Technische Universität München **Winter term 2017/18**
I7
Prof. J. Esparza / Dr. M. Blondin **16.12.2017**

Automata and Formal Languages — Homework 9

Due 19.12.2017

Exercise 9.1

Suppose there are n processes being executed concurrently. Each process has a critical section and a non critical section. At any time, at most one process should be in its critical section. In order to respect this mutual exclusion property, the processes communicate through a channel c. Channel c is a queue that can store up to m messages. A process can send a message x to the channel with the instruction $c \,!\, x$. A process can also consume the first message of the channel with the instruction $c \,?\, x$. If the channel is full when executing $c \,!\, x$, then the process blocks and waits until it can send x. When a process executes $c \,?\, x$, it blocks and waits until the first message of the channel becomes x.

Consider the following algorithm. Process i declares its intention of entering its critical section by sending i to the channel, and then enters it when the first message of the channel becomes i:

```
1 process(i):
2     while true do
3         c ! i
4         c ? i
5         /* critical section */
6         /* non critical section */
```

(a) Sketch an automaton that models a channel of size $m > 0$ where messages are drawn from some finite alphabet Σ.

(b) Model the above algorithm, with $n = 2$ and $m = 1$, as a network of automata. There should be three automata: one for the channel, one for `process`(0) and one for `process`(1).

(c) ★ Use `Spin` to simulate and verify `naive_mutex.pml`.

(d) Construct the asynchronous product of the network obtained in (b).

(e) Use the automaton obtained in (d) to show that the above algorithm violates mutual exclusion, i.e. the two processes can be in their critical sections at the same time.

(f) Design an algorithm that makes use of a channel to achieve mutual exclusion for two processes ($n = 2$). You may choose m as you wish.

(g) Model your algorithm from (f) as a network of automata.

(h) ★ Model your algorithm from (f) in Promela. Use `Spin` to simulate and verify the algorithm.

(i) Construct the asynchronous product of the network obtained in (g).

(j) Use the automaton obtained in (i) to show that your algorithm achieves mutual exclusion.

Exercise 9.2

In the previous question, we have seen that mutual exclusion can be achieved by communicating through channels. Let us now consider processes communicating through shared variables. Suppose there are two processes sharing a variable x initialized to 0. Mutual exclusion can be achieved using the following algorithm:

```
1 process(i):
2     while true do
3         while x = 1 − i do
4             skip
5         /* critical section */
6         x ← 1 − i
7         /* non critical section */
```

(a) Model the above algorithm as a network of automata. There should be three automata: one for the channel, one for `process`(0) and one for `process`(1).

(b) Construct the asynchronous product of the network obtained in (a).

(c) ★ Use `Spin` to simulate and verify `mutex.pml`.

(d) Use the automaton obtained in (b) to show that the algorithm achieves mutual exclusion.

(e) If a process wants to enter its critical section, is it always the case that it will eventually enter it? You should reason in terms of infinite executions.

Exercise 9.3

The following algorithm attempts to achieve mutual exclusion for two processes. The processes share variables b_0, b_1 and k initialized respectively to *false*, *false* and 0.

```
1  process(i):
2      while true do
3          b_i ← true
4          while k ≠ i do
5              while b_{1−i} do
6                  skip
7              k ← i
8          /* critical section */
9          b_i ← false
10         /* non critical section */
```

(a) Model the above algorithm as a network of automata.

(b) Find an execution where both processes end up in their critical sections at the same time.

(c) ★ Model the algorithm in Promela.

(d) ★ Can you find an execution violating mutual exclusion by simulating the algorithm with `Spin`?

(e) ★ Verify the algorithm with `Spin`.

Exercise 9.4

The following algorithm attempts to achieve mutual exclusion for two processes. These processes share variables x, y and z which are initialized to 0.

```
1  process(i):
2      while true do
3          /* non critical section */
4          x ← i + 1
5          if y ≠ 0 and y ≠ i + 1 then
6              goto 4
7          z ← i + 1
8          if x ≠ i + 1 then
9              goto 4
10         y ← i + 1
11         if z ≠ i + 1 then
12             goto 4
13         /* critical section */
```

(a) Model the above algorithm as a network of automata.

(b) ★ Find an execution where both processes end up in their critical sections at the same time.

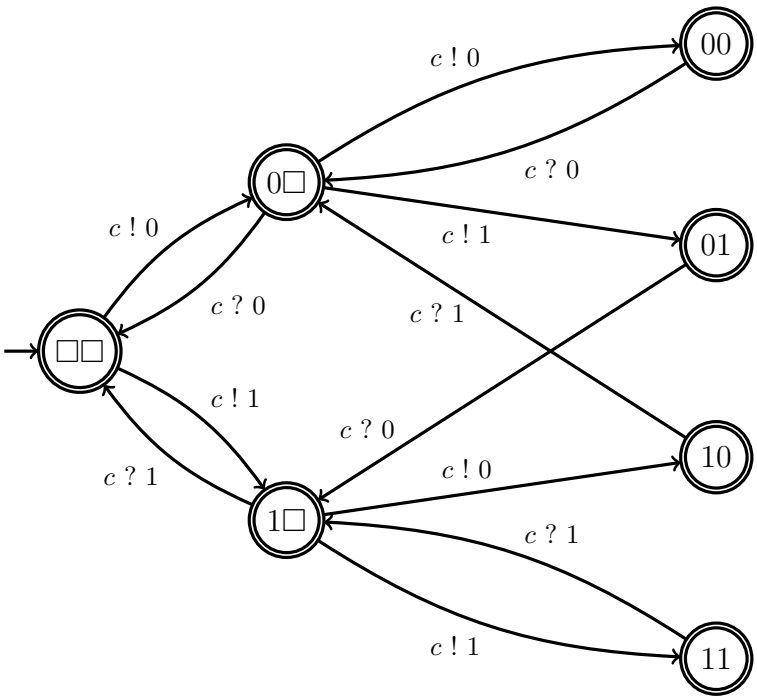
(c) ★ Model the algorithm in Promela.

(d) ★ Can you find an execution violating mutual exclusion by simulating the algorithm with `Spin`?

(e) ★ Verify the algorithm with `Spin`.

Solution 9.1

(a) We construct an automaton $A_{\Sigma,m}$ that stores the content of the channel within its states. For example, the automaton for $\Sigma = \{0, 1\}$ and $m = 2$ is as follows:



More formally, $A_{\Sigma,m} = (Q, \Gamma, \delta, q_0, F)$ is defined as:

$$
\begin{aligned}
Q &= \{w \in (\Sigma \cup \square)^m : (w_i = \square) \implies (w_{i+1} = \square) \text{ for every } 1 \le i < m\},\\
\Gamma &= \{c\,!\,\sigma : \sigma \in \Sigma\} \cup \{c\,?\,\sigma : \sigma \in \Sigma\},\\
q_0 &= \square^m,\\
F &= Q.
\end{aligned}
$$

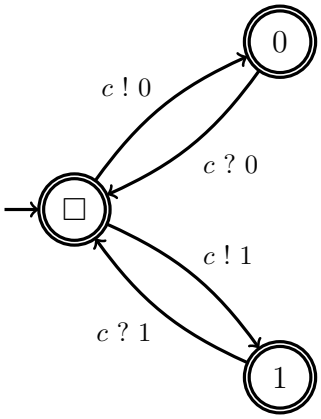
Let $\ell\colon Q \to \{1, 2, \ldots, m\}$ be the function that associates to each state q the position of the last letter of q which is not $\square$. For example, $\ell(abb\square\square) = 3$. The transitions are formally defined as follows:

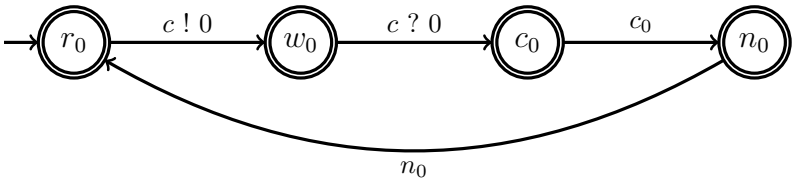
$$
\delta(q, c\,!\,\sigma) = \begin{cases} q_1 q_2 \cdots q_{\ell(q)} \sigma \square^{m-\ell(q)-1} & \text{if } \ell(q) < m,\\ \text{none} & \text{otherwise,} \end{cases}
$$

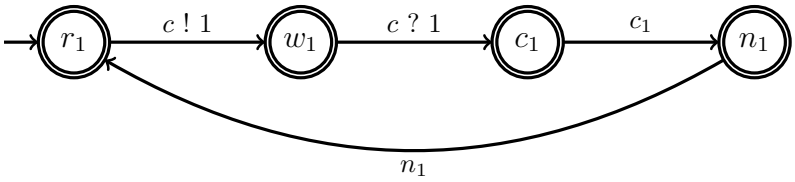
$$
\delta(q, c\,?\,\sigma) = \begin{cases} q_2 q_3 \cdots q_m \square & \text{if } q_1 = \sigma,\\ \text{none} & \text{otherwise.} \end{cases}
$$

★ Note that $A_{\Sigma,m}$ grows exponentially since $|Q| = \sum_{i=0}^{m} |\Sigma|^i = (|\Sigma|^{m+1} - 1)/(|\Sigma| - 1)$.

(b) The automata for the channel, `process`(0) and `process`(1) are respectively:



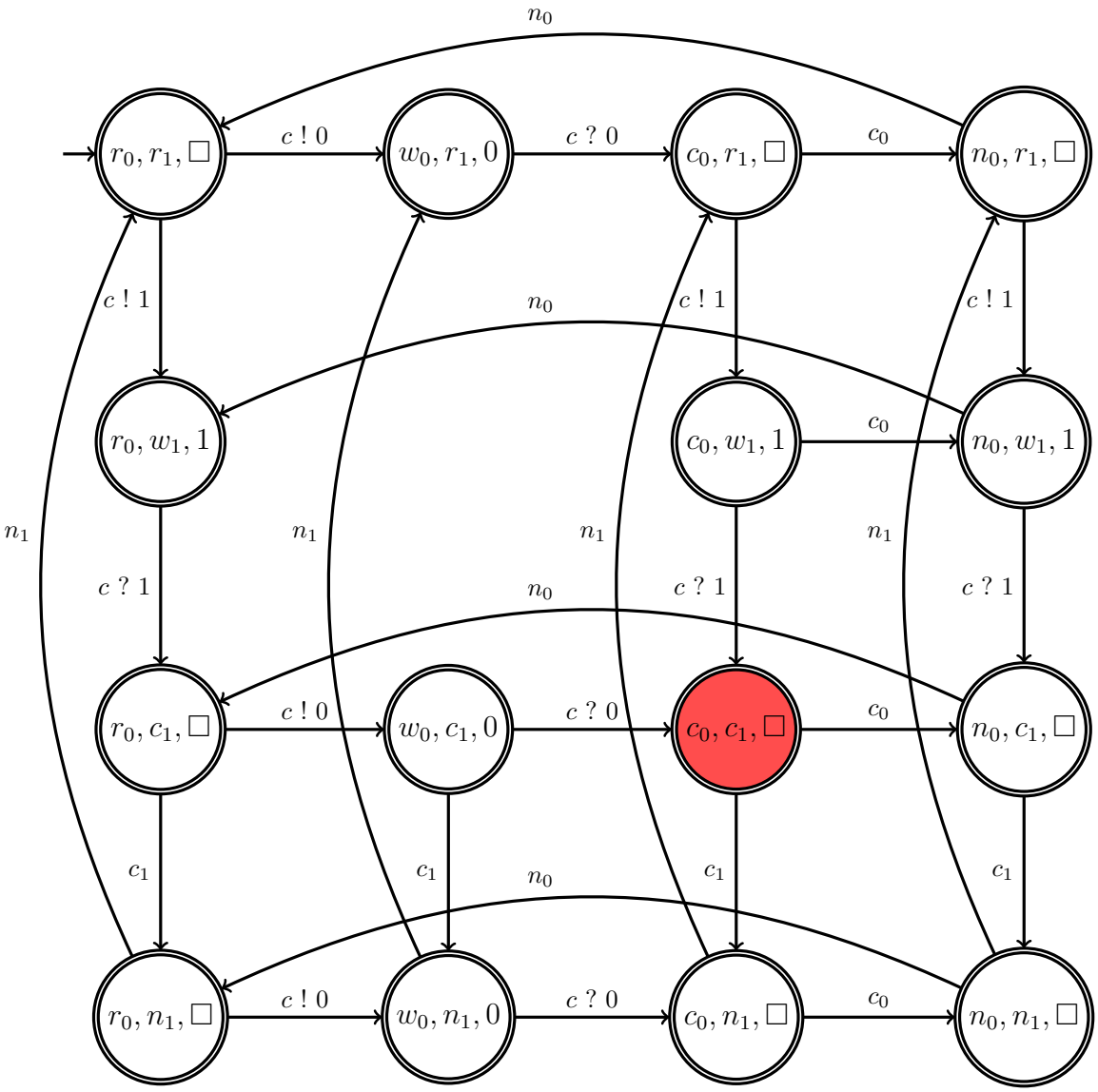




(c) A simulation quickly finds a problem with the algorithm:

```
 0:        proc  - (:root:) creates proc  0 (:init:)
Starting process with pid 1
  1:        proc  0 (:init::1) creates proc  1 (process)
  1:        proc  0 (:init::1) naive_mutex.pml:8 (state 1)        [(run process(0))]
Starting process with pid 2
  2:        proc  0 (:init::1) creates proc  2 (process)
  2:        proc  0 (:init::1) naive_mutex.pml:9 (state 2)        [(run process(1))]
  3:        proc  1 (process:1) naive_mutex.pml:16 (state 1)        [c!i]
  4:        proc  1 (process:1) naive_mutex.pml:19 (state 2)        [c?i]
  5:        proc  2 (process:1) naive_mutex.pml:16 (state 1)        [c!i]
  6:        proc  1 (process:1) naive_mutex.pml:24 (state 3)        [crit = (crit+1)]
  7:        proc  1 (process:1) naive_mutex.pml:25 (state 4)        [assert((crit==1))]
  8:        proc  2 (process:1) naive_mutex.pml:19 (state 2)        [c?i]
  9:        proc  2 (process:1) naive_mutex.pml:24 (state 3)        [crit = (crit+1)]
spin: naive_mutex.pml:25, Error: assertion violated
spin: text of failed assertion: assert((crit==1))

[variable values, step   9]

 crit  =  2
 process(1):i  =  0
 process(2):i  =  1
```

(d)

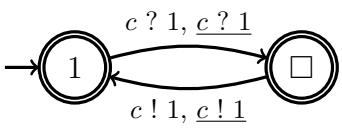


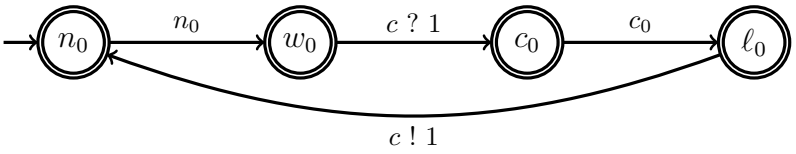
(e) The algorithm violates mutual exclusion since state $(c_0, c_1, \square)$ is reachable in the above automaton.

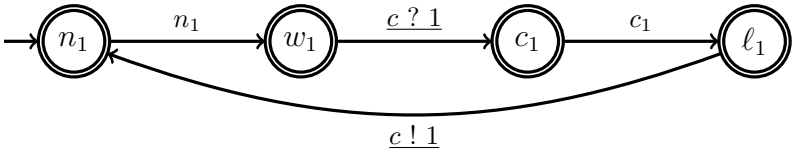
(f) We initialize a channel c of size one with message 1. When a process wants to enter its critical section, it simply consumes 1 from the channel and sends it back once it is done:

```
1 process():
2     while true do
3         /* non critical section */
4         c ? 1
5         /* critical section */
6         c ! 1
```

(g) The automata modeling the channel and the two processes are respectively:







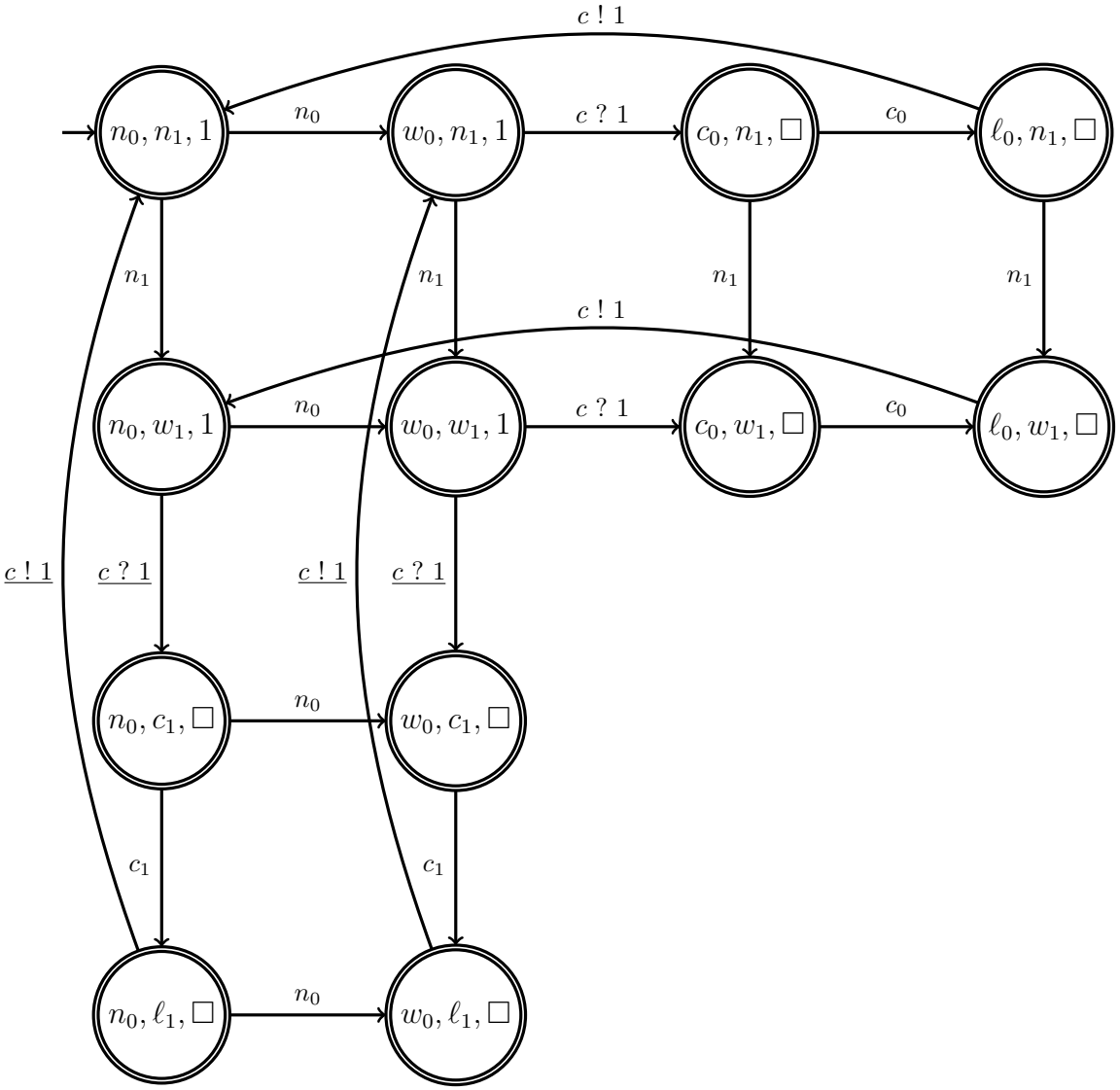
★ Note that we have introduced the new letters $\underline{c \text{ ! } 1}$ and $\underline{c \text{ ? } 1}$. We could have simply used letters c ! 1 and c ? 1. However, these new letters will be important when considering the asynchronous product of the network. If the two automata modeling the processes both used c ! 1 and c ? 1, then the asynchronous product would force them to synchronize on these letters.

★ In class, a student suggested an alterative solution: to simply swap line 4 and 5 of the processes described in #9.1. This also works. You can verify this solution either manually or with `Spin`.

(h) `Spin` successfully verifies the following Promela modeling:

```
chan c = [1] of { bit };
byte crit = 0;

init
{
  atomic
  {
    c ! 1
    run process()
    run process()
  }
}

proctype process()
{
noncritical:
  skip

wait:
  c ? 1

critical:
  atomic
  {
    crit++
    assert(crit == 1)
  }

leave:
  atomic
  {
    c ! 1
    crit--
    goto noncritical
  }
}
```

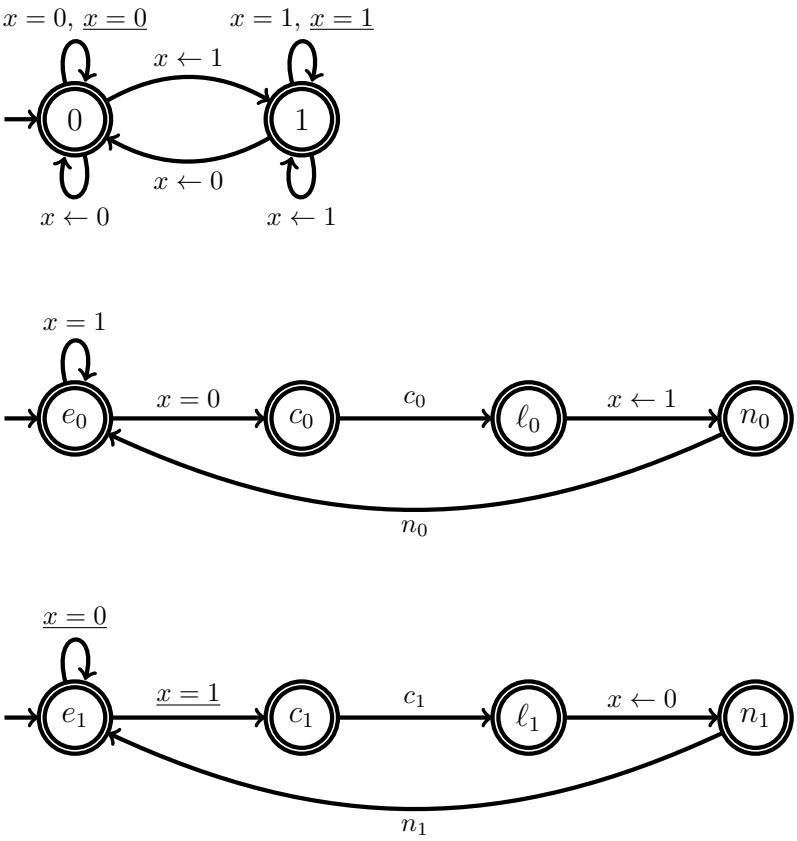
(i)



(j) None of the state of the above automaton is of the form (c_0, c_1, σ) where $\sigma \in \{\square, 1\}$. This implies that both processes cannot be in their critical sections at the same time.

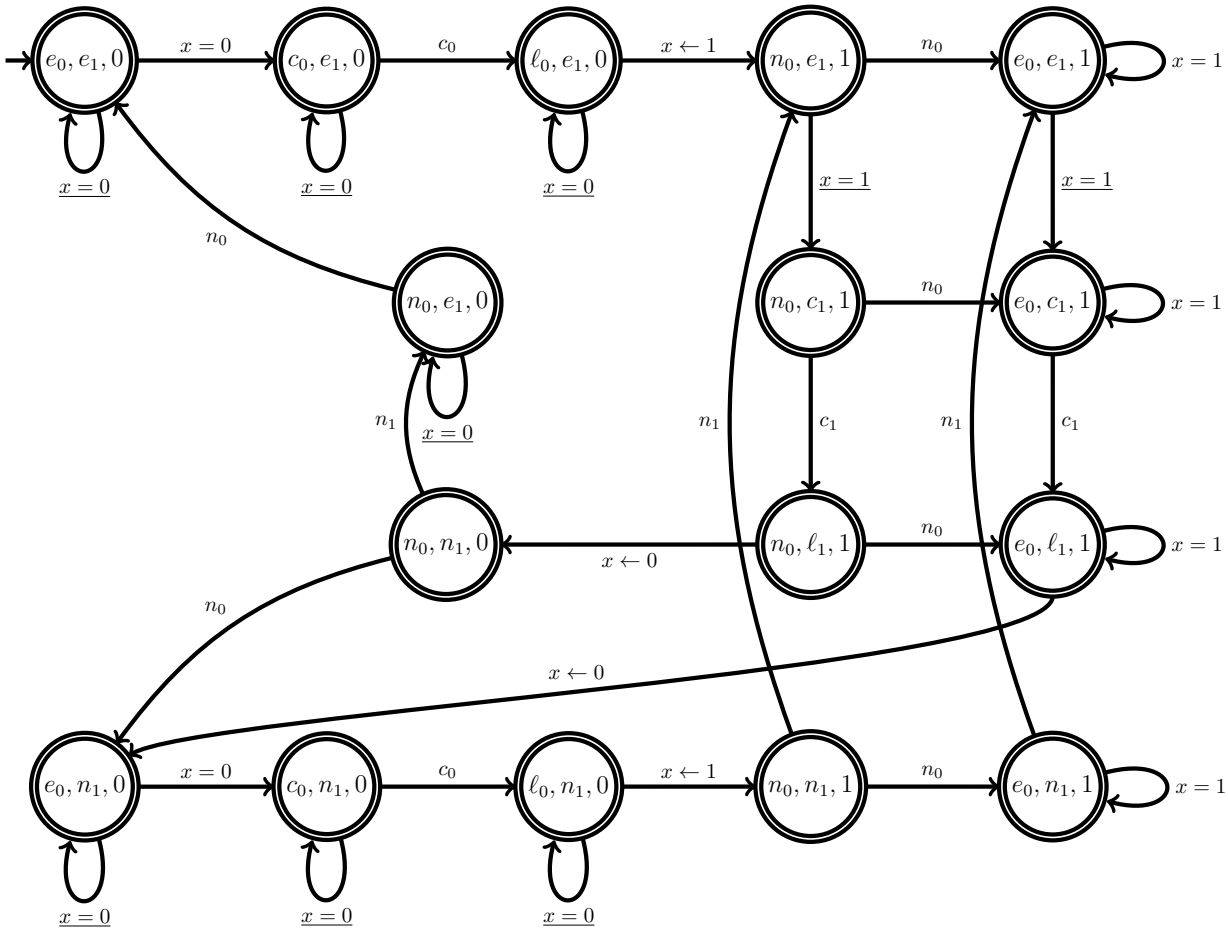
Solution 9.2

(a) The automata modeling the channel, `process`(0) and `process`(1) are respectively:



★ Note that we have introduced the new letters $\underline{x = 0}$ and $\underline{x = 1}$. We could have simply used letters $x = 0$ and $x = 1$. However, these new letters will be important when considering the asynchronous product of the network. If the two automata modeling the processes both used $x = 0$ and $x = 1$, then the asynchronous product would force them to synchronize on these letters.

(b)



(c) `Spin` successfully verifies `mutex.pml`.

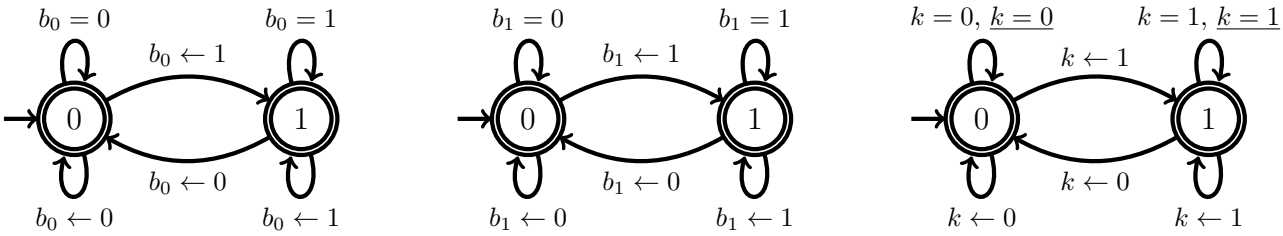
(d) None of the state of the above automaton is of the form (c_0, c_1, σ) where $\sigma \in \{0, 1\}$. This implies that both processes cannot be in their critical sections at the same time.

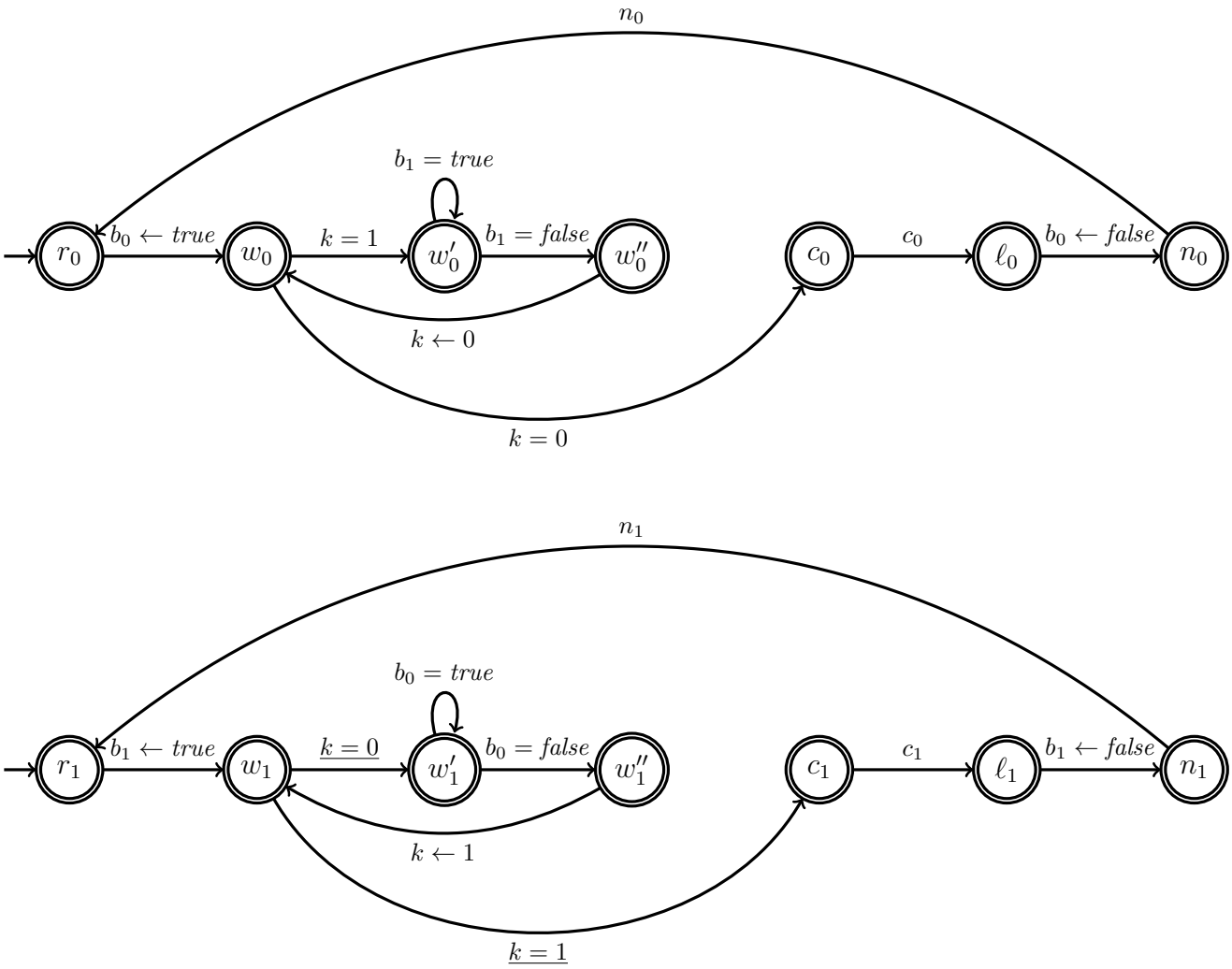
(e) Technically, *no*, since e.g. $(e_0, e_1, 0) \xrightarrow{x=0} (e_0, e_1, 0) \xrightarrow{x=0} \cdots$ is an infinite execution where both processes never reach their critical section. This kind of behaviour occurs when the scheduler eventually let only one process run.

