

# Solutions to Homework 1 – CIS 540 Fall 2009

## 1 Problem 1

The reactive component has 4 states, called  $\{1, 2, 3, 4\}$ , and its state machine is as in Fig. 1.

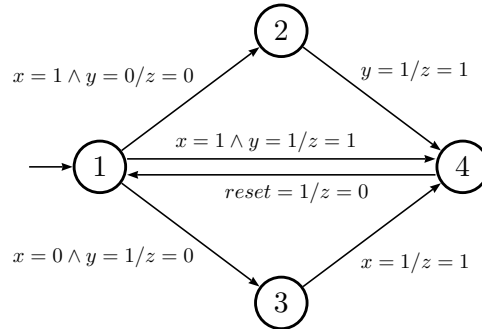


Figure 1: State machine of the component in Problem 1.

The component can be designed as follows:

- Input variables:

bool  $x, y, reset$

- Output variables:

bool  $z$

$\{1, 2, 3, 4\} \quad s : s = 1$
$(s = 4 \wedge reset = 1) \rightarrow (s' = 1 \wedge z = 0)  $ $((s = 1 \wedge x = 1 \wedge y = 1) \vee (s = 2 \wedge y = 1) \vee (s = 3 \wedge x = 1)) \rightarrow (s' = 4 \wedge z = 1)  $ $((s = 1) \rightarrow ((x = 1) \rightarrow (s' = 2)   ((y = 1) \rightarrow (s' = 3)   (s' = 1)))   (s' = s)) \wedge (z = 0))$

## 2 Problem 2

**ClockedDelay** is an event-triggered component: everytime there is an event *clock*, the component saves the current input (variable  $x$ ) in a state variable ( $s'$ ) and outputs the (old) value of  $s$ . Its design is as follows:

- Input variables:

bool  $x$   
 event  $clock$

- Output variables:

event(bool)  $y$

bool $s : s = 0$
$clock? \rightarrow (s' = x \wedge y = s)   (s' = s \wedge y = \perp)$

### 3 Problem 3

There might be different protocols, and below is just one of them.

The protocol works as follows:

- Each node keeps track of the number of rounds so far, in a state variable  $r$ .
- Each message is a set of pairs  $(id, dist)$  of type  $(nat, nat)$ , each means that there exists a path of length  $dist$  from node  $id$  to some node (not necessarily the node sending the message out) so far (i.e. from beginning until the current round).
- Each node  $n$  keeps the current value of  $D_n$  so far, in a state variable  $D$ .
- Each node keeps a set of ID's of nodes that have reached it so far (i.e. from each node in the set, there exists a path of length at most  $r$  to the current node); we store this set in a state variable  $reach$ .
- Upon receiving an input which is a set of pairs  $(id, dist)$ , each node does the following:
  - If its own ID is in the set then removes all those pairs and updates  $D$  to the maximum distance in those pairs.
  - For each  $id$  in the set that is not in  $reach$ , i.e. the corresponding node has not yet reached this node, inserts it into  $reach$  and updates corresponding  $dist$  to  $r$ .
  - Saves the remaining pairs in a state variable  $x$ , to send them out in the next round.
- Initially, each node sends out  $\{(myID, 0)\}$ ; in each round, each node sends out the value of  $x$ .
- After at most  $2N - 2$  rounds, the value of  $D$  of each node is the maximum length of paths from it to other nodes.

Before writing the algorithm for each network node component, we need to define some notations:

- For each node, an input  $in$  is a set of sets of pairs  $(id, dist)$ ; we write  $\cup in$  to denote the union of the sets in  $in$  such that if there are multiple pairs with the same  $id$ , only the one with maximum  $dist$  is retained.
- If  $s$  is a set of pairs  $(id, dist)$ 
  - $s.id$  is the set of the ID's in the pairs;
  - $s[ids]$ , where  $ids$  is a set of ID's, returns the set of pairs with  $id$  in  $ids$ , and returns  $\emptyset$  if there is no such pair.
  - $(s.dist \leftarrow r)$  returns the same pairs in  $s$  but with  $dist$  being changed to  $r$ .

The algorithm for each node component is then as following

- Message type:  $msg := \text{set}((\text{nat}, \text{nat}))$ .

- Input variables

$\text{set}(msg) \text{ in}$

- Output variables:

$\text{event}(msg) \text{ out}$

$\begin{aligned} \text{nat } D : D = 0 \\ \text{set}(\text{nat}) \text{ reach} : \text{reach} = \{myID\} \\ \text{nat } r : r = 1 \\ \text{msg } x : x = \{(myID, 0)\} \end{aligned}$
$\begin{aligned} \text{out} = (r \leq 2N - 2) \rightarrow x \perp \\ \wedge r' = (r \leq 2N - 2) \rightarrow r + 1   r \\ \wedge (\cup in = \emptyset) \rightarrow [(reach' = reach) \wedge (D' = D) \wedge (x' = x)]   \\ [(reach' = (reach \cup (\cup in.id))) \wedge (D' = \max(D, \cup in[myID].id)) \wedge \\ (x' = (\cup in[\cup in.id \setminus reach]) \cup (\cup in[reach \setminus \{myID\}] \leftarrow r))] \end{aligned}$

To illustrate the protocol, consider the following example: a network with 3 nodes ( $N = 3$ ), connected as in Fig. 2. With this network topology, the algorithm requires the maximum number of rounds,  $2N - 2 = 4$  in this case, to determine the value  $D_n$  for each node  $n$ . The execution of the algorithm for this example is given in Table 1. Notice that after 4 rounds,  $D_1 = 2$ ,  $D_2 = 1$ , and  $D_3 = 2$ .



