CMSC 712
Distributed Algorithms and Verification
(Writing correct distributed programs)

Shankar

May 24, 2014

Outline

Objective

- Rigorous and practical method to write correct distributed algorithms and programs
- Correctness: Can prove that algorithm/program satisfies desired properties for *all* possible variations of thread speeds
- Compositionality: Define the external behavior, or service, of a program A such that the service suffices for writing programs that use A.
- Accessible to programmers
- Apply the method to various distributed systems problems
  - locks, producer-consumer, termination detection, shared memory, network protocols, database concurrency control, $\cdots$

Algorithm vs Program?

- Early CS: there was no difference between them
  - Knuth (MIX), sorting (Algol, Pascal), locks (asm, C), $\cdots$
- Later on: algorithms were written in pseudo-code, accurately
  - could obtain program without understanding the algorithm
- Currently, dist algs are written in pseudo-code, but vaguely
  - obtaining program requires understanding algorithm
  - making design decisions about the algorithm
- We will bridge the gap between distributed algorithms and distributed programs
  - algorithms are programs written in high-level syntax

Approach

- Programs written in high-level syntax (e.g., Java/Python)
  - threads and inter-process communication are explicit
- Services are written as programs with special structure
  - employ non-determinism and powerful atomicity
  - service programs meant to be easily understood, not efficient
- Define "program implements service" such that in any program C that uses B, replacing B by any program that implements B preserves correctness properties of C.
- Proving "A implements B" reduces to proving properties of $A|B$
  - use assertional reasoning: old-fashioned, but works
  - requires mental effort (just like programming)
  - automated tools (perhaps) useful for low-level steps of a proof

Outline

Programs and Systems

- Program: instructions organized in main code and functions
- Threads: active agents that execute instructions
- Instructions, in addition to the usual, can
  - create threads to execute functions
  - instantiate programs
- System: instantiation of a program // e.g., process in OS
  - instantiating thread executes main code and returns
  - system exists until it is explicitly terminated
- Systems interact via function calls and returns
- Thread in system A calls a function of system B
  - call is an output of A, an input of B
  - return is an input of A, an output of B

External Behavior of Programs

- External behaviour of program P is given by the set of all its possible input-output sequences, say $io(P)$
  - infinite: due to parameters, non-terminating threads
  - non-deterministic: many sequences for same input subsequence

- $io(P)$ is sufficient for using P in other programs
- But not helpful: P is likely the only way to express $io(P)$

- We really want the set of *acceptable* io sequences, say $xio(P)$
  - $io(P) \neq xio(P)$ due to efficiency issues or errors in P

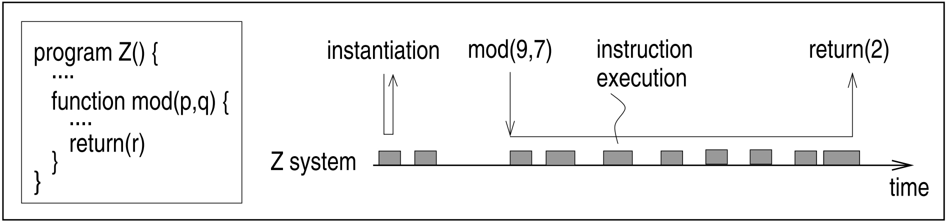
- We want a convenient expression, say Q, that generates $xio(P)$
  - Q would be the service of P
  - use Q, instead of P, when writing programs
  - replace Q by P when executing those programs

Approach

- Services expressed by service programs
  - programs with a special structure, non-determinism, powerful atomicity
- Define "P implements Q" to achieve compositionality
  - candidate: $io(P) \subseteq xio(P)$ // doesn't quite work
- Develop a program-level version of "P implements Q"
  - reduces to proving properties of a program $P|Q$
- Assertional reasoning to prove properties of programs

