Reasoning about Concurrency in (RT) Java

A description of past work on reasoning about concurrent Java and ongoing in the EU FP6 Artimis “CHARTER” project

Joe Kiniry
Outline

- Java vs. JavaCard vs. RT Java
- reasoning about concurrency in the past
- modern models of concurrency
- what do developers (do/understand)?
- recent work and next steps
Concurrency in Java
Early Java

• only two classes: Thread and ThreadGroup

• ThreadGroups are sets of threads, arranged hierarchically, with a pseudo-multicast façade

• synchronized and volatile keywords exist, but the latter is ignored

• hand-waving about scheduling: “Every thread has a priority. Threads with higher priority are executed in preference to threads with lower priority.”

• thread has several surprising methods: checkAccess, destroy, interrupt, setPriority, stop
Safety

• every object contains a monitor

• a thread attempts to lock an object’s monitor either via a call to a synchronized method or an explicit synchronized block

• recall that classes are represented by singleton objects of type \textit{Class}, thus synchronized static methods lock these singletons
Safety Problems

• no guarantees on lock ordering, fairness, complex semantics on lock release, etc.

• no support for identifying race conditions

• loose and very complex semantics on memory consistency (Manson, Pugh, and Adve in POPL’05)

• reordering during and after JIT permitted
Liveness

- threads communicate via calls to low-level methods like `sleep`, `wait`, and `notify/notifyAll`
- no support to identifying or avoiding deadlock or livelock
- no semantics for priorities and scheduling
Communication

• superposition via volatile shared variables with non-atomic updates for nearly all primitive types
• wait/notify patterns often misused
• manual encoding of barriers, callbacks, etc.
Teenager Java (1.2–1.4)

- Thread now has access to its ClassLoader and gets the **holdLock** method
- ThreadGroup now has **interrupt** method
- ThreadLocal class introduced
- Thread’s suspend, resume, and stop methods are all deprecated
- Doug Lea starts work on his Java concurrency library (EDU.oswego.cs.dl.util.concurrent)
Mature Java (1.5)

- *Thread* gets structured access to stack traces, ID, state, and exception handlers
- thread state is now explicit and exactly one of the following states:
  - `new`, `Runnable`, `blocked`, `waiting`, `timed_waiting`, `terminated`
- *ThreadLocal* gets a `remove` method
- Doug’s concurrency library standardized
The Java Concurrency API

- lifts the level of abstraction away from raw threads, monitors, and synchronized regions

- new constructs available include:
  - atomic wrapper classes for some primitive types and references, barriers, concurrent collections, conditions, copy-on-write collections, executors, futures, latches, (pairs of) locks, queues, semaphores, threadpools, etc.

- implementation is VM-invariant and somewhat lock-free
Java Variants

- JavaCard
  - no concurrency or floating point and a different memory model than normal Java (no allocation, permanent and transient memory)

- MIDP
  - normal, concurrent Java, but on small devices

- Real-time Java (RT Java)
  - soft & hard real-time with priorities with deterministic scheduling and mixed thread model
  - Thread’s API becomes well-defined again
  - triplet (scoped/immortal/heap) memory model
Reasoning about Concurrency
Early Efforts

• Model Checking
  • Bandera and Bogor from Corbett, Dwyer, Hatcliff, Laubach, Pasareanu, Robby, and Zheng (ICSE’00—CAV’05)
  • Java PathFinder from Havelund and Pressburger (SPIN’99—TACAS’07)

• Proof Systems
  • Ábrahám, de Boer, de Roever and Steffen (CONCUR’00—Fund. Info.’08)
Finding Key Abstractions

- Race Condition Checking from Flanagan and Freund (ESOP’99—PLDI’00)
- Atomicity from Flanagan, Freund, and Qadeer (PLDI’03—PLDI’08)
- Immutability (Ernst and many colleagues OOPSLA’05—ESEC-FSE’07; Haack, Poll, Schäfer, and Schubert, ESOP’07; et al.)
“Purported” Tools

• there is a terrible dearth of released, supported tools in this research area

• dozens of tools for reasoning about concurrency have been discussed in papers, but have not been released to the research community