Figure 2: Example of Problem 3: network with 3 nodes.

## 4 Problem 4

The safety monitor can be designed as follows

$r$	Node 1	Node 2	Node 3
1	$in = \{(2, 0)\}$ $D = 0$ $reach = \{1\}$ $out = x = \{(1, 0)\}$	$in = \{(1, 0), (3, 0)\}$ $D = 0$ $reach = \{2\}$ $out = x = \{(2, 0)\}$	$in = \{(2, 0)\}$ $D = 0$ $reach = \{3\}$ $out = x = \{(3, 0)\}$
2	$in = \{(1, 1), (3, 1)\}$ $D = 0$ $reach = \{1, 2\}$ $out = x = \{(2, 1)\}$	$in = \{(2, 1), (2, 1)\}$ $D = 0$ $reach = \{1, 2, 3\}$ $out = x = \{(1, 1), (3, 1)\}$	$in = \{(1, 1), (3, 1)\}$ $D = 0$ $reach = \{2, 3\}$ $out = x = \{(2, 1)\}$
3	$in = \{\}$ $D = 1$ $reach = \{1, 2, 3\}$ $out = x = \{(3, 2)\}$	$in = \{(1, 2), (3, 2)\}$ $D = 1$ $reach = \{1, 2, 3\}$ $out = x = \{\}$	$in = \{\}$ $D = 1$ $reach = \{1, 2, 3\}$ $out = x = \{(1, 2)\}$
4	$in = \{(1, 2), (3, 2)\}$ $D = 1$ $reach = \{1, 2, 3\}$ $out = x = \{\}$	$in = \{\}$ $D = 1$ $reach = \{1, 2, 3\}$ $out = x = \{(1, 2), (3, 2)\}$	$in = \{(1, 2), (3, 2)\}$ $D = 1$ $reach = \{1, 2, 3\}$ $out = x = \{\}$
5	$in = \perp$ $D = 2$ $reach = \{1, 2, 3\}$ $out = \perp$	$in = \perp$ $D = 1$ $reach = \{1, 2, 3\}$ $out = \perp$	$in = \perp$ $D = 2$ $reach = \{1, 2, 3\}$ $out = \perp$

Table 1: Execution of Example 3 of Problem 3

- Input variables

event( $\{\text{arrive, leave}\}$ )  $out_E, out_W$   
 $\{\text{green, red}\}$   $signal_E, signal_W$

- Output variables: none

$\{\text{East, West, None}\}$ $lasttrain : lasttrain = \text{None}$
bool $error, waiting_E, waiting_W : error = 0 \wedge waiting_E = 0 \wedge waiting_W = 0$
$waiting'_E = (out_E = \text{leave}) \rightarrow 0 \mid ((out_E = \text{arrive}) \rightarrow 1 \mid waiting_E)$
$\wedge waiting'_W = (out_W = \text{leave}) \rightarrow 0 \mid ((out_W = \text{arrive}) \rightarrow 1 \mid waiting_W)$
$\wedge lasttrain' = (out_E = \text{leave}) \rightarrow \text{East} \mid ((out_W = \text{leave}) \rightarrow \text{West} \mid lasttrain)$
$\wedge error' = (waiting'_E = 1 \wedge waiting'_W = 1 \wedge error = 0) \rightarrow$ $((signal_W = \text{green} \wedge lasttrain' = \text{West}) \vee (signal_E = \text{green} \wedge lasttrain' = \text{East}) \rightarrow 1 \mid 0)$ $\mid error$

In this monitor,  $waiting_E$  ( $waiting_W$ ) is 1 if and only if the East train (the West train) is waiting for entering the bridge,  $lasttrain$  is the last train that entered the bridge. If  $error = 1$  in any round, the requirement has been violated.

We modify the controller `Controller2` to resolve the contention in a round-robin manner:

$\{\text{green, red}\} \text{ west, east} : \text{west} = \text{red} \wedge \text{east} = \text{ref}$ $\text{bool } \text{near}_E, \text{near}_W : \text{near}_W = 0 \wedge \text{near}_E = 0$ $\{\text{East, West}\} \text{ lasttrain} : \text{lasttrain} = \text{West}$
$\text{signal}_W = \text{west}$ $\wedge \text{signal}_W = \text{west}$ $\wedge \text{near}'_E = [(\text{out}_E = \text{arrive}) \rightarrow 1   ((\text{out}_E = \text{leave} \rightarrow 0   \text{near}_E)]$ $\wedge \text{near}'_W = [(\text{out}_W = \text{arrive}) \rightarrow 1   ((\text{out}_W = \text{leave} \rightarrow 0   \text{near}_W)]$ $\wedge \text{lasttrain}' = [(\text{out}_E = \text{leave}) \rightarrow \text{East}   ((\text{out}_W = \text{leave}) \rightarrow \text{West}   \text{lasttrain})]$ $\wedge \text{east}' = \neg \text{near}'_E \rightarrow \text{red}   ((\neg \text{near}'_W \vee \text{lasttrain}' = \text{West}) \rightarrow \text{green}   \text{east})$ $\wedge \text{west}' = \neg \text{near}'_W \rightarrow \text{red}   ((\neg \text{near}'_E \vee \text{lasttrain}' = \text{East}) \rightarrow \text{green}   \text{west})$

The controller remembers the last train that crossed the bridge, and allows one train to enter the bridge if and only if the other train is not waiting or it is not that last train.