Construction and Verification of Software

2017 - 2018

MIEI - Integrated Master in Computer Science and Informatics
Consolidation block

Lecture 7 - Concurrent Abstract Data Types

João Costa Seco (joao.seco@fct.unl.pt)
based on previous editions by **Luís Caires** (lcaires@fct.unl.pt)

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Concurrent
Abstract Data Types

Concurrency

Concurrency

- Several threads of control **share** the same state
- **Interference:**
  - the local view of a thread may change without notice (another thread may act “under the hood”).
- “No-sharing concurrency” (a.k.a. “parallelism”)
  - not treated in this course
- Interference is **the essence** of concurrency
- **Key issue**
  - how to keep state consistency in the presence of sharing and interference
- Reasoning about concurrency is **challenging**!

Verification of Concurrent ADTs

- Check the consistency of stateful objects, when subject to (concurrent) operation requests.
- Any method call must preserve consistency of the ADT
- Consistency is precisely expressed by the representation invariant (and abstraction mapping).
- Every method body *mbody* of an ADT operation must preserve the RepInv

  { RepInv && requires-cond } *mbody* { RepInv && ensures-cond}

- This line of reasoning works well under the assumption of sequentiality. What if methods overlap in time?
- **Challenge**: how to program and reason about ADTs with interfering methods

Interference

- Consider a Stack ADT
  - **push(v), pop(), isEmpty()**
  - push() interferes with pop() ?
  - pop() interferes with isEmpty() ?
  - pop() interferes with pop() ?
- Consider a Dictionary ADT
  - **assoc(key,data), find(key)**
  - assoc() interferes with find() ?
  - assoc() interferes with assoc() ?
  - find() interferes with find() ?

Operation Level Behaviour

- + useful to reason at the level of ADT operations, not about unstructured low level code and state
- Each ADT operation is performed in three steps
  - The operation is called (by the client thread)
  - The operation is executed (inside the ADT)
  - The operation returns
- Example:
  - push_call(2)
    - ... execute (internally to the ADT)
  - push_return
  - pop_call()
    - .... execute (internally to the ADT)
  - pop_return(2)

Operation Level Behaviour

- We may consider several levels of concurrency
  - Several threads are invoking ADT operations but only one may actually be executing the operation
    - strict serialisation, easier to implement and reason about
    - less chances of “unsound” interference
  - Several threads are invoking ADT operations but more than one may be executing an operation
    - more parallelism, more concurrency, harder to implement and reason about
    - more chances of “unsound” interference
- How does the concurrent object behaviour relate to the intended sequential object specification ?

Two Basic Models

- **Serializability**
  - The global trace is always consistent with some sequential serialisation of previous operations (no overlaps of calls and returns), compatible with the sequential specification.
- **Linearizability**
  - The global trace is always consistent with a view in which previous operations appear to occur instantaneously between calls and returns, and the obtained serialisation is compatible with the sequential specification.
- Linearizability is more flexible than serializability, as it allows for more parallel behaviour.

Desired properties of op execution

- Let us reinterpret the "classical" ACID story:
- **Atomicity**
  – No intermediate states are visible (clearly, they are not compatible with the representation invariant)
- **Consistency**
  – Operations lead from a sound state to a sound state (invariant and soundness are preserved)
- **Isolation**
  – This is another word for "no unsafe interference"
- **Durability** (this goes without saying)
  – Effects are undoable (N.B: this is more useful to highlight in the context of database transactions)

Correctness of Concurrent ADTs

- With naive concurrency, it is hard (or impossible) for client code to be sure if a specific **post-condition** holds.
- E.g: two clients modify the concrete state at the same time, bringing the state inconsistent, breaking the representation invariant, or even crashing the code.
- Solution using serialisation:
  - **serialize** usages of concrete states, so that just a **single** thread may be accessing the state at each given moment (mutual exclusion of concrete state)
  - We may then safely reason about such mutually exclusive code fragments as we have done for sequential code.

Correctness of Concurrent ADTs

- With naive concurrency, it is hard (or impossible) for client code to be sure if a specific **pre-condition** holds.
- E.g: client checks that a buffer is not empty, but other thread empties it under the woods.
- Solution:
  - Concurrency control replaces pre-condition checking (on the client side) by explicit waiting for the precondition to hold (inside the ADT).
  - The pre-condition for some ADT op can only be enabled by executing some other ADT op
  - So waiting for a pre-condition must be managed by special programming language or system support, in a coordinated way with other ADT operations

Concurrent Programming

- Reasoning about concurrency is **hard**
- Making sure the code is right is much more difficult than in sequential code
- Trying to simulate the program running in your head and debug it does not work anymore :-)
  - It does not work in the sequential case either, actually..., although you may still believe.
- We will now study how to design and construct correct concurrent code, based on **monitors**
- monitor = invariant preserving concurrent ADT
- Nicely supported by java.concurrent.util

Monitors

Operating Systems — C. Weissman, Editor

Monitors: An Operating System Structuring Concept

C.A.R. Hoare
The Queen's University of Belfast

This paper develops Brinch-Hansen's concept of a monitor as a method of structuring an operating system. It introduces a form of synchronization, describes a possible method of implementation in terms of semaphores and gives a suitable proof rule. Illustrative examples include a single resource scheduler, a bounded buffer, an alarm clock, a buffer pool, a disk head optimizer, and a version of the problem of readers and writers.

Key Words and Phrases: monitors, operating systems, scheduling, mutual exclusion, synchronization, system implementation languages, structured multiprogramming
CR Categories: 4.31, 4.22




Monitors

- An ADT where operations may be called concurrently
- **2 key mechanisms** provided for ensuring consistency:

synchronization (a.k.a. mutual exclusion)

  – only a single thread may “own” the shared state at any time object, and has permission to change it

– All that client code may expect from shared state is **the invariant, and nothing more than the invariant**

  – any context switches must preserve the invariant

concurrency control

  – pre-condition checking must be usually replaced by explicit waiting for the pre-condition to hold.

  – conditions refine the invariant into finer partitions.

Implementation of monitors

- To implement monitors in Java, we will use locks
  - You have already heard about locks (FSO, CP)
    - A lot harder to reason about programs if we just think of using locks in an unstructured way
- We may latter refine the borders of serialisability to get more concurrency (approach linearisability)
  - Still, useful to only use locks as delimiters of abstract operations on the shared state, thought of as a ADT
  - We will use the **java.util.concurrent** API (Doug Lea)
  - We will learn how to design concurrent ADTs without thinking "operationally", but rather in terms of (partitioned) ownership, invariants, and conditions.

Example (Bounded Counter)

```
class BCounter {
  int N;
  int MAX;
  BCounter(int max) { N = 0 ; MAX = max; }
  void inc() { N++; }
  void dec() { N--; }
  int get() { return N; }
}
```

Example (Bounded Counter)

```
/*@
    predicate BCounterInv(BCounter c; int v,int m) =
        c.N |-> v &*& c.MAX |-> m &*& v>=0 &*& v<=m;
@*/
class BCounter {
  int N;
  int MAX;
  BCounter(int max)
  //@ requires 0 <= max;
  //@ ensures BCounterInv(this,0,max);
  { N = 0 ; MAX = max; }

  void inc()
  //@ requires BCounterInv(this,?n,?m) &*& n < m;
  //@ ensures BCounterInv(this,n+1,m);
  { N++; }

  void dec()
  //@ requires BCounterInv(this,?n,?m) &*& n > 0;
  //@ ensures BCounterInv(this,n-1,m);
  { N--; }

  int get()
  //@ requires BCounterInv(this,?n,?m);
  //@ ensures BCounterInv(this,n,m) &*& 0<=result &*& result<=m;
  { return N; }
}
```

Example (Bounded Counter)

```
public static void main(String[] args)
//@ requires true;
//@ ensures true;
{
     int MAX = 100;
     BCounter c = new BCounter(MAX);
     //@ assert BCounterInv(c,0,MAX);
      if (c.get() < MAX) {
          c.inc(); // this is ok, precondition satisfied
      }
}
```

Example (Bounded Counter)

```
public static void main(String[] args)
//@ requires true;
//@ ensures true;
{
     int MAX = 100;
     BCounter c = new BCounter(MAX);
     //@ assert BCounterInv(c,0,MAX);
     giveaway(c); // potentially give other thread access to c
      if (c.get() < MAX) {
          //@ assert BCounterInv(c,?v,MAX) &*& v < MAX;
          c.inc();
          // not safe any more as other thread may have acted
      }
}
```

1st: Serialise access to shared state

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

class BCounter {
  int N;
  int MAX;
  ReentrantLock mon;