• IMO, this is not kosher
“Real” Tools

• JProbe’s deadlock analysis and race condition detection
• lightweight static checkers like PMD and FindBugs do conservative race analysis and concurrency anti-pattern detection
• RCC does type-based race condition checking
• ESC/Java2 does deadlock and locking discipline analysis
Recent Efforts

• the Mobius approach

• ensure programs are properly synchronized through the use of ESC/Java2 and RCC, then reason about program sequentially

• deductive verification from Beckert and Klebenov (threading) in the KeY group

• the separation logic camp

• Berdine, Bierman, Calcagno, Distefano, Huisman, Hurlin, Jacobs, O’Hearn, Parkinson, Smans, Reynolds, Vafeiadis, Yang
Properly Synchronized

• several definitions of *properly synchronized* have been floated in the literature

• within Mobius, it meant:
  • a program has no race conditions
  • every shared variable is monitored by one or more locks
  • lock ordering is consistent
  • the locking discipline is respected
Specification
Constructs and Use

• JML permits one to specify through a small set of primitive constructs your own locking discipline

• if ESC/Java2 reports that your annotated code has no race conditions, then it likely has none

• if ESC/Java2 reports that your annotated code has no deadlocks, then it likely has none

• if RCC reports your annotated code has no race conditions, then it definitely has none
Locking Disciplines

• a locking discipline is a means by which concurrently accessed data is guarded

• a locking disciplines answers the questions

  • what data is (not) encapsulated?
  
  • how does one access said data?
  
  • which constructs are used for access control and which ones for data?
Discipline Flavors

• conservative/pessimistic
• liberal/optimistic
• strict separation of data and access control
• the data is the access control
• hierarchical structuring/ownership
• permission granting
Expressing and Reasoning about Locking Disciplines
class C {
    // this annotation means that read or write
    // accesses to field ‘f’ must only happen
    // after the containing object is locked
    // Only legal for instance fields.
    float f; //@ monitored

    synchronized void m() { f = 1.0; }        // ok
    synchronized static void n(C c)
            { c.f = 1.0; }                          // error!
}

// client code
C c = new C();
float g = c.f;                  // error!
c.f = 0.0;                      // error!
synchronized (c) { c.f = 1.0; } // ok
static monitors_for

class C {
    // 'aLock' must be locked for all access to field 'i'
    // If 'i' is static, then its lock must be static.
    //@ monitors_for i <- aLock;
    static int i;
    static Object aLock;

    static void m() {
        synchronized(aLock) { i = 1; }  // ok
    }
    static void n() { i = 2; }        // error!
}

C c = new C();
C.i = 1.0;                           // error!
synchronized(C.class) { c.d = 1.0; } // error!
synchronized(C.aLock) { c.d = 1.0; } // ok
simple dynamic monitors_for

```java
class C {
    double d;
    //@ monitors_for d <- this;
    // equivalent to just annotating with "monitored"

    synchronized void m() { d = 1.0; } // ok
    void n() { d = 1.0; }              // error!
    void o() { synchronized(this) { d = 1.0; } } // ok
    void p(D d) { synchronized(d.aLock) {
        d = 1.0; }
    } // error unless we can prove d.aLock
     // always is equal to this.
    void q() { Object o = this;
        synchronized(o) { d = 1.0; }
    } // ok
}
```

```java
C c = new C();
c.d = 1.0; // error!
synchronized(c) { c.d = 1.0; } // ok
```
class C {
    String aLock;
    double d;
    //@ monitors_for d <- aLock;

    synchronized void m() { d = 1.0; } // error!
    void n() { d = 1.0; } // error!
    void o() { synchronized(this) { d = 1.0; } } // error!
    void p() { synchronized(aLock) { d = 1.0; } } // ok
    void q() { Object o = anotherLock;
        synchronized(o) { d = 1.0; } } // ok
}

... 
C c = new C();
c.d = 1.0; // error!
synchronized(c.aLock) { c.d = 1.0; } // ok
class C {
    static Object aLock;
    String anotherLock;
    double d;
    //@ monitors_for d <- aLock, anotherLock;
    //@ axiom aLock < anotherLock;

    synchronized void m() { d = 1.0; }     // error!
    static void n() {
        synchronized(aLock) { d = 1.0; } }   // error!
    void o() { synchronized(anotherLock) {
        synchronized(aLock) { d = 1.0; } } } // error!
}

