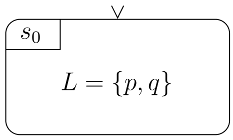
T–79.4301 **Spring 2006**

Parallel and Distributed Systems

Tutorial 8 – Solutions

1. a) Let $M_a = (S_a, s_a^0, R_a, L_a)$, where

$$\begin{aligned} S_a &= \{s_0\}, \\ s_a^0 &= s_0, \\ R_a &= \emptyset, \text{ and} \\ L(s_0) &= \{p, q\}. \end{aligned}$$



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 $M_a \models \mathbf{G}\, p$ 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.)

$$\begin{array}{lll}
 & M_a \models \mathbf{G}\, p & \\
\text{iff} & \pi \models \mathbf{G}\, p \text{ for all execution paths } \pi \text{ in } M_a & \textit{(semantics of} \models\textit{)} \\
\text{iff} & \pi_1 \models \mathbf{G}\, p & \textit{(}M_a \textit{ has the unique execution path } \pi_1\textit{)} \\
\text{iff} & \pi_1^i \models p \text{ for all } 0 \leq i < |\pi_1| & \textit{(semantics of } \mathbf{G}\textit{)} \\
\text{iff} & \pi_1^0 \models p & \textit{(}|\pi_1| = 1\textit{)} \\
\text{iff} & p \in L(s_0) & \textit{(semantics of} \models\textit{)} \\
\text{iff} & p \in \{p, q\} & \textit{(definition of} \models\textit{)}
\end{array}$$

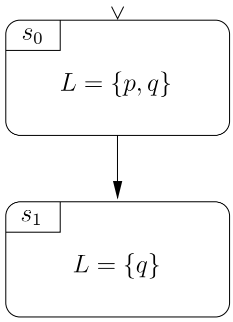
Because $p \in \{p, q\}$ holds, $M_a \models \mathbf{G}\, p$ holds. Similarly,

$$\begin{array}{lll}
 & M_a \models \mathbf{G}\,(p \Rightarrow q) & \\
\text{iff} & \pi \models \mathbf{G}\,(p \Rightarrow q) \text{ for all execution paths } \pi \text{ in } M_a & \textit{(semantics of} \models\textit{)} \\
\text{iff} & \pi_1 \models \mathbf{G}\,(p \Rightarrow q) & \textit{(}M_a \textit{ has the unique execution path } \pi_1\textit{)} \\
\text{iff} & \pi_1^i \models p \Rightarrow q \text{ for all } 0 \leq i < |\pi_1| & \textit{(semantics of } \mathbf{G}\textit{)} \\
\text{iff} & \pi_1^0 \models p \Rightarrow q & \textit{(}|\pi_1| = 1\textit{)} \\
\text{iff} & \pi_1^0 \models (\neg p) \vee q & \textit{(semantics of} \Rightarrow\textