Atomicity for Reliable Concurrent Software

Cormac Flanagan
UC Santa Cruz

Shaz Qadeer
Microsoft Research

Joint work with
Stephen Freund

Towards Reliable Multithreaded Software

- Multithreaded software
  - increasingly common (Java, C#, GUIs, servers)
  - decrease latency
  - exploit underlying hardware
    - multi-core chips
- Heisenbugs due to thread interference
  - race conditions
  - atomicity violations
- Need tools to verify atomicity
  - dynamic analysis
  - type systems

Motivations for Atomicity

1. Beyond Race Conditions

Race Conditions

```
class Ref {
  int i;
  void inc() {
    int t;
    t = i;
    i = t+1;
  }
}
```


Race Conditions

```
class Ref {
  int i;
  void inc() {
    int t;
    t = i;
    i = t+1;
  }
}

Ref x = new Ref(0);

    x.inc();
    x.inc();

assert x.i == 2;
```


Race Conditions

```
class Ref {
  int i;
  void inc() {
    int t;
    t = i;
    i = t+1;
  }
}

Ref x = new Ref(0);
parallel {
    x.inc();    // two calls happen
    x.inc();    // in parallel
}
assert x.i == 2;
```

A race condition occurs if
- two threads access a shared variable at the same time
- at least one of those accesses is a write

Lock-Based Synchronization

```
class Ref {
  int i;              // guarded by this
  void inc() {
    int t;
    synchronized (x) {
      t = i;
      i = t+1;
    }
  }
}

Ref x = new Ref(0);
parallel {
    x.inc();    // two calls happen
    x.inc();    // in parallel
}
assert x.i == 2;
```

- Field guarded by a lock
- Lock acquired before accessing field
- Ensures race freedom

Limitations of Race-Freedom

```
class Ref {
  int i;              // guarded by this
  void inc() {
    int t;
    synchronized (x) {
      t = i;
      i = t+1;
    }
  }
}

Ref x = new Ref(0);
parallel {
    x.inc();    // two calls happen
    x.inc();    // in parallel
}
assert x.i == 2;
```

Ref.inc()
- race-free
- behaves correctly in a multithreaded context

Limitations of Race-Freedom

```
class Ref {
  int i;
  void inc() {
    int t;
    synchronized (this) {
      t = i;
    }
    synchronized (this) {
      i = t+1;
    }
  }
  ...
}
```

Ref.inc()
- race-free
- behaves incorrectly in a multithreaded context

Race freedom does not prevent errors due to unexpected interactions between threads

Limitations of Race-Freedom

```
class Ref {
  int i;
  void inc() {
    int t;
    synchronized (this) {
      t = i;
      i = t+1;
    }
  }
  void read() { return i; }
  ...
}
```

Ref.read()
- has a race condition
- behaves correctly in a multithreaded context

Race freedom is not necessary to prevent errors due to unexpected interactions between threads

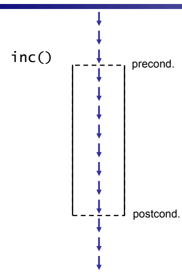
Race-Freedom

- Race-freedom is neither *necessary* nor *sufficient* to ensure the absence of errors due to unexpected interactions between threads
- Is there a more fundamental semantic correctness property?

Motivations for Atomicity

2. Enables Sequential Reasoning

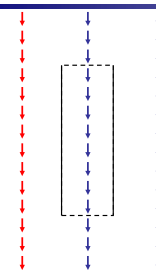
Sequential Program Execution




```
void inc() {
  ..

  ..
}
```

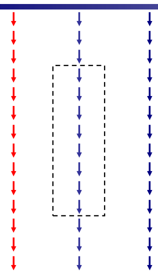

Multithreaded Execution



```
void inc() {
  ..

  ..
}
```

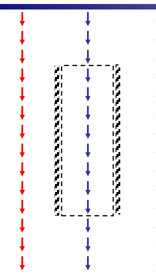

Multithreaded Execution



```
void inc() {
  ..

  ..
}
```


Multithreaded Execution



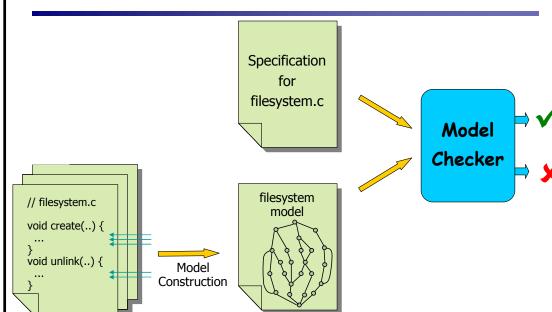
Atomicity

- guarantees concurrent threads do not interfere with atomic method
- enables sequential reasoning
- matches existing methodology