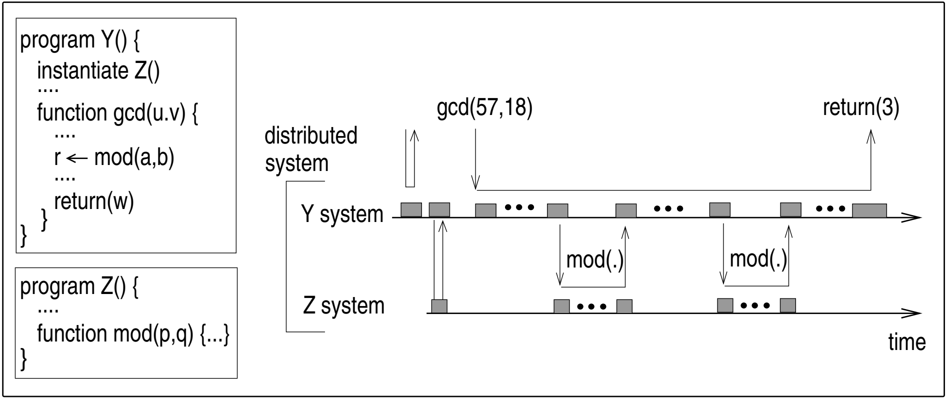
Outline

Program Z



- Program Z
  - inputs: instantiation call, `mod(p,q)` call
  - outputs: instantiation return, `mod(p,q)` return
- $xio(Z)$: sequences of call-return pairs s.t.
  - return of `mod(p,q)` call has value equal to remainder(p/q)
  - every call is eventually followed by a return
- $xio(Z)$ places constraints on Z and its environment

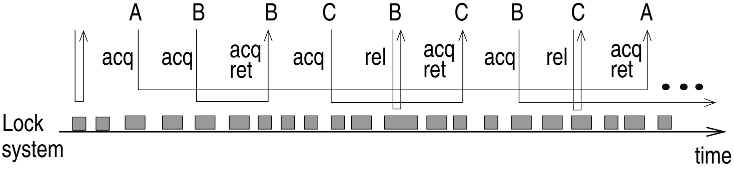
Program Y



- Evolution is Y-evolution and Z-evolution “stitched together”
- Distributed system consists of one or more “basic” systems
- $xio(Y)$: similar in structure to $xio(Z)$

Program *Lock* – 1

```
program Lock() {
  ....
  function acq() {
    ....
    return
  }
  function rel() {
    ....
    return
  }
}
```



- Lock system can have more than 1 thread active simultaneously
- inputs: instantiation call, acq call, `rel` call
- outputs: instantiation return, acq return, `rel` return

xio(*Lock*): io sequences such that

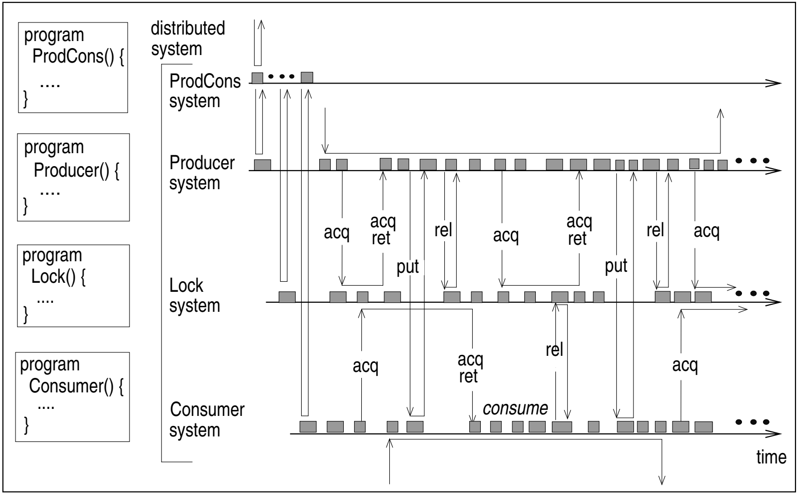
1. for each user, interactions cycle through
   acq call, acq return, `rel` call, and `rel` return
   // request only if not holding lock, release only if holding lock

2. between every two acq returns, there is a `rel` call.
   // at most one user has the lock at any time.