  BCounter(int max)
 {
      N = 0 ;
      MAX = max;
      mon = new ReentrantLock();
  }
…
```

Example (Bounded Counter)

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

class BCounter {
  int N;
  int MAX;
  ReentrantLock mon;
…
  void inc()
 {
      mon.enter(); //request permission to the shared state
      N++;
      mon.leave(); //release ownership of the shared state
  }

  void dec()
 {
      mon.enter(); //request permission to the shared state
      N--;
      mon.leave(); //release ownership of the shared state
  }
```

Example (Bounded Counter)

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

class BCounter {
  int N;
  int MAX;
  ReentrantLock mon;
…
  void inc()
 {
      mon.lock(); //request permission to the shared state
      N++;
      mon.unlock(); //release ownership of the shared state
  }

  void dec()
 {
      mon.lock(); //request permission to the shared state
      N--;
      mon.unlock(); //release ownership of the shared state
  }
```

Example (Bounded Counter)

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

class BCounter {
  int N;
  int MAX;
  ReentrantLock mon;
…
  int get()
  {
      int r;
      mon.enter();
      r = N; // put a copy on the stack, private to the thread
      mon.leave();
      return r;
  }
```

Hoare Rule for enter / leave

{ emp } m.enter () { *SharedStateInv* }

{ *SharedStateInv* } m.leave () { emp }

SharedStateInv is the representation invariant.

In our example ...

```
//@ predicate BCounterInv(BCounter c) =
    c.N |-> ?v &*& c.MAX |-> ?m &*& v>=0 &*& v<=m;
```

so:

{ `emp` } m.enter () { `BCounterInv(this)` }

Issue: Red assertions not available!

```
class BCounter {
  int N;
  int MAX;
  ReentrantLock mon;

 void inc()
  //@ requires BCounterInv(this,?n,?m) &*& n < m;
  //@ ensures BCounterInv(this,n+1,m);
  {
      mon.enter(); //@ request permission to the shared state
      //@ assert BCounterInv(this,?n,?m)
      N++;
      //@ assert BCounterInv(this,n+1,m)
      mon.leave(); //@ release ownership of the shared state
  }
```

How can a client check n<m ?

- With naive concurrency, it is hard (or impossible) for client code to be sure a **pre-condition** holds.
- E.g: client checks that a buffer is not empty, but other thread empties it under the hood.
- Solution:
  - Concurrency control replaces pre-condition checking (on the client side) by explicit waiting for the precondition to hold (inside the ADT).
  - The pre-condition for some ADT op can only be enabled by executing some other ADT op
  - So waiting for a pre-condition must be managed by special programming language or system support, in a coordinated way with other ADT operations

“invisible” abstract state

- Many threads may be interfering, so the only thing one may assume is the invariant, only after entering the shared state a client may know extra details about the concrete state.
- In fact, nothing specific about the abstract state may be revealed to client code, and we need to be less informative about the abstract state (e.g., no current val)
- Inside the object, the only unprotected objects are the locks (or the single lock).
- Each lock can be used to ask permission to access a disjoint part of the shared state.
- We must precisely define which part of the shared state is separately owned by each lock.

Example (Bounded Counter)

```
/*@
    predicate_ctor BCounter_shared_state (BCounter c) () =
        c.N |-> ?v &*& v >= 0 &*& c.MAX |-> ?m &*& m > 0 &*& v <= m;
*@/

/*@ predicate BCounterInv(BCounter c) =
    c.mon |-> ?l &*&
    l != null &*& lck(l,1,BCounter_shared_state(c))
    @*/
class BCounter {
  int N;
  int MAX;
  ReentrantLock mon;
  BCounter(int max)
  //@ ensures BCounterInv(this);
  {
    N = 0 ;
    MAX = max;
    mon = new ReentrantLock();
  }
}
```

Example (Bounded Counter)

```
class BCounter {
  int N;
  int MAX;
  ReentrantLock mon;

  void inc()
  //@ requires BCounterInv(this);
  //@ ensures BCounterInv(this);
  {
    mon.enter(); // request permission to the shared state
    //@ open BCounter_shared_state(this)();
    N++;
    //@ close BCounter_shared_state(this)();
    mon.leave(); // release ownership of the shared state
  }
}
```

Example (Bounded Counter)

```
class BCounter {
  int N;
  int MAX;
  ReentrantLock mon;

  void dec()
  //@ requires BCounterInv(this);
  //@ ensures BCounterInv(this);
  {
    mon.enter();
    //@ open BCounter_shared_state(this)();
    N--;
    //@ close BCounter_shared_state(this)();
    mon.leave();
  }
}
```

