Automata and Formal Languages — Exercise Sheet 6

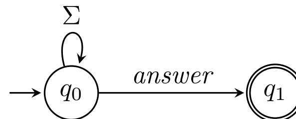
Exercise 6.1
Let $\Sigma = \{request, answer, working, idle\}$.

(1) Build a regular expression and an automaton recognizing all words with the property P_1: for every occurrence of *request* there is a later occurrence of *answer*.

(2) P_1 does not imply that every occurrence of *request* has "its own" *answer*: for instance, the sequence *request request answer* satisfies P_1, but both *request*s must necessarily be mapped to the same *answer*. But, if words were infinite and there were infinitely many *request*s, would P_1 guarantee that every *request* has its own *answer*?
More precisely, let $w = w_1 w_2 \cdots$ satisfying P_1 and containing infinitely many occurrences of *request*, and define $f : \mathbb{N} \to \mathbb{N}$ such that $w_{f(i)}$ is the ith *request* in w. Is there always an injective function $g : \mathbb{N} \to \mathbb{N}$ satisfying $w_{g(i)} = answer$ and $f(i) < g(i)$ for all $i \in \{1, \ldots, k\}$?

(3) Build an automaton recognizing all words with the property P_2: there is an occurrence of *answer* before which only *working* and *request* occur.

(4) Using automata theoretic constructions, prove that all words accepted by the automaton A below satisfy P_1, and give a regular expression for all words accepted by the automaton A that violate P_2.



Exercise 6.2
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 6.3

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 shared variable, 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 6.4

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.

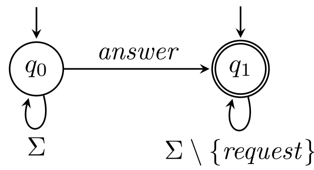
(c) ★ Model the algorithm in Promela.

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

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

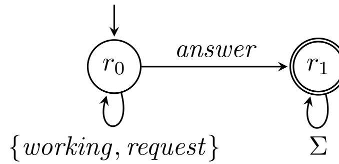
Solution 6.1

(1) A possible regular expression is $(\Sigma^* \mathit{answer})^*(\Sigma \setminus \{\mathit{request}\})^*$. (Observe that we must also describe the sequences containing no occurrence of *request*.) A minimal NFA for the property is

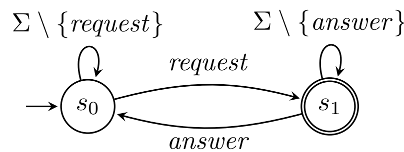


(2) Yes. We define g inductively. Define $g(1)$ as the smallest index such that $w_{g(1)} = \mathit{answer}$ and $g(1) > f(1)$. For every $i > 1$, define $g(i)$ as the smallest index such that $w_{g(i)} = \mathit{answer}$, $g(i) > f(i)$, and $g(i) > g(i-1)$.

(3) A minimal NFA for P_2 is

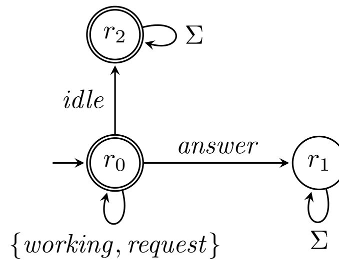


(4) Determinizing and complementing the automaton for P_1 we get

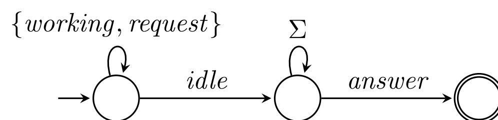


The intersection of A and the automaton for P_1 is empty: indeed, we can only reach a final state of A by executing *request*, while we can only reach a final state of the automaton for P_1 by executing *answer*. So we cannot simultaneously reach final states in both.

For the second half, since the automaton for P_2 is deterministic, we can complement it by exchanging final and non-final states (and not forgetting that the trap state now becomes an accepting state). We get:



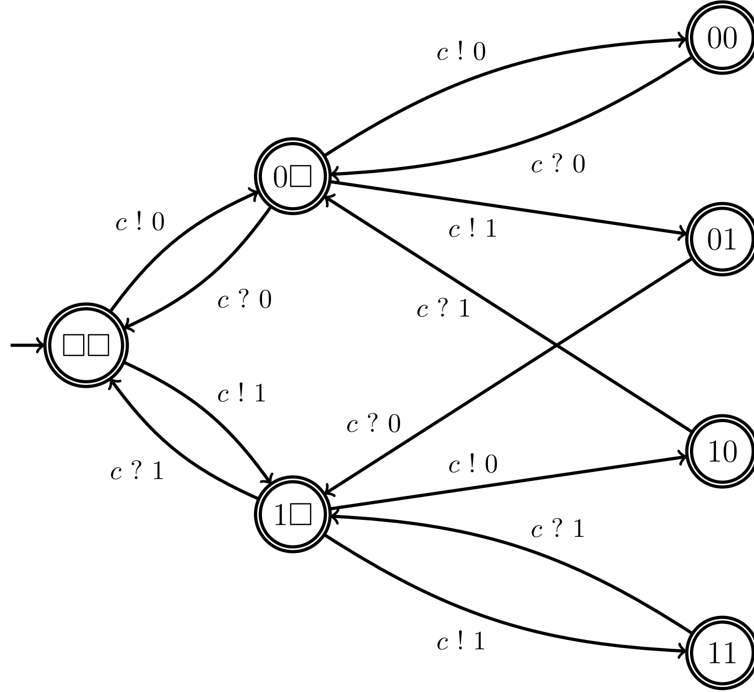
The intersection with A yields



and the regular expression is $(\textit{working} + \textit{request})^* \; \textit{idle} \, \Sigma^* \; \textit{answer}$.

Solution 6.2

(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:

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

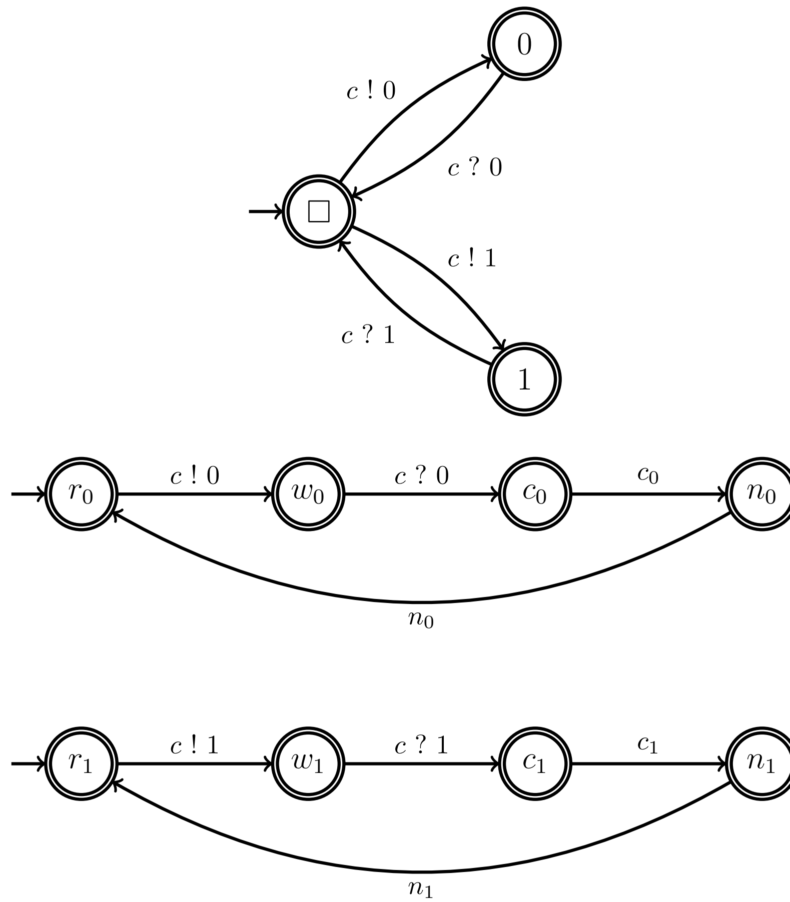
Let $\ell\colon Q \to \{1, 2, \ldots, m\}$ be the function that associates to each state q the position of the last letter of q which is not $\Box$. For example, $\ell(abb\Box\Box) = 3$. The transitions are formally defined as follows:

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

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

★ 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:

data.txt

```
Starting process with pid 1
  1:        proc  0 (:init::1) naive_mutex.pml:8 (state 1)        [(run process(0))]
Starting process with pid 2
  2:        proc  0 (:init::1) naive_mutex.pml:9 (state 2)        [(run process(1))]
  3:        proc  2 (process:1) naive_mutex.pml:16 (state 1)        [c!i]
  4:        proc  2 (process:1) naive_mutex.pml:19 (state 2)        [c?i]
  5:        proc  1 (process:1) naive_mutex.pml:16 (state 1)        [c!i]
  6:        proc  1 (process:1) naive_mutex.pml:19 (state 2)        [c?i]
  7:        proc  1 (process:1) naive_mutex.pml:24 (state 3)        [crit = (crit+1)]
  8:        proc  1 (process:1) naive_mutex.pml:25 (state 4)        [assert((crit==1))]
  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))
#processes: 3
                queue 1 (c):
                crit = 2
 10:        proc  2 (process:1) naive_mutex.pml:25 (state 4)
 10:        proc  1 (process:1) naive_mutex.pml:29 (state 8)
 10:        proc  0 (:init::1) naive_mutex.pml:11 (state 4) <valid end state>
3 processes created
```

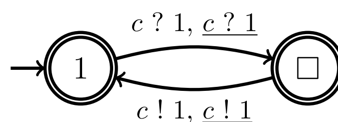
(d)

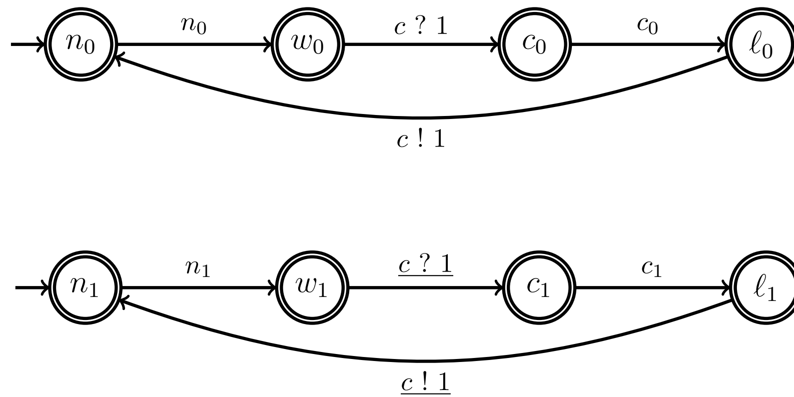
(e) The algorithm violates mutual exclusion since state $(c_0, c_1, \square)$ is reachable in the above automaton.

(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 $\underline{c\ !\ 1}$ and $\underline{c\ ?\ 1}$. 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.

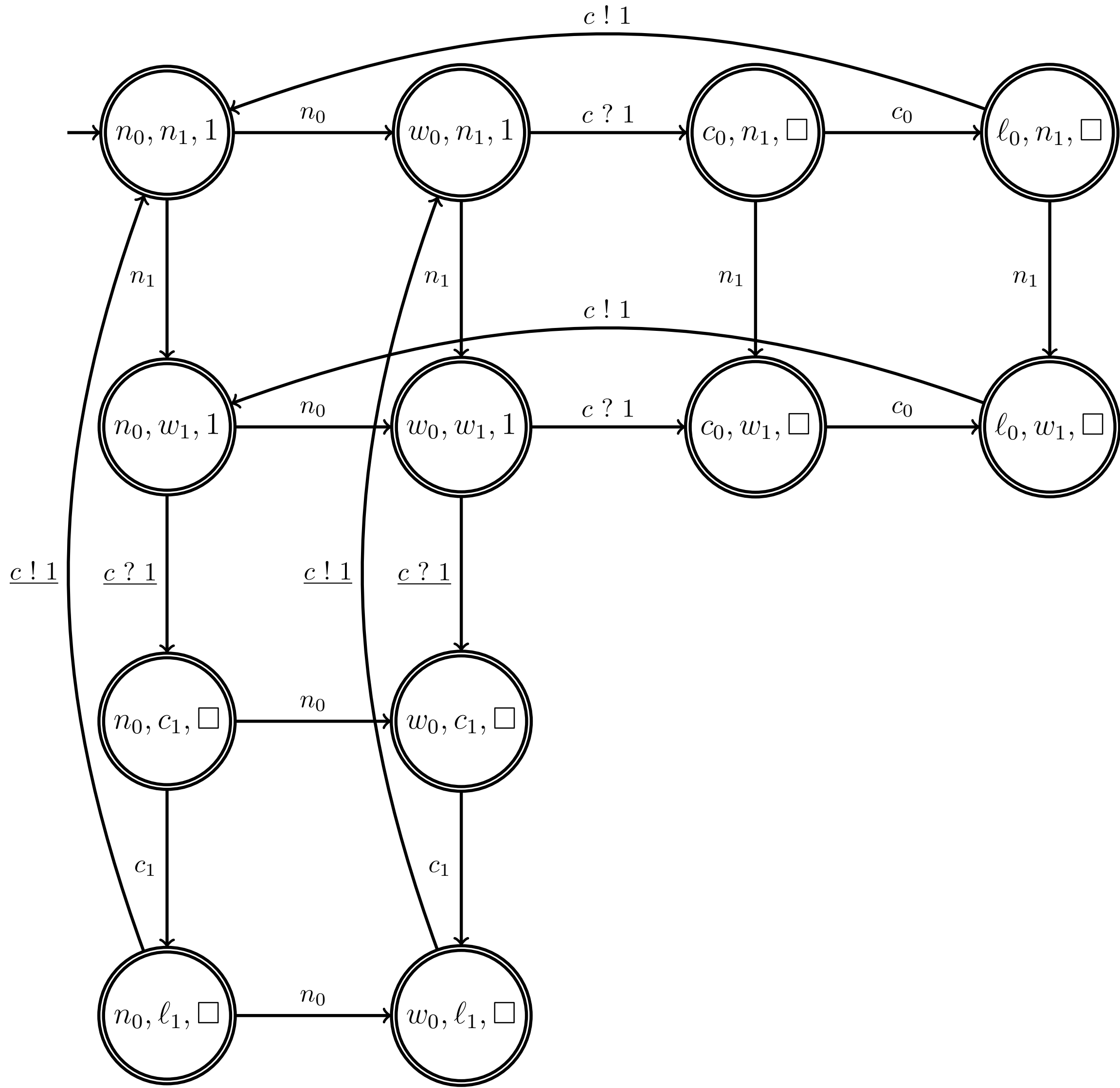
★ In class, we have seen an alternative solution: to simply swap line 4 and 5 of the processes described in #6.2. This also works. You can verify this solution either manually or with `Spin`.

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

data.txt

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

init
{
  atomic
  {
    c ! 1
    run process(0)
    run process(1)
  }
}

proctype process(bit i)
{
noncritical:
  skip

wait:
  c ? 1

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

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

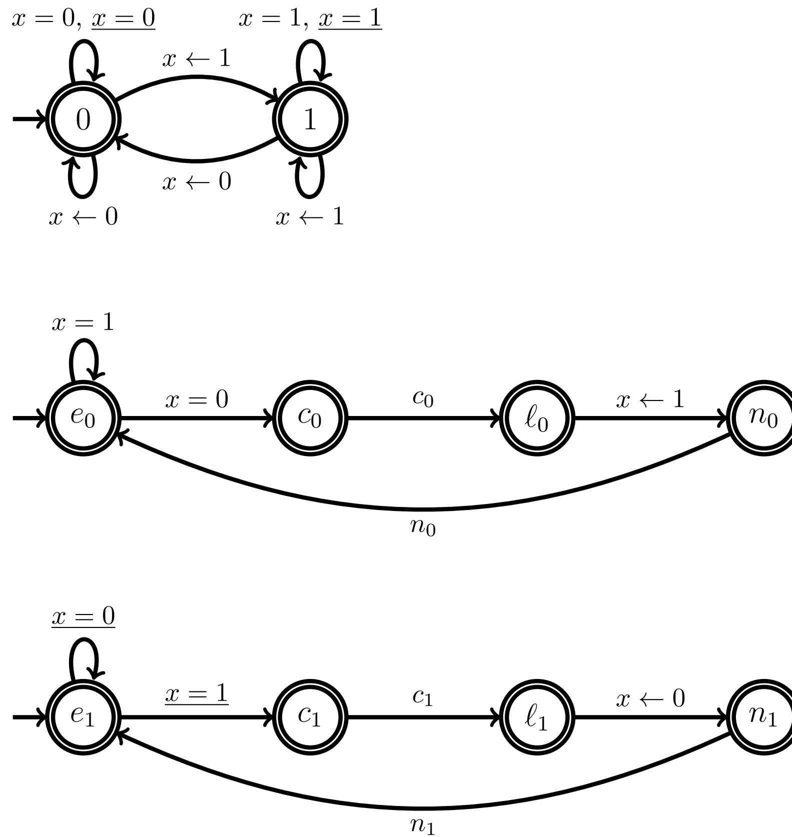
(i)



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

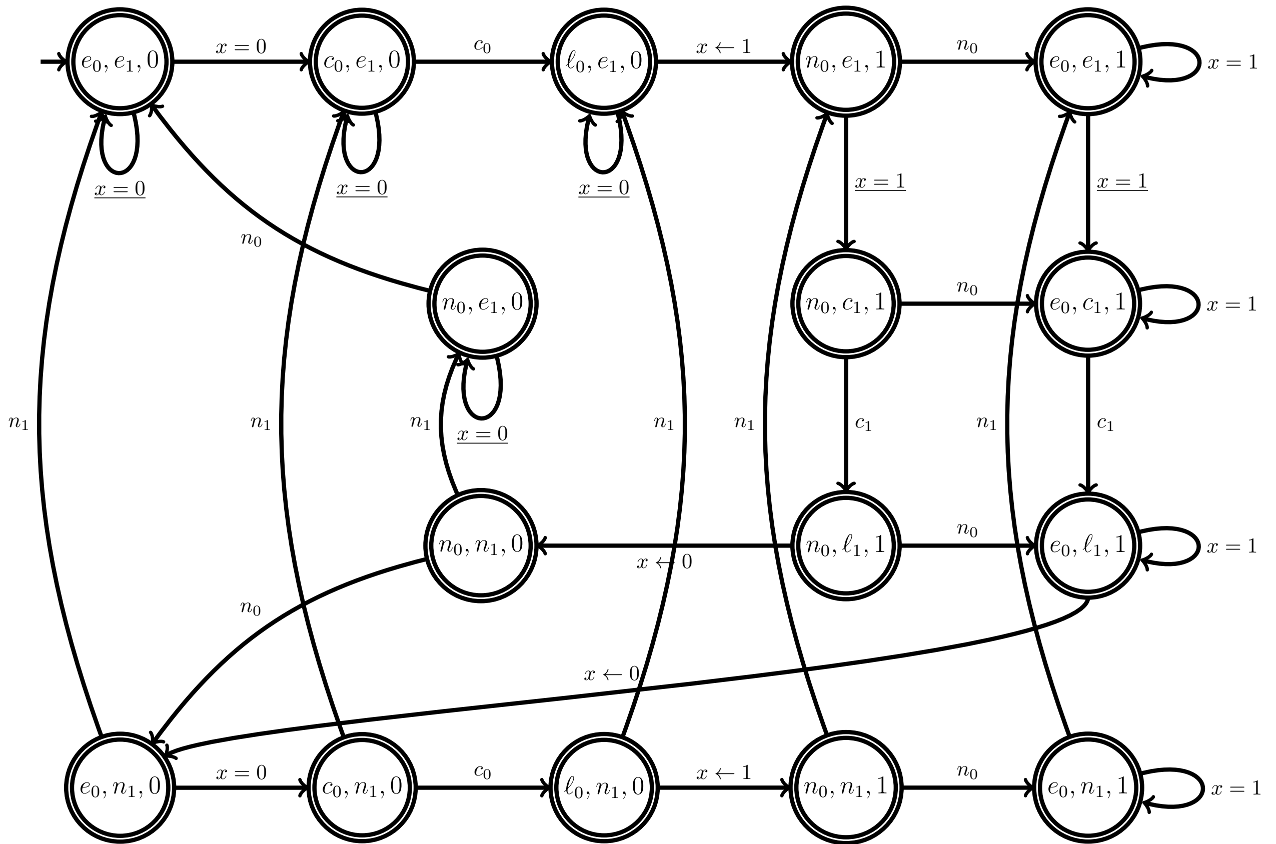
Solution 6.3

(a) The automata modeling the shared variable, `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`.

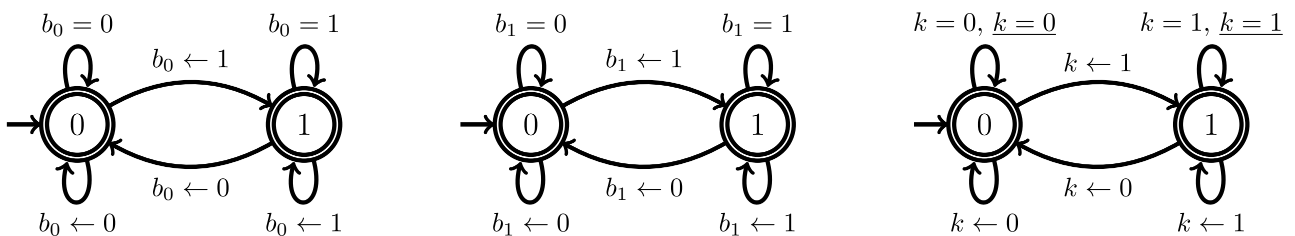
(d) None of the state of the above automaton is of the form (c_0, c_1, σ) where $\sigma \in \{0, 1\}$. This implies that both processes cannot be in their critical sections at the same time.

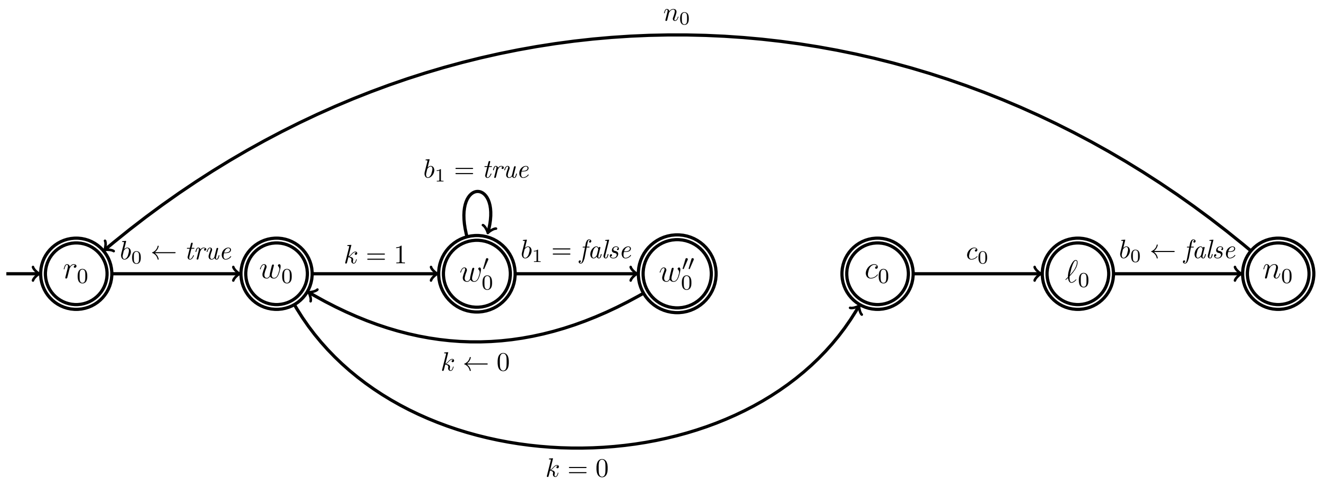
(e) Technically, *no*, since e.g. $(e_0, e_1, 0) \xrightarrow{\underline{x=0}} (e_0, e_1, 0) \xrightarrow{\underline{x=0}} \cdots$ is an infinite execution where both processes never reach their critical section. This kind of behaviour occurs when the scheduler eventually let only one process run.

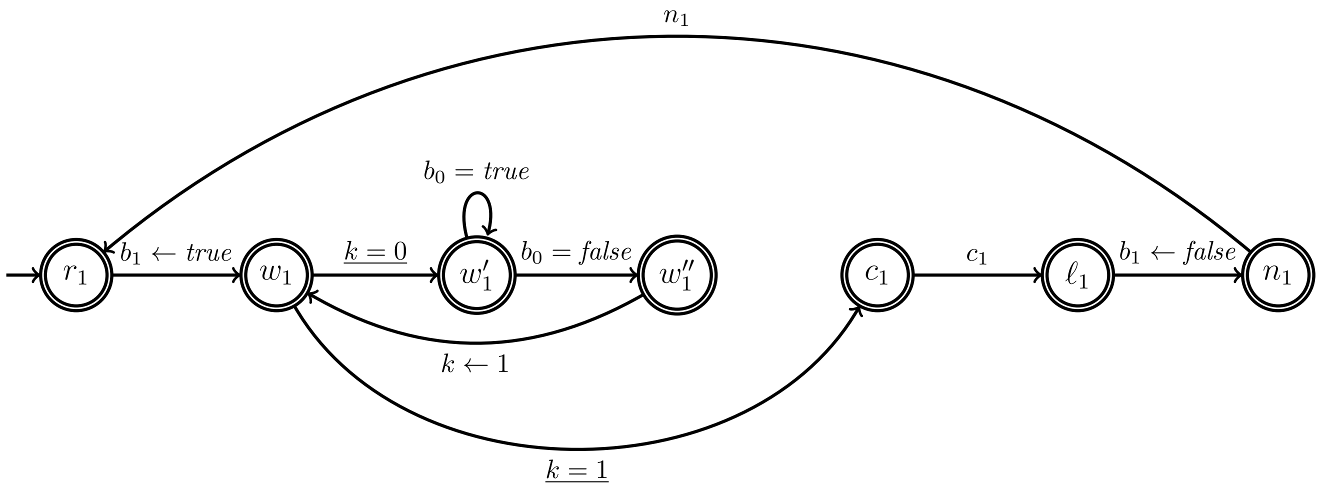
If the scheduler is "fair", then an execution cannot get stuck into any of the self-loops of the automaton. In this case, the answer is *yes* since every cycle of the automaton contains a state of the form $(c_0, \star, \star)$ *and* a state of the form $(\star, c_1, \star)$. Later this semester, we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

Solution 6.4

(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 $(\texttt{process}(0), \texttt{process}(1), b_0, b_1, k)$:

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

data.txt

(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:

data.txt

```
spin: hyman.pml:22, redundant skip
  1:        proc  0 (:init::1) hyman.pml:9 (state 1)        [b[0] = 0]
  2:        proc  0 (:init::1) hyman.pml:10 (state 2)        [b[1] = 0]
  3:        proc  0 (:init::1) hyman.pml:11 (state 3)        [k = 0]
Starting process with pid 1
  4:        proc  0 (:init::1) hyman.pml:12 (state 4)        [(run process(0))]
Starting process with pid 2
  5:        proc  0 (:init::1) hyman.pml:13 (state 5)        [(run process(1))]
  6:        proc  2 (process:1) hyman.pml:20 (state 1)        [(1)]
  7:        proc  2 (process:1) hyman.pml:24 (state 3)        [b[i] = 1]
  8:        proc  2 (process:1) hyman.pml:37 (state 16)        [.(goto)]
  9:        proc  1 (process:1) hyman.pml:20 (state 1)        [(1)]
 10:        proc  1 (process:1) hyman.pml:24 (state 3)        [b[i] = 1]
 11:        proc  2 (process:1) hyman.pml:27 (state 4)        [((k!=i))]
 12:        proc  2 (process:1) hyman.pml:32 (state 10)        [.(goto)]
 13:        proc  1 (process:1) hyman.pml:37 (state 16)        [.(goto)]
 14:        proc  1 (process:1) hyman.pml:33 (state 13)        [else]
 15:        proc  2 (process:1) hyman.pml:29 (state 5)        [(b[(1-i)])]
 16:        proc  1 (process:1) hyman.pml:33 (state 14)        [goto :b1]
 17:        proc  2 (process:1) hyman.pml:29 (state 6)        [(1)]
 18:        proc  1 (process:1) hyman.pml:39 (state 18)        [crit = (crit+1)]
 19:        proc  1 (process:1) hyman.pml:40 (state 19)        [assert((crit==1))]
 20:        proc  1 (process:1) hyman.pml:45 (state 21)        [crit = (crit-1)]
 21:        proc  1 (process:1) hyman.pml:46 (state 22)        [b[i] = 0]
 22:        proc  2 (process:1) hyman.pml:32 (state 10)        [.(goto)]
 23:        proc  1 (process:1) hyman.pml:49 (state 25)        [.(goto)]
 24:        proc  1 (process:1) hyman.pml:20 (state 1)        [(1)]
 25:        proc  1 (process:1) hyman.pml:24 (state 3)        [b[i] = 1]
 26:        proc  1 (process:1) hyman.pml:37 (state 16)        [.(goto)]
 27:        proc  2 (process:1) hyman.pml:29 (state 5)        [(b[(1-i)])]
 28:        proc  2 (process:1) hyman.pml:29 (state 6)        [(1)]
 29:        proc  2 (process:1) hyman.pml:32 (state 10)        [.(goto)]
 30:        proc  1 (process:1) hyman.pml:33 (state 13)        [else]
 31:        proc  2 (process:1) hyman.pml:29 (state 5)        [(b[(1-i)])]
 32:        proc  1 (process:1) hyman.pml:33 (state 14)        [goto :b1]
 33:        proc  1 (process:1) hyman.pml:39 (state 18)        [crit = (crit+1)]
 34:        proc  1 (process:1) hyman.pml:40 (state 19)        [assert((crit==1))]
 35:        proc  2 (process:1) hyman.pml:29 (state 6)        [(1)]
 36:        proc  2 (process:1) hyman.pml:32 (state 10)        [.(goto)]
 37:        proc  2 (process:1) hyman.pml:29 (state 5)        [(b[(1-i)])]
 38:        proc  2 (process:1) hyman.pml:29 (state 6)        [(1)]
 39:        proc  2 (process:1) hyman.pml:32 (state 10)        [.(goto)]
 40:        proc  2 (process:1) hyman.pml:29 (state 5)        [(b[(1-i)])]
 41:        proc  1 (process:1) hyman.pml:45 (state 21)        [crit = (crit-1)]
 42:        proc  1 (process:1) hyman.pml:46 (state 22)        [b[i] = 0]
 43:        proc  2 (process:1) hyman.pml:29 (state 6)        [(1)]
 44:        proc  2 (process:1) hyman.pml:32 (state 10)        [.(goto)]
 45:        proc  1 (process:1) hyman.pml:49 (state 25)        [.(goto)]
 46:        proc  2 (process:1) hyman.pml:30 (state 7)        [else]
 47:        proc  2 (process:1) hyman.pml:30 (state 8)        [goto :b2]
 48:        proc  2 (process:1) hyman.pml:32 (state 12)        [k = i]
 49:        proc  2 (process:1) hyman.pml:37 (state 16)        [.(goto)]
 50:        proc  1 (process:1) hyman.pml:20 (state 1)        [(1)]
 51:        proc  1 (process:1) hyman.pml:24 (state 3)        [b[i] = 1]
 52:        proc  2 (process:1) hyman.pml:33 (state 13)        [else]
 53:        proc  1 (process:1) hyman.pml:37 (state 16)        [.(goto)]
 54:        proc  1 (process:1) hyman.pml:27 (state 4)        [((k!=i))]
 55:        proc  2 (process:1) hyman.pml:33 (state 14)        [goto :b1]
 56:        proc  2 (process:1) hyman.pml:39 (state 18)        [crit = (crit+1)]
 57:        proc  2 (process:1) hyman.pml:40 (state 19)        [assert((crit==1))]
 58:        proc  1 (process:1) hyman.pml:32 (state 10)        [.(goto)]
 59:        proc  2 (process:1) hyman.pml:45 (state 21)        [crit = (crit-1)]
 60:        proc  2 (process:1) hyman.pml:46 (state 22)        [b[i] = 0]
 61:        proc  2 (process:1) hyman.pml:49 (state 25)        [.(goto)]
 62:        proc  2 (process:1) hyman.pml:20 (state 1)        [(1)]
 63:        proc  1 (process:1) hyman.pml:30 (state 7)        [else]
 64:        proc  2 (process:1) hyman.pml:24 (state 3)        [b[i] = 1]
 65:        proc  1 (process:1) hyman.pml:30 (state 8)        [goto :b2]
 66:        proc  1 (process:1) hyman.pml:32 (state 12)        [k = i]
 67:        proc  2 (process:1) hyman.pml:37 (state 16)        [.(goto)]
```

```
 68:        proc  2 (process:1) hyman.pml:27 (state 4)        [((k!=i))]
 69:        proc  1 (process:1) hyman.pml:37 (state 16)        [.(goto)]
 70:        proc  2 (process:1) hyman.pml:32 (state 10)        [.(goto)]
 71:        proc  2 (process:1) hyman.pml:29 (state 5)        [(b[(1-i)])]
 72:        proc  2 (process:1) hyman.pml:29 (state 6)        [(1)]
 73:        proc  2 (process:1) hyman.pml:32 (state 10)        [.(goto)]
 74:        proc  1 (process:1) hyman.pml:33 (state 13)        [else]
 75:        proc  2 (process:1) hyman.pml:29 (state 5)        [(b[(1-i)])]
 76:        proc  1 (process:1) hyman.pml:33 (state 14)        [goto :b1]
 77:        proc  1 (process:1) hyman.pml:39 (state 18)        [crit = (crit+1)]
 78:        proc  1 (process:1) hyman.pml:40 (state 19)        [assert((crit==1))]
 79:        proc  2 (process:1) hyman.pml:29 (state 6)        [(1)]
 80:        proc  1 (process:1) hyman.pml:45 (state 21)        [crit = (crit-1)]
 81:        proc  1 (process:1) hyman.pml:46 (state 22)        [b[i] = 0]
 82:        proc  2 (process:1) hyman.pml:32 (state 10)        [.(goto)]
 83:        proc  1 (process:1) hyman.pml:49 (state 25)        [.(goto)]
 84:        proc  1 (process:1) hyman.pml:20 (state 1)        [(1)]
 85:        proc  1 (process:1) hyman.pml:24 (state 3)        [b[i] = 1]
 86:        proc  1 (process:1) hyman.pml:37 (state 16)        [.(goto)]
 87:        proc  2 (process:1) hyman.pml:29 (state 5)        [(b[(1-i)])]
 88:        proc  2 (process:1) hyman.pml:29 (state 6)        [(1)]
 89:        proc  1 (process:1) hyman.pml:33 (state 13)        [else]
 90:        proc  2 (process:1) hyman.pml:32 (state 10)        [.(goto)]
 91:        proc  1 (process:1) hyman.pml:33 (state 14)        [goto :b1]
 92:        proc  2 (process:1) hyman.pml:29 (state 5)        [(b[(1-i)])]
 93:        proc  1 (process:1) hyman.pml:39 (state 18)        [crit = (crit+1)]
 94:        proc  1 (process:1) hyman.pml:40 (state 19)        [assert((crit==1))]
 95:        proc  2 (process:1) hyman.pml:29 (state 6)        [(1)]
 96:        proc  1 (process:1) hyman.pml:45 (state 21)        [crit = (crit-1)]
 97:        proc  1 (process:1) hyman.pml:46 (state 22)        [b[i] = 0]
 98:        proc  1 (process:1) hyman.pml:49 (state 25)        [.(goto)]
 99:        proc  1 (process:1) hyman.pml:20 (state 1)        [(1)]
100:        proc  2 (process:1) hyman.pml:32 (state 10)        [.(goto)]
101:        proc  2 (process:1) hyman.pml:30 (state 7)        [else]
102:        proc  1 (process:1) hyman.pml:24 (state 3)        [b[i] = 1]
103:        proc  2 (process:1) hyman.pml:30 (state 8)        [goto :b2]
104:        proc  2 (process:1) hyman.pml:32 (state 12)        [k = i]
105:        proc  2 (process:1) hyman.pml:37 (state 16)        [.(goto)]
106:        proc  1 (process:1) hyman.pml:37 (state 16)        [.(goto)]
107:        proc  2 (process:1) hyman.pml:33 (state 13)        [else]
108:        proc  1 (process:1) hyman.pml:27 (state 4)        [((k!=i))]
109:        proc  2 (process:1) hyman.pml:33 (state 14)        [goto :b1]
110:        proc  2 (process:1) hyman.pml:39 (state 18)        [crit = (crit+1)]
111:        proc  2 (process:1) hyman.pml:40 (state 19)        [assert((crit==1))]
112:        proc  1 (process:1) hyman.pml:32 (state 10)        [.(goto)]
113:        proc  1 (process:1) hyman.pml:29 (state 5)        [(b[(1-i)])]
114:        proc  2 (process:1) hyman.pml:45 (state 21)        [crit = (crit-1)]
115:        proc  2 (process:1) hyman.pml:46 (state 22)        [b[i] = 0]
116:        proc  1 (process:1) hyman.pml:29 (state 6)        [(1)]
117:        proc  2 (process:1) hyman.pml:49 (state 25)        [.(goto)]
118:        proc  1 (process:1) hyman.pml:32 (state 10)        [.(goto)]
119:        proc  2 (process:1) hyman.pml:20 (state 1)        [(1)]
120:        proc  2 (process:1) hyman.pml:24 (state 3)        [b[i] = 1]
121:        proc  1 (process:1) hyman.pml:29 (state 5)        [(b[(1-i)])]
122:        proc  1 (process:1) hyman.pml:29 (state 6)        [(1)]
123:        proc  1 (process:1) hyman.pml:32 (state 10)        [.(goto)]
124:        proc  1 (process:1) hyman.pml:29 (state 5)        [(b[(1-i)])]
125:        proc  2 (process:1) hyman.pml:37 (state 16)        [.(goto)]
126:        proc  2 (process:1) hyman.pml:33 (state 13)        [else]
127:        proc  2 (process:1) hyman.pml:33 (state 14)        [goto :b1]
128:        proc  2 (process:1) hyman.pml:39 (state 18)        [crit = (crit+1)]
129:        proc  2 (process:1) hyman.pml:40 (state 19)        [assert((crit==1))]
130:        proc  1 (process:1) hyman.pml:29 (state 6)        [(1)]
131:        proc  2 (process:1) hyman.pml:45 (state 21)        [crit = (crit-1)]
132:        proc  2 (process:1) hyman.pml:46 (state 22)        [b[i] = 0]
133:        proc  2 (process:1) hyman.pml:49 (state 25)        [.(goto)]
134:        proc  1 (process:1) hyman.pml:32 (state 10)        [.(goto)]
135:        proc  1 (process:1) hyman.pml:30 (state 7)        [else]
136:        proc  1 (process:1) hyman.pml:30 (state 8)        [goto :b2]
137:        proc  2 (process:1) hyman.pml:20 (state 1)        [(1)]
138:        proc  1 (process:1) hyman.pml:32 (state 12)        [k = i]
139:        proc  1 (process:1) hyman.pml:37 (state 16)        [.(goto)]
140:        proc  2 (process:1) hyman.pml:24 (state 3)        [b[i] = 1]
141:        proc  1 (process:1) hyman.pml:33 (state 13)        [else]
142:        proc  1 (process:1) hyman.pml:33 (state 14)        [goto :b1]
```

```
143:        proc  1 (process:1) hyman.pml:39 (state 18)        [crit = (crit+1)]
144:        proc  1 (process:1) hyman.pml:40 (state 19)        [assert((crit==1))]
145:        proc  2 (process:1) hyman.pml:37 (state 16)        [.(goto)]
146:        proc  1 (process:1) hyman.pml:45 (state 21)        [crit = (crit-1)]
147:        proc  1 (process:1) hyman.pml:46 (state 22)        [b[i] = 0]
148:        proc  1 (process:1) hyman.pml:49 (state 25)        [.(goto)]
149:        proc  1 (process:1) hyman.pml:20 (state 1)        [(1)]
150:        proc  2 (process:1) hyman.pml:27 (state 4)        [((k!=i))]
151:        proc  2 (process:1) hyman.pml:32 (state 10)        [.(goto)]
152:        proc  2 (process:1) hyman.pml:30 (state 7)        [else]
153:        proc  2 (process:1) hyman.pml:30 (state 8)        [goto :b2]
154:        proc  2 (process:1) hyman.pml:32 (state 12)        [k = i]
155:        proc  1 (process:1) hyman.pml:24 (state 3)        [b[i] = 1]
156:        proc  1 (process:1) hyman.pml:37 (state 16)        [.(goto)]
157:        proc  1 (process:1) hyman.pml:27 (state 4)        [((k!=i))]
158:        proc  2 (process:1) hyman.pml:37 (state 16)        [.(goto)]
159:        proc  1 (process:1) hyman.pml:32 (state 10)        [.(goto)]
160:        proc  2 (process:1) hyman.pml:33 (state 13)        [else]
161:        proc  2 (process:1) hyman.pml:33 (state 14)        [goto :b1]
162:        proc  2 (process:1) hyman.pml:39 (state 18)        [crit = (crit+1)]
163:        proc  2 (process:1) hyman.pml:40 (state 19)        [assert((crit==1))]
164:        proc  2 (process:1) hyman.pml:45 (state 21)        [crit = (crit-1)]
165:        proc  2 (process:1) hyman.pml:46 (state 22)        [b[i] = 0]
166:        proc  1 (process:1) hyman.pml:30 (state 7)        [else]
167:        proc  2 (process:1) hyman.pml:49 (state 25)        [.(goto)]
168:        proc  2 (process:1) hyman.pml:20 (state 1)        [(1)]
169:        proc  2 (process:1) hyman.pml:24 (state 3)        [b[i] = 1]
170:        proc  1 (process:1) hyman.pml:30 (state 8)        [goto :b2]
171:        proc  2 (process:1) hyman.pml:37 (state 16)        [.(goto)]
172:        proc  2 (process:1) hyman.pml:33 (state 13)        [else]
173:        proc  2 (process:1) hyman.pml:33 (state 14)        [goto :b1]
174:        proc  1 (process:1) hyman.pml:32 (state 12)        [k = i]
175:        proc  1 (process:1) hyman.pml:37 (state 16)        [.(goto)]
176:        proc  1 (process:1) hyman.pml:33 (state 13)        [else]
177:        proc  1 (process:1) hyman.pml:33 (state 14)        [goto :b1]
178:        proc  2 (process:1) hyman.pml:39 (state 18)        [crit = (crit+1)]
179:        proc  2 (process:1) hyman.pml:40 (state 19)        [assert((crit==1))]
180:        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))
#processes: 3
                b[0] = 1
                b[1] = 1
                k = 0
                crit = 2
181:        proc  2 (process:1) hyman.pml:43 (state 23)
181:        proc  1 (process:1) hyman.pml:40 (state 19)
181:        proc  0 (:init::1) hyman.pml:15 (state 7) <valid end state>
3 processes created
```