1. a) Let $M_a = (S_a, s_0^a, R_a, L_a)$, where

\[
S_a = \{s_0\},
\]  
\[
s_0^a = s_0,
\]  
\[
R_a = \emptyset,
\]  
\[
L(s_0) = \{p, q\}.
\]

The Kripke structure $M_a$ has the unique execution $\sigma_1 = s_0$, which corresponds to the execution path $\pi_1 = L(s_0) = \{p, q\}$. We check that $M_a \models G p$ holds. (Throughout the discussion, we denote the length of a finite sequence $x$ by $|x|$: for example, $|\sigma_1| = |\pi_1| = 1$ in this case.)

\[
M_a \models G p
\]  
iff $\pi \models G p$ for all execution paths $\pi$ in $M_a$ \hspace{1cm} (semantics of $\models$)
\iff $\pi_1 \models G p$ \hspace{1cm} ($M_a$ has the unique execution path $\pi_1$)
\iff $\pi_i^1 \models p$ for all $0 \leq i < |\pi_1|$ \hspace{1cm} (semantics of $G$)
\iff $\pi_0^1 \models p$ \hspace{1cm} ($|\pi_1| = 1$)
\iff $p \in L(s_0)$ \hspace{1cm} (semantics of $|$)
\iff $p \in \{p, q\}$ \hspace{1cm} (definition of $|$)

Because $p \in \{p, q\}$ holds, $M_a \models G p$ holds. Similarly,

\[
M_a \models G (p \Rightarrow q)
\]  
iff $\pi \models G (p \Rightarrow q)$ for all execution paths $\pi$ in $M_a$ \hspace{1cm} (semantics of $|$)
\iff $\pi_1 \models G (p \Rightarrow q)$ \hspace{1cm} ($M_a$ has the unique execution path $\pi_1$)
\iff $\pi_i^1 \models p \Rightarrow q$ for all $0 \leq i < |\pi_1|$ \hspace{1cm} (semantics of $G$)
\iff $\pi_0^1 \models p \Rightarrow q$ \hspace{1cm} ($|\pi_1| = 1$)
\iff $\pi_i^1 \models (\neg p) \vee q$ \hspace{1cm} (semantics of $\Rightarrow$)
\iff $\pi_i^0 \models \neg p$ or $\pi_i^0 \models q$ \hspace{1cm} (semantics of $\vee$)
\iff $\pi_i^0 \models \neg p$ or $\pi_i^0 \models q$ \hspace{1cm} (semantics of $\neg$)
\iff $p \notin L(s_0)$ or $q \in L(s_0)$ \hspace{1cm} (semantics of $|$)
\iff $p \notin \{p, q\}$ or $q \in \{p, q\}$ \hspace{1cm} (definition of $L$)

Because $q \in \{p, q\}$ holds, it follows that $M_a \models G (p \Rightarrow q)$ holds.
b) Let \( M_b = (S_b, s_0^b, R_b, L_b) \), where

\[
\begin{align*}
S_b &= \{s_0, s_1, s_2\}, \\
S_{s_0}^b &= s_0, \\
R_b &= \{(s_0, s_1)\}, \\
L(s_0) &= \{p, q\}, \text{ and} \\
L(s_1) &= \{q\}.
\end{align*}
\]

The Kripke structure \( M_b \) has two executions \( \sigma_1 = s_0 \) and \( \sigma_2 = s_0s_1 \) corresponding to the execution paths \( \pi_1 = L(s_0) \) and \( \pi_2 = L(s_0)L(s_1) \), respectively.

\[
\begin{align*}
M_b \not\models G p \\
\text{iff not } (M_b \models G p) & \quad \text{(semantics of } \models \text{)} \\
\text{iff not } (\pi \models G p \text{ for all execution paths } \pi \text{ in } M_b) & \quad (-) \\
\text{iff } \pi \not\models G p \text{ for some execution path } \pi \text{ in } M_b & \quad (-)
\end{align*}
\]

In this case, we see that \( \pi_2 \not\models G p \):

\[
\begin{align*}
\pi_2 \not\models G p \\
\text{iff not } (\pi_2 \models G p) & \quad \text{(semantics of } \models \text{)} \\
\text{iff not } (\pi_i^2 \models p \text{ for all } 0 \leq i < |\pi_2|) & \quad \text{(semantics of } G \text{)} \\
\text{iff } \pi_i^2 \not\models p \text{ for some } 0 \leq i < |\pi_2| & \quad \text{(semantics of } \models \text{)} \\
\text{iff } \pi_2^0 \not\models p \text{ or } \pi_2^1 \not\models p & \quad (|\pi_2| = 2) \\
\text{iff } p \not\in L(s_0) \text{ or } p \not\in L(s_1) & \quad \text{(semantics of } \models \text{)} \\
\text{iff } p \not\in \{p, q\} \text{ or } p \not\in \{q\} & \quad \text{(definition of } L \text{)}
\end{align*}
\]

Because \( p \not\in \{q\} = L(s_1) \) holds, \( \pi_2 \not\models G p \) holds, and it follows that \( M_b \not\models G p \).

To check that \( M_b \models G (p \lor Y q) \) holds, we need to check that both \( \pi_1 \models G (p \lor Y q) \) and \( \pi_2 \models G (p \lor Y q) \) hold in the model \( M_b \). This can be seen as follows:

\[
\begin{align*}
\pi_1 \models G (p \lor Y q) \\
\text{iff } \pi_i \models p \lor Y q \text{ for all } 0 \leq i < |\pi_1| & \quad \text{(semantics of } G \text{)} \\
\text{iff } \pi_1^0 \models p \lor Y q & \quad (|\pi_1| = 1) \\
\text{iff } \pi_1^0 \models p \text{ or } \pi_1^0 \models Y q & \quad \text{(semantics of } \lor \text{)}
\end{align*}
\]

Because \( p \in L(s_0) = \{p, q\} \) (i.e., \( \pi_1^0 \models p \)) holds, it follows that \( \pi_1 \models G (p \lor Y q) \).
\(\pi_2 \models G(p \lor Y q)\) iff \(\pi_i^2 \models p \lor Y q\) for all \(0 \leq i < |\pi_2|\) (semantics of \(G\))

iff \(\pi_i^0 \models p \lor Y q\) and \(\pi_i^1 \models p \lor Y q\) (\(\pi_2 = 2\))

iff \((\pi_i^0 \models p \lor p_1^0 = Y q)\) and \((\pi_i^1 \models p \lor p_1^1 = Y q)\) (semantics of \(\lor\))

iff \((\pi_i^0 \models p \lor (0 > 0 \text{ and } \pi_i^0 \text{ or } \pi_i^0 \text{ or } \pi_i^0 = q)\) and

\((\pi_i^1 \models p \lor (1 > 0 \text{ and } \pi_i^1 \text{ or } \pi_i^1 \text{ or } \pi_i^1 = q)\) (semantics of \(Y\))

iff \((\pi_i^0 \models p)\) and \((\pi_i^1 \models p \lor \pi_i^0 \models q)\) (0 \(\neq 0, 1 > 0\))

Because \(\{p, q\} \subseteq L(s_0) = \{p, q\}\) holds (i.e., \(\pi_i^0 \models p\) and \(\pi_i^0 \models q\)), it follows that also \(\pi_2 \models G(p \lor Y q)\). Therefore, \(M_b \models G(p \lor Y q)\).

c) Let \(M_c = (S_c, s_c^0, R_c, L_c)\), where

\[
S_c = \{s_0, s_1, s_2\},
\]

\[
s_c^0 = s_0.
\]

\[
R_c = \{(s_0, s_1), (s_1, s_2)\},
\]

\[
L(s_0) = \emptyset,
\]

\[
L(s_1) = \{q\}, \text{ and}
\]

\[
L(s_2) = \{p, q\}.
\]

The Kripke structure \(M_c\) has three executions \(\sigma_1 = s_0, \sigma_2 = s_0, s_1, s_2\) and \(\sigma_3 = s_0, s_1, s_2\) corresponding to the execution paths \(\pi_1 = L(s_0), \pi_2 = L(s_0) L(s_1)\) and \(\pi_3 = L(s_0) L(s_1) L(s_2)\), respectively.

\(M_c \models G(p \Rightarrow (q S \neg p))\) iff \(\pi_i \models G(p \Rightarrow (q S \neg p))\) for all \(i \in \{1, 2, 3\}\) (semantics of \(\Rightarrow\))

iff \(\pi_i^j \models p \Rightarrow (q S \neg p)\) for all \(i \in \{1, 2, 3\}\) and \(0 \leq j < |\pi_i|\) (semantics of \(\Rightarrow\))

iff \(\pi_i^j \models (\neg p) \lor (q S \neg p)\) for all \(i \in \{1, 2, 3\}\) and \(0 \leq j < |\pi_i|\) (semantics of \(\lor\))

iff \(\pi_i^j \models \neg p\) or \(\pi_i^j \models q S \neg p\) for all \(i \in \{1, 2, 3\}\) and \(0 \leq j < |\pi_i|\) (semantics of \(\neg\))

Because \(p \notin L(s_0) = \emptyset\) and \(p \notin L(s_1) = \{q\}\) hold, it follows that \(\pi_i^j \models \neg p\) holds for all \(i \in \{1, 2, 3\}\) and \(0 \leq j < |\pi_i|\). Therefore, \(\pi_1 \models G(p \Rightarrow (q S \neg p))\) and \(\pi_2 \models G(p \Rightarrow (q S \neg p))\) hold, and \(\pi_3 \models p \Rightarrow (q S \neg p)\) holds for all \(j \in \{0, 1\}\). Thus, \(\pi_3 \models G(p \Rightarrow (q S \neg p))\) (and therefore, \(M_c \models G(p \Rightarrow (q S \neg p))\)) holds iff \(\pi_2^3 \models p \Rightarrow (q S \neg p)\).
\[ \pi^2_3 \models p \Rightarrow (q \mathbb{S} \neg p) \]

if \( \pi^1_3 \models \neg p \) or \( \pi^2_3 \models q \) (see above)

if \( p \notin L(s_2) \) or (there exists an index \( 0 \leq k \leq 2 \) such that \( \pi^k_3 \models \neg p \) and \( \pi^k_3 \models q \) for all \( k < n \leq 2 \) (semantics of \( \models \), \( \mathbb{S} \))

if there exists an index \( 0 \leq k \leq 2 \) such that \( \pi^k_3 \not\models p \) and \( \pi^k_3 \models q \) for all \( k < n \leq 2 \) (\( p \in L(s_2) = \{p, q\} \), semantics of \( \models \))

if \( (\pi^0_3 \not\models p \text{ and } \pi^1_3 \models q \) \) or \( (\pi^2_3 \not\models p \text{ and } \pi^2_3 \models q) \) or \( (\pi^2_3 \not\models p) \)

\((k \text{ is one of } 0, 1, 2)\)

Because \( p \notin L(s_0) = \emptyset \) and \( p \notin L(s_1) = \{q\} \) (i.e., \( \pi^0_3 \not\models p \) and \( \pi^1_3 \not\models p \) hold, but \( q \in L(s_1) \) and \( q \in L(s_2) = \{p, q\} \) (i.e., \( \pi^1_3 \models q \) and \( \pi^2_3 \models q \) hold), it follows that the above condition is satisfied. Therefore \( \pi^2_3 \models p \Rightarrow (q \mathbb{S} \neg p) \), and it follows that \( M_e \models \mathbf{G} (p \Rightarrow (q \mathbb{S} \neg p)) \).

As above, because \( p \notin L(s_0) \) and \( p \notin L(s_1) \), it is easy to check that \( \pi^j_i \models p \Rightarrow \mathbf{Y} \mathbf{Y} \neg p \) holds for all \( i \in \{1, 2, 3\} \) and \( 0 \leq j < \min \{2, \lvert \pi^i_2 \rvert \} \). Therefore, \( M_e \models \mathbf{G} (p \Rightarrow \mathbf{Y} \mathbf{Y} \neg p) \) holds iff \( \pi^2_3 \models p \Rightarrow \mathbf{Y} \mathbf{Y} \neg p \).

\[ \pi^2_3 \models p \Rightarrow \mathbf{Y} \mathbf{Y} \neg p \]

if \( \pi^3_3 \models (\neg p) \lor \mathbf{Y} \mathbf{Y} \neg p \) (semantics of \( \Rightarrow \))

if \( \pi^3_3 \models \neg p \) or \( \pi^3_3 \models \mathbf{Y} \mathbf{Y} \neg p \) (semantics of \( \lor \))

if \( \pi^3_3 \not\models p \) or \( 2 > 0 \) and \( \pi^{3-1}_3 \models \mathbf{Y} \neg p \) (semantics of \( \neg, \mathbf{Y} \))

if \( p \notin L(s_2) \) or \( \pi^1_3 \models \mathbf{Y} \neg p \) (2 > 0, semantics of \( \models \))

if \( 1 > 0 \) and \( \pi^{3-1}_3 \models \neg p \) (\( p \in L(s_2) = \{p, q\} \), semantics of \( \mathbf{Y} \))

if \( \pi^3_3 \not\models p \) (1 > 0, semantics of \( \neg \))

if \( p \notin L(s_0) \) (semantics of \( \models \))

The result now follows because \( p \notin L(s_0) = \emptyset \) holds by the definition of \( L \).

(This solution was designed for illustrating the semantics of the various operators of the logic. A simpler solution is given by any Kripke model which consists of a single state in which the atomic proposition \( p \) is false.)
d) Let $M_d = (S_d, s_d^0, R_d, L_d)$, where

$$S_d = \{s_0, s_1\},$$
$$s_d^0 = s_0,$$
$$R_d = \{(s_0, s_1)\},$$
$$L(s_0) = \{q\},$$
$$L(s_1) = \emptyset.$$

The Kripke structure $M_d$ has two executions $\sigma_1 = s_0$ and $\sigma_2 = s_0s_1$ corresponding to the execution paths $\pi_1 = L(s_0)$ and $\pi_2 = L(s_0)L(s_1)$, respectively.

Suppose that $\pi_2 \models G (p \mathbf{S} q)$ holds. In particular (by the semantics of $G$), $\pi_2^1 \models p \mathbf{S} q$ holds in this case, and there exists an index $0 \leq i \leq 1$ such that $\pi_2^i \models q$, and $\pi_2^0 \models p$ for all $i < n \leq 1$. Clearly, $i = 0$ is the only index such that $\pi_2^0 \models q$ holds. Because $p \notin L(s_1) = \emptyset$, however, $\pi_2^1 \not\models p$, and thus it cannot be the case that $\pi_2^i \models p$ for all $i < n \leq 1$, contrary to the assumption. Therefore $\pi_2 \not\models G (p \mathbf{S} q)$, and thus also $M_d \not\models G (p \mathbf{S} q)$.

On the other hand,

$$M_d \models G \mathbf{O} q$$

iff $\pi \models G \mathbf{O} q$ for all execution paths $\pi$ in $M_d$ (semantics of $\models$)
iff $\pi_i \models G \mathbf{O} q$ for all $i \in \{1, 2\}$ (M has the execution paths $\pi_1$ and $\pi_2$)
iff $\pi_i^j \models \mathbf{O} q$ for all $i \in \{1, 2\}$ and $0 \leq j < |\pi_i|$ (semantics of $\mathbf{O}$)
iff $\pi_i^j \models \mathbf{T} \mathbf{S} q$ for all $i \in \{1, 2\}$ and $0 \leq j < |\pi_i|$ (semantics of $\mathbf{T}$)
iff for all $i \in \{1, 2\}$ and $0 \leq j < |\pi_i|$ and $\pi_i^j \models q$ for some $0 \leq k < j$, and $\pi_i^j \models \mathbf{T}$ for all $k < n \leq j$ (semantics of $\mathbf{S}$)
iff for all $i \in \{1, 2\}$ and $0 \leq j < |\pi_i|$ and $\pi_i^j \models q$ for some $0 \leq k < j$, and $\pi_i^j \models p \lor \neg p$ for all $k < n \leq j$ (semantics of $\lor$)
iff for all $i \in \{1, 2\}$ and $0 \leq j < |\pi_i|$ and $\pi_i^j \models q$ for some $0 \leq k < j$, and $\neg (\pi_i^j \models p)$ and $\neg (\pi_i^j \models \neg p)$ for all $k < n \leq j$ (semantics of $\rightarrow$)
iff for all $i \in \{1, 2\}$ and $0 \leq j < |\pi_i|$ and $\pi_i^j \models q$ for some $0 \leq k < j$, and $\neg (\pi_i^j \models p)$ and $\neg (\pi_i^j \models \neg p)$ always holds for all $k < n \leq j$ (semantics of $\Rightarrow$)

Because $q \in \{q\} = L(s_0)$ holds, it is easy to see that $\pi_1^0 \models q$ and $\pi_2^0 \models q$ hold, and thus the above requirement is satisfied in all execution paths in $M_d$. Therefore $M_d \models G \mathbf{O} q$ holds.

2. We first characterise the finite words that are counterexamples to the formula $\varphi$. Let $\pi = x_0 x_1 \ldots x_n \in (2^{2\mathbf{A}})^*$ be a finite word over the alphabet $2^{2\mathbf{A}} = \{\emptyset, \{\text{alarm}\}, \{\text{crash}\}, \{\text{alarm, crash}\}\}$. The word $\pi$ is a counterexample to the formula $\varphi$, i.e., $\pi \not\models G (\text{alarm} \Rightarrow \mathbf{O} \text{crash})$, where
iff not (π | G (alarm ⇒ O (crash))) (semantics of |)
iff not (for all 0 ≤ i ≤ n: πi | alarm ⇒ O (crash)) (semantics of G)
iff not (for all 0 ≤ i ≤ n: πi | (¬alarm) ∨ O (crash)) (semantics of ⇒)
iff not (for all 0 ≤ i ≤ n: (πi | ¬alarm or πi | O (crash))) (semantics of ∨)
iff not (for all 0 ≤ i ≤ n: (πi | ¬alarm or there exists an index 0 ≤ j ≤ i such that πj | crash)) (semantics of ¬, O)
iff there exists an 0 ≤ i ≤ n: (πi | alarm, and πj | crash for all 0 ≤ j ≤ i).

The counterexamples to ϕ are therefore those finite words in which the symbol \{alarm\} appears before a symbol that contains the atomic proposition crash, i.e., the words that match the regular expression

\( \emptyset^* \{alarm\} (\emptyset \cup \{alarm\} \cup \{crash\} \cup \{alarm, crash\})^*. \)

A deterministic finite automaton that accepts the counterexamples to ϕ can thus read its input one symbol at a time until (i) the input is exhausted (in which case the automaton will not accept its input), or (ii) until it encounters a symbol that differs from \emptyset. The automaton then enters one of two states in which it simply consumes the rest of the input and either accepts or rejects the input word depending on whether the first input symbol different from \emptyset was \{alarm\} or not.

![Diagram of the automaton](image)

3. Suppose that we wish to check a system which consists of the following Promela process for violations of the safety property ϕ from exercise 2:

```promela
bool alarm = false;
bool crash = false;
active proctype system() {
```
do
  :: true -> skip
  :: crash = true; break
od;
crash = false;
alarm = true
}

It is easy to see that—in every execution of this system—the variable alarm will never have the value true before crash has been set to true at some previous step. This system therefore satisfies the safety property φ, which expresses the requirement that a state in which alarm is true should always be preceded by (or coincide with) a state in which the variable crash has the value true.

Since we already have a deterministic finite state automaton which accepts violations of the safety property (see exercise 2), we would like to use this automaton as a “monitor process” that observes the global state of the system and reports a failure if the safety property is ever violated. Obviously, this requires coupling the monitor process with the system. Translating the automaton from the previous exercise into a proctype definition, we obtain the Promela code

active proctype monitor() {
  do
    :: (!alarm && !crash)
    :: (alarm && !crash) -> assert(false)
    :: (crash) ->
      do
        :: true -> skip
      od
  od
}

The behaviour of this process mimics the behaviour of the automaton: the outer do-loop is executed until one of the global variables alarm and crash becomes true. If (alarm && !crash) is true, the monitor process executes the assertion (reporting a failure); if crash is true, the process enters an infinite loop from which the assertion can no longer be reached (since it becomes impossible to violate the safety property in this case).

However, analysing a model that consists of the definitions of the two above processes yields an unexpected verification result.\(^1\)

\(^1\)The -DREACH option for the compiler and the -i option for the verifier are used only to optimize the length of the counterexample. They are not necessary to uncover the error.
$ spin -a 3.pml
$ cc -DREACH -o pan pan.c
$ ./pan -i
hint: this search is more efficient if pan.c is compiled -DSAFETY
pan: assertion violated 0 (at depth 4)
pan: wrote 3.pml.trail

Analysing the error trail gives the following result:

$ spin -t -p 3.pml
Starting system with pid 0
Starting monitor with pid 1
1: proc 0 (system) line 7 "3.pml" (state 3) [crash = 1]
2: proc 0 (system) line 9 "3.pml" (state 8) [crash = 0]
3: proc 0 (system) line 10 "3.pml" (state 9) [alarm = 1]
4: proc 1 (monitor) line 16 "3.pml" (state 2) [((alarm&&!(crash)))]
spin: line 16 "3.pml", Error: assertion violated
spin: text of failed assertion: assert(0)
5: proc 1 (monitor) line 16 "3.pml" (state 3) [assert(0)]
spin: trail ends after 5 steps
#processes: 2
alarm = 1
crash = 0
5: proc 1 (monitor) line 14 "3.pml" (state 10)
5: proc 0 (system) line 11 "3.pml" (state 10) <valid end state>
2 processes created
$

In this error trail, the system process already reaches the end of its code before the monitor process takes even its first execution step. At this point, the global variables alarm and crash have the values true and false, respectively, which leads the monitor process to execute the assertion statement. Thus, our monitor process does not appear to work as intended: it fails to observe that the variable crash was set to true at a previous step.