What if N==0 ?

```
class BCounter {
  int N;
  int MAX;
  ReentrantLock mon;

  void dec()
  //@ requires BCounterInv(this); // no way to reveal a pre-cond!
  //@ ensures BCounterInv(this);
  {
    mon.enter();
    //@ open BCounter_shared_state(this)();
    N--;
    //@ close BCounter_shared_state(this)(); // must ensure N>=0!
    mon.leave();
  }
}
```

Partition shared state using conditions

```
class BCounter {
  int N;
  int MAX;
  ReentrantLock mon;
  Condition notzero;
  Condition notmax;

  void dec()
  //@ requires BCounterInv(this);
  //@ ensures BCounterInv(this);
  {
    mon.enter();
    //@ open BCounter_shared_state(this)();
    if (N==0) notzero.wait();
    N--;
    //@ close BCounter_shared_state(this)();
    mon.leave();
  }
}
```

Partition shared state using conditions

```
/*@

predicate_ctor BCounter_shared_state (BCounter c) () =
   c.N |-> ?v &*& v >= 0 &*& c.MAX |-> ?m &*& m > 0 &*& v <= m;

predicate_ctor BCounter_nonzero (BCounter c) () =
   c.N |-> ?v &*& c.MAX |-> ?m &*& v > 0 &*& m > 0 &*& v <= m;

predicate_ctor BCounter_nonmax (BCounter c) () =
   c.N |-> ?v &*& c.MAX |-> ?m &*& v < m &*& m > 0 &*& v >= 0;

predicate BCounterInv(BCounter c) =
        c.mon |-> ?l
    &*& l != null
    &*& lck(l,1, BCounter_shared_state(c))
    &*& c.notzero |-> ?cc
    &*& cc !=null
    &*& cond(cc, BCounter_shared_state(c), BCounter_nonzero(c))
    &*& c.notmax |-> ?cm
    &*& cm !=null
    &*& cond(cm, BCounter_shared_state(c), BCounter_nonmax(c));
@*/
```

Partition shared state using conditions

```
class BCounter {
  int N;
  int MAX;
  ReentrantLock mon;
  Condition notzero;
  Condition notmax;

  void dec()
  //@ requires BCounterInv(this);
  //@ ensures BCounterInv(this);
  {
    mon.enter();
    //@ open BCounter_shared_state(this)();
    if (N==0) notzero.wait();
    N--;
    //@ close BCounter_shared_state(this)();
    mon.leave();
  }
}
```

Example (Bounded Counter)

```
class BCounter {
  int N;
  int MAX;
  ReentrantLock mon;
  Condition notzero;
  Condition notmax;

  BCounter(int max)
  //@ requires max > 0;
  //@ ensures BCounterInv(this);
  {
   MAX = max; mon = new ReentrantLock();
   //@ close BCounter_shared_state(this)();
   //@ close set_cond(BCounter_shared_state(this),BCounter_nonzero(this));
   notzero = mon.newCondition(); // notzero set to mean N > 0  !!
   //@ close set_cond(BCounter_shared_state(this),BCounter_nonmax(this));
   notmax = mon.newCondition();  // notmax set to mean N < MAX !!
  }
}
```

Partition shared state using conditions

```
class BCounter {
  int N;
  int MAX;
  ReentrantLock mon;
  Condition notzero; Condition notmax;
  void dec()
  //@ requires BCounterInv(this);
  //@ ensures BCounterInv(this);
  {
    mon.enter();
    //@ open BCounter_shared_state(this)();
    if (N==0) notzero.wait();
    //@ open BCounter_notzero(this)(); // refined state >=0
    N--;
    //@ close BCounter_shared_state(this)();
    mon.leave();
  }
}
```

Partition shared state using conditions

```
class BCounter {
  int N;
  int MAX;
  ReentrantLock mon;
  Condition notzero; Condition notmax;
  void inc()
  //@ requires BCounterInv(this);
  //@ ensures BCounterInv(this);
  {
    mon.enter();
    //@ open BCounter_shared_state(this)();
    if (N==MAX) notmax.wait();
    //@ open BCounter_notmax(this)(); // refined state <= max
    N++;
    //@ close BCounter_shared_state(this)();
    mon.leave();
  }
}
```

Hoare Rule for await

{ SharedStateInv } C.await() { SharedStateInv && cond(C) }

cond(C) is the refined state property denoted by condition C.