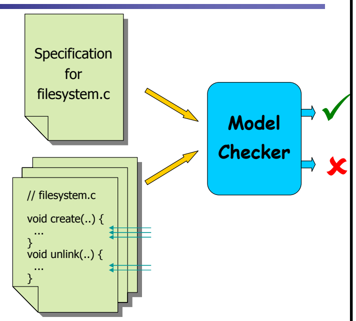
Motivations for Atomicity

3. Simple Specification

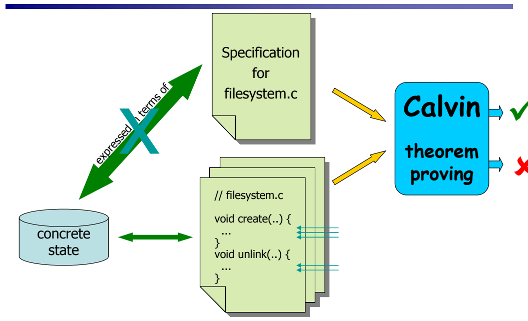
Model Checking of Software *Models*



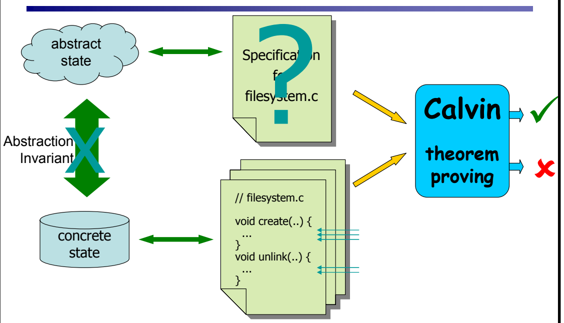
Model Checking of Software



Experience with Calvin Software Checker

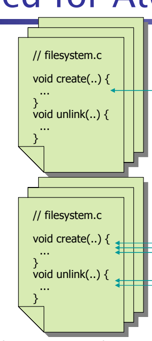


Experience with Calvin Software Checker



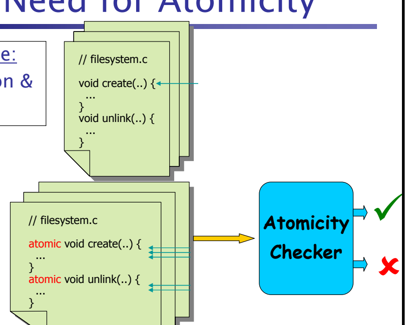
The Need for Atomicity

Sequential case:
code inspection &
testing mostly ok



The Need for Atomicity

Sequential case:
code inspection &
testing ok



Motivations for Atomicity

1. Beyond Race Conditions
2. Enables Sequential Reasoning
3. Simple Specification

Atomicity

- The method inc() is atomic if concurrent threads do not interfere with its behavior
- Guarantees that for every execution

acq(this) x t=i y i=t+1 z rel(this)

acq(this) x y t=i i=t+1 z rel(this)

- there is a *serial* execution with same behavior

x y acq(this) t=i i=t+1 rel(this) z

Atomicity

- Canonical property
  - (cmp. linearizability, serializability, ...)
- Enables sequential reasoning
  - simplifies validation of multithreaded code
- Matches practice in existing code
  - most methods (80%+) are atomic
  - many interfaces described as "thread-safe"
- Can verify atomicity statically or dynamically
  - atomicity violations often indicate errors
  - leverages Lipton's theory of reduction

Reduction [Lipton 75]

acq(this) X t=i Y i=t+1 Z rel(this)

acq(this) X Y t=i i=t+1 Z rel(this)

X Y acq(this) t=i i=t+1 Z rel(this)

X Y acq(this) t=i i=t+1 rel(this) Z

Checking Atomicity

```
atomic void inc() {
  int t;
  synchronized (this) {
    t = i;
    i = t + 1;
  }
}
```

R: right-mover	lock acquire
L: left-mover	lock release
B: both-mover	race-free variable access
A: atomic	conflicting variable access

$\underbrace{\text{acq(this)}_R \;\; \text{t=i}_B \;\; \text{i=t+1}_B \;\; \text{rel(this)}_L}_{A}$

- Reducible blocks have form: (R|B)* [A] (L|B)*

Checking Atomicity (cont.)

```
atomic void inc() {
  int t;
  synchronized (this) {
    t = i;
  }
  synchronized (this) {
    i = t + 1;
  }
}
```

R: right-mover	lock acquire
L: left-mover	lock release
B: both-mover	race-free variable access
A: atomic	conflicting variable access

