Automata and Formal Languages — Exercise Sheet 6

Exercise 6.1

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

(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 requests must necessarily be mapped to the same answer. But, if words were infinite and there were infinitely many requests, would $P_1$ guarantee that every request has its own answer?
More precisely, let $w = w_1w_2\cdots$ 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 $i$th request in $w$. Is there always an injective function $g : \mathbb{N} \to \mathbb{N}$ satisfying $w_{g(i)} = \text{answer}$ 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 $P_2$.

![Automaton Diagram]

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 $\Sigma$. 
(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          \( x \leftarrow 1 - i \)
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.

```c
1 process(i):
2   while true do
3      $b_i$ ← true
4      while $k \neq i$ 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 $(\Sigma^{*} \text{answer})^*(\Sigma \setminus \{\text{request}\})^*$. (Observe that we must also describe the sequences containing no occurrence of \text{request}.) A minimal NFA for the property is

\begin{center}
\begin{tikzpicture}
  \node[state, initial, accepting] (q0) at (0,0) {$q_0$};
  \node[state, accepting] (q1) at (1,0) {$q_1$};
  \draw (q0) edge[above] node {$\text{answer}$} (q1);
  \draw (q1) edge[below] node {$\Sigma \setminus \{\text{request}\}$} (q0);
  \draw (q0) edge[loop above] node {$\Sigma$} (q0);
  \draw (q1) edge[loop above] node {$\Sigma \setminus \{\text{request}\}$} (q1);
\end{tikzpicture}
\end{center}

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

(3) A minimal NFA for $P_2$ is

\begin{center}
\begin{tikzpicture}
  \node[state, initial] (r0) at (0,0) {$r_0$};
  \node[state, accepting] (r1) at (1,0) {$r_1$};
  \node[state] (r2) at (0.5,-1) {};\node[state] (r3) at (1.5,-1) {};
  \draw (r0) edge[above] node {$\text{answer}$} (r1);
  \draw (r0) edge[-loop below] node {$\{\text{working, request}\}$} (r0);
  \draw (r1) edge[loop below] node {$\Sigma$} (r1);
  \draw (r2) edge[above, bend left] node {$\text{idle}$} (r0);
  \draw (r2) edge[below, bend left] node {$\Sigma$} (r3);
  \draw (r3) edge[above, bend left] node {$\text{request}$} (r0);
  \draw (r3) edge[below, bend left] node {$\Sigma \setminus \{\text{answer}\}$} (r1);
\end{tikzpicture}
\end{center}

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

\begin{center}
\begin{tikzpicture}
  \node[state, initial] (s0) at (0,0) {$s_0$};
  \node[state, accepting] (s1) at (1,0) {$s_1$};
  \draw (s0) edge[loop above] node {$\Sigma \setminus \{\text{request}\}$} (s0);
  \draw (s0) edge[above] node {$\text{request}$} (s1);
  \draw (s1) edge[below] node {$\Sigma \setminus \{\text{answer}\}$} (s0);
\end{tikzpicture}
\end{center}

The intersection of $A$ and the automaton for $P_1$ is empty: indeed, we can only reach a final state of $A$ by executing $\text{request}$, while we can only reach a final state of the automaton for $P_1$ by executing $\text{answer}$. 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:

\begin{center}
\begin{tikzpicture}
  \node[state, initial] (r2) at (0,0) {$r_2$};
  \node[state, accepting] (r0) at (0.5,-1) {};\node[state] (r1) at (1.5,-1) {};
  \draw (r2) edge[above] node {$\Sigma$} (r0);
  \draw (r2) edge[below, bend left] node {$\text{idle}$} (r0);
  \draw (r0) edge[above] node {$\text{answer}$} (r1);
  \draw (r0) edge[below, bend left] node {$\{\text{working, request}\}$} (r3);
  \draw (r1) edge[loop above] node {$\Sigma$} (r1);
  \draw (r3) edge[above, bend left] node {$\text{request}$} (r0);
  \draw (r3) edge[below, bend left] node {$\Sigma \setminus \{\text{answer}\}$} (r1);
\end{tikzpicture}
\end{center}

The intersection with $A$ yields

\begin{center}
\begin{tikzpicture}
  \node[state, initial] (r2) at (0,0) {$\emptyset$};
  \node[state, accepting] (r0) at (0.5,-1) {};\node[state] (r1) at (1.5,-1) {};
  \draw (r2) edge[above] node {$\Sigma$} (r0);
  \draw (r2) edge[below, bend left] node {$\text{idle}$} (r0);
  \draw (r0) edge[above] node {$\{\text{working, request}\}$} (r3);
  \draw (r0) edge[below, bend left] node {$\Sigma$} (r1);
  \draw (r1) edge[loop above] node {$\text{answer}$} (r1);
\end{tikzpicture}
\end{center}
and the regular expression is \((working + request)^* \text{ idle } \Sigma^* \text{ answer}\).

**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:

![Automaton Diagram]

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

\[
Q = \{w \in (\Sigma \cup \square)^m : (w_i = \square) \implies (w_{i+1} = \square) \text{ for every } 1 \leq i < m\},
\]

\[
\Gamma = \{c! \sigma : \sigma \in \Sigma\} \cup \{c? \sigma : \sigma \in \Sigma\},
\]

\[
q_0 = \square^m,
\]

\[
F = Q.
\]

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

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

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

\(\star\) 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, \texttt{process(0)} and \texttt{process(1)} are respectively:
(c) A simulation quickly finds a problem with the algorithm:

```
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))] spin: naive_mutex.pml:25, Error: assertion violated
9: proc 2 (process:1) naive_mutex.pml:24 (state 3) [crit = (crit+1)]

#processes: 3
queue 1 (c):
crit = 2

spin: text of failed assertion: assert((crit==1))

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
```
(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:

```plaintext
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 $c!1$ and $c?1$. 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
        }
}
```
(j) None of the state of the above automaton is of the form $(c_0, c_1, \sigma)$ where $\sigma \in \{\square, 1\}$. 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, \texttt{process(0)} and \texttt{process(1)} are respectively:

\begin{align*}
&x = 0, x = 0 \quad x = 1, x = 1 \\
&e_0 \quad c_0 \quad \ell_0 \quad x \leftarrow 1
\end{align*}

\begin{align*}
&x = 1 \\
&c_0 \quad \ell_0
\end{align*}

\begin{align*}
&x = 0 \\
&c_1 \quad \ell_1
\end{align*}

\begin{align*}
&x \leftarrow 0 \\
&x \leftarrow 1
\end{align*}

\begin{center}
\begin{tikzpicture}
  \node[state] (0) at (0,0) {$0$};
  \node[state] (1) at (1,0) {$1$};
  \node[state] (e0) at (-1,-1) {$e_0$};
  \node[state] (c0) at (-1,-2) {$c_0$};
  \node[state] (ell0) at (0,-2) {$\ell_0$};
  \node[state] (n0) at (1,-2) {$n_0$};
  \node[state] (e1) at (-1,-3) {$e_1$};
  \node[state] (c1) at (-1,-4) {$c_1$};
  \node[state] (ell1) at (0,-4) {$\ell_1$};
  \node[state] (n1) at (1,-4) {$n_1$};
  \draw[->] (0) edge[bend left] node[above] {$x \leftarrow 1$} (1);
  \draw[->] (1) edge[bend left] node[above] {$x \leftarrow 0$} (0);
  \draw[->] (0) edge node[left] {$x \leftarrow 0$} (e0);
  \draw[->] (e0) edge node[below] {$x = 0$} (c0);
  \draw[->] (c0) edge node[below] {$c_0$} (ell0);
  \draw[->] (ell0) edge node[below] {$x \leftarrow 1$} (n0);
  \draw[->] (e1) edge node[left] {$x \leftarrow 1$} (c1);
  \draw[->] (c1) edge node[below] {$c_1$} (ell1);
  \draw[->] (ell1) edge node[below] {$x \leftarrow 0$} (n1);
\end{tikzpicture}
\end{center}

\begin{itemize}
  \item Note that we have introduced the new letters $x = 0$ and $x = 1$. 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.
\end{itemize}
(c) Spin successfully verifies mutex.pml.

(d) None of the state of the above automaton is of the form \((c_0, c_1, \sigma)\) 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. \((e_0, e_1, 0) \xrightarrow{\sigma=0} (e_0, e_1, 0) \xrightarrow{\sigma=0} \cdots\) is an infinite execution where both processes never reach their critical section. This kind of behaviour occurs when the scheduler eventually lets 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 \((c_0, *, *)\) and a state of the form \((*, c_1, *)\). 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 \((\text{process}(0), \text{process}(1), b_0, b_1, k)\):

\[
\begin{align*}
(r_0, r_1, false, false, 0) & \xrightarrow{b_1 = true} (r_0, w_1, false, true, 0) \\
& \xrightarrow{k = 0} (r_0, w_1', false, true, 0) \\
& \xrightarrow{b_0 = false} (r_0, w_1'', false, true, 0) \\
& \xrightarrow{b_0 = true} (w_0, w_1'', true, true, 0) \\
& \xrightarrow{k = 0} (c_0, w_1'', true, true, 0) \\
& \xrightarrow{k = 1} (c_0, w_1, true, true, 1) \\
& \xrightarrow{c_0 = 1} (c_0, c_1, true, true, 1).
\end{align*}
\]
(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.
Spin finds the following violation:

Starting process with pid 1
proc 0 (:init::1) hyman.pml:12 (state 4) [(run process(0))]

Starting process with pid 2
proc 0 (:init::1) hyman.pml:13 (state 5) [(run process(1))]

proc 2 (process:1) hyman.pml:20 (state 1) [(1)]
proc 2 (process:1) hyman.pml:24 (state 3) [b[i] = 1]
proc 2 (process:1) hyman.pml:27 (state 4) [((k!=i))]
proc 2 (process:1) hyman.pml:32 (state 10) [(goto)]
proc 2 (process:1) hyman.pml:33 (state 13) [else]
proc 2 (process:1) hyman.pml:37 (state 16) [(goto)]
proc 1 (process:1) hyman.pml:20 (state 1) [(1)]
proc 1 (process:1) hyman.pml:24 (state 3) [b[i] = 1]
proc 2 (process:1) hyman.pml:27 (state 4) [((k!=i))]
proc 1 (process:1) hyman.pml:32 (state 10) [(goto)]
proc 2 (process:1) hyman.pml:33 (state 13) [else]
proc 2 (process:1) hyman.pml:37 (state 16) [(goto)]
proc 1 (process:1) hyman.pml:30 (state 7) [else]
proc 2 (process:1) hyman.pml:32 (state 10) [(goto)]
proc 1 (process:1) hyman.pml:30 (state 8) [goto :b2]
proc 2 (process:1) hyman.pml:32 (state 12) [k = i]
proc 2 (process:1) hyman.pml:37 (state 16) [(goto)]
proc 1 (process:1) hyman.pml:20 (state 1) [(1)]
proc 1 (process:1) hyman.pml:24 (state 3) [b[i] = 1]
proc 1 (process:1) hyman.pml:30 (state 8) [goto :b2]
proc 2 (process:1) hyman.pml:32 (state 12) [k = i]
proc 2 (process:1) hyman.pml:37 (state 16) [(goto)]
proc 2 (process:1) hyman.pml:24 (state 3) [b[i] = 1]
proc 1 (process:1) hyman.pml:30 (state 8) [goto :b2]
proc 1 (process:1) hyman.pml:32 (state 12) [k = i]
proc 1 (process:1) hyman.pml:37 (state 16) [(goto)]
proc 2 (process:1) hyman.pml:24 (state 3) [b[i] = 1]
proc 1 (process:1) hyman.pml:30 (state 8) [goto :b2]
proc 1 (process:1) hyman.pml:32 (state 12) [k = i]
proc 1 (process:1) hyman.pml:37 (state 16) [(goto)]
proc 2 (process:1) hyman.pml:27 (state 4) \[((k!=i))\]
proc 1 (process:1) hyman.pml:37 (state 16) \[.(goto)\]
proc 2 (process:1) hyman.pml:32 (state 10) \[.(goto)\]