... 
C c = new C();
floa g = c.d;           // error, even for reads!
synchronized (C.aLock) {
    synchronized (c.anotherLock) {
        c.d = 1.0; } } }                  // ok
Subtleties

• locks are arbitrary objects
• must deal with aliasing of locks
• if the lock reference is null, one cannot lock
• field hiding matters
• spec-accessibility matters
• multiple monitors_for are permitted
Locksets

- \texttt{lockset} is of type \texttt{LockSet}
- denotes the set of locks held by the current thread
- membership in locksets
  - expression of the form $S[L]$ where $S$ is a spec-expr of type \texttt{LockSet} and $L$ is a spec-expr of ref-type denotes that $L$ is a member of $S$
  - $\max(S)$ denotes the maximum element of $S$
Complex Example

```java
public class Tree {
    public /*@ monitored */ Tree left, right;
    public /*@ monitored */ Thing contents;
    //@ axiom (∀ Tree t; t.left != null ==> t < t.left);
    //@ axiom (∀ Tree t; t.right != null ==> t < t.right);

    Tree(Thing c) {
        contents = c;
    }

    //@ requires \max(\lockset) <= this;
    public synchronized void visit() {
        contents.mungle();
        if (left != null) left.visit();
        if (right != null) right.visit();
    }
}
```
RCC Annotations

- variant of monitors_for, guarded_by, is used, e.g.,
  Type VarName /** guarded_by */ LockSet

- lockset names are used as shorthand, e.g.,
  /** requires LockSet */
  ReturnType MethodName(Args) {Body}

- class-level annotations of locksets possible
  ClassName /** {ghost Object LockSet} */ {ClassBody}
  ClassName /** {LockSet} */ VarName

- thread-local and thread-shared are introduced
  /** thread_local */ ClassDeclaration
  /** thread_shared */ ClassDeclaration
But what do developers do and understand?
Old Concurrent Java

• deadlock and system failure was pervasive before Thread’s methods were deprecated

• Flanagan et al. found that most Java methods are written as if they are atomic

• unsafe idioms and assumptions were and are pervasive (assumptions about immutability, atomicity, double-checking locks, lazy instantiation)
Open Questions

• Which concurrency constructs are used most often?

• What are the most common concurrency idioms witnessed in RT Java code?

• Does the Java concurrency library work at all in an RT Java setting?

• If it does not, why not, and how might it be changed to accommodate priorities?
Case Study

• a tool called the Histogram System has been formally specified (from formal requirements refined down to JML-annotated Java) to analyze developers’ use of concurrency

• our plan is to...
  • analyze >100M NCSS of off-the-shelf Java
  • analyze all known public examples of RT Java (probably 10,000s of NCSS)
Recent Work and Next Steps
Reasoning about RT Java

• Java PathFinder has been applied to RT Java by Lindstrom, Mehlitz, and Visser (ATVA’05)

• timing and dataflow analysis for WCET and schedulability by many researchers

• deductive verification about memory from Engel in the KeY group

• no work on deductive verification of safety and liveness, especially at the model level
Recent Work

• concurrency annotations at the model level
• traceable formal refinement to JML
• foundation for reasoning about concurrent architectures at the model level
The Concurrency Semantic Property

- six kinds of annotations
- annotations permitted at class or method level
- class-level annotations are not inherited

<table>
<thead>
<tr>
<th>concurrent</th>
<th>sequential</th>
<th>guarded</th>
</tr>
</thead>
<tbody>
<tr>
<td>failure</td>
<td>atomic</td>
<td>special</td>
</tr>
</tbody>
</table>
Preliminaries

• *Feature Thread Count*: The thread count for a feature $f$ of object $o$ is the number of threads simultaneously executing $f$ on object $o$.

• *Broken Object*: When an object is broken, the object’s invariants and feature postconditions are no longer guaranteed.
Semantics via Examples: Concurrent and Sequential

interface I {
  // @concurrent 5
  void m();
  // indicates that, if more than 5 threads enter m,
  // then this is broken

  // @sequential
  void n();
  // equivalent to @concurrent 1
  ...
Locks

...  
// @locks a, b, c  
void o();  
// o will attempt to acquire no more than  
// locks a, b, and c  

// @locks Void  
void p();  
// p will not attempt to acquire any locks  

// @locks *  
void q();  
// q may acquire any set of locks  
...
Guarded

... 
// @guarded a, b, c
void r();
// caller must hold a, b, and c prior to r being 
// executed, but need not acquire them before 
// calling r, and a, b, and c are released prior 
// to r terminating

// @guarded 3
void s();
// equivalent to guarded Semaphore(3)
...

