Lecture 9: Critical Sections revisited, and Reasoning about Programs

K. V. S. Prasad
Dept of Computer Science
Chalmers University
Monday 23 Feb 2015

Plan for today

Chap 2, 3 recap and complete

Chap 4 intro to logic

REMINDER: Class rep meeting later today.

Schedule: booked extra slots on 9 and 11 March

Request: exercises 8.1 and 8.2
  dining-sole-channel-pml-pseudo.txt
  sort-pml-pseudo.txt, also slide 8.18

Recap – state diagrams

- (Discrete) computation = states + transitions
  - Both sequential and concurrent
    - Can two frogs move at the same time?
  - We use labelled or unlabelled transitions
    - According to what we are modelling
    - Chess games are recorded by transitions alone (moves)
      - States used occasionally for illustration or as checks
  - In message passing, the (labelled) transitions
    - Are what we see, from the outside, of a (sub)system
    - So they matter more than the states

How to program multiple processes

- Concurrent vs. sequential
  - Concurrent has more states due to interleaving
- But a concurrent sort program should sort
  - No matter which interleaving
  - So cut out unwanted interleavings
    - through synchronisation (waits)

What is interleaved? Atomic statements

- The thing that happens without interruption
  - Can be implemented as high priority
- We must say what the atomic statements are
  - In the book, assignments and boolean conditions
  - How to implement these as atomic?

Correctness - safety

- A safety property must always hold
  - In every state of every computation
- = “nothing bad ever happens”
  - Typically, partial correctness
    - Program is correct if it terminates
    - E.g., “loop until head, toss”
      - sure to produce a toss if it terminates
      - But not sure it will terminate
        - » Will do so with increasing probability the longer we go on
    - How about “loop until sorted, shuffle deck”?
      - Sure to produce sorted deck if it terminates
      - Needs much longer expected run to terminate
      - Can guarantee neither progress nor termination

Correctness - Liveness

- A liveness property must eventually hold
  - Every computation has a state where it holds
- = a good thing happens eventually
  - Termination
  - Progress = get from one step to the next
  - Non-starvation of individual process
- Sort by shuffle is safe but cannot guarantee liveness - either progress or termination

Safety and Liveness are duals

- Dualism in classical logic
  - not (P and Q) = (not P) or (not Q)
  - not (P or Q) = (not P) and (not Q)
- Let *P* be a safety property
  - Then *not P* is a liveness property
- Let *P* be a liveness property
  - Then *not P* is a safety property
- Safety typically proved via invariants (assertions)
- Absence of liveness typically proved by finding loop of states none of which make the progress

(Weak) Fairness assumption

- If at any state in the scenario, a statement is continuously enabled, that statement will eventually appear in the scenario.
- So an unfair version of coin tossing cannot guarantee we will eventually see a head.
- We usually assume fairness

What is the critical section problem?

- Specification
  - Both p and q cannot be in their CS at once (mutex)
  - If p and q both wish to enter their CS, one must succeed eventually (no deadlock)
  - If p tries to enter its CS, it will succeed eventually (no starvation)
- GIVEN THAT
  - A process in its CS will leave eventually (progress)
  - Progress in non-CS optional

Different kinds of requirement

- Safety:
  - Nothing bad ever happens on any path
  - Example: mutex
    - In no state are p and q in CS at the same time
    - If state diagram is being generated incrementally, we see more clearly that this says ”in every path, mutex”
- Liveness
  - A good thing happens eventually on every path
  - Example: no starvation
    - If p tries to enter its CS, it will succeed eventually
  - Often bound up with fairness
    - We can see a path that starves, but see it is unfair

Deadlock?

- With higher level of process
  - Processes can have a blocked state
  - If all processes are blocked, deadlock
  - So require: no path leads to such a state
- With independent machines (always running)
  - Can have livelock
    - Everyone runs but no one can enter critical section
  - So require: no path leads to such a situation

Language, logic and machines

- Evolution
  - Language fits life – why?
  - What is language?
- What is logic?
  - Special language
- What are machines?
  - Why does logic work with them?
- What kind of logic?

