

### Problem 1

(a) Define a module *SyncNor* that models a zero-delay Nor gate. Use the variable names  $in_1$  and  $in_2$  for input, and use *out* for output.

(b) Why is

```
hide z in
  || SyncNor[in1, in2, out := set, z, out]
  || SyncNor[in1, in2, out := reset, out, z]
```

not a legal definition of a module?

(c) Consider the module

```
module SyncDelay is
  private state : B
  interface out : B
  external in : B
  atom ComputeOutput controls out reads state
  atom ComputeNextState controls state awaits in
  initupdate
  || true → state' := in'
```

which shares the atom *ComputeOutput* with the module *SyncLatch* from Figure 1.17 (Chapter 1). Characterize, in precise words, the set of all initialized trajectories of *SyncDelay*. Is the module *SyncDelay* finite? Closed? Deterministic? Asynchronous? Passive?

(d) Consider the module

```
module SyncLatch2 is
  —interface out
  —external set, reset
  hide z1, z2, z3 in
  || SyncNor[in1, in2, out := set, z3, z1]
  || SyncNor[in1, in2, out := reset, out, z2]
  || SyncDelay[in, out := z1, out]
  || SyncDelay[in, out := z2, z3]
```

How does the behavior of *SyncLatch2* differ from that of *SyncLatch*?

### Problem 2

In this question, you are asked to design a solution to the agreement problem.

#### Informal Description.

In the agreement problem, there are  $n$  processes, and each process  $i$  has a boolean input value. The problem requires each process to decide a boolean output value by coordinating with the remaining processes. Processes are connected by a point-to-point synchronous network. Thus, in every round, every process can send a message to every other process. Processes can fail at arbitrary points. In fact, a process can fail after sending messages only to some of the processes in a round. This is what makes the protocol design challenging. Failed processes stop participating in coordination scheme (this is the so-called stopping-failure). There are two correctness requirements:

1. Validity: The output value should be input value of some process. Thus, if all processes start with input 0, then no process can output 1.
2. Agreement: Output values of all nonfaulty processes coincide.

## Guide to Formalization in Reactive Modules.

Corresponding to each process  $i$ , there are two modules  $P_i$  and  $B_i$ . The interface variables of  $P_i$  are  $pc_i$ ,  $x_i$ ,  $v_i$ , and  $m_i$ , and it has a single external variable  $b_i$ . The interface variable of  $B_i$  is  $b_i$ , and its external variables are  $x_j$ ,  $m_j$  for  $j \neq i$ .

- The type of  $pc_i$  is  $\{normal, fail, decided\}$ . Initially,  $pc_i$  is *normal*. In any update round,  $pc_i$  can change from *normal* to *fail* nondeterministically. Once  $pc_i$  equals *fail*, it stays *fail*. The design problem is to figure out condition for changing  $pc_i$  from *normal* to *decided*. Once  $pc_i$  becomes *decided*, it stays *decided*.
- The module  $P_i$  has a boolean variable  $v_i$  that stores its preference. Initially,  $v_i$  equals the input value, which can be either 0 or 1. As long as  $pc_i$  equals *normal*,  $v_i$  can be updated in each round (you need to figure out a strategy to update it). Once  $pc_i$  changes to *decided* or *fail*,  $v_i$  cannot be changed.
- The module  $P_i$  has an interface event  $x_i$  and associated message bit  $m_i$ . As long as  $pc_i$  is *normal* or *decided*, the module issues, in every update round, the event  $x_i$  with the message  $m_i$  set to its current preference  $v_i$ . We say that the process  $i$  sends the message 0 in an update round if  $P_i$  issues the event  $x_i$  with the message  $m_i$  set to 0.
- The buffer  $B_i$  has an interface variable  $b_i$  that can take 4 values  $\emptyset$ ,  $\{0\}$ ,  $\{1\}$ , and  $\{0,1\}$ , denoting the messages that the process  $i$  receives in each round. If a process  $j$  fails in an update round, then the message issued by  $P_j$  may not get to all processes. We capture this scenario by making update of  $b_i$  nondeterministic. If there is a process  $P_j$  ( $j \neq i$ ) such that  $P_j$  sends 0 and  $P_j$  does not fail in the current round, then  $b_i$  is guaranteed to contain 0. If no process  $P_j$  ( $j \neq i$ ) sends 0 then  $b_j$  is guaranteed not to contain 0. In the remaining case (i.e. some processes sends 0, but all such processes fail in the current round),  $b_i$  may or may not contain 0.

### Assignment.

First figure out a strategy to update the preference, and when to decide. Note that the only external variable of  $P_i$  is  $b_i$ , and thus, it can learn about preferences of other processes only via  $b_i$ . Then, build a formal model of your solution in reactive modules for  $n = 5$  (use mocha to simulate, and check if the model is doing what you think it does). Then, formalize the requirements of validity and agreement as invariants (you may need to add monitors). Then, use mocha to prove that the model satisfies the invariants (if not, fix your solution!). You can use either enumerative or symbolic search capability of mocha. For efficiency, ensure that the only latched variables are  $v_i$  and  $pc_i$  (thus, there are no more than  $6^5$  latched states). To test whether your solution is an optimal one, consider the following property: if overall  $f < n$  processes fail, all the nonfaulty are guaranteed to decide in  $(f + 1)$ -th update round. Verify whether or not your model satisfies this property for  $f = 2$  (hint: use a monitor processes that keeps track of number of rounds, number of failed processes, etc).

### Using Mocha.

The tool and the documentation is available on the web.

### Using Alternative Model Checkers.

If you prefer, you can use an alternative model checker (e.g. Spin, NuSMV, Murphi). You will need to model the problem in the input language of the chosen tool. You can use the informal description as a starting point, and make design decisions appropriately.