In our example:

- cond(**notzero**) = (N > 0)
- cond(**notmax**) = (N < MAX)

Ensure progress using signalling

```
void inc()
//@ requires BCounterInv(this);
//@ ensures BCounterInv(this);
{
  mon.enter();
  //@ open BCounter_shared_state(this)();
  if (N==MAX) notmax.wait();
  //@ assert BCounter_notmax(this)();
  N++;
  //@ close BCounter_notzero(this)();
  notzero.signal();
  //@ close BCounter_shared_state(this)();
  mon.leave();
}
```

Ensure progress using signalling

```
void dec()
//@ requires BCounterInv(this);
//@ ensures BCounterInv(this);
{
  mon.enter();
  //@ open BCounter_shared_state(this)();
  if (N==0) notzero.wait();
  //@ assert BCounter_notzero(this)();
  N--;
  //@ close BCounter_notmax(this)();
  notmax.signal();
  //@ close BCounter_shared_state(this)();
  mon.leave();
}
```

Hoare Rule for signal

{ SharedStateInv && cond(C) } C.signal() { SharedStateInv }

cond(C) is the refined state property denoted by condition C.

In our example:

- cond(**`notzero`**) = (N > 0)
- cond(**`notmax`**) = (N < MAX)

Defending against unsound implementation

Excerpt from Java API documentation:

Implementation Considerations

*When waiting upon a `Condition`, a "spurious wakeup" is permitted to occur, in general, as a concession to the underlying platform semantics. This has little practical impact on most application programs as a `Condition` **should always be waited** upon in a loop, testing the state predicate that is being waited for.*
An implementation is free to remove the possibility of spurious wakeups but it is recommended that applications programmers always assume that they can occur and so always wait in a loop.

Defending against unsound implementation

```
void inc()
//@ requires BCounterInv(this);
//@ ensures BCounterInv(this);
{
  mon.enter();
  //@ open BCounter_shared_state(this)();
  while(N==MAX) notmax.wait();
  //@ assert BCounter_notmax(this)();
  N++;
  //@ close BCounter_notzero(this)();
  notzero.signal();
  //@ assert BCounter_shared_state(this)();
  mon.leave();
}
```

Defending against unsound implementation

```
void dec()
//@ requires BCounterInv(this);
//@ ensures BCounterInv(this);
{
  mon.enter();
  //@ open BCounter_shared_state(this)();
  while (N==0) notzero.wait();
  //@ assert BCounter_notzero(this)();
  N--;
  //@ close BCounter_notmax(this)();
  notmax.signal();
  //@ assert BCounter_shared_state(this)();
  mon.leave();
}
```

Concurrent ADT Construction Steps

Concurrent ADT Construction Steps

- Associate a monitor to the ADT (mon)
- Determine the CADT Representation Invariant (RI), which now talks about the **shared state**

  The RI describes the memory footprint of the shared state, subject to other various conditions.

- In the implementation of each operation of the CADT

  To get access to the RI, **you must** mon.lock()

  When done and **only if the RI holds** you mon.unlock()

- Replace the ADT op pre-conditions by conditions inside the monitor (this part must be carefully though).

Summary (key monitor primitives)

- **mon.enter(); // a.k.a. mon.lock()**

  asks for exclusive access to the shared state

  ```
  { P } mon.enter() { P &*& SSInv }
  ```

- **mon.leave(); // a.k.a. mon.unlock()**

  releases exclusive access to the shared state

  ```
  { P &*& SSInv } mon.leave(); { P }
  ```

- **cond.wait();**

  releases exclusive access to the shared state

  ```
  { P &*& SSInv} cond.await(); { P &*& SSCond}
  ```

- **cond.signal()**

  releases exclusive access to the shared state

  ```
  { P &*& SSCond} cond.signal(); { P &*& SSInv}
  ```

Concurrent ADT Construction Steps

- To replace ADT op pre-conditions by conditions inside the monitor, we must consider the following aspects:
- When a thread enters a CADT op and gets ownership of the RI, it may find that the state does not satisfy the pre-condition (e.g., wants to dec but counter value is zero)
- The thread **must then await for the condition to hold** (e.g, for the value to be > 0).
- Conversely, whenever a thread running inside the CADT establishes any one of the monitor conditions (e.g., inc establishes value >0), **it has the duty to signal the condition** (so that the runtime system, may awake a waiting thread)
- Notice: signaling is there to help the system to progress, and simplify the implementation of monitors.