This verification result can be explained by examining the composition of the two processes the model checker Spin uses for verification. The control structure of the two processes can be depicted as the following two extended labelled transition systems in which we decorate the states in the “system” LTS with the values of the global variables. The transitions of the LTSs are labelled with the expressions that appear in the Promela code of the processes. These expressions form the alphabets of the LTSs; for each process, we use an alphabet that is disjoint from the alphabet of the other process. This is denoted by prefixing every expression used as an alphabet symbol
with “s:” or “m:” depending on whether the expression originates from the system process or the monitor process.

\[ \Sigma_{\text{system}} = \{ s:\text{true}, s:\text{skip}, s:\text{crash}=\text{true}, s:\text{crash}=\text{false}, s:\text{alarm}=\text{true} \} \]
\[ \Sigma_{\text{monitor}} = \{ m:(\neg\text{alarm} \land \neg\text{crash}), m:(\neg\text{alarm} \land \neg\text{crash}), m:\text{assert}(\text{false}), m:(\text{crash}), m:\text{true}, m:\text{skip} \} \]

The verifier analyses a structure which can be described as a parallel composition of the extended LTSs corresponding to the Promela processes. (When forming the product of the extended LTSs, we consider a transition referring to the global system variables in the monitor process to be enabled only if the expression labelling the transition evaluates to true in the current state of the system LTS.) This parallel composition has the following structure (the solid lines correspond to transitions of the system process, the dashed lines to transitions of the monitor process):
Even though the system process satisfies the safety property, the parallel composition of the LTSs contains a path (for example, \((s_0, m_0) \rightarrow (s_2, m_0) \rightarrow (s_3, m_0) \rightarrow (s_4, m_0) \rightarrow (s_4, m_1) \rightarrow (s_4, m_0))\) in which the monitor process executes the assertion. The reason for this is the interleaving of the transitions of the two processes in the parallel composition: there is no mechanism to ensure that the monitor process will always observe the change in the value of the variable \(\text{crash}\) in the state \((s_2, m_0)\) before the system process resets the value of the variable again to \(\text{false}\). In other words, the usual parallel composition of LTSs does not guarantee that the monitor process remains synchronised with the changes in the state of the system it is supposed to observe.

The never claim construct of Promela provides a direct way to add to a Promela model a process which is guaranteed to execute synchronously with the rest of the system (only one such process per model is allowed; furthermore, because never claims are intended to be used to observe the behaviour of models—intuitively, to detect behaviour that should “never” happen in a system, a never claim may not contain statements that effect changes in the system state). Instead of using a proctype definition for the monitor process, we can thus define a monitor process that will execute synchronously with the rest of the system as a never claim with the following syntax:

```
never {
  do
  :: (!\_alarm && !\_crash)
  :: (\_alarm && !\_crash) -> assert(false)
  :: (\_crash) ->
    do
    :: true -> skip
    od
  od
}
```

A verifier generated by Spin forms the composition of a never claim declaration with a system comprising one or more processes by executing the never claim synchronously with the (LTS-like) parallel composition of the system processes. Every transition in the structure analysed by the verifier then corresponds to a pair of transitions taken by the never claim and a process in the rest of the model. (If either the system or the never claim cannot execute a transition in a system state, no transition is generated in the composition.) The composition of a system with a never claim thus resembles more closely the product of finite automata (see notes from lecture 2) instead of the parallel composition of LTSs.
In our example system, we obtain the composition

(Note that the transition taken by the never claim in a pair of transitions is always chosen from the transitions enabled in the system state corresponding to the source state of the pair of transitions.)

From this synchronous composition of the system with the monitor process (specified as a never claim) we see that the failing assertion can no longer be reached.

(As stated in the assignment, assert statements are rarely used in never claims. A more conventional way to write the never claim would be to use a break statement in place of the assertion: a verifier generated by Spin will report an error if the never claim is able to reach the end of its code while observing the system.)

More information on never claims and the formal definitions of the various product constructions used by Spin-generated verifiers can be found in Appendix A of Gerard J. Holzmann’s textbook The Spin Model Checker: Primer and Reference Manual, Addison-Wesley, 2003.