The Semantics of Concurrent Programming, 2

K. V. S. Prasad
Dept of Computer Science
Chalmers University
February – March 2013

Background

- We covered imperative and functional programming, (which is Erlang?), referential transparency, why it is hard to reason about imperative programs, and how logic applies to program execution (x=5 but when?).

Invariants

- Do you know what they are?
  - Help to prove loops correct
  - Game example
  - Insertion sort
- Semaphore invariants
  - k >= 0
  - k = k.init + #signals - #waits
  - Proof by induction
    - Initially true
    - The only changes are by signals and waits

Propositional logic

- Assignment – atomic props mapped to T or F
  - Extended to interpretation of formulae (B.1)
- Satisfiable – f is true in some interpretation
- Valid - f is true in all interpretations
- Logically equal
  - same value for all interpretations
  - P -> q is equivalent to (not p) or q
- Material implcation
  - p -> q is true if p is false

Proof methods

- State diagram
  - Large scale: ”model checking”
  - A logical formula is true of a set of states
- Deductive proofs
  - Including inductive proofs
  - Mixture of English and formulae
    - Like most mathematics

Atomic Propositions (true in a state)

- *wantp* is true in a state
  - iff (boolean) var wantp has value true
- *p4* is true iff the program counter is at p4
    - p4 is the command about to be executed
    - Then pj is false for all j =/= 4
- *turn=2* is true iff integer var turn has value 2
- *not (p4 and q4)* in alg 4.1, slide 4.1
    - Should be true in all states to ensure mutex

Box and Diamond

- A request is eventually granted
  - For all t. req(t) -> exists t'. (t' >= t) and grant(t')
  - New operators indicate time relationship implicitly
    - box (req -> diam grant)
- If "successor state" is reflexive,
  - box A -> A (if it holds indefinitely, it holds now)
  - A -> diam A (if it holds now, it holds eventually)
- If "successor state" is transitive,
  - box A -> box box A
    - if not transitive, A might hold in the next state, but not beyond
  - diam diam A -> diam A

Progress proof for Dekker's algorithm

- From http://fmt.cs.utwente.nl/courses/cdp/

First try (slides 3.3 – 3.11)

- Mutex
  - Full state diagram -> only 16 states reachable (of 32)
  - States (p3,q3,*) not reachable, so mutex.
- Absence of deadlock
  - Abbreviate program to reduce state space
  - someone escapes from each wait state
  - Need progress assumption!
- Starvation, from fragment of full diagram
  - If q1 is stuck in NCS with turn=2, p starves

Mutex for Alg 4.1

- Invariant Inv1: (p3 or p4 or p5) -> wantp
  - Base: p1, so antecedent is false, so Inv1 holds.
  - Step: Process q changes neither wantp nor Inv1.

    Neither p1 nor p3 nor p4 change Inv1.

    p2 makes both p3 and wantp true.

    p5 makes antecedent false, so keeps Inv1.

So by induction, Inv1 is always true.

Mutex for Alg 4.1 (contd.)

- Invariant Inv2: wantp -> (p3 or p4 or p5)
  - Base: wantp is initialised to false , so Inv2 holds.
  - Step: Process q changes neither wantp nor Inv1.
    Neither p1 nor p3 nor p4 change Inv1.
    p2 makes both p3 and wantp true.
    p5 makes antecedent false, so keeps Inv1.

  So by induction, Inv2 is always true.

  Inv2 is the converse of Inv1.

  Combining the two, we have

  Inv3: wantp <-> (p3 or p4 or p5) and
  wantq <-> (q3 or q4 or q5)

Mutex for Alg 4.1 (concluded)

- Invariant Inv4: not (p4 and q4)
  - Base: p4 and q4 is false at the start.
  - Step: Only p3 or q3 can change Inv4.

    p3 is ”await (not wantq)”. But at q4, wantq is true by Inv3, so p3 cannot execute at q4.