3. every `rel` call is followed eventually by its return.
   // not necessarily immediately

4. if every acq return is followed eventually by a `rel` call,
   then every acq call is followed eventually by its return.
   // if no user holds the lock indefinitely,
   // then every acquire request is satisfied

Program *ProdCons*



- *ProdCons* starts three systems: producer, consumer, lock
- Producer and consumer use lock to synchronize

Outline

Program Model

- Program is modeled as a state machine
- State
  - <values of variables, locations of threads> of all its systems
- Transition
  - an atomic step, i.e., atomic execution of a code chunk
  - may involve interaction with environment (or inside program)
  - changes the state
- Evolution: path in state machine
  - $<$*initial state*, [*transition*, *next state*]^n$>$, $n = 0, \cdots, \infty$
  - evolution can be finite or infinite
- Undefined transition: evolution ends in a fault state

Input Assumptions of a Program

- Program invariably has input assumptions, e.g.,
  - input is an integer or a prime number
  - lock release called only if lock is not free
  - at most one call ongoing at any time
- An input occurence is allowed in an evolution if its input assumption holds at that point
- Allowed evolution: one where all input occurrences are allowed
- Typically interested only in the allowed evolutions of the program

Correctness Properties

- Correctness property:
  - true/false statement about an evolution
  - holds for a program iff holds for every allowed evolution
- Correctness properties are of two kinds: safety and progress
- Safety: nothing bad happens
  - two users cannot hold lock at the same time
- Progress (aka liveness): something good happens
  - every release call eventually returns
  - every acquire call eventually returns if
    every request return is eventually followed by release call
- We use assertions to express correctness properties

Assertions

- Predicates: formulas in variables and threads
  - true or false for each state
  - `x = 2 or (thread t at a1)` $\Rightarrow$ `(y prefixOf z)`
- Assertions: formulas in predicates and "temporal" operators
  - true or false for each evolution
- *Inv* P, for predicate P // safety assertion
  - holds for an evolution if every state of the evolution satisfies P
- P *leads-to* Q, for predicates P, Q // progress assertion
  - holds for an evolution if for every state that satisfies P, that state or a later state satisfies Q
- Fault state does not satisfy any predicate (not even *false*)
- Program satisfies assertion if every allowed evolution satisfies it

Proving an Assertion about a Program

- Assertional reasoning
  - generate a sequence of intermediate assertions $A_1, \cdots, A_n$ leading to desired assertion
  - prove that each A_j follows from program and previous A_i's
- Proof of A_j can be operational or assertional
- Operational proof of A_j
  - natural to humans: if u does this then v did that and so $\cdots$
  - can give insight but is error-prone (implicit assumptions)
  - checkable only by humans
- Assertional proof of A_j
  - apply a proof rule to program and previous assertions
  - checkable without understanding the program or the assertions
  - mechanically checkable by theorem provers (but arduous)

Outline

Program U


```
program U(int N) {
 // input assumption:
 // at most one thread
 // in program

 x ← na ← nr ← 0;

 function add() {
   if (x < N)
      x ← x+1;
      na ← na+1;
}

 function rmv() {
   if (x > 0)
      x ← x−1;
      nr ← nr+1;
}}
```

- Atomic code chunks
  - `main` // memory isolation
  - `add()` // input assumption
  - `rmv()` // input assumption
- Desired properties
  - Inv nr $\leq$ na
  - `forall(k:`

    na $=$ k

    leads-to nr $=$ k)

Operational proof of Inv nr $\leq$ na

- Proof
  - nr incremented only if x can be decremented, which is only if x has been incremented, which is only if na has been incremented.
  - so nr incremented only if na has incremented.
  - so Inv nr $\leq$ na holds.
- Implicit assumption
  - above proof does not mention initial values
  - but Inv nr $\leq$ na does not hold if x $>$ 0 initially

Assertional proof of Inv nr $\leq$ na – 1

Rule 1

Program satisfies Inv P if

- P holds after the initial atomic step
- every atomic step *unconditionally* preserves P (i.e., establishes P after assuming only P before)

Rule 2

Program satisfies Inv R if

- program satisfies Inv P and Inv Q
- $(P \text{ and } Q) \Rightarrow R$ holds

- Does nr $\leq$ na satisfy rule 1?
  No: not unconditionally preserved by `rmv()`

- Proof
  - key observation: nr+x equals na
  - U satisfies Inv nr+x = na via rule 1
    - x, na and nr are initialized to zero
    - add increases both x and na
    - rmv decreases x and increases na
  - U satisfies Inv x $\geq$ 0 via rule 1 // similarly
  - predicates nr+x = na and x $\geq$ 0 imply nr $\leq$ na
  - hence U satisfies Inv nr $\leq$ na via rule 2
- Coming up with intermediate assertions requires invention
- Checking the proof does not