Logic Review

- How to check that our programs are correct?
  - Testing
    - Can show the presence of errors, but never absence
      - Unless we test every path, usually impractical
  - How do you show math theorems?
    - For *every* triangle, … (wow!)
    - For *every* run
      - Nothing bad ever happens (safety)
      - Something good eventually happens (liveness)

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 implication
  - p -> q is true if p is false
  - Don't be hung up on names
    - Just as "work" in physics is not "work" in real life
    - So "imply" here is just the name of a function of p and q

Liveness via Progress

- Invariants can prove safety properties
  - Something good is always true
  - Something bad   is always false
- But invariants cannot state liveness
  - Something good happens eventually
- Progress A to B
  - if we are in state A, we will progress to state B.
- Weak fairness assumed
  - to rule out trivial starvation because process never scheduled.
  - A scenario is weakly fair if
    - B is continually enabled at state Ain scenario ->
      B will eventually appear in the scenario

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
- See Wikipedia page on LTL

Formalising Fairness

- Absolute Fairness: every process should be executed infinitely often:
  - **for all** i: GF ex_i
  - But a process might not be enabled.
- Strong Fairness: a process that is infinitely often enabled executes infinitely often when enabled:
  - **for all** i : (GF en_i) **) =>** (GF(en_i **^**ex_i)) :
- Weak Fairness: a process that is ultimately always enabled should execute infinitely often:
  - **for all** i : (FG en_i) **) =>** (GF ex_i)

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
  - But can be formalised
    - Theorem provers
    - Proof checkers

Invariants recap

- Help to prove loops correct
  - Game example with straight and wavy lines
- Semaphore invariants
  - k >= 0
  - k = k.init + #signals - #waits
  - Proof by induction
    - Initially true
    - The only changes are by signals and waits

CS correctness via sem invariant

- Let #CS be the number of procs in their CS's.
  - Then #CS + k = 1
    - True at start
    - Wait decrements k and increments #CS; only one wait possible before a signal intervenes
    - Signal
      - Either decrements #CS and increments k
      - Or leaves both unchanged
  - Since k>=0, #CS <= 1. So mutex.
  - If a proc is waiting, k=0. Then #CS=1, so no deadlock.
  - No starvation – see book, page 113

CS correctness (contd.)

- No starvation (if just two processes, p and q)
  - If p is starved, it is indefinitely blocked
  - So k = 0 and p is on the sem queue, and #CS=1
  - So q is in its CS, and p is the only blocked process
  - By progress assumption, q must exit CS
  - Q will signal, which immediately unblocks p
- Why “immediately”?

Why two proofs?

- The state diagram proof
  - Looks at each state
  - Will not extend to large systems
    - Except with machine aid (model checker)
- The invariant proof
  - In effect deals with sets of states
    - E.g., all states with one proc is CS satisfy #CS=1
  - Better for human proofs of larger systems
  - Foretaste of the logical proofs we will see (Ch. 4)