If the scheduler is "fair", then an execution cannot get stuck into any of the self-loops of the automaton. In this case, the answer is *yes* since every cycle of the automaton contains a state of the form $(c_0, \star, \star)$ *and* a state of the form $(\star, c_1, \star)$. Later this semester, we will see how such properties can be verified formally and we will briefly discuss what assumptions can be made on the scheduler.

Solution 9.3

(a) The automata modeling b_0, b_1, k, `process`(0) and `process`(1) are respectively:





(b) Here is such an execution where configurations are of the form $(\texttt{process}(0), \texttt{process}(1), b_0, b_1, k)$:

$$
\begin{aligned}
(r_0, r_1, \mathit{false}, \mathit{false}, 0) &\xrightarrow{b_1 \leftarrow \mathit{true}} (r_0, w_1, \mathit{false}, \mathit{true}, 0) \\
&\xrightarrow{\underline{k=0}} (r_0, w'_1, \mathit{false}, \mathit{true}, 0) \\
&\xrightarrow{b_0 = \mathit{false}} (r_0, w''_1, \mathit{false}, \mathit{true}, 0) \\
&\xrightarrow{b_0 \leftarrow \mathit{true}} (w_0, w''_1, \mathit{true}, \mathit{true}, 0) \\
&\xrightarrow{k=0} (c_0, w''_1, \mathit{true}, \mathit{true}, 0) \\
&\xrightarrow{k \leftarrow 1} (c_0, w_1, \mathit{true}, \mathit{true}, 1) \\
&\xrightarrow{\underline{k=1}} (c_0, c_1, \mathit{true}, \mathit{true}, 1).
\end{aligned}
$$

(c)
```
bool b[2];
bit k;
byte crit;

init
{
  atomic
  {
    b[0] = false
    b[1] = false
    k = 0
    run process(0)
    run process(1)
  }
}

proctype process(bit i)
{
  do
    :: true ->
       // Non critical section
       skip

       b[i] = true

       do
         :: k != i ->
            do
              :: b[1-i] -> skip
              :: else -> break
            od
            k = i
         :: else -> break
       od

       // Critical section
       atomic
       {
         crit++
         assert(crit == 1)
       }

       atomic
       {
         crit--
         b[i] = false
       }
  od
}
```

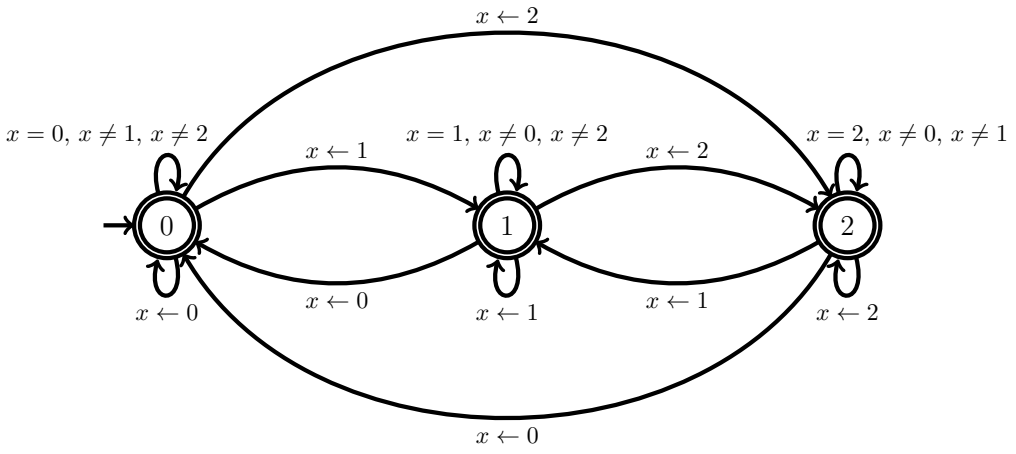
(d) Yes, `Spin` finds a violation on most simulations.

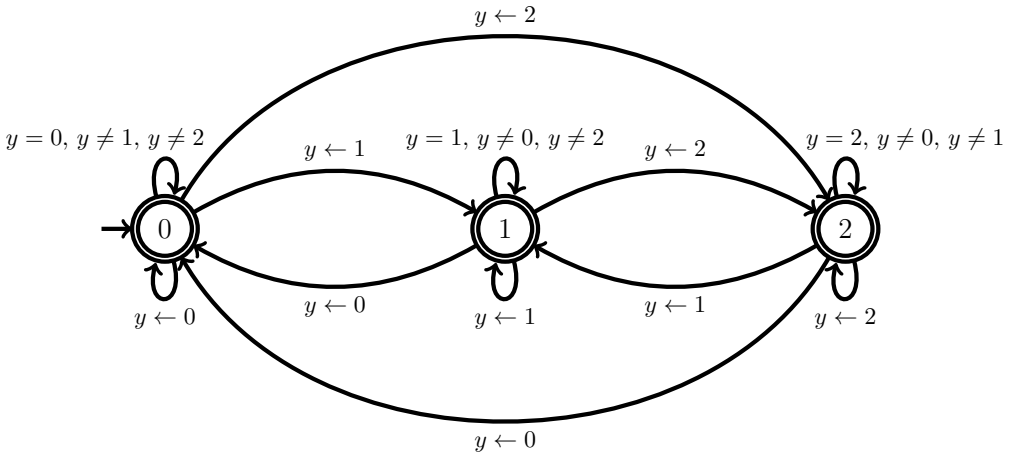
(e) `Spin` finds the following violation:

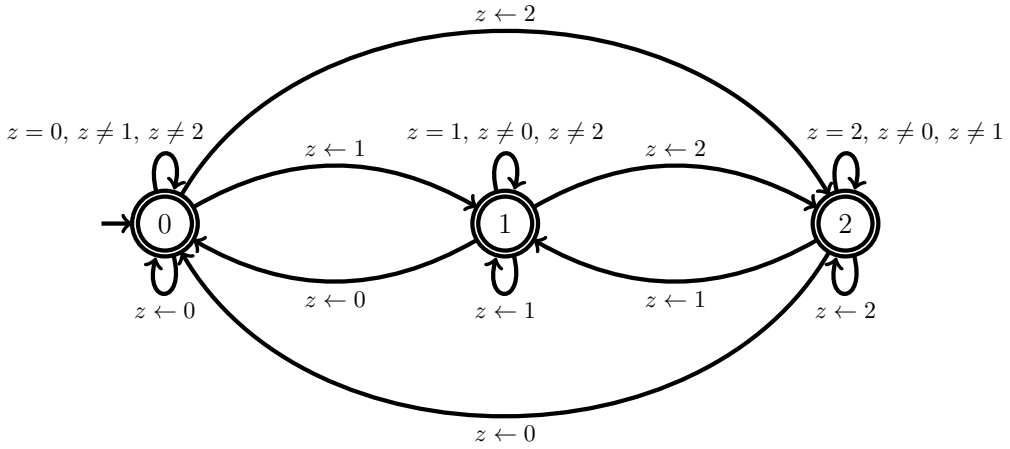
```
using statement merging
  1:        proc  0 (:init::1) hyman.pml:9 (state 1)        [b[0] = 0]
  1:        proc  0 (:init::1) hyman.pml:10 (state 2)        [b[1] = 0]
  1:        proc  0 (:init::1) hyman.pml:11 (state 3)        [k = 0]
Starting process with pid 1
  2:        proc  0 (:init::1) hyman.pml:12 (state 4)        [(run process(0))]
Starting process with pid 2
  3:        proc  0 (:init::1) hyman.pml:13 (state 5)        [(run process(1))]
  4:        proc  2 (process:1) hyman.pml:20 (state 1)        [(1)]
  5:        proc  1 (process:1) hyman.pml:20 (state 1)        [(1)]
  6:        proc  2 (process:1) hyman.pml:24 (state 3)        [b[i] = 1]
  7:        proc  2 (process:1) hyman.pml:27 (state 4)        [((k!=i))]
  8:        proc  2 (process:1) hyman.pml:30 (state 7)        [else]
  9:        proc  1 (process:1) hyman.pml:24 (state 3)        [b[i] = 1]
 10:        proc  1 (process:1) hyman.pml:33 (state 13)        [else]
 11:        proc  2 (process:1) hyman.pml:32 (state 12)        [k = i]
 12:        proc  2 (process:1) hyman.pml:33 (state 13)        [else]
 13:        proc  2 (process:1) hyman.pml:39 (state 18)        [crit = (crit+1)]
 13:        proc  2 (process:1) hyman.pml:40 (state 19)        [assert((crit==1))]
 14:        proc  1 (process:1) hyman.pml:39 (state 18)        [crit = (crit+1)]
spin: hyman.pml:40, Error: assertion violated
spin: text of failed assertion: assert((crit==1))

[variable values, step  14]

 b[0]  =  1
 b[1]  =  1
 crit  =  2
 k  =  1
```

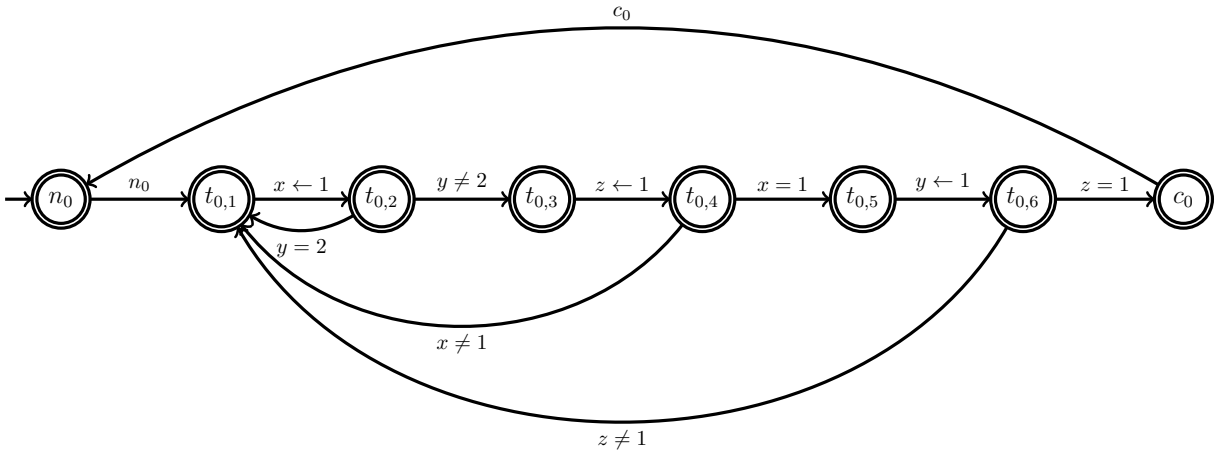
Solution 9.4

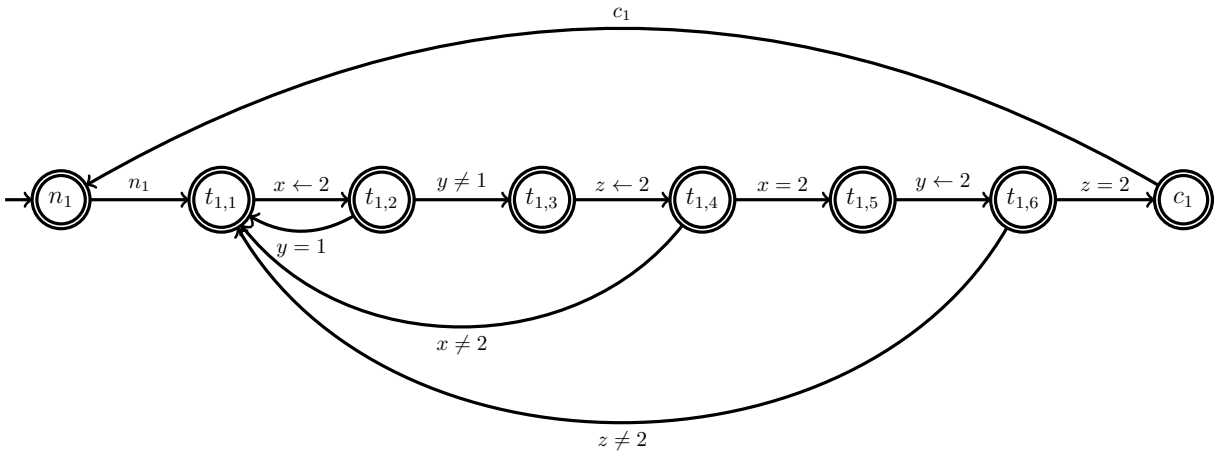
(a) The automata modeling x, y, z, `process`(0) and `process`(1) are respectively:




$y \leftarrow 2$
$y = 0,\ y \neq 1,\ y \neq 2$
$y \leftarrow 1$
$y = 1,\ y \neq 0,\ y \neq 2$
$y \leftarrow 2$
$y = 2,\ y \neq 0,\ y \neq 1$
0
1
2
$y \leftarrow 0$
$y \leftarrow 0$
$y \leftarrow 1$
$y \leftarrow 1$
$y \leftarrow 2$
$y \leftarrow 0$


$z \leftarrow 2$
$z = 0,\ z \neq 1,\ z \neq 2$
$z \leftarrow 1$
$z = 1,\ z \neq 0,\ z \neq 2$
$z \leftarrow 2$
$z = 2,\ z \neq 0,\ z \neq 1$
0
1
2
$z \leftarrow 0$
$z \leftarrow 0$
$z \leftarrow 1$
$z \leftarrow 1$
$z \leftarrow 2$
$z \leftarrow 0$


c_0
n_0
n_0
$t_{0,1}$
$x \leftarrow 1$
$t_{0,2}$
$y \neq 2$
$t_{0,3}$
$z \leftarrow 1$
$t_{0,4}$
$x = 1$
$t_{0,5}$
$y \leftarrow 1$
$t_{0,6}$
$z = 1$
c_0
$y = 2$
$x \neq 1$
$z \neq 1$



(b) Here is such an execution where configurations are of the form $(\texttt{process}(0), \texttt{process}(1), x, y, z)$:

$$
\begin{aligned}
(n_0, n_1, 0, 0, 0) &\xrightarrow{n_1} (n_0, t_{1,1}, 0, 0, 0) \\
&\xrightarrow{x \leftarrow 2} (n_0, t_{1,2}, 2, 0, 0) \\
&\xrightarrow{y \neq 1} (n_0, t_{1,3}, 2, 0, 0) \\
&\xrightarrow{z \leftarrow 2} (n_0, t_{1,4}, 2, 0, 2) \\
&\xrightarrow{x = 2} (n_0, t_{1,5}, 2, 0, 2) \\
&\xrightarrow{n_0} (t_{0,1}, t_{1,5}, 2, 0, 2) \\
&\xrightarrow{x \leftarrow 1} (t_{0,2}, t_{1,5}, 1, 0, 2) \\
&\xrightarrow{y \neq 2} (t_{0,3}, t_{1,5}, 1, 0, 2) \\
&\xrightarrow{y \leftarrow 2} (t_{0,3}, t_{1,6}, 1, 2, 2) \\
&\xrightarrow{z = 2} (t_{0,3}, c_1, 1, 2, 2) \\
&\xrightarrow{z \leftarrow 1} (t_{0,4}, c_1, 1, 2, 1) \\
&\xrightarrow{x = 1} (t_{0,5}, c_1, 1, 2, 1) \\
&\xrightarrow{y \leftarrow 1} (t_{0,6}, c_1, 1, 1, 1) \\
&\xrightarrow{z = 1} (c_0, c_1, 1, 1, 1)
\end{aligned}
$$

(c)
```
// Adapted from Ex. 3(c) of http://spinroot.com/spin/Man/1_Exercises.html
byte x, y, z
byte crit

init
{
  atomic
  {
    x = 0
    y = 0
    z = 0
    run process(0)
    run process(1)
  }
}

proctype process(bit i)
```

```
{
noncritical:
  skip

try:
  x = i + 1

  if
    :: (y != 0 && y != i + 1) -> goto try
    :: else
  fi

  z = i + 1

  if
    :: (x != i + 1) -> goto try
    :: else
  fi

  y = i + 1

  if
    :: (z != i + 1) -> goto try
    :: else
  fi

  atomic // Critical section
  {
    crit++
    assert(crit == 1)
  }

  atomic
  {
    crit--
    goto noncritical
  }
}
```

(d) No, even though it is theoretically possible.

(e) `Spin` finds the following violation:

```
using statement merging
  1:        proc  0 (:init::1) mutex_three.pml:9 (state 1)        [x = 0]
  1:        proc  0 (:init::1) mutex_three.pml:10 (state 2)        [y = 0]
  1:        proc  0 (:init::1) mutex_three.pml:11 (state 3)        [z = 0]
Starting process with pid 1
  2:        proc  0 (:init::1) mutex_three.pml:12 (state 4)        [(run process(0))]
Starting process with pid 2
  3:        proc  0 (:init::1) mutex_three.pml:13 (state 5)        [(run process(1))]
  4:        proc  2 (process:1) mutex_three.pml:20 (state 1)        [(1)]
  5:        proc  1 (process:1) mutex_three.pml:20 (state 1)        [(1)]
  6:        proc  2 (process:1) mutex_three.pml:23 (state 2)        [x = (i+1)]
  7:        proc  2 (process:1) mutex_three.pml:27 (state 5)        [else]
  8:        proc  2 (process:1) mutex_three.pml:30 (state 8)        [z = (i+1)]
  9:        proc  2 (process:1) mutex_three.pml:34 (state 11)        [else]
 10:        proc  1 (process:1) mutex_three.pml:23 (state 2)        [x = (i+1)]
 11:        proc  1 (process:1) mutex_three.pml:27 (state 5)        [else]
```

```
 12:        proc  2 (process:1) mutex_three.pml:37 (state 14)        [y = (i+1)]
 13:        proc  2 (process:1) mutex_three.pml:41 (state 17)        [else]
 14:        proc  2 (process:1) mutex_three.pml:46 (state 20)        [crit = (crit+1)]
 14:        proc  2 (process:1) mutex_three.pml:47 (state 21)        [assert((crit==1))]
 15:        proc  1 (process:1) mutex_three.pml:30 (state 8)        [z = (i+1)]
 16:        proc  1 (process:1) mutex_three.pml:34 (state 11)        [else]
 17:        proc  1 (process:1) mutex_three.pml:37 (state 14)        [y = (i+1)]
 18:        proc  1 (process:1) mutex_three.pml:41 (state 17)        [else]
 19:        proc  1 (process:1) mutex_three.pml:46 (state 20)        [crit = (crit+1)]
spin: mutex_three.pml:47, Error: assertion violated
spin: text of failed assertion: assert((crit==1))

[variable values, step  19]

 crit  =  2
 x  =  1
 y  =  1
 z  =  1
```