Outline

Distributed Diffusing Computation

- Systems $a0, \cdots, aN$ connected by fifo channel
- A system can be active or inactive
  - active: send data messages, do computation, become inactive
  - inactive: become active upon receiving a data message
- Initially only system $a0$ is active and no messages in transit
- Activity spreads out from system $a0$
- Any system can switch between active and inactive many times
- Define termination: all systems inactive, no messages in transit
- Computation may never terminate

Distributed Termination Detection

- Augment the diffusing computation to detect termination
- Maintain a distributed dynamic out-tree
  - rooted at system a0
  - includes all active systems
  - each system tracks number of incoming tree edges
  - so system a0 detects termination when it has no incoming edges
- System j responds to every data msg with an "ack" message
  - if the data message causes j to join the tree, j sends the ack only when it next leaves the tree
  - otherwise, j sends the ack immediately

Variables at system j

- active: initially true iff j = a0
- engager: initially a0 if j = a0 otherwise null
  // points to its "down-stream" system if j is in the tree
  // null otherwise
- unAcked: initially 0
  // # of unacked outgoing data messages

Rules at system j (atomically executed)

- only if active = true:
  active ← false;

- only if active:
  send [j,dmsg] to k; unAcked++;

- receive [k,dmsg]:
  active ← true;
  if (engager = null) engager ← k;
  else send ack to k;

- receive ack:
  unAcked −−;

- only if (not active and unAcked = 0 and engager ≠ null):
  if (j = a0) signal termination;
  else send ack to engager; engager ← null;

Analysis Helper Quantities

- termination: `forall(j: not j.active and numDAT(j) = 0)`
- `numDAT(j)`: number of data messages in transit outgoing from `j`
- `numACK(j)`: number of ack messages in transit incoming to `j`
- `eNodes`: `set(j: j.engager` $\neq$ `null)` // set of "engaged" nodes
- `eEdges`: `bag([k.engager, k]: k` $\neq$ `a0, k.engager` $\neq$ `null)`
  // set of "engagement" edges
- `eGraph`: `[eNodes, eEdges]` // "engagement" digraph

Assertions to be proved

Safety

A_1 : Inv ((a0.unAcked = 0 and not a0.active)
⇒ termination)

Progress

A_2 : termination
leads-to (a0.unAcked = 0 and not a0.active)

Proof of A_1

- Intermediate predicates

B_1 : `outTree(eGraph)  and  root(eGraph) = a0`

B_2 : `j.unAcked = numDAT(j) + numACK(j)`
`+ sum([j,k]: [j,k] in eEdges)`

B_3 : `j.engager = [] ⇒ (not j.active  and  j.unAcked = 0)`

1. Inv B_1–B_2 holds // because B_1–B_3 satisfies rule 1; do details
2. B_1–B_3 implies A_1's predicate // do details
3. A_1 holds // from 1, 2 and rule 2

Proof of A_2

A_2 : `termination`
leads-to `(a0.unAcked = 0 and not a0.active)`

- Assume `termination` holds: all inactive, no data msgs in transit
  - need to show that `a0.unAcked` becomes 0
- Assume eEdges is not empty; so there is a leaf node, say `j`.
  - `j` has no outgoing data msgs or incoming edges
  - `j`'s incoming acks are eventually received
  - so `j.unAcked` becomes 0 eventually
  - so `j` sends an ack to its engager and leaves the tree
- Eventually eEdges is empty and `a0.unAcked` is 0

Assertional proof of A_2

Rule 3

Program satisfies P *leads-to* Q if
from any state satisfiying (P and not Q):

- every atomic step that can execute establishes (P or Q)
- there is an atomic step of non-zero speed that establishes Q

If *Inv* R holds, then P can be replaced by (P and R)

1. Define α = [|eEdges|, # acks in transit]   // lexicographic order
   Define H = (terminated and B_1–B_3)
2. (H and α = k > 0) *leads-to* (H and α < k)
   // via rule 3, helper {receive ack, send ack to engager}
3. H *leads-to* (H and $\alpha = 0$)   // induction on 2
4. 3's rhs implies A_2's rhs