Modeling HexNet in ACL2

Presentation by Ebele Esimai

ebele@cs.utexas.edu
OUTLINE

- Introduction into HexNet and its features
- Implementation in ACL2
- Packet movement in a sample network
- Proof properties
- Some ACL2 events
The network plan is based on a hexagonal topology, hence is called HexNet. This hexagonal topology can also be seen in the connections in a brick wall.
HexNet

- Each node in the network is called a JUNCTION.

- A Junction can either be
  - REPEATER (degree of 3) : This accepts data and forwards it to its appropriate output.
  - DESTINATION (degree of 1) : This is an entry or exit point in the network.

- Usually, branching and merging in the network is two-way.

- Therefore, data on only two input links compete for each output link and data on each input link can leave through only two possible output links.

Our goal: Modeling the high-level description of this network and packet movement in the network
Features of the network

- **Arbitration**: Correct decision behavior when data arrives on two input links at nearly the same time. Also, a mechanism for fairness is needed.
Features of the network

- **Packet Routing**: Packet coming through one input link has two possible output links to pass through. This decision is based on the comparison of the final destination and the current junction's location.
The network is modeled as a graph.

Addressing is based on a cartesian grid, as such, each junction is list of its x coordinate and its y coordinate.

There are two links between any two connected junctions to represent data movement in each direction.
Implementation of HexNet in ACL2

(defconst *network* ;; junct dest in out
'(((0 2) (2 2) S20 S02))
((2 2) (0 2) S02 S20)
((4 2) S42 S24)
((2 4) T24 T42))
((4 2) (2 2) S24 S42)
((6 2) S64 S46)
((4 3) T23 T32))
((6 2) (4 2) S46 S64)
((8 2) S86 S68)
((6 0) T02 T20))
((8 2) (6 2) S68 S86)
((10 2) S108 S810)
((8 3) T83 T38))
...

![HexNet diagram]
Packets

- A packet is modelled as a list that contains:
  - A turn signal, which is the next direction the packet should take
  - An address of final destination of the packet
  - An address of the origin of the packet
  - The information to be passed along
- The packets are stored on the links.
- A link can either be full or empty.
- A linkmap shows the current state of all the links in the network at a given time.

Pkt1 - ‘(E (8 7) (0 2) data)
E – turn_signal
(8 7) – coordinates of the final destination
(0 2) – coordinates of the origin of the packet

(defconst *linkmap*
'((S02 (E (12 2) (0 2) data))
 (S20) (S24) (S42) (S46) (S64)
 (S68) (S86) (S810) (S108)
 (S1012) (S1220) (T20)
 (T02) (T32) (T23) (T38)
 (T83 (W (3 4) (8 3) data)) ...
Arbitration in the Model

- Arbitration is based on the output direction
- One of the packets is chosen to proceed while the second waits
Arbitration in the Model

- Arbitration is based on the output direction
- One of the packets is chosen to proceed while the second waits
- However, to facilitate fairness, the model has to remember the decision at the previous step if there is another tie at that junction
Arbitration in the Model

- Arbitration is based on the output direction
- One of the packets is chosen to proceed while the second waits
- However, to facilitate fairness, the model has to remember the decision at the previous step if there is another tie at that junction

Outlink – link in the output direction
Inputs – links with packets
St – statemap that stores previous step decisions

(defun arbiter (outlink inputs st)
  (cond
    ((atom inputs) (mv nil st))
    ((atom (cdr inputs)) (mv (car inputs) st))
    (t (let* ((entry (assoc-eq outlink st))
              (pref (cdr entry)))
        (if (and pref
             (member-eq pref inputs))
            (let* ((new-pref (car (remove-eq pref inputs)))
                   (new-st (update-alist outlink new-pref st)))
              (mv new-st))
            (let* ((new-pref (cadr inputs))
                   (new-st (update-alist outlink new-pref st)))
              (mv (car inputs) new-st)))))))

12
Packet Routing

- The network uses **Advanced Address Decoding**
- This means that the routing function calculates the next direction, i.e. turn-signal, a step before it makes the turn.
- Comparison is made between the final destination of the packet and the address of next junction.

*Turn_signal (pkt1) = East => curr*

*Routing_function (dest1 dest2 curr final g) = (compare curr final) => East or South*
(defun routing-normal (dest1 dest2 curr final g)
  (let* ((dest1-x (car dest1))
          (dest1-y (cadr dest1))
          (dest2-x (car dest2))
          (dest2-y (cadr dest2))
          (dir1 (get-direction curr dest1 g))
          (dir2 (get-direction curr dest2 g))
          (curr-x (car curr))
          (curr-y (cadr curr))
          (final-x (car final))
          (final-y (cadr final)))
    (if (isTerminal dest1 curr g)
        (if (equal dest1 final)
            dir1
            (if (isTerminal dest2 curr g)
                (if (equal dest2 final) dir2 (cw "Bad route "))
                dir2)
        (if (isTerminal dest2 curr g)
            (if (equal dest2 final) dir2 dir1)
            (cond ((< curr-y final-y)
                    (if (< curr-y dest1-y) dir1 dir2))
                  ((> curr-y final-y)
                    (if (> curr-y dest1-y) dir1 dir2))
                  ((< curr-x final-x)
                    (if (< dest1-x dest2-x) dir2 dir1))
                  ((> curr-x final-x)
                    (if (< dest1-x dest2-x) dir1 dir2))))))))
Example

- Linkmap before execution

(S02 (E 12 2) (0 2) data))
(T83 (W 3 4) (8 3) data))
(T68 (E 12 2) (6 8) data))
Example

- Linkmap before execution
  (S02 (E (12 2) (0 2) data))
  (T83 (W (3 4) (8 3) data))
  (T68 (E (12 2) (6 8) data))