proc 2 (process:1) hyman.pml:29 (state 5) \[b[(1-i)]\]
proc 2 (process:1) hyman.pml:29 (state 6) \[(i)\]
proc 3 (process:1) hyman.pml:32 (state 10) \[.(goto)\]
proc 1 (process:1) hyman.pml:33 (state 13) \[else\]
proc 2 (process:1) hyman.pml:29 (state 5) \[b[(1-i)]\]
proc 1 (process:1) hyman.pml:33 (state 13) \[else\]
proc 1 (process:1) hyman.pml:33 (state 14) \[goto :b1\]
proc 1 (process:1) hyman.pml:39 (state 19) \[assert((crit==1))\]
proc 2 (process:1) hyman.pml:29 (state 6) \[(i)\]
proc 1 (process:1) hyman.pml:39 (state 18) \[crit = (crit+1)\]
proc 1 (process:1) hyman.pml:40 (state 19) \[assert((crit==1))\]
proc 2 (process:1) hyman.pml:29 (state 5) \[b[(1-i)]\]
proc 1 (process:1) hyman.pml:45 (state 21) \[b[i] = 0\]
proc 2 (process:1) hyman.pml:37 (state 16) \[.(goto)\]
proc 1 (process:1) hyman.pml:40 (state 19) \[assert((crit==1))\]
proc 2 (process:1) hyman.pml:33 (state 14) \[goto :b1\]
proc 2 (process:1) hyman.pml:32 (state 10) \[(i)\]
proc 2 (process:1) hyman.pml:39 (state 19) \[crit = (crit+1)\]
proc 1 (process:1) hyman.pml:33 (state 13) \[else\]
proc 1 (process:1) hyman.pml:27 (state 4) \[((k!=i))\]
proc 1 (process:1) hyman.pml:37 (state 16) \[.(goto)\]
proc 1 (process:1) hyman.pml:32 (state 10) \[.(goto)\]
proc 2 (process:1) hyman.pml:29 (state 5) \[b[(1-i)]\]
proc 1 (process:1) hyman.pml:33 (state 14) \[goto :b1\]
proc 2 (process:1) hyman.pml:32 (state 10) \[(i)\]
proc 2 (process:1) hyman.pml:37 (state 16) \[.(goto)\]
proc 2 (process:1) hyman.pml:40 (state 19) \[assert((crit==1))\]
proc 2 (process:1) hyman.pml:29 (state 6) \[(i)\]
proc 1 (process:1) hyman.pml:39 (state 18) \[crit = (crit+1)\]
proc 1 (process:1) hyman.pml:33 (state 13) \[else\]
proc 1 (process:1) hyman.pml:33 (state 14) \[goto :b1\]
proc 1 (process:1) hyman.pml:29 (state 5) \[1\]
proc 1 (process:1) hyman.pml:39 (state 18) \[assert((crit==1))\]
proc 2 (process:1) hyman.pml:30 (state 7) \[else\]
proc 2 (process:1) hyman.pml:30 (state 8) \[goto :b2\]
proc 1 (process:1) hyman.pml:33 (state 13) \[else\]
proc 2 (process:1) hyman.pml:30 (state 8) \[goto :b2\]
proc 2 (process:1) hyman.pml:30 (state 8) \[goto :b2\]
proc 1 (process:1) hyman.pml:30 (state 12) \[k = i\]
proc 1 (process:1) hyman.pml:37 (state 16) \[.(goto)\]
proc 1 (process:1) hyman.pml:33 (state 13) \[else\]
proc 1 (process:1) hyman.pml:33 (state 14) \[goto :b1\]
proc 1 (process:1) hyman.pml:29 (state 5) \[1\]
proc 2 (process:1) hyman.pml:45 (state 21) \[b[i] = 0\]
proc 2 (process:1) hyman.pml:46 (state 22) \[b[i] = 0\]
proc 2 (process:1) hyman.pml:32 (state 10) \[.(goto)\]
proc 1 (process:1) hyman.pml:40 (state 19) \[assert((crit==1))\]
proc 2 (process:1) hyman.pml:33 (state 14) \[goto :b1\]