Java Monitors Interface

```
package java.util.concurrent.locks;

interface Lock:

void lock() // Acquires the lock.
void unlock() // Releases the lock.
Condition newCondition() // Returns a new Condition instance that is
bound to this Lock instance.

interface Condition:
void await() // Causes the current thread to wait until the
condition holds (is signaled to hold).
void signal() // Signals to the runtime system that the condition
holds.This may cause a waiting thread on this condition to wakeup.
```

These are just a few hints, read the java docs!

Java Monitors Verifast Interface

```
package java.util.concurrent.locks;

/*@
predicate lck(ReentrantLock s; int p, predicate() inv);

predicate cond(Condition c; predicate() inv, predicate() p);

predicate enter_lck(int p, predicate() inv) = (p == 0 ? emp : inv()) ;

predicate set_cond(predicate() inv, predicate() p) = true;

@*/
```

enter_lock: to associate Representation Invariant to monitor

set_cond: to associate logical assertion to Condition object

Java Monitors Verifast Interface

```
public class ReentrantLock {

    public ReentrantLock();
    //@ requires enter_lck(1,?inv);
    //@ ensures lck(this, 1, inv);

    public void lock();
    //@ requires [?f]lck(?t, 1, ?inv);
    //@ ensures [f]lck(t, 0, inv) &*& inv();

    public void unlock();
    //@ requires [?f]lck(?t, 0, ?inv) &*& inv();
    //@ ensures [f]lck(t, 1, inv);

    public Condition newCondition();
    //@ requires lck(?t, 1, ?inv) &*& set_cond(inv, ?pred);
    //@ ensures lck(t, 1, inv) &*& result != null &*& cond(result,inv,pred);
}
```

Java Monitors Verifast Interface

```
package java.util.concurrent.locks;

public interface Condition {

    public void await();
        //@ requires cond(this,?inv,?acond) &*& inv();
        //@ ensures cond(this,inv, acond) &*& acond();

    public void signal();
        //@ requires cond(this,?inv,?acond) &*& acond();
        //@ ensures cond(this,inv,acond) &*& inv();

}
```

Counter ADT (Java + Verifast)

```
package CCounterMain;

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

/*@

predicate_ctor CCounter_shared_state (CCounter c) () = c.N |-> ?v &*& v >= 0;

predicate_ctor CCounter_notzero_state (CCounter c) () = c.N |-> ?v &*& v > 0;

predicate CCounterInv(CCounter c;) =
 c.mon |-> ?l &*& l!=null &*& lck(l,1, CCounter_shared_state(c)) &*&
 c.notzero |-> ?cc &*& cc != null &*&
   cond(cc, CCounter_shared_state(c),CCounter_notzero_state(c));

@*/
```

Counter ADT (Java + Verifast)

```
public class CCounter {

int N;
ReentrantLock mon;
Condition notzero;

public CCounter()
//@ requires true;
//@ ensures CCounterInv(this);
{
//@ close CCounter_shared_state(this)();
//@ close enter_lck(1,CCounter_shared_state(this));
 mon = new ReentrantLock();
 //@ close set_cond(CCounter_shared_state(this),CCounter_notzero_state(this));
notzero = mon.newCondition();
//@ close CCounterInv(this);
}
...
```

Counter ADT (Java + Verifast)

```
public class CCounter {

  int N;
  ReentrantLock mon;
  Condition notzero;

public void inc()
//@ requires [?f]CCounterInv(this);
//@ ensures [f]CCounterInv(this);
  {
  //@ open CCounterInv(this);
  mon.lock();
  //@ open [f] CCounter_shared_state(this)();
  N++;
  //@ close CCounter_notzero_state(this)();
  notzero.signal();
  mon.unlock();
  //@ close [f]CCounterInv(this);
  }
```

Counter ADT (Java + Verifast)

```
public class CCounter {
...
public void dec()
//@ requires [?f]CCounterInv(this);
//@ ensures [f]CCounterInv(this);
{
  try {
//@ open [f]CCounterInv(this);
   mon.lock();
//@ open CCounter_shared_state(this)();
   if (N==0) {
//@ close CCounter_shared_state(this)();
    notzero.await(); }
//@ open CCounter_notzero_state(this)();
    N--;
  } catch (java.lang.InterruptedException e){}
//@ close CCounter_shared_state(this)();
  mon.unlock();
//@ close [f]CCounterInv(this);
  }
```