- Linkmap after 1 step
  (S24 (E (12 2) (0 2) DATA))
  (S86 (W (3 4) (8 3) DATA))
  (S686 (E (12 2) (6 8) DATA))
Example

- Linkmap before execution
  (S02 (E (12 2) (0 2) data))
  (T83 (W (3 4) (8 3) data))
  (T68 (E (12 2) (6 8) data))

- Linkmap after 1 step
  (S24 (E (12 2) (0 2) DATA))
  (S86 (W (3 4) (8 3) DATA))
  (S686 (E (12 2) (6 8) DATA))

- Linkmap after 2 steps
  (S46 (E (12 2) (0 2) DATA))
  (S64 (W (3 4) (8 3) DATA))
  (S610 (S (12 2) (6 8) DATA))
Example

- Linkmap before execution
  (S02 (E (12 2) (0 2) data))
  (T83 (W (3 4) (8 3) data))
  (T68 (E (12 2) (6 8) data))

- Linkmap after 1 step
  (S24 (E (12 2) (0 2) DATA))
  (S86 (W (3 4) (8 3) DATA))
  (S686 (E (12 2) (6 8) DATA))

- Linkmap after 2 steps
  (S46 (E (12 2) (0 2) DATA))
  (S64 (W (3 4) (8 3) DATA))
  (S610 (S (12 2) (6 8) DATA))

- Linkmap after 3 steps
  (S68 (E (12 2) (0 2) DATA))
  (S42 (N (3 4) (8 3) DATA))
  (T106 (S (12 2) (6 8) DATA))
Example

- Linkmap before execution
  (S02 (E (12 2) (0 2) DATA))
  (T83 (W (3 4) (8 3) DATA))
  (T68 (E (12 2) (6 8) DATA))

- Linkmap after 1 step
  (S24 (E (12 2) (0 2) DATA))
  (S86 (W (3 4) (8 3) DATA))
  (S686 (E (12 2) (6 8) DATA))

- Linkmap after 2 steps
  (S46 (E (12 2) (0 2) DATA))
  (S64 (N (3 4) (8 3) DATA))
  (S610 (S (12 2) (6 8) DATA))

- Linkmap after 3 steps
  (S68 (E (12 2) (0 2) DATA))
  (S42 (N (3 4) (8 3) DATA))
  (T106 (S (12 2) (6 8) DATA))

- Linkmap after 4 steps
  (S810 (E (12 2) (0 2) DATA))
  (T42 (E (3 4) (8 3) DATA))
  (T104 (E (12 2) (6 8) DATA))
Example

- Linkmap after 4 steps
  (S810 (E (12 2) (0 2) DATA))
  (T42 (E (3 4) (8 3) DATA))
  (T104 (E (12 2) (6 8) DATA))

- Linkmap after 5 steps
  (S1012 (DONE (12 2) (0 2) DATA))
  (S23 (DONE (3 4) (8 3) DATA))
  (T104 (E (12 2) (6 8) DATA))

The statemap records that preference should be given to pkt2 if there is a tie in the next step.

(S1012 . T104)
Example

Linkmap after 4 steps
(S810 (E (12 2) (0 2) DATA))
(T42 (E (3 4) (8 3) DATA))
(T104 (E (12 2) (6 8) DATA))

Linkmap after 5 steps
(S1012 (DONE (12 2) (0 2) DATA))
(S23 (DONE (3 4) (8 3) DATA))
(T104 (E (12 2) (6 8) DATA))

The statemap records that preference should be given to pkt2 if there is a tie in the next step.
(S1012 . T104)

Linkmap after 6 steps
(S1012 (DONE (12 2) (6 8) DATA))
Proof Properties

- Reachability: Every information source can send data to every destination.

- Diameter: The number of steps that the communication from source to destination will take.
(defun update-link (junct dest lt st g) "Calculates update for the next link"
  (let* ((outlink (outlink-lookup junct dest g)))
    (if (cdr (assoc-eq outlink lt))
      (if (equal (car (cadr (assoc-eq outlink lt))) 'Done)
        (let ((new-lt (update-alist outlink nil lt)))
          (mv new-lt st))
        (mv lt st))
    (let* ((stack (remove-equal dest (sources junct g)))
           (inputs (current-inputs junct dest stack lt g)))
      (if (endp inputs)
        (mv lt st)
      (let* ((result (mv-list 2 (oracle outlink inputs st)))
              (input (car result))
              (new-st (cadr result)))
        (if (atom (cdr (assoc-eq input lt)))
          (mv nil nil)
        (let* ((pkt (cadr (assoc-eq input lt)))
                (Tinal (cadr pkt))
                (turn_signal (routing junct dest Tinal g)))
          (if turn_signal
            (let* ((new-pkt (cons turn_signal (cdr pkt)))
                   (new-link-state (update-alist outlink (list new-pkt) lt))
                   (new-lt (update-alist input nil new-link-state)))
              (mv new-lt new-st))
          (mv nil nil)))))))
(defun step-junct (junct neighbors lt st g)  "Updates all output links in a junction"
  (if (endp neighbors)
      (mv lt st)
      (let ((dest (car neighbors)))
        (if (junctp dest g)
            (let* ((result (mv-list 2 (update-link junct dest lt st g)))
                   (new-lt (car result))
                   (new-st (cadr result)))
              (step-junct junct (cdr neighbors) new-lt new-st g))
            (mv nil nil))))

(defun run (juncts lt st g)  "Updates the Junctions by a step"
  (if (endp juncts)
      (mv lt st)
      (let* ((junct (car juncts))
              (neighbors (sources junct g))
              (result (mv-list 2 (step-junct junct neighbors lt st g)))
              (new-lt (car result))
              (new-st (cadr result)))
      (run (cdr juncts) new-lt new-st g))))