Invariants and semaphores
Mon 4 Sep 2017

K. V. S. Prasad
Dept of Computer Science
Chalmers University
Lecture 3 of TDA384/DIT301, August –October 2017

Recap – state diagrams

- (Discrete) computation = states + transitions
  - Both sequential and concurrent
    - Can two frogs move at the same time? (slide 2.38, p 42)
  - 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

State diagrams – a reasoning tool

- Note that states (in diagrams and scenarios) describe variable values before the next command is executed.
- In concurrent programs, the commands are tuples, one for each of the processes
- Not all thinkable states are reachable from the start state

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)
- Concurrency brings interleaving, which has to be trimmed.
  - This is the downside; what was the upside, again?
  - Faithful modelling, or speed by parallel processing.

On system design

- What do you want the system to do?
  - How do you say this? In what language?
    - Logic (requirement)
    - or another program (equivalence).
- What does the system do?
  - How do you say this?
    - Operational semantics
      - (walk through a state diagram).

More on system design

- Build the right system (validation)
  - Does the formal (math) spec capture user wants?
  - Is it consistent? Complete?
  - These can be checked before you build anything
- Build the system right (verification)
  - Does it do what you want it to?
    - According to the spec.

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

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 head 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

Specification of the critical section problem

- REQUIRE
  - At most one process can be in its CS at any time (mutex)
  - If more than one process wishes to enter their CS, one must succeed eventually (no deadlock)
  - Any process trying to enter its CS will succeed eventually (no starvation)
- GIVEN THAT
  - A process in its CS will leave eventually (progress)
  - Progress in non-CS optional

Pet examples (mostly of CS)

- Passing a door from opposite directions
  - If both sleep until the other passes – deadlock
  - If both eager – livelock (busy waiting)
- Library
- The knife (atomic; deadlock if fork+knife picked up in either order)
- The printer (grab then print file, or atomic per sheet?)
- Count up to 20
- Max, sort by chemical machine
- Max and grabbing by broadcast

Earlier attempt at the bank problem

flag := free

- ATM code
- loop

```
a1: loop until flag=free;
a2: flag := busy;
a3: temp := bal; temp--; bal:=temp;
a4: flag := free
```

- ATM code
- loop

```
b1: loop until flag=free;
b2: flag := busy;
b3: temp := bal; temp--; bal:=temp;
b4: flag := free
```

Consider the scenario a1, b1, a2, b2, … We get interference.

The solution is that a1 and a2 must happen in one step. Atomic action to prevent unwanted interleaving. Can you solve this with semaphores?

Invariants

- Help to prove loops correct
  - Game example with straight and wavy lines
- Example: insertion sort
  - Invariant: the array so far is sorted
    - Empty array to start
    - Every step preserves sort
  - To complete, we show termination: when input over.
- Try bubble sort on your own
- Try linear program to extract max of a set

A hardware example – the swap instruction

- Ben-Ari slide 3.23
- Try to show this is correct
- Beautiful example of an invariant – there is only one green token.
- But there is a busy wait for the green token.

Semaphore ops

- Signal (S)
  - If S. L = {} then S.V := S.V+1
    else S.L:= S.L-{q}; //for some q in S.L
    q.state := ready
- Wait(S)
  - If S.V > 0 then S.V := S.V-1
    else S.L:= S.L U {p};
    p.state := blocked

Semaphore invariants

- S.V >= 0
- S.V = S.V.init + #signals - #waits
- Proof by induction
  - Initially true
  - Only changes by signals and waits

Mergesort using semaphores

- See p 115, alg 6.5 (s 6.8)
  - The two halves can be sorted independently
    - No need to synch
  - Merge, the third process,
    - has to wait for both halves
  - Note semaphores initialised to 0
    - Signal precedes wait
    - Done by process that did not do a wait
  - Not a CS problem, but a synchronisation one

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

CS by semaphore

- Slides 6.2 through 6.7
- Why 5 states in slide 6.4?
- Mutex means there is no state with p2&q2
- Deadlock would be p1 & q1 & S.V=0

CS correctness via sem invariant

- Let #CS be the number of procs in their CS's.
  - Then #CS + k = 1 is an invariant. (writing k for S.V)
    - 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 next slide

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"?
  - The sem. op. is taken to be atomic

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

CS correctness via sem invariant for N≥2 processes

- 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 for N=2
    - But possible for N>2. P blocks, while Q and R alternate.

CS problem for n processes

- See alg 6.3 (p 113, s 6.5)
  - The same algorithm works for n procs
  - The proofs for mutex and deadlock freedom work
    - We never used special properties of binary sems
  - But starvation is now more likely
    - p and q can release each other and leave r blocked
- Exercise: If k is set to m initially, at most m processes can be in their CS's.

Dining Philosophers

- Obvious solution deadlocks (alg 6.10)
- Break by limiting 4 phils at table (6.11)
- Or by asymmetry (6.12)

Dining philosophers with semaphores

- Slide 6.14 to 6.18 (p. 124 to 128)
- Requirements
  - Can only eat with lhs and rhs fork
  - Mutex over each fork
  - Deadlock-free
  - Starvation-free
  - (efficient if no contention)