    Similarly for q3.

So we have mutex for Alg 4.1

Second try: alg 3.6, slide 3.12

- Error in first try
  - p and q both set and test ”turn”
  - if one dies the other is stuck
- So second try uses independent flags
- Wantp => p4 (CS) or p5
- Not wantp => p1 (NCS) or p2 (await) or p3
- Sadly, no mutex
  - try running p and q in lockstep
- Can we see this in the states? Abbreviate first!

Second try (slides 3.13 – 3.15)

- Abbreviate again to only the protocols
  - Now p1 and p2 are NCS, and p3 is CS
- See path from start to (p3, q3, true, true)
  - Or see scenario
- So mutex fails: lock set too late on entering CS
- So let's try preemptively setting the lock

Third try: alg 3.8, slide 3.16

- Flip p2 and p3 of second try; book your place before trying to enter CS
- Exercise: abbreviate, do state diagram

Abbreviated third try

Loop

p1. p := T

p2. await not q

p3. p := F

Loop

q1. q := T

q2. await not p

q3. q := F

One path through states for try 3

- See state diag in slide 3.18
- See scenario in slide 3.17
  - Deadlock by definition
    - (both want CS, neither gets it)
  - Actually, worth calling it ”livelock”
    - If await is a busy wait
- Maybe p should declare intention but not insist on entering CS
  - Instead, try. If fail, release CS.

Fourth try: alg 3.9, slide 3.19

- Again, lockstep gets p and q into trouble
  - Mutex is fine (show by state diagram)
  - No deadlock : p or q *can* enter CS
  - But they can starve in parallel
  - Loop in state diagram (slide 3.20) shows we cannot say ”it *must* eventually succeed”.
- Just when it is beginning to look like a bad joke …

Dekker's alg (3.10, slide 3.21)

- Modify try 4 by adding the turn from try 1
  - To arbitrate away from the parallel starvation
- Prove correctness by state diagram
  - Deductive proof in Sec 4.5
    - Using temporal logic

Rethink

- P checks wantq
  - Finds it false, enters CS,
    - but q enters before p can set wantp
- Could we prevent that?
  - When I find the book free, I take it
    - Before anyone else even sees it free
- Test-and-set(common, local) = atomic{local:=common; common:=1}
  - Now see Ben-Ari p76, slide 3.22, alg 3.11
  - See Wikipedia article, also Herlihy 1991

Exchange and other atomics

- Slides 3.22 and 3.23
- Other atomic instructions
  - Compare and swap
  - Fetch-and-add
- All use busy waits
  - OK in multiprocessors
    - Particularly if low contention

Semaphore definition

- Is a pair < value, set of blocked processes>
  - Doesn't make sense until you have a software process with a blocked state (others being ready, running, terminated)
- Initialised to <k, empty>
  - k depends on application
    - For a binary semaphore, k=1 or 0, and k=1 at first
- Two operations. When proc p calls sem S
  - Wait (S) =
    - if k>0 then k:=k-1 else block p and add it to set
  - signal (S)
    - If empty set then k:=k+1 else take a q from set and unblock it
- Signal undefined on a binary sem when k=1

Processes revisited

- We didn't really say what "waiting" was
  - Define it as "blocked for resource"
    - If run will only busy-wait
  - If not blocked, it is "ready"
    - Whether actually running depends on scheduler
  - Running -> blocked transition done by process
  - Blocked -> ready transition due to external event
- Now see B-A slide 6.1
- Define "await" as a non-blocking check of boolean condition

Critical Section with semaphore

- See alg 6.1 and 6.2 (slides 6.2 through 6.4)
- Semaphore is like alg 3.6
  - The second attempt at CS without special ops
  - There, the problem was
    - P checks wantq
      - Finds it false, enters CS,
      - but q enters before p can set wantp
- We can prevent that by compare-and-swap
- Semaphores are high level versions of this