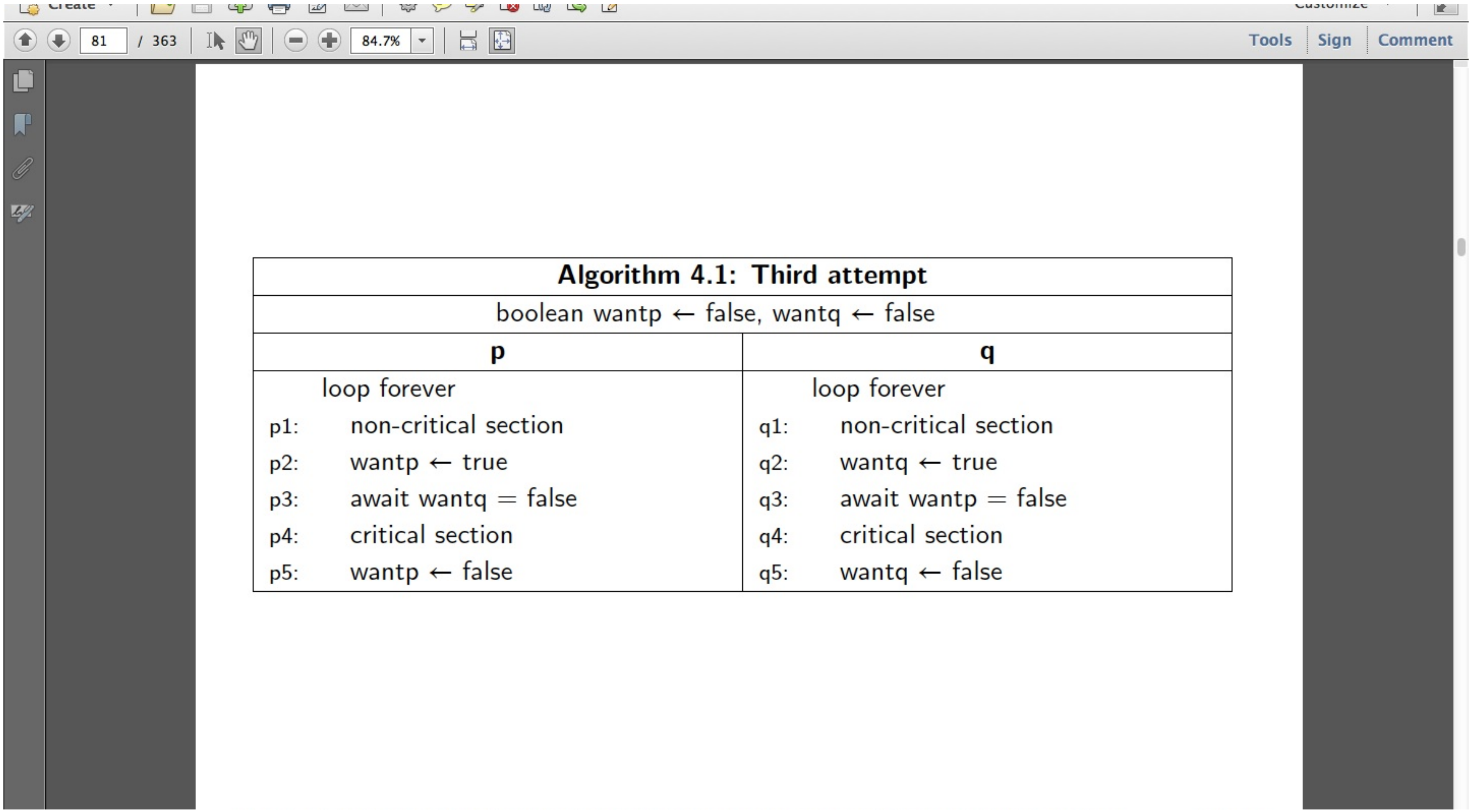
Infinite buffer is correct

- Invariant
  - #sem = #buffer
    - 0 initially
    - Incremented by append-signal
      - Need more detail if this is not atomic
    - Decremented by wait-take
- So cons cannot take from empty buffer
- Only cons waits – so no deadlock or starvation, since prod will always signal

Bounded buffer

- See alg 6.8 (p 119, s 6.12)
  - Two semaphores
    - Cons waits if buffer empty
    - Prod waits if buffer full
  - Each proc needs the other to release ”its” sem
    - Different from CS problem
  - ”Split semaphores”
  - Invariant
    - notEmpty + notFull = initially empty places

Algorithm 4.1 = Third CS attempt



Algorithm 4.1: Third attempt	
boolean wantp ← false, wantq ← false	
p	**q**
loop forever	loop forever
p1: non-critical section	q1: non-critical section
p2: wantp ← true	q2: wantq ← true
p3: await wantq = false	q3: await wantp = false
p4: critical section	q4: critical section
p5: wantp ← false	q5: wantq ← false

Algorithm 3.1

- We can prove mutex by checking all the states
- How to prove absence of deadlock?
  - Do we have to look at all scenarios?
    - Would be even worse; all paths through the state diagram
  - Fortunately, only paths from states (p2,q2,i)
  - Then we can argue from the text of the program
    - Don't need state diagram
- Starvation?
  - Sadly, a loop in the state diagram can starve a process

Algorithms 3.2 – 3.4

- 3.2 : No mutex
  - easy to see scenario that leads to this
  - just do both processes in step
- 3.3 : Deadlocks
  - Again, by doing both processes in step
- 3.4 : Both can starve
  - Again, by doing both processes in step

Dekker's algorithm (3.10)

- Each process in turn has the right to insist on entering its critical section
- Perhaps surprisingly, this does the trick!
- Can prove by state diagram
  – but will do by logic
- Dekker, Peterson, etc.
  – These algorithms are now only nice examples
- Actual mutex etc. achieved by hardware
  – instructions such as test-and-set, compare-and-swap.