Monday, 11 April 2011
Failure, Atomic, and Special

...  

// @concurrent 2
// @failure MyException
void t();
// t throws an exception of type MyException
// immediately when the number of threads
// executing t is 2 and a new thread calls t

// @atomic
void t();
// indicates that t is serializable wrt some
// definition of atomicity

// @special This method locks the database DB.
void u();
...
Specification
Expressions

• \textit{\textbackslash broken}

• \textit{\textbackslash semaphore} and \textit{\textbackslash semaphore}(\textit{descriptor})
  \textit{descriptor} is a string representation of a method signature

• \textit{\textbackslash thread\_count}

• \textit{\textbackslash thread\_limit} and \textit{\textbackslash thread\_limit}(\textit{descriptor})
Ex. Semantic Mapping

class C {
//@ invariant I;

//@ requires P; assignable A; ensures Q;
//@ concurrency concurrent 4
public void m() {}
}

class C {
//@ invariant !\broken ==> I;

/*@ normal_behavior
@   requires P; assignable A; ensures !\broken ==> Q;
@   also normal_behavior
@   requires \thread_count == \thread_limit;
@   assignable A;
@   ensures \broken; */
public void m() {} 

/*@ invariant typeof(this) == \type(C) ==> 
@   \thread_limit("m()") == 4; */
//@ invariant 4 <= \thread_limit("m()");
}
class BARBER
    feature
cut_hair
        -> c: CUSTOMER
        concurrency guarded Current
        ensure not c.needs_hair_cut
        ensure delta c.needs_hair_cut
end
class BARBER_SHOP
  feature
    barber: BARBER
    num_seats: INTEGER
      ensure 0 < Result
  end
  get_hair_cut
    -> c: CUSTOMER
      concurrency concurrent (num_seats + 1)
      concurrency failure NO_SEATS
      concurrency locks barber
      ensure not c.needs_hair_cut; delta c.needs_hair_cut
      -- behavior: barber.cut_hair(c)
  end
  make
    -> the_barber: BARBER
    -> the_num_seats: INTEGER
      ensure barber = the_barber; num_seats = the_num_seats
  end
end
class CUSTOMER
  concurrency guarded Current
  feature
    shop: BARBER_SHOP
      ensure Result /= Void
  end
  needs_hair_cut: BOOLEAN
  set_hair_cut
    ensure not needs_hair_cut; delta needs_hair_cut
  end
  regular_activities
    require not needs_hair_cut
    ensure needs_hair_cut; delta needs_hair_cut
    -- behavior: whatever the customer does between haircuts
  end
end
run
  concurrency locks shop.barber
  ensure delta needs_hair_cut
  -- behavior: repeatedly execute the sequence
  -- "regular_activities; retry shop.get_hair_cut until
  -- not needs_hair_cut"
end
make
  -> the_shop: BARBER_SHOP
  ensure shop = the_shop; not needs_hair_cut
end
end
class MAIN
    feature
        make
            -> the_seats: INTEGER
            -> the_customers: INTEGER
            require 0 < the_seats; 0 < the_customers
            -- behavior: create a barber, create a barber shop with
            -- the_seats seats, create the_customers customers,
            -- and "run" each customer in a separate thread
    end
end

class NO_SEATS inherit EXCEPTION end
Contributions and Limitations

• contributions
  • model-level concurrency annotations
  • design matches the typical set of concurrency patterns witnessed in code

• limitations
  • no notion of fairness is specified
  • unable to reason without a well-specified locking discipline
  • tool support still underway
Next Steps

• How to reason about concurrency at the architecture level?

• Which constructs do developers use (correctly) and how might they be modeled in our reasoning engines (HOL or SMT)?

• How might one introduce separation into JML? Should it be explicit or implicit?

• How might one introduce orchestration into JML? Does such help with respect to concurrency?