Assignment 2

Submitted by:
Mathieu Thibault-Marois – 5049388

Presented to
Professor Gregor Von Bochmann
for
ELG7187C

Monday February 25th, 2013
University of Ottawa

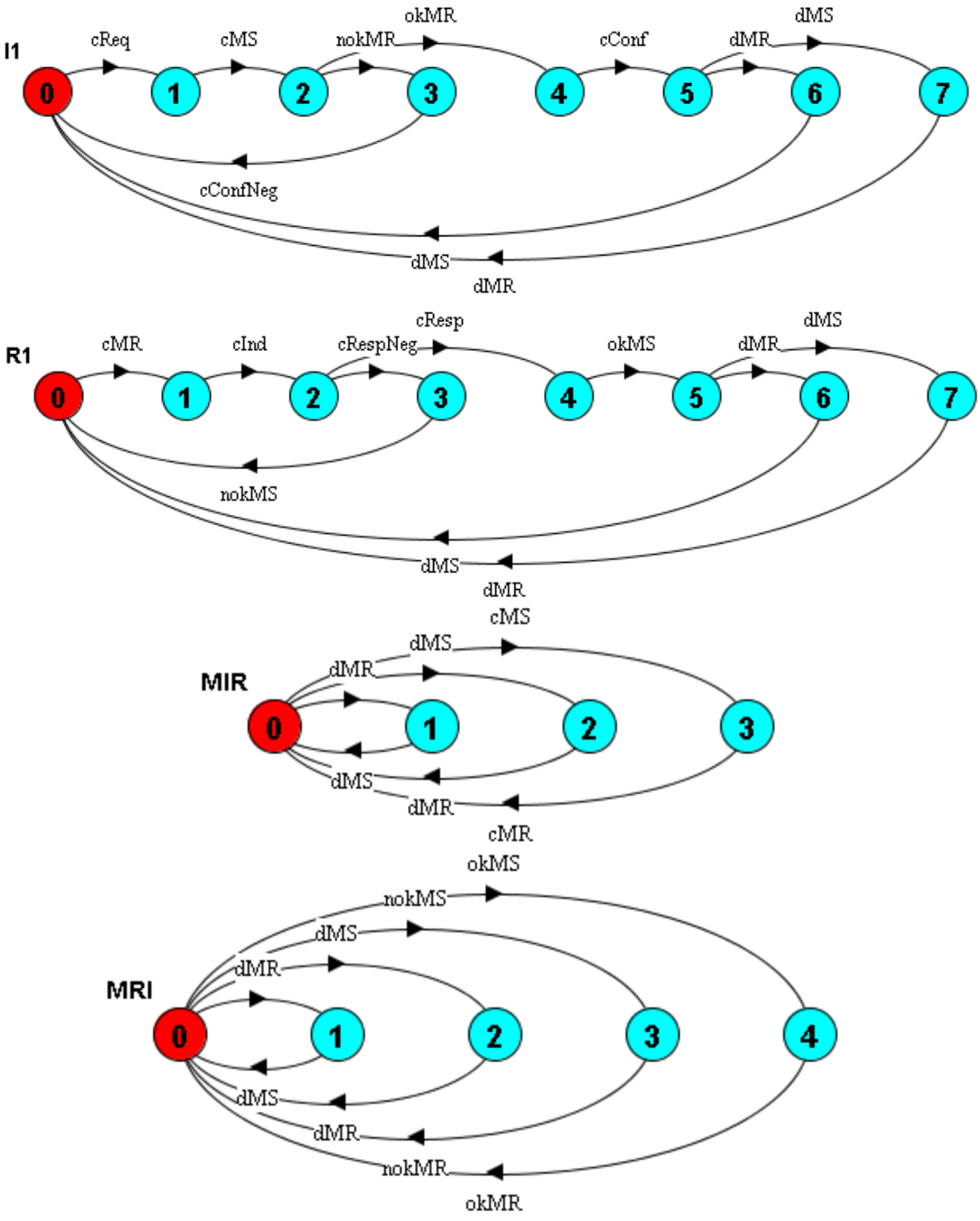
Question 1

At first sight, there seem to be only one difference in the modeling of the receiver and sender and that is the fact that after a successful connection, the LTSA automaton goes back to its initial state for another connection to begin. In the SDL model, if the connection is successful, the model does not go back to its initial state. It only returns if the connection fails. Another difference is the fact that the LTSA automaton models the underlying communication medium, something that is not needed in the SDL model.

Running a safety check on the LTS machine, LTSA reports a deadlock after the following actions: cReq, cMS, cMR, cInd, cResp, okMS, okMR, cConf, dMS, cReq. The reason is that after sending dMS, both the initiator and the receiver go back to the initial state, but the underlying communication is still waiting for an appropriate reception for the signal dMR and thus both MIR and MRI are stuck in state 1. Since Both I1 and R1 are back to state 0, there are no specified reception for dMR and thus the automaton deadlocks. So what the model does is that both the receiver and the initiator disconnect instantly when either one of them disconnect (receives dMR or dMS), but the communication medium actually only expect that the Initiator will disconnect and then the receiver will disconnect. There multiple ways to fix this as to fix it, one must make assumptions on what is the expected behavior of the whole system. For example, is "disconnect" a message that should transit through the medium or is it instantaneous that when one disconnects, the other instantly also does.

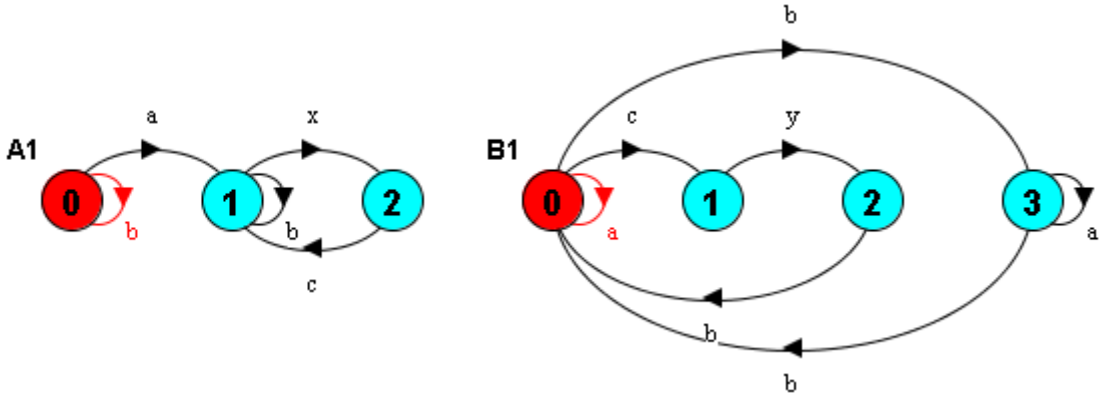
The proposed solution enforces that both dMR and dMS must be received before returning to the initial state, in any order. So if the receiver disconnects (dMR sent), then then dMS must be seen before trying to re-establish a connection. Same goes if dMS is received first, then dMR must be received before continuing. This solution required change to all the LTS machines. First, the precedence relationship is enforced in both the initiator and the receiver. Second, both communication channels are made to accept dMR -> dMS and dMS -> dMR transitions. The model would also work by removing all reference to dMR and dMS from the communication medium but that would mean that both terminals are aware of the disconnect state of each other and can wait on each other to disconnect without relying on it and it would not make much sense. In short, this model ensures that both the initiator and the receiver disconnect properly before returning to their initial state. The revised model is show on the next page.

LTSA's safety and supertrace checks do not report any errors or deadlock with the revised system. The progress check also does not report any livelock with the model. LTSA reports that the total state space is 1280 (8 for I1, 8 for R1, 4 for MIR and 5 for MRI) which requires at least 11 bits to represent. The fact that LTSA report no deadlock is not a warranty that the system actually behaves like we would like it too, it just merely ensure that no blocking exists. In order to ensure that the system behaves as expected, all possible execution path were manually verified.

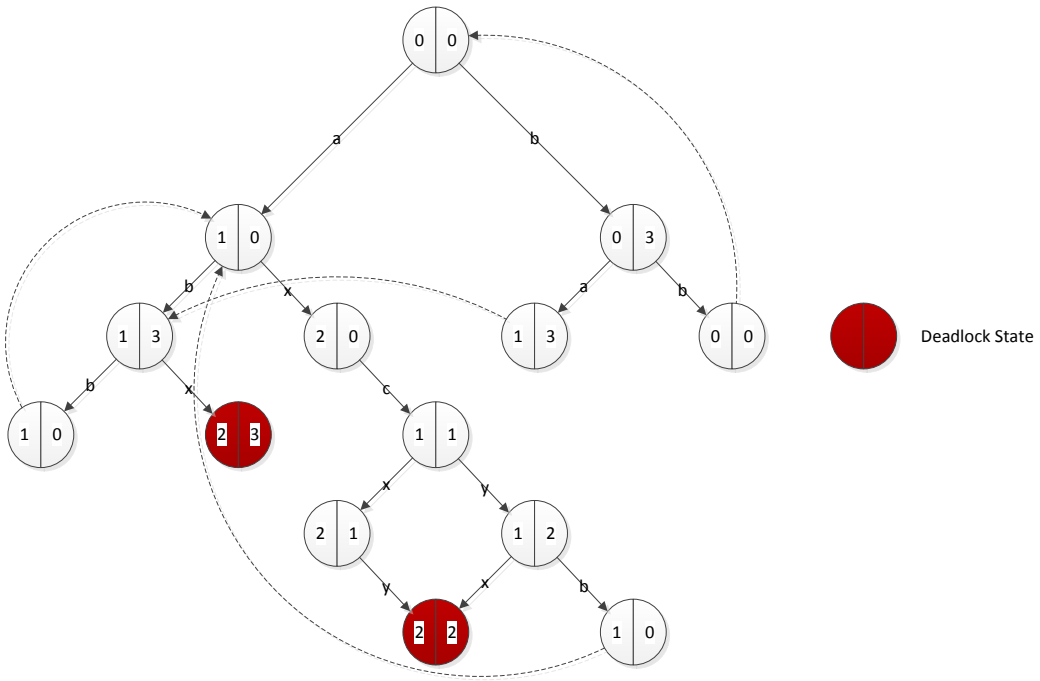

I1
cReq
cMS
okMR
nokMR
cConf
dMR
dMS
0
1
2
3
4
5
6
7
cConfNeg
dMS
dMR
R1
cMR
cInd
cResp
cRespNeg
okMS
dMR
dMS
0
1
2
3
4
5
6
7
nokMS
dMS
dMR
MIR
cMS
dMS
dMR
0
1
2
3
dMS
dMR
cMR
MRI
okMS
nokMS
dMS
dMR
0
1
2
3
4
dMS
dMR
nokMR
okMR

Question 2

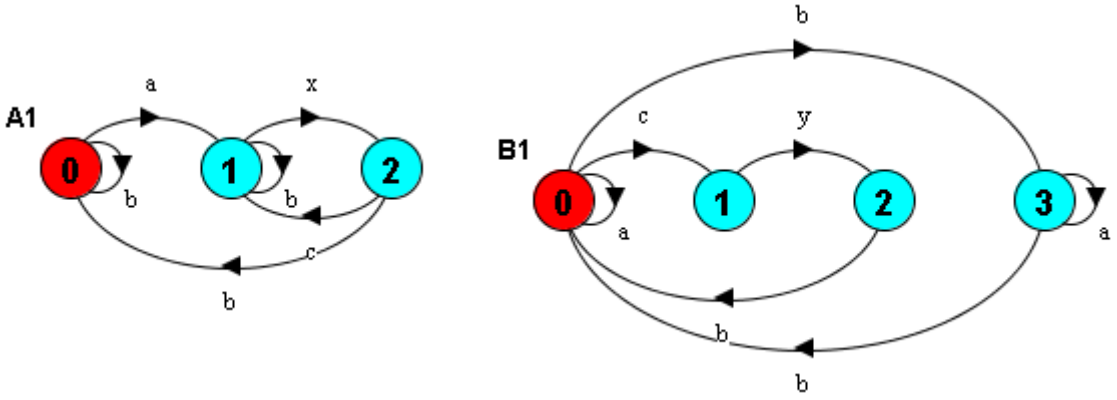
For the sake of clarity, instead of using the machines drawn by hand shown on the webpage, I will be using the equivalent machines shown below, redone in LTSA.

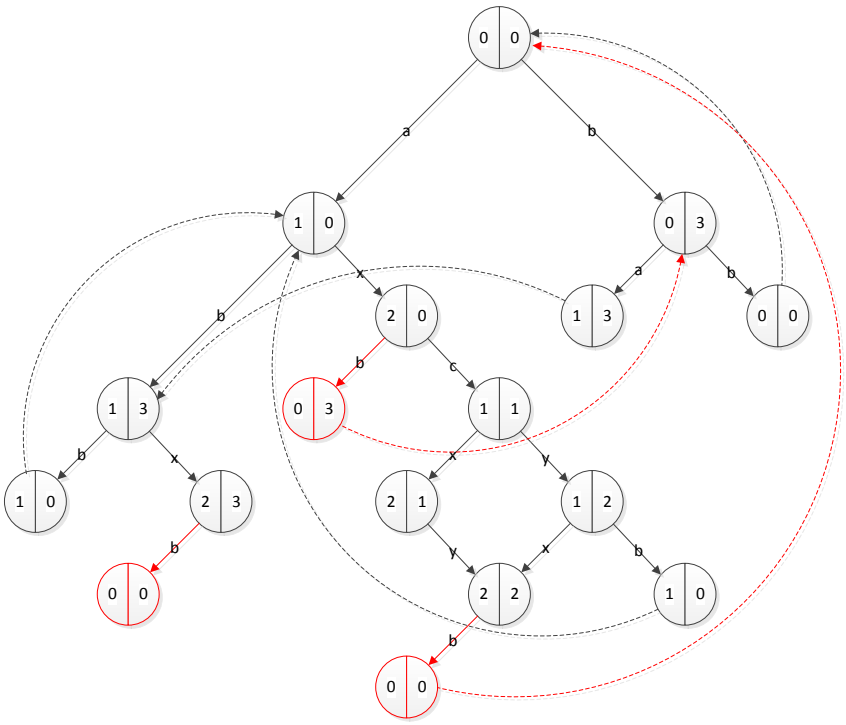


The composed LTS machine is show below (The state numbers have been modified to fit with the notation in LTSA which start a 0, not 1):



There are two states that, if reached, will cause a deadlock of the system. In both state 2,2 and 2,3, machine A can only transition on a C while machine B can only transition on a B. In order to solve the problem, there should either be a transition for C in states 2 and 3 of machines B or a transition for B in state 2 of machine A. Since the second option requires fewer modifications, it will be the one implemented. Now the second decision is to decide to which state that transition should take machine A. Since there are no explanations on what the machine is actually supposed to do, any choice of state will do, as long as the deadlocks are removed. The redesigned machines are shown below, as well as their composition, with state 2 of machine A now transitioning on B to state 0. Since this modification affects other transition than the ones in the deadlocks states, the composition is necessary to show that additional states that may have been introduced do not create new deadlocks.





The new states and transitions are shown in red. As expected, the added transition added new transitions not only in the deadlock states. However, these new added transitions do not create any new deadlock situations. The composed machine is now safe.

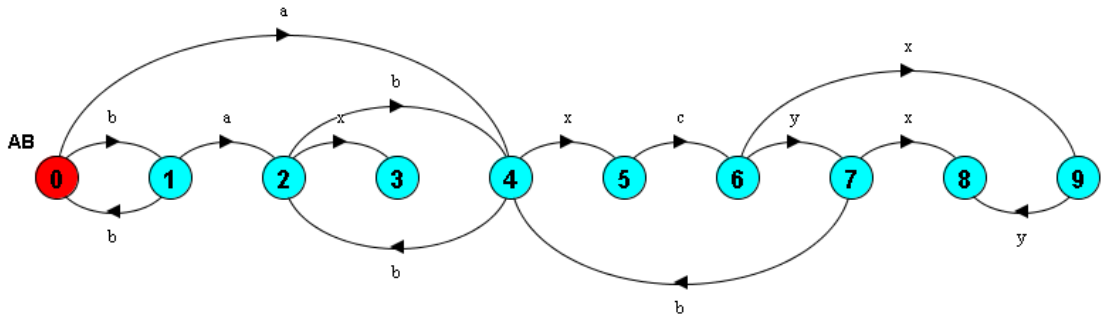
Using LTSA, the two initial machines are modeled as follows:

```
A1 = (a -> A2 | b ->A1), A2 = (x -> A3 | b -> A2), A3 = (c -> A2).

B1 = (a -> B1 | b -> B2 | c -> B3), B2 = (a -> B2 | b -> B1),
B3 = (y -> B4), B4 = (b -> B1).
```

And to tell LTSA that the two models interact with each by rendez-vous, one needs simply to add the following line:

```
||AB = (A1 || B1).
```

LTSA has the capability to compose the machines A1 and B1 into the behavior of a single machine AB. Using this ability wields the following graph:



This is identical to the first composed machines derived by hand. Running LTSA's safety and supertrace checks shows that the system will deadlock for either one of these two sequence of event : a,b,x or b,a,x. This however is limited since it only highlights states 3 (2,3 in the machine derived by hand) as a deadlock state. State 8 (2,2 in the machine derived by hand) is not reported since the tool stops at the first deadlock state it encounters. In short, if LTSA does not report any deadlock, then there are none, however, if LTSA reports a deadlock, there might actually be an arbitrary number of deadlocks states still undetected. Implementing the revised system presented before, LTSA reports no deadlock, as expected. The implementation of the revised system is shown below.

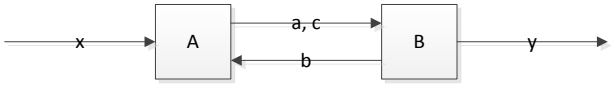
```
A1 = (a -> A2 | b ->A1), A2 = (x -> A3 | b -> A2),
A3 = (c -> A2 | b -> A1).

B1 = (a -> B1 | b -> B2 | c -> B3), B2 = (a -> B2 | b -> B1),
B3 = (y -> B4), B4 = (b -> B1).

||AB = (A1 || B1).
```

While the revised system reports no deadlock, LTSA still reports a livelock (progress check), for the sequence ab(bb)+.

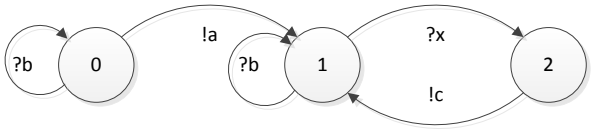
The number of executions of x and y are indeed related to each other. Looking that the original diagram for the composed machine, one can easily see that, in the sequence that do not lead to deadlock, x is always followed by y. Hence for each execution of x, there will be an execution of y. The exception to this is the sequences containing x, c, x, y in which two executions of x follow each other, but these sequences results in a deadlock of the system. In the revised specification, the execution x and y are no longer related. There can be an infinite number of executions for x with no execution of y, one x for every y, two x for every y, etc. The two becomes completely unrelated.

Question 3

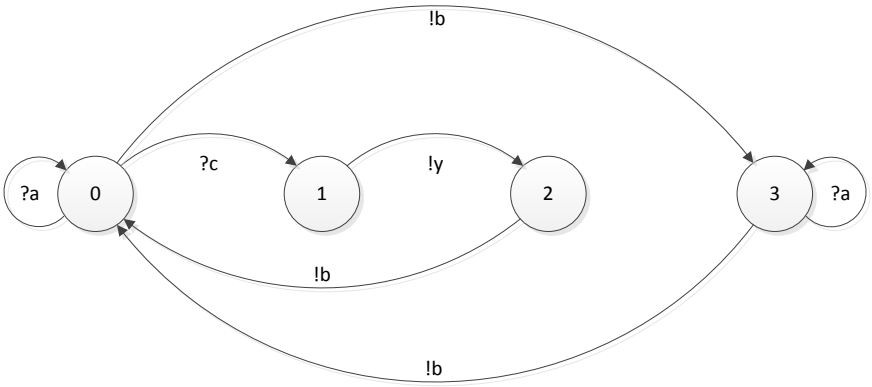
In IOAs, a machine can produce an output for which the receiver does not have a specified reception. For this question, we take the assumption that if an IOA receive an input for which it has no specified reception, the input is simply dropped and the receiving IOA stays in the same state, as if nothing happened. The IOA that sent the input moves on to its new state. Since IOAs specify input and output, unlike LTS machines, the behavior of the system must be extended from the one described in Question 2:



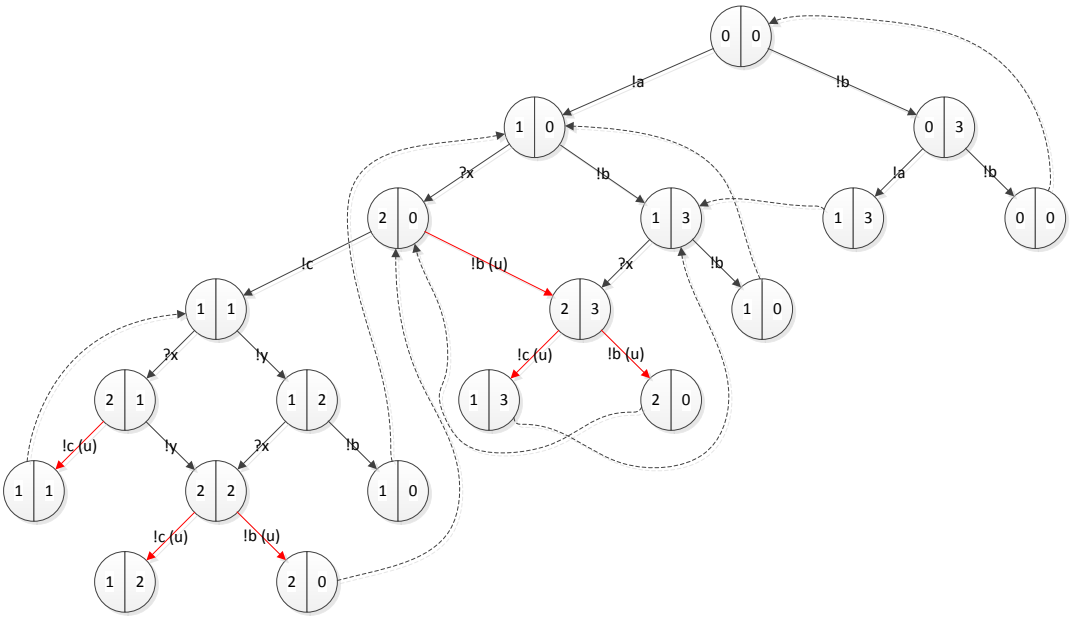
This diagram shows that machine A take x and b as inputs and outputs a and c, and that machine B takes a, c as inputs and outputs b and y. This information can then be added to the IOAs describing machine A and B. Machine A is as follow:



And machine B is:



Doing a reachability analysis of the new system, we get the following:

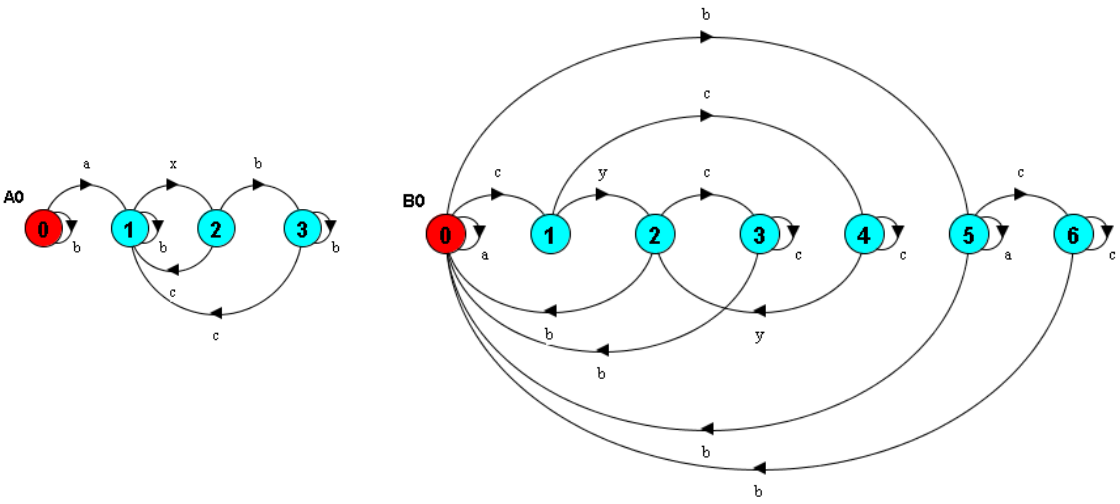


In the diagram, dashed lines represent transitions to states that were already presented previously in the graph and red lines represent transitions where an unspecified reception occurs. The reachability analysis shows that under the assumption that unspecified receptions are dropped, there is no deadlock in the system. Technically, in all states shown above, the message x could be received from the environment. However, since nothing at all happens unless there is a reception specified, these are not represented.

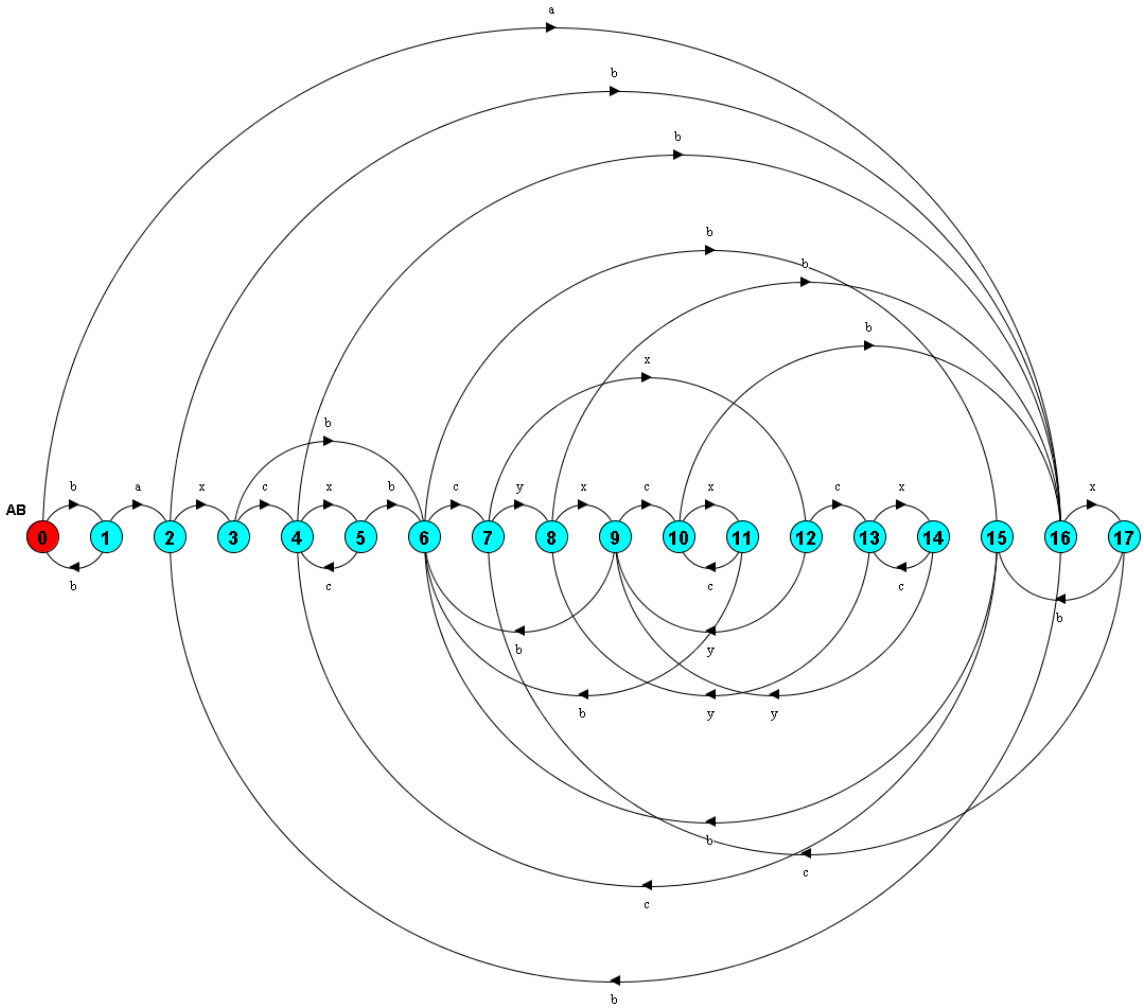
In LTSA, IOA can be represented by adding non specified receptions as transitions to additional states. For example, in state 2 of machine A, b is unexpected, so to model this, we would add another state to which A would transition if b is received and would stay there until c is received, which was the expected reception. The whole model in LTSA is shown below:

```
A0 = (a -> A1 | b ->A0), A1 = (x -> A2 | b -> A1),
A2 = (c -> A1 | b -> A2_NR), A2_NR = (b -> A2_NR | c -> A1).

B0 = (a -> B0 | b -> B3 | c -> B1), B1 = (y -> B2 | c -> B1_NR),
B2 = (b -> B0 | c -> B2_NR), B3 = (a -> B3 | b -> B0 | c -> B3_NR),
B1_NR = (c -> B1_NR | y -> B2), B2_NR = (c -> B2_NR | b -> B0),
B3_NR = (c -> B3_NR | b -> B0).

||AB = (A0 || B0).
```

The resulting LTS machines for this LTS code are show on the next page.

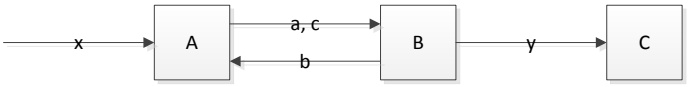


Due to all the added transitions and states, the composed machine is now much more complicated than the one shown in Question 2.

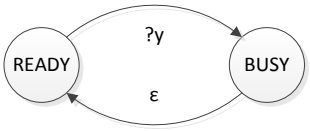


As expected, LTSA does not report any deadlock with the system. However, just like in Question 2, LTSA reports a livelock for the sequence ab(bb)+.

For the last part of Question 3, the system is now assumes to have a component that receives the y interaction sent by B. The architecture thus is now described as:

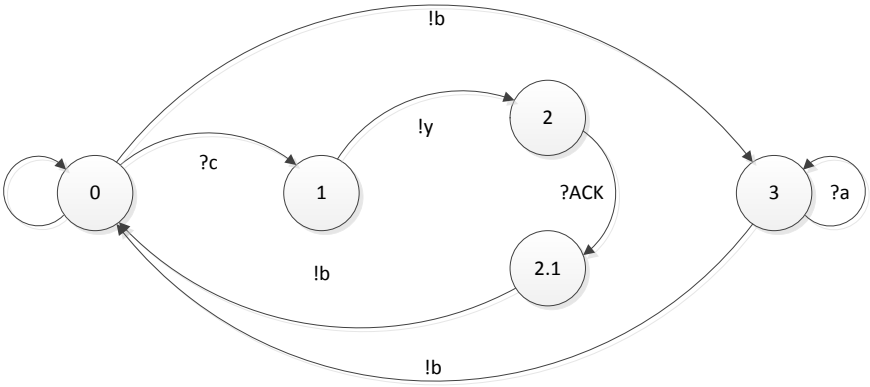


This new component has two states: ready and busy. When it is ready, it can receive y and become busy and spontaneously comes back to ready.

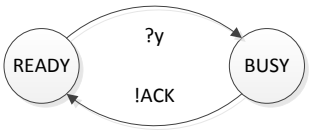


When in busy however, it cannot receive y and thus this situation may lead to an unspecified reception for C. Obviously, this can only happen when B is in a position to send, which means B is in state 1. This means that y can be sent only when the system is in state 1,1 and 2,1 since 3,1 is not a possible state. For proof of this, take a look back at the composition of the original system done at the start of this section.

An easy way to ensure that no receptions are made in the busy state would be for component C to send an ACK to B once it is done with y and for B to wait on this ACK before continuing. B would become this:



And C would now be:

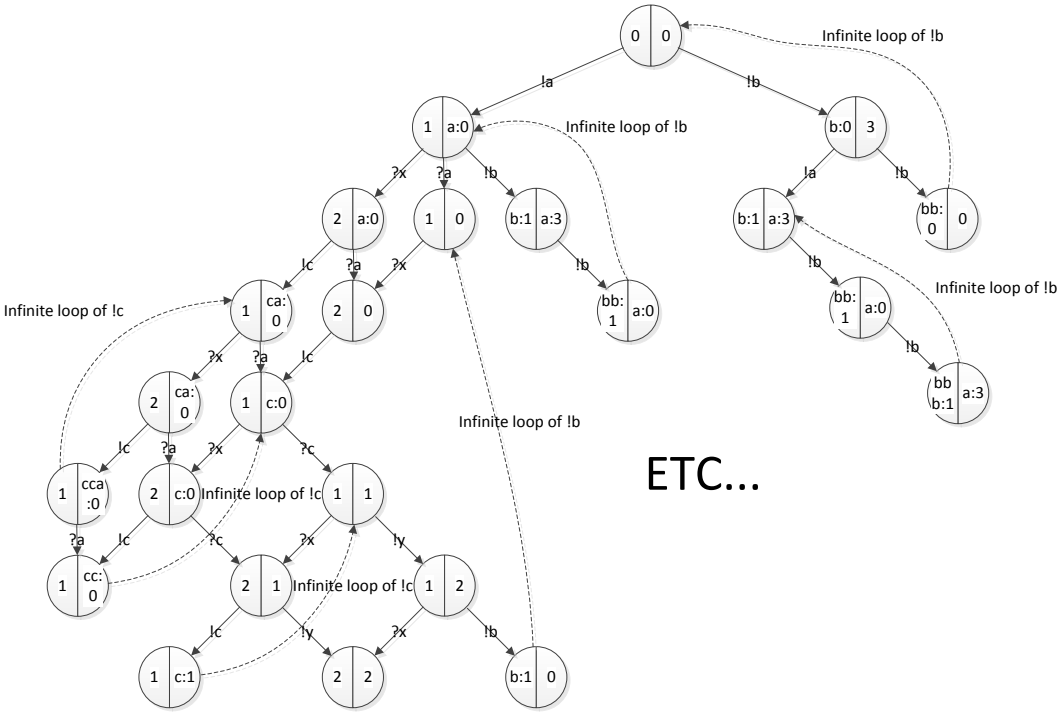


It is thus now impossible for B to send a y to C in the busy state since it needs to wait for C to leave the busy state before moving on and maybe sending a 2nd y.

Question 4

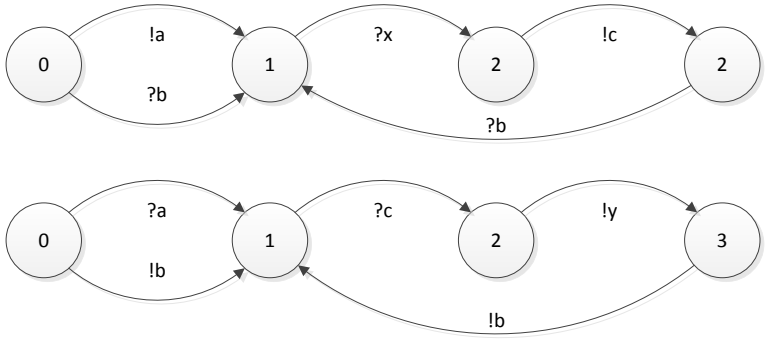
Assuming that the system described in Question 3 now uses asynchronous communication with FIFO buffers then doing a complete reachability analysis is impossible. First, the behavior of the environment is not defined, so in Machine A, the environment could send an infinite number of message x before A as time to consume any. Same goes for y, for which the environment technically needs an infinite queue. In order to simplify the analysis, these queues will not be considered and only the queues in between A and B will be considered. This is in order to analyze the inner workings of the system since we already know that the interaction with the environment is broken.

The diagram below shows a partial reachability analysis of the system. Not all transitions from all states are shown. It highlights the fact that even internally; the system does have multiple infinite loops possible resulting infinite queue lengths.



The problem with the design is two folds. Externally, the environment is not modeled and so is assumed to be able to send / receive an infinite number of messages with no protocol. On the output, this is not a big issue since the infinite queue for y is outside of the system scope and can simply be ignored. However, for the input, the queue for x is inside the system and is infinite. To resolve the problem, one could either model the environment more restrictively or state that inputs to the system do not go into a queue but are simply ignored when not expected. Internally, machine A and B both need infinite queues to communicate with each other. This is due to a bad internal design. Machine B can send an infinite number of message b and Machine A can send an infinite number of message c. One should always ensure that the protocol is balanced, that is one side cannot

overwhelm the other. To solve the problem, one would have to redesign the protocol. However, no knowing the goal of this protocol, one has no ideas what manipulations will preserve the desired behavior. Below is one possible way (purely hypothetical) to change the protocol to eliminate the infinite queues. Machine A is first, followed by Machine B:



As said before, this new protocol changes the behavior of the system and thus may be inappropriate for the task. However, it shows that it is possible to design a protocol without encountering infinite queues if one is careful. If the design is not to be modified, then one must make assumptions as to the environment and the behavior. For example, let assume that x arrives at a very low frequency, then in state 1, machine A might actually be able to deal with all the arriving b message if the frequency of those is also not too high (but larger than for x).

The case where non-specified receptions are dropped has been considered in Question 3. Please refer to that section for reference. In short, since there are no queues, the infinite loops do not cause problem.