Complex atomic hardware instructions

- Show correctness of 3.11 and 3.12
- Test-and-set(common, local) is

```
local := common
common := 1
```

Exchange(a, b) is

```
local temp
temp := a
a := b
b := temp
```

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

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

4.1 deadlocks

- Prove (p1 and q1) => <> [] (p3 and q3)
- p1 => <> p2   (similarly for q)
- p2 => <> p3   (similarly for q)
- So (p1 and q1 and not wp and not wq)
=> <> (p2 and q1 and not wp and not wq)
=> <> (p2 and q2 and not wp and not wq) …
=> <> (p3 and q3 and wp and wq)
=> <> [] (p3 and q3 and wp and wq)
=> <> [] (p3 and q3)

In 4.1, [] p3 can result no matter where q is

- Prove (p3 and q4) => <> p4
  - Note: cannot prove p3 => <> p4
    - which we might like
    - but it’s not true!
    - because of the deadlock: p3 and q3 => [] (p3 and q3)
- q4 => <> q5 => <> q1
- (p3 and q4) => <> (p3 and q5)

  => <> (p3 and q1 and not wq) …

  => <> (p4 and q1) or (p3 and q3)

Proof of Dekker's Algorithm (outline)

- Invariant Inv2: (turn = 1) or (turn = 2)
- Invariant Inv3: wantp <-> p3..5 or p8..10
- Invariant Inv4: wantq <-> q3..5 or q8..10
- Mutex follows as for Algorithm 4.1
- Will show neither p nor q starves
  - Effectively shows absence of livelock

Proof of Dekker’s Algorithm (outline)

- Invariant Inv2: (turn = 1) or (turn = 2)
- Invariant Inv3: wantp <-> p3..5 or p8..10
- Invariant Inv4: wantq <-> q3..5 or q8..10
- Mutex follows as for Algorithm 4.1
  - NB: ”turn” alone won’t prove it.
- Will show neither p nor q starves
  - Effectively shows absence of livelock

Progress in (non-)critical section

- The following are notes re Ben-Ari's proof, but I prefer the liveness proof in the Utwente notes.
- Progress in critical section
  - box (p8 -> diam p9)
  - It is always true that if we are at p8, we will eventually progress to p9
- Non-progress in non-critical section
  - diam (box p1)
  - It is possible that we will stay at p1 indefinitely

Progress through control statements

- For ”p1: if A then s” to progress to s, need
  - p1 and box A
  - p1 and A   is not enough
    - does not guarantee A holds by the time p1 is scheduled
- So in Dekker’s algorithm
  - p4 and box (turn = 2) -> diam p5
  - But turn = 2 is not true forever!
    - It doesn’t have to be. Only as long as p4.

Lemma 4.11

- box wantp and box (turn = 1) -> diam box (not wantq)
  - If it is p’s turn, and it wants to enter its CS, q will eventually defer
- Note that at q1, wantq is always false
  - Both at init and on looping
- q will progress through q2..q5 and wait at q6
  - Inv4: wantq <-> q3..5 or q8..10
    - Implies box (not wantq) at q
- Lemma follows

Progress to CS in Dekker's algorithm

- Suppose p2 and box (turn=2)
  - If p3 and not wantq then diam p8
  - p2 and box (turn=2 and wantq) ->
    diam box p6 <-> diam box (not wantp)
  - p6 and box (turn=2 and not wantp) -> diam q9
  - p2 and box (turn=2) -> diam box (p6 and turn=1)
  - Lemma 4.11 now yields diam p8