VERIFICATION OF SHARED-READING SYNCHRONISERS

MARIEKE HUISMAN
UNIVERSITY OF TWENTE, NETHERLANDS

JOINT WORK WITH AFSHIN AMIGHI AND STEFAN BLOM
An important challenge: Reliable software with parallel computations

All software has errors!

Software failures can have enormous impact
SYNCHRONISERS

- Major building block for concurrent software
- Role:
  - Tame the inferences between threads
  - Prevent unwanted interleavings
- Well-known examples (in Java API):
  - Reentrant Locks
  - Barriers
  - Countdown Latch
  - Semaphore
- Exclusive Access vs Shared Reading
Implementation uses AtomicInteger

Need to ensure that synchroniser protects access to shared resources

Threads can have a different view on shared state

Verification support for atomics

```java
public class AtomicInteger {
    private volatile int value;

    public AtomicInteger(int v);
    public int get();
    public void set(int v);
    public boolean compareAndSet(int x, int n);
}
```

Atoms rule:
```
emp |- {P * I} C {Q * I}
I |- {P} atomic(C) {Q}
```
Identification of typical synchronisation patterns (using combinations of get, set and compareAndSet)

Specification of AtomicInteger class

- Value of the atomic variable (synchroniser state)
- Views of the participating threads on the atomic variable

Client specifies synchronisation protocol

- Roles of the thread
- Resource invariant: protected shared memory location

Verification of various exclusive access synchroniser implementations
CONTRIBUTIONS

- Extension to shared-reading synchronisers
- Adaptation of AtomicInteger specification
  - Support both partial and exclusive access to shared state
- Verification of several synchroniser implementations
  - Semaphore
  - CountDown Latch
- Verification is tool-supported
  - VerCors tool set: http://www.utwente.nl/vercors
public class Semaphore{
    private AtomicInteger sync;
    Semaphore(int n){
        sync = new AtomicInteger(n);
    }

    public void acquire() { ... }
    public void release() { ... }
}
public void acquire(){
    boolean stop = false;
    int c = 0;
    while(!stop) {
        c = sync.get(); // how many parts are still left
        if( c > 0 ){ // any part left?
            int nextc = c-1;
            stop = sync.compareAndSet(c,nextc); //try to obtain 1 part
        }
    }
}

public void release(){
    boolean stop = false;
    while(!stop) {
        int c = sync.get(); // how much space is left
        int nextc = c+1;
        stop = sync.compareAndSet(c,nextc); // try to increase this by 1
    }
}
Groups of threads involved in the synchronisation can be abstracted by their **behavioural role**

If threads with identical role share a resource, they have to participate in a **compare-and-set competition** (for acquire and release)

If threads have different roles access can be controlled by **collaboration** between get and set

Internal volatile counter determines **remaining portion** of shared state

- **CompareAndSet**: obtain resources by competition
- **Get**: obtain resources as prescribed by the protocol
- **Set**: release resources
/*@
  given Set<role> rs;  // abstract thread roles
  given group (frac->group) inv;  // protected shared resource
  given (role, int->frac) share;  // relation counter and resource fraction
  given (role, int, int-> boolean) trans;  // valid transitions
@*/
class AtomicInteger {
  private volatile int value;
 /*@  group handle(role r, int d, frac p); @*/  // token from last access

 /*@  requires inv(share(S,v));  // resources to be protected  
      ensures (\forall* r in rs: handle(r,v,1));  // handle for all threads
  @*/
  AtomicInteger(int v);
  ...
}
ATOMIC INTEGER SPECIFICATION: GET AND SET

/**@  given role r, int d, frac p;
    requires handle(r,d,p) ** inv(share(r,d));
    ensures handle(r,result,p) ** inv(share(r,result));
@*/
public int get();

/**@  given role r, int d, frac p;
    requires handle(r,d,p) ** trans(r,d,v);
    requires inv(share(S,v)) ** inv(share(r,d));
    ensures handle(r,v,p);@*/
public void set(int v);
/*@
given role r, int m, frac p;
requires handle(r,x,p) ** trans(r,x,n)
requires inv(share(S,n)-share(S,x));
ensures \result===> (handle(r,n,p) ** inv(share(S,x) - share(S,n));
ensures !\result===> (handle(r,x,p) ** inv(share(S,n) - share(S,x));
@*/

boolean compareAndSet(int x, int n);
VERIFICATION OF SEMAPHORE IMPLEMENTATION
CLASS LEVEL SPECIFICATIONS

/*@ given group (frac -> resource) rinv; @*/
public class Semaphore{
 /*@ ghost final int num; 
  ghost Set<role> roles = {T}; 
  group initialized(int d,frac p) = sync.handle(T,d,p); 
  resource held(int d,frac p) = initialized(d,p); 
  group inv(frac p) = rinv(p); 
  frac share(role r, int c){
    return (r==S && c>=0 && c<num)?(c/num):0; } 
  boolean trans(role r, int c, int n){
    return (r==T && c>0 && n==c-1) ||
      (r==T && c<max && n==c+1); } @*/
private AtomicInteger /*@ <roles,inv,share,trans> @*/ sync;

UNIVERSITY OF TWENTE.
CONSTRUCTOR

/*@ requires rinv(1) ** n>0;
  ensures initialized(n,1) ** num == n;
@*/
Semaphore(int n){
   /*@ set num = n; fold sync.inv(share(n)); @*/
    sync=new AtomicInteger /*@<roles,inv,share,trans>@*/ (n);
   /*@ fold initialized(n,1); @*/
}
IF YOU READ THE PAPER

- More details about how the specification for AtomicInteger is derived
- Full verification (also acquire and release of Semaphore)
- CountDown Latch implementation with online verification
CONTRIBUTIONS

- Extension to shared-reading synchronisers
- Adaptation of AtomicInteger specification
  - Support both partial and exclusive access to shared state
- Verification of several synchroniser implementations
  - Semaphore
  - CountDown Latch
- Verification is tool-supported
  - VerCors tool set: http://www.utwente.nl/vercors