$\underbrace{\underbrace{\text{acq(this)}_R \;\; \text{t=i}_B \;\; \text{rel(this)}_L}_{A} \;\; \underbrace{\text{acq(this)}_R \;\; \text{i=t+1}_B \;\; \text{rel(this)}_L}_{A}}_{\text{Compound}}$

java.lang.StringBuffer

```
/**
  ... used by the compiler to implement the binary
  string concatenation operator ...

  String buffers are safe for use by multiple
  threads. The methods are synchronized so that
  all the operations on any particular instance
  behave as if they occur in some serial order
  that is consistent with the order of the method
  calls made by each of the individual threads
  involved.
*/
/*# atomic */ public class StringBuffer { ... }
```

FALSE

java.lang.StringBuffer

```
public class StringBuffer {
  private int count;
  public synchronized int length() { return count; }
  public synchronized void getChars(...) { ... }

  atomic public synchronized void append(StringBuffer sb){

    int len = sb.length();
    ...
    ...
    ...
    sb.getChars(...,len,...);
    ...
  }
}
```

sb.length() acquires lock on sb, gets length, and releases lock

other threads can change sb

use of stale len may yield StringIndexOutOfBoundsException inside getChars(...)

java.lang.StringBuffer

```
public class StringBuffer {
  private int count;
  public synchronized int length() { return count; }
  public synchronized void getChars(...) { ... }

  atomic public synchronized void append(StringBuffer sb){

    int len = sb.length();          A
    ...
    ...
    ...
    sb.getChars(...,len,...);       A
    ...
  }
}
```

Compound

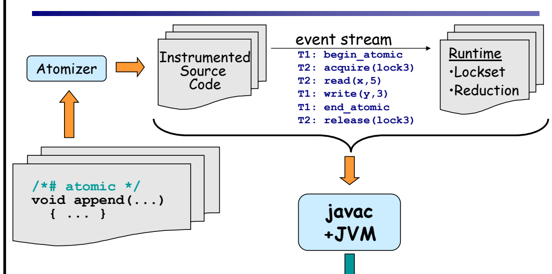
Tutorial Outline

- Part 1
  - Introduction
  - Runtime analysis for atomicity
- Part 2
  - Model checking for atomicity
- Part 3
  - Type systems for concurrency and atomicity
- Part 4
  - Beyond reduction – atomicity via “purity”

Part I continued:

Runtime Analysis for Atomicity

Atomizer: Instrumentation Architecture



Warning: method “append” may not be atomic at line 43

Atomizer: Dynamic Analysis

- Lockset algorithm
  - from Eraser [Savage et al. 97]
  - identifies race conditions
- Reduction [Lipton 75]
  - proof technique for verifying atomicity, using information about race conditions

Analysis 1: Lockset Algorithm

- Tracks *lockset* for each field
  - lockset = set of locks held on all accesses to field
- Dynamically infers protecting lock for each field
- Empty lockset indicates possible race condition

Lockset Example

Thread 1

```
synchronized(x) {
  synchronized(y) {
    o.f = 2;
  }
  o.f = 11;
}
```

Thread 2

```
synchronized(y) {
  o.f = 2;
}
```

- First access to `o.f`:

LockSet(`o.f`) = Held(curThread)
= { x, y }

Lockset Example

Thread 1

```
synchronized(x) {
  synchronized(y) {
    o.f = 2;
  }
  o.f = 11;
}
```

Thread 2

```
synchronized(y) {
  o.f = 2;
}
```

- Subsequent access to `o.f`:

LockSet(`o.f`) := LockSet(`o.f`) ∩ Held(curThread)
= { x, y } ∩ { x }
= { x }

Lockset Example

Thread 1

```
synchronized(x) {
  synchronized(y) {
    o.f = 2;
  }
  o.f = 11;
}
```

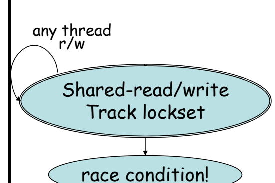
Thread 2

```
synchronized(y) {
  o.f = 2;
}
```

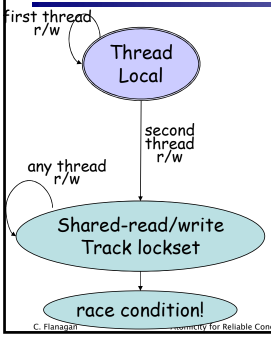
- Subsequent access to `o.f`:

LockSet(`o.f`) := LockSet(`o.f`) ∩ Held(curThread)
= { x } ∩ { y }
= { } => race condition

