- Program order
  - intra-thread ordering of actions
- Synchronization order
  - global total order on synch. actions

During execution

- Program order
  - intra-thread ordering of actions
- Synchronization order
  - global total order on synch. actions
- Action *a* happens before *b* if:

  (1) $a \leq_{po} b$

  (2) $a \leq_{po} rel \leq_{so} acq \leq_{hb} b$, where *rel* and *acq* act on the same lock
- Data race = accesses to some variable not related by happens before

Trace Preserving Refactoring

- Does not alter the set of memory traces of a program
- Theorem: After such a refactoring every possible behavior of the original program is a behavior of the refactored program and vice versa. This holds even in the presence of data races.

Trace Preserving Refactoring II

- JMM has no notion of methods
- Trace preserving if just reorganizes code:
  - Pull Up & Push Down Method
  - Move Method
  - Extract & Inline Method

- Not trace preserving if field accesses are reordered:
  - Extract & Inline Local

Restructuring Refactoring

- Partial function mapping actions from the original execution *E* to *E'*

- A restructuring transformation is said to respect synchronization dependencies if its mapping fulfills:
  1. If $a \leq_{so} b$, then also $f(a) \leq'_{so} f(b)$.
  2. If *acq* is an acquire action and $acq \leq_{po} b$, then also $f(acq) \leq'_{po} f(b)$.
  3. If *rel* is a release action and $a \leq_{po} rel$, then also $f(a) \leq'_{po} f(rel)$.

Restructuring Refactoring II

- Theorem: If there is a data race between two actions f(*a*) and f(*b*) in execution *E'*, then there is already a data race between *a* and *b* in *E*.

- Corollary: A restructuring transformation that does not introduce any new actions will map correctly synchronized programs to correctly synchronized programs
  - Newly introduced actions ??
  - Proof for Extract & Inline Local

Implementation

- Extend a refactoring engine
  - Intra-procedural approach
  - Use control flow analysis to calculate dependence on mutex enters/exits

- Desugaring:
  (1) Desugar
  (2) Refactor
  (3) Resugar if possible

Implementation II

- Dependence edge preservation:
  (1) Calculate dependencies
  (2) Refactor
  (3) Verify new dependencies
  (4) Accept or reject refactoring

- Can not extract/inline expressions containing calls to methods involving synchronization

Implementation III

- Method involves synchronization if:

1. is declared synchronized or contains a synchronized block
2. contains an access to a volatile field
3. calls a thread management method from the standard library
4. calls a method which involves synchronization

- Measurements show <30% such methods (DaCapo benchmark and Apache Ant)

Conclusion

- Contribution
  - Idea of concurrency-aware refactoring
  - Synchronized keyword desugaring
  - Dependence edge preservation technique
  - Proofs and detailed discussion

- Criticism
  - Complex refactorings not discussed (Variable To Field)
  - Nice read, but too much “marketing”
  - Cited mostly by same authors