Prof. Javier Esparza
Philipp Czerner

Technical University of Munich
Chair for Foundations of Software Reliability

Automata and Formal Languages

Winter Term 2023/24 – Exercise Sheet 8

Exercise 8.1.

(a) Give a recursive algorithm for the following operation:

INPUT: States p and q of the master automaton.
OUTPUT: State r of the master automaton such that $L(r) = L(p) \cdot L(q)$.

Observe that the languages $L(p)$ and $L(q)$ can have different lengths. Try to reduce the problem for p, q to the problem for p^a, q.

(b) Give a recursive algorithm for the following operation:

INPUT: A state q of the master automaton.
OUTPUT: State r of the master automaton such that $L(r) = L(q)^R$

where R is the reverse operator.

(c) A *coding* over an alphabet Σ is a function $h\colon \Sigma \mapsto \Sigma$. A coding h can naturally be extended to a function over words, i.e., $h(\varepsilon) = \varepsilon$ and $h(w) = h(w_1)h(w_2)\cdots h(w_n)$ for every $w \in \Sigma^n$. Give an algorithm for the following operation:

INPUT: A state q of the master automaton and a coding h.
OUTPUT: State r of the master automaton such that $L(r) = \{h(w) : w \in L(q)\}$.

Can you make your algorithm more efficient when h is a permutation?

Solution.

(a) Let L and L' be fixed-length languages. The following holds:

$$L \cdot L' = \begin{cases} \emptyset & \text{if } L = \emptyset, \\ L' & \text{if } L = \{\varepsilon\}, \\ \bigcup_{a \in \Sigma} \{a\} \cdot L^a \cdot L' & \text{otherwise.} \end{cases}$$

These identities give rise to the algorithm on the next page, where we added an extra case for $q = q_\varepsilon$ and $q = q_\emptyset$ because the operation *make* is not defined for $q_\emptyset$:

(b) Let L be a fixed-length language. The following holds:

$$L^R = \begin{cases} \emptyset & \text{if } L = \emptyset, \\ \{\varepsilon\} & \text{if } L = \{\varepsilon\}, \\ \bigcup_{a \in \Sigma} (L^a)^R \cdot \{a\} & \text{otherwise.} \end{cases}$$

These identities give rise to the following algorithm:

[hard] Note that Lines 11 and 12 are introduced in order to represent the language $\{a_i\}$ in Line 13 as a state $\texttt{make}(s_1, s_2, ..., s_n)$ of the master automaton. This can be avoided by using the algorithm from Exercise 8.1, namely the state that represents $\{a_i\}$ is $\texttt{add-lang}(\{a_i\})$. Thus, Lines 11-13 can be replaced just by $r \leftarrow \texttt{concat}(\texttt{reverse}(q^{a_i}), \texttt{add-lang}(\{a_i\}))$

Input: States p and q of the master automaton.
Output: State r of the master automaton such that $L(r) = L(p) \cdot L(q)$.

```
concat(p, q):
    if G(p, q) is not empty then
        return G(p, q)
    else if p = q_∅ then
        return q_∅
    else if p = q_ε then
        return q
    else if q = q_∅ then
        return q_∅
    else if q = q_ε then
        return p
    else
        for a_i ∈ Σ do
            s_i ← concat(p^{a_i}, q)
        G(p, q) ← make(s_1, s_2, ..., s_n)
        return G(p, q)
```

Input: A state q of the master automaton.
Output: State r of the master automaton such that $L(r) = L(q)^R$.

```
reverse(q):
    if G(q) is not empty then
        return G(q)
    else if q = q_∅ then
        return q_∅
    else if q = q_ε then
        return q_ε
    else
        p ← q_∅
        for a_i ∈ Σ do
            s_i ← q_ε
            s_j ← q_∅ for every i ≠ j
            r ← concat(reverse(q^{a_i}), make(s_1, s_2, ..., s_n))
            p ← union(p, r)
        G(q) ← p
        return G(q)
```

(c) Let L be a fixed-length language and let h be a coding. The following holds:

$$h(L) = \begin{cases} \emptyset & \text{if } L = \emptyset, \\ \{\varepsilon\} & \text{if } L = \{\varepsilon\}, \\ \bigcup_{a \in \Sigma} h(a) \cdot h(L^a) & \text{otherwise.} \end{cases}$$

These identities give rise to the following algorithm:

Input: A state q of the master automaton and a coding h.
Output: State r of the master automaton such that $L(r) = \{h(w) : w \in L(q)\}$.

```
coding(q, h):
    if G(q) is not empty then
        return G(q)
    else if q = q_∅ then
        return q_∅
    else if q = q_ε then
        return q_ε
    else
        p ← q_∅
        for a ∈ Σ do
            r ← coding(q^a, h)
            s_{h(a)} ← r
            s_b ← q_∅ for every b ≠ h(a)
            p ← union(p, make(s))
        G(q) ← p
        return G(q)
```

The above algorithm makes use of `union` because the coding may be the same for distinct letters, i.e. $h(a) = h(b)$ for $a \neq b$ is possible. However, if the coding is a permutation, then this is not possible, and thus each letter maps to a unique residual. Therefore, the algorithm can be adapted as follows:

Input: A state q of the master automaton and a coding h which is a permutation.
Output: State r of the master automaton such that $L(r) = \{h(w) : w \in L(q)\}$.

```
coding-permutation(q, h):
    if G(q) is not empty then
        return G(q)
    else if q = q_∅ then
        return q_∅
    else if q = q_ε then
        return q_ε
    else
        for a ∈ Σ do
            s_{h(a)} ← coding-permutation(q^a, h)
        G(q) ← make(s)
        return G(q)
```

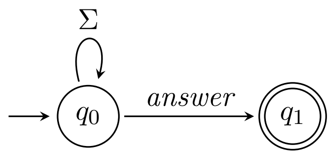
Exercise 8.2.

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

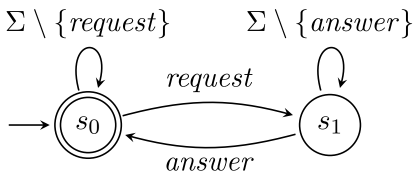
(1) Build a regular expression and an automaton recognizing all words with the property P_1: for every occurrence of *request* there is a later occurrence of *answer*.

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

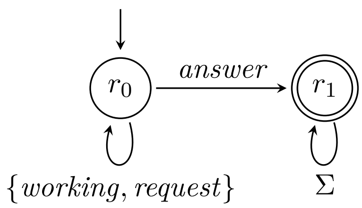
(3) Using automata theoretic constructions, prove that all words accepted by the automaton A below satisfy P_1, and give a regular expression for all words accepted by the automaton A that violate P_2.



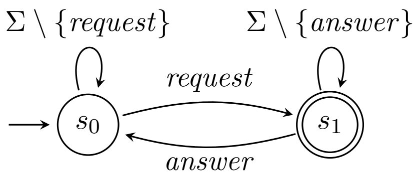
Solution. (1) A possible regular expression is $(\Sigma^* \mathit{answer})^*(\Sigma \setminus \{\mathit{request}\})^*$. (Observe that we must also describe the sequences containing no occurrence of *request*.) A minimal DFA for the property is



(3) A minimal NFA for P_2 is

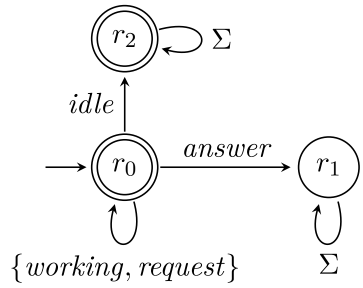


(4) Complementing the automaton for P_1 we get

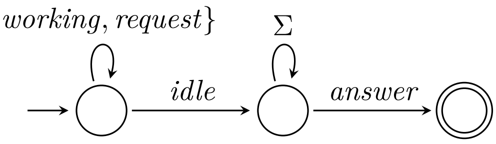


The intersection of A and the automaton for P_1 is empty: indeed, we can only reach a final state of A by executing *request*, while we can only reach a final state of the automaton for P_1 by executing *answer*. So we cannot simultaneously reach final states in both.

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 $(\mathit{working} + \mathit{request})^* \, \mathit{idle} \, \Sigma^* \, \mathit{answer}$.

Exercise 8.3.