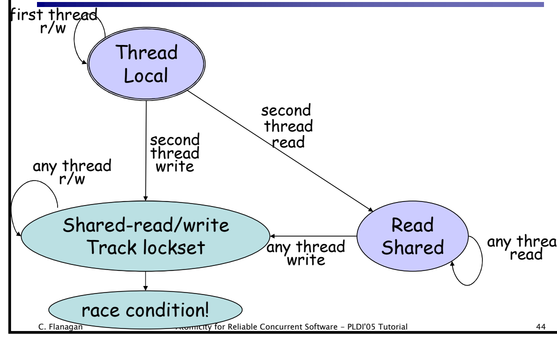
Lockset



Lockset with Thread Local Data



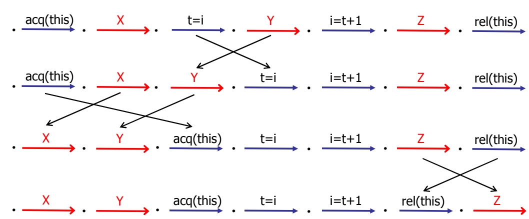
Lockset with Read Shared Data



Atomizer: Dynamic Analysis

- Lockset algorithm
  - from Eraser [Savage et al. 97]
  - identifies race conditions
- Reduction [Lipton 75]
  - proof technique for verifying atomicity, using information about race conditions

Reduction [Lipton 75]

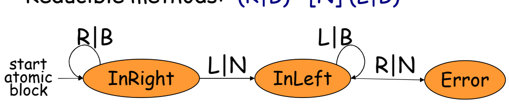


Performing Reduction Dynamically

- R: right-mover
  - lock acquire
- L: left-mover
  - lock release
- B: both-mover
  - race-free field access
- N: non-mover
  - access to "racy" fields

acq(lock)	j=bal	bal=j+n	rel(lock)
R	B	B	L

- Reducible methods: (R|B)* [N] (L|B)*



Atomizer Review

- Instrumented code calls Atomizer runtime
  - on field accesses, sync ops, etc
- Lockset algorithm identifies races
  - used to classify ops as movers or non-movers
- Atomizer checks reducibility of atomic blocks
  - warns about atomicity violations

Evaluation

- 12 benchmarks
  - scientific computing, web server, std libraries, ...
  - 200,000+ lines of code
- Heuristics for atomicity
  - all synchronized blocks are atomic
  - all public methods are atomic, except `main` and `run`
- Slowdown: 1.5x – 40x

Performance

Benchmark	Lines	Base Time (s)	Slowdown
elevator	500	11.2	-
hedc	29,900	6.4	-
tsp	700	1.9	21.8
sor	17,700	1.3	1.5
moldyn	1,300	90.6	1.5
montecarlo	3,600	6.4	2.7
raytracer	1,900	4.8	41.8
mtrt	11,300	2.8	38.8
jigsaw	90,100	3.0	4.7
specJBB	30,500	26.2	12.1
webl	22,300	60.3	-
lib-java	75,305	96.5	-

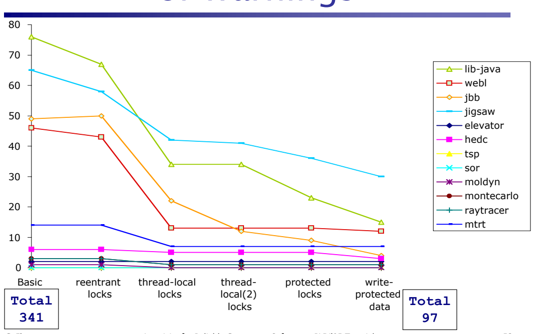
Extensions

- Redundant lock operations are both-movers
  - re-entrant acquire/release
  - operations on thread-local locks
  - operations on lock A, if lock B always acquired before A
- Write-protected data

Write-Protected Data

```
class Account {
    int bal;
    /*# atomic */ int read() { return bal; }
    /*# atomic */ void deposit(int n) {
R     synchronized (this) {
B       int j = bal;
N       bal = j + n;
L     }
    }
}
```


Extensions Reduce Number of Warnings



Evaluation

- Warnings: 97 (down from 341)
- Real errors (conservative): 7
- False alarms due to:
  - simplistic heuristics for atomicity
    - programmer should specify atomicity
  - false races
  - methods irreducible yet still "atomic"
    - eg caching, lazy initialization
- No warnings reported in more than 90% of exercised methods

java.lang.StringBuffer

```
public class StringBuffer {
  private int count;
  public synchronized int length() { return count; }
  public synchronized void getChars(...) { ... }
  /*# atomic */
  public synchronized void append(StringBuffer sb){

    int len = sb.length();
    ...
    ...
    ...
    sb.getChars(...,len,...);
    ...
  }
}
```

```
StringBuffer.append is not atomic:
  Start:
    at StringBuffer.append(StringBuff
    at Thread1.run(Example.java:17)

  Commit: Lock Release
    at StringBuffer.length(StringBuff
    at StringBuffer.append(StringBuff
    at Thread1.run(Example.java:17)

  Error: Lock Acquire
    at StringBuffer.getChars(StringBu
    at StringBuffer.append(StringBuff
    at Thread1.run(Example.java:17)
```