Model Checking

Mark Greenstreet

CpSc 418 – Nov. 27, 2012

Lecture Outline

Model Checking & Termination Detection

- Erlang Workshop
- Model Checking
  - Dining Philosophers (again)
  - Model Checking
  - Parallel Model Checking (Stern & Dill)
- Termination Detection
  - Brainstorm
  - Stern & Dill's method
  - Termination detection without timers

Erlang Workshop

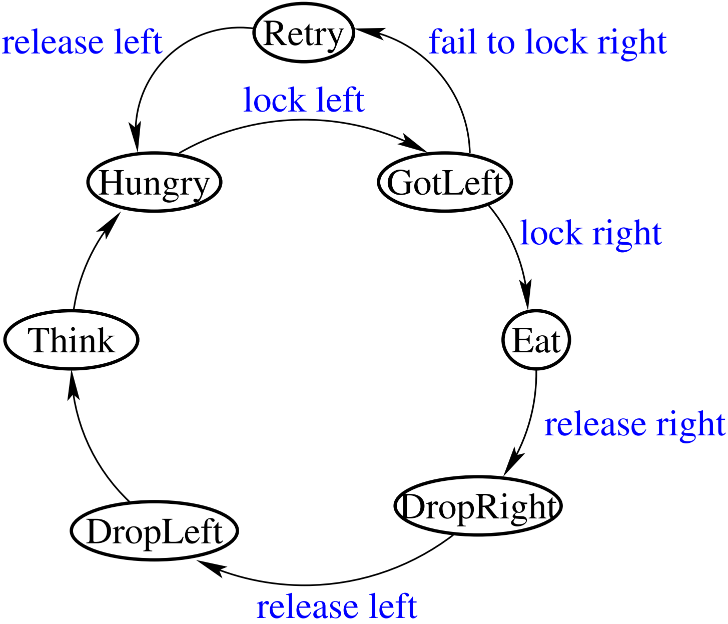
- Worker pools.
- Worker process state
    - `workers:put` and `workers:get`: called by the worker process, accesses that process's state.
    - `workers:update` and `workers:retrieve`: called by the master process, accesses the state of all processes.
- Distributed lists:
    - Creating with `workers:update`.
    - Creating with `workers:seq`.
    - Creating with `workers:rlist`.
- Reduce and scan
- Debugging tips

Dining Philosophers

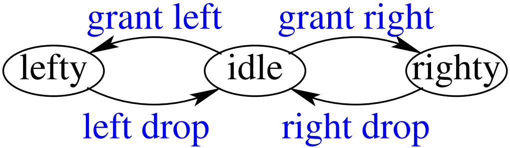
A classic illustration of deadlock and livelock for parallel programs, operating systems, and other concurrent programming problems.

- Setting:
  - Five philosophers sit at a round dinner table.
  - Each philosopher has a plate of spaghetti in front of him/her.
  - There is one fork between each pair of philosophers.
- Eating:
  - To eat, each philosopher must pick up the fork to his/her left and the fork to his/her right (in either order).
  - In other words, the philosopher must acquire two locks.

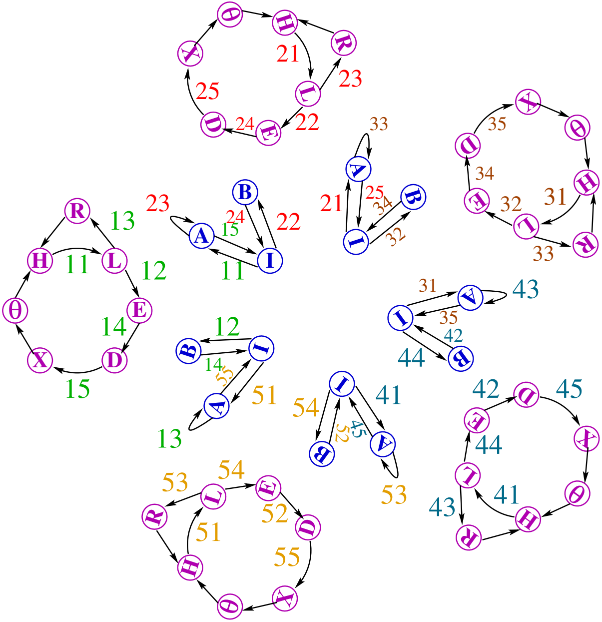
A Philosopher's FSM



A Fork's FSM


grant left
grant right
lefty
idle
righty
left drop
right drop

Combining the FSMs



Finding all reachable states

- Find all reachable states:

```
P := initialStates; // P = pending states
Q := $\emptyset$; // Q = processed states
while(P $\neq \emptyset$) {
    x := any element of P;
    P := P - {x};
    if(x $\notin$ Q) {
            Q := Q $\cup$ {x};
            for each successor, y, of x {
                P := P $\cup$ {y};
            }
        }
    }
```

Model Checking

- Let ϕ be a property (a predicate over states).
- To show that ϕ holds in all reachable states:
  - Compute the set of reachable states.
  - Verify that ϕ holds in each one.
- To find states from which ϕ eventually holds:

```
Q := all reachable states;
P := $\{x \in Q \mid \phi \text{ holds in } x\}$;
repeat {
    $\psi$ := $\{y \in Q - \phi \mid \text{all successors of } y \text{ are in P}\}$;
    P := P $\cup\ \psi$;
} until($\psi = \emptyset$);
```

- For predicates α and β we can show
  - If α holds in some state, β will hold in some future state.
  - If α holds in some state, α will continue to hold (at least) until a state is reached where β holds.
  - …

Model Checking

- Model Checking is awesome!
  - It can automatically check for errors.
  - It is exhaustive: no error left undetected.
  - It can generate counter-examples.
- BUT state-space explosion is a big problem
  - Our dining philosopher's example had:
    - 5 philosophers and 5 forks
    - 6 states per philosopher, 3 per fork.
    - $6^5 * 3^5 \approx 1.9 \mathrm{million}$ states.
    - Even more if we added priority counters, etc., to prevent livelock and starvation.
  - A few million states can be readily explored by a computer.
    - But not a few quadrillion or more.

Avoiding the state space explosion

- Symmetry reductions.
- Symbolic methods.
- Lots of other clever tricks:
  - Model checking is used in industry to find bugs in code and hardware where it's really critical.
  - Often used for HW and/or SW that is intrinsically parallel.
  - But, still an area where experts are needed:
    - If interested, do your grad work with Alan Hu or me ☺.
- Model checking can work for 10^{30} or more states!
- Parallelism:
  - 1000 machines have 1000 times as much memory as one.
  - At a big company (e.g. Intel), there are 1000's of people with computers, and these computers are idle most of the time.
  - Might as well put them to work doing something useful.

Parallel Model Checking

- Stern and Dill's algorithm (w/o termination detection):

```
reach(Q, Nprocs) ->
   Pids = map(fun(I) -> spawn(doCheck, []) end, seq(
   map(fun(Pid) -> Pid ! {allPids, Pids} end, Pids),
   sendStates(Pids, Q).

sendStates(Pids, [Hd | Tl]) ->
   nth(hashToPid(Hd), Pids) ! {state, Hd},
   sendStates(Pids, Tl).

doCheck() -> receive {allPids, Pids} -> doCheck(Pids,

doCheck(Pids, Q) ->
   receive {state, State} ->
      if
         State ∈ Q -> doCheck(Pids, Q);
         true ->
            sendState(Pids, successors(State)),
            doCheck(Pids, Q U {State})
      end
```

Termination Detection

How do we know when we're done?

- Simple idea:
  - If root process is idle for a while
    - It sends messages to all other processes asking if they are idle.
    - If they all reply `idle`, then we conclude that we're done.
- What's wrong with this "solution"?

Stern & Dill's Approach

Make `doCheck` keep track of how many messages sent and received:

```
doCheck() -> receive {allPids, Pids} -> doCheck(Pids, ∅,

doCheck(Pids, Q, Nsent, Nrecv, ToSend) ->
   TimeOut = if (hd(Pids) == self()) -> 1000; true -> in
   receive
      {state, State} ->
         if
            State ∈ Q -> doCheck(Pids, Q, Nsent, Nrecv+1
            true ->
               doCheck(Pids, Q ∪ {State}, Nsent+length(Q
                  sendStates(Pids, successors(State), To
         end;
      doneQuery ->
         hd(Pids) !  (Nsent - Nrecv),
         doCheck(Pids, Q, Nsent, Nrecv, []);
      continue ->
         sendStates(Pids, ToSend, noDelay),
         doCheck(Pids, Q, Nsent + length(ToSend), Nrecv,
      die -> ok
```

Stern & Dill's Approach (cont.)

- The rest of the code

```
doCheck(Pids, Q, Nsent, Nrecv, ToSend) ->
   TimeOut = if (hd(Pids) == self()) -> 1000; true -
   receive
      ...
      after TimeOut ->
         [Pid !  doneQuery || Pid <- tl(Pids)], % sen
         % check message counts
         M = lists:sum([ receive MessageCount, C ->
            M == 0 -> map(fun(Pid) -> Pid !  die end
            M /= 0 -> map(fun(Pid) -> Pid !  continu
         end
   end.

sendStates(State, Pids, ToSend) ->
   if
      ToSend == noDelay ->
            nth(hashToPid(State), Pids) !  {state, St
            noDelay;
      true -> [State, ToSend]
```

Why Stern & Dill's approach is correct

PReach: scaling it up to hundreds of processors

- Issues
  - Input queue overflow
  - Load balancing
  - Message batching
- Results
  - Used at Intel on clusters with several hundred machines.
  - Current work on
    - Adding checks for liveness properties.
    - Performance: we always want bigger and faster.
    - Robustness: how to keep going if a machine crashes.

Summary