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:

```
process(i):
    while true do
        c ! i
        c ? i
        /* critical section */
        /* 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) Construct the asynchronous product of the network obtained in (b).

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

(e) 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.

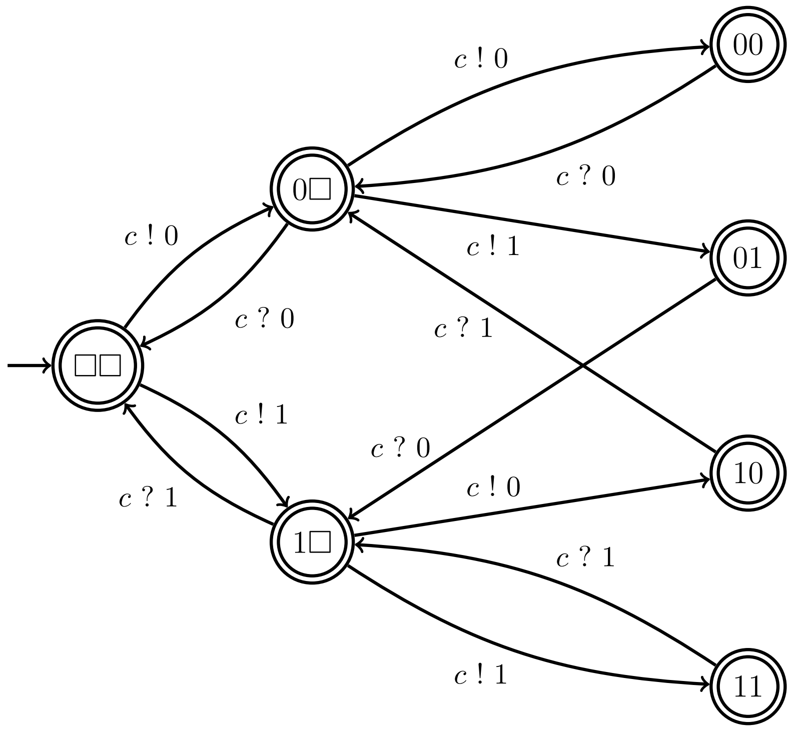
(f) Model your algorithm from (e) as a network of automata.

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

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

Solution.

(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:

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

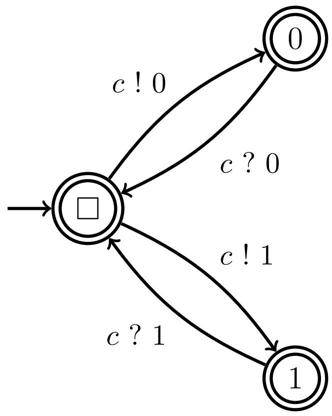
Let $\ell \colon Q \to \{1, 2, \ldots, m\}$ be the function that associates to each state q the position of the last letter of q which is not $\square$. For example, $\ell(abb\square\square) = 3$. The transitions are formally defined as follows:

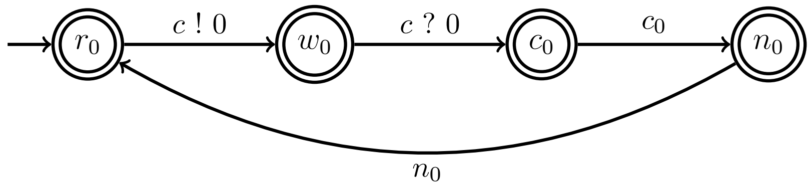
$$
\delta(q, c\ !\ \sigma) = \begin{cases} q_1 q_2 \cdots q_{\ell(q)} \sigma \square^{m-\ell(q)-1} & \text{if } \ell(q) < m, \\ \text{none} & \text{otherwise,} \end{cases}
$$

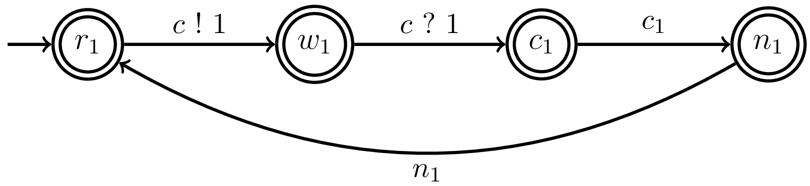
$$
\delta(q, c\ ?\ \sigma) = \begin{cases} q_2 q_3 \cdots q_m \square & \text{if } q_1 = \sigma, \\ \text{none} & \text{otherwise.} \end{cases}
$$

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

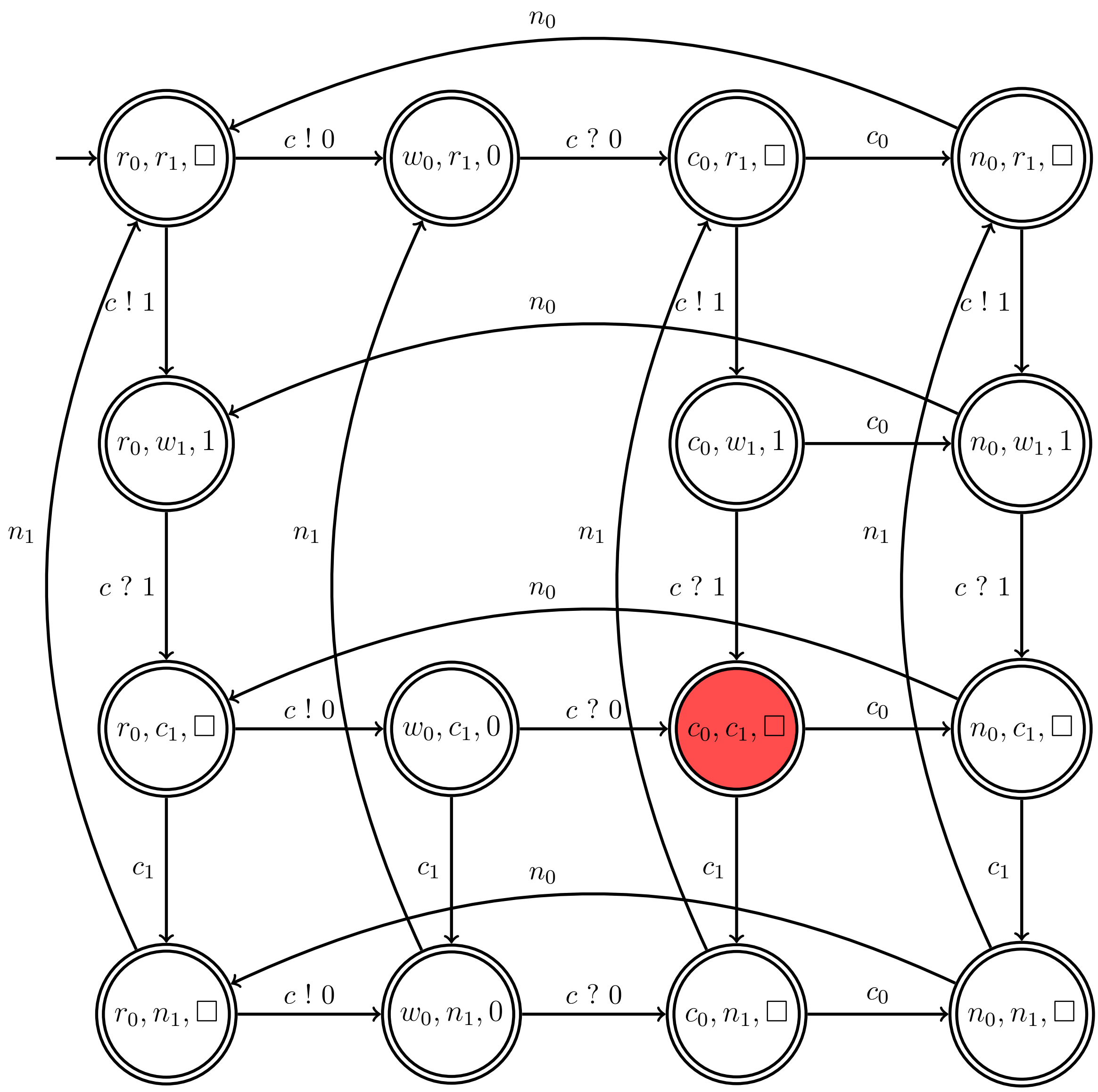
(b) The automata for the channel, `process`(0) and `process`(1) are respectively:







(c)

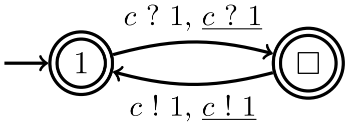


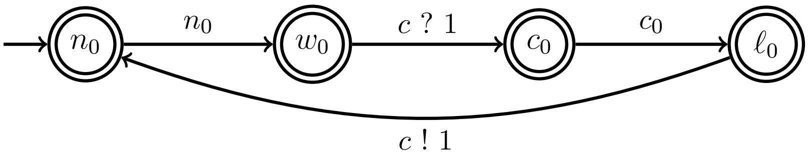
(d) The algorithm violates mutual exclusion since state $(c_0, c_1, \square)$ is reachable in the above automaton.

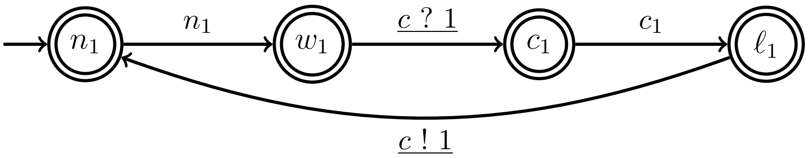
(e) We initialize a channel c of size one with message 1. When a process wants to enter its critical section, it simply consumes 1 from the channel and sends it back once it is done:

```
process():
  while true do
    /* non critical section */
    c ? 1
    /* critical section */
    c ! 1
```

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



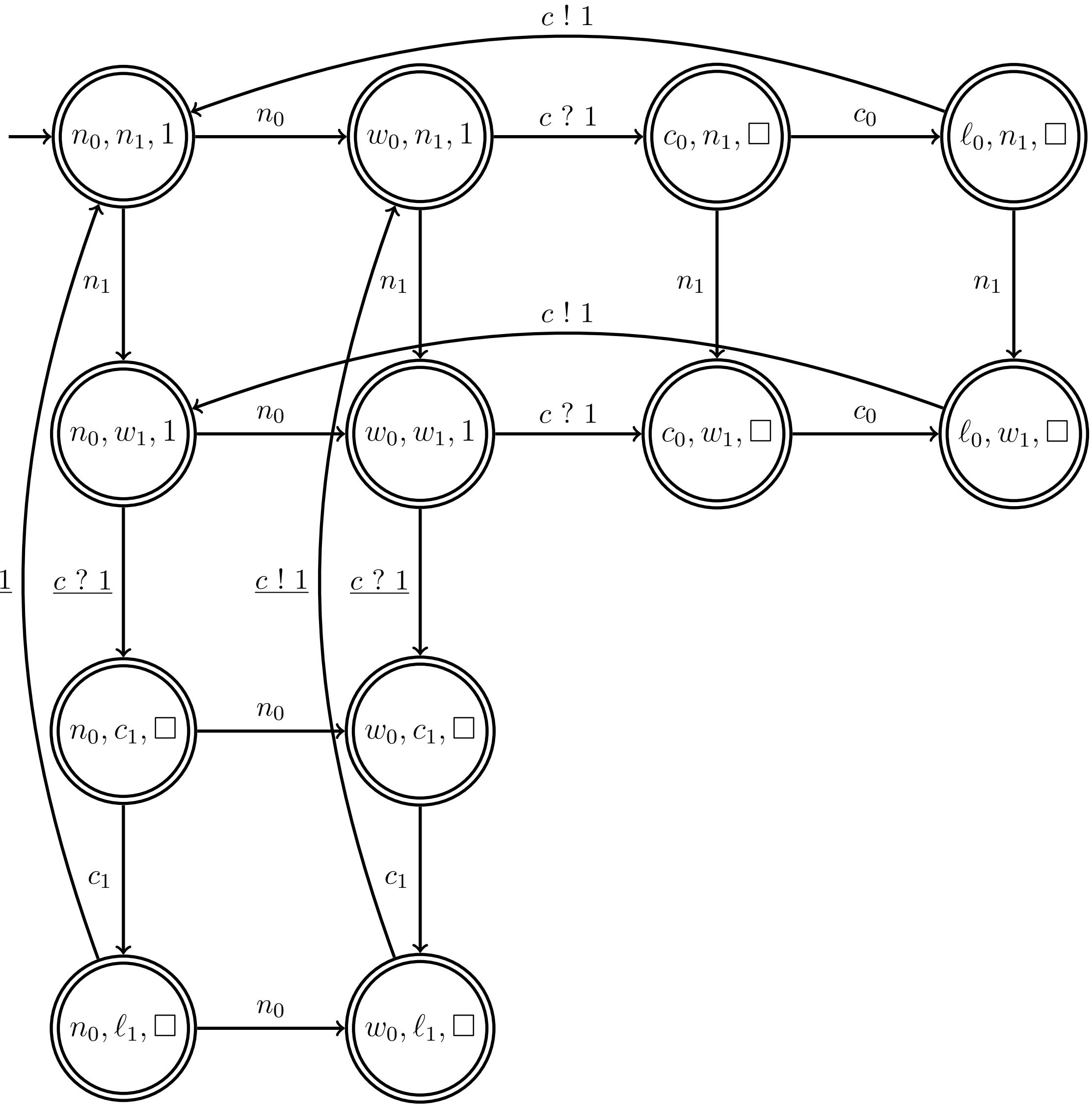




[hard] Note that we have introduced the new letters $\underline{c\ !\ 1}$ and $\underline{c\ ?\ 1}$. We could have simply used letters c ! 1 and c ? 1. However, these new letters will be important when considering the asynchronous product of the network. If the two automata modeling the processes both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

[hard] 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`.

(g)



(h) 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.