proc 1 (process:1) hyman.pml:33 (state 13) \[else\]
proc 1 (process:1) hyman.pml:29 (state 5) \[1\]
proc 1 (process:1) hyman.pml:33 (state 14) \[goto :b1\]
proc 1 (process:1) hyman.pml:39 (state 18) [crit = (crit+1)]
proc 1 (process:1) hyman.pml:40 (state 19) [assert((crit==1))]
proc 1 (process:1) hyman.pml:37 (state 16) [. (goto)]
proc 1 (process:1) hyman.pml:45 (state 21) [crit = (crit-1)]
proc 1 (process:1) hyman.pml:46 (state 22) [b[i] = 0]
proc 1 (process:1) hyman.pml:49 (state 25) [. (goto)]
proc 1 (process:1) hyman.pml:20 (state 1) [(i)]
proc 2 (process:1) hyman.pml:27 (state 4) [(k==i)]
proc 2 (process:1) hyman.pml:32 (state 10) [(goto)]
proc 2 (process:1) hyman.pml:30 (state 7) [else]
proc 2 (process:1) hyman.pml:30 (state 8) [goto :b2]
proc 2 (process:1) hyman.pml:32 (state 12) [k = i]
proc 1 (process:1) hyman.pml:24 (state 3) [b[i] = 1]
proc 1 (process:1) hyman.pml:37 (state 16) [. (goto)]
proc 1 (process:1) hyman.pml:27 (state 4) [(k!=i)]
proc 2 (process:1) hyman.pml:37 (state 16) [. (goto)]
proc 1 (process:1) hyman.pml:32 (state 10) [. (goto)]
proc 2 (process:1) hyman.pml:30 (state 7) [else]
proc 2 (process:1) hyman.pml:33 (state 13) [else]
proc 2 (process:1) hyman.pml:33 (state 14) [goto :b1]
proc 2 (process:1) hyman.pml:39 (state 18) [crit = (crit+1)]
proc 2 (process:1) hyman.pml:40 (state 19) [assert((crit==1))]
proc 2 (process:1) hyman.pml:45 (state 21) [crit = (crit-1)]
proc 2 (process:1) hyman.pml:46 (state 22) [b[i] = 0]
proc 1 (process:1) hyman.pml:30 (state 7) [else]
proc 2 (process:1) hyman.pml:24 (state 3) [b[i] = 1]
proc 1 (process:1) hyman.pml:37 (state 16) [. (goto)]
proc 2 (process:1) hyman.pml:20 (state 1) [(i)]
proc 2 (process:1) hyman.pml:24 (state 3) [b[i] = 1]
proc 1 (process:1) hyman.pml:30 (state 8) [goto :b2]
proc 2 (process:1) hyman.pml:37 (state 16) [. (goto)]
proc 2 (process:1) hyman.pml:33 (state 13) [else]
proc 2 (process:1) hyman.pml:33 (state 14) [goto :b1]
proc 1 (process:1) hyman.pml:32 (state 12) [k = i]
proc 1 (process:1) hyman.pml:37 (state 16) [. (goto)]
proc 1 (process:1) hyman.pml:33 (state 13) [else]
proc 1 (process:1) hyman.pml:33 (state 14) [goto :b1]
proc 2 (process:1) hyman.pml:39 (state 18) [crit = (crit+1)]
proc 2 (process:1) hyman.pml:40 (state 19) [assert((crit==1))]
proc 1 (process:1) hyman.pml:39 (state 18) [crit = (crit+1)]
proc 2 (process:1) hyman.pml:40 (state 19) [assert((crit==1))]
proc 2 (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
proc 2 (process:1) hyman.pml:43 (state 23)
proc 1 (process:1) hyman.pml:40 (state 19)
proc 0 (:init::1) hyman.pml:15 (state 7) <valid end state>
3 processes created