Lecture 9: Critical Sections revisited and Reasoning about Programs

K. V. S. Prasad
Dept of Computer Science
Chalmers University
Monday 1 Oct 2012

Questions?

- Have you looked at the detailed syllabus?
- Have you looked at the questionnaire?
  - Answer it if you haven't
    - at least the parts you think will help you
    - Or free form notes, related to the questionnaire or not
- Start posting questions on the Google group
- Today's "and finally"
  - Up to you, but prefer urgent issues re course

Plan for today

Chap 3 recap and complete

Chap 4 intro to logic

REMINDER: Read the Appendices in the book!

REMINDER: exercises in Chaps. 1, 2, 3, 6, 7, 8, 9

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
- Concurrent or sequential
  - Concurrent just has more states due to interleaving
  - But sort program sorts no matter which interleaving,

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?

What is the 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 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

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

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

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

Progress in (non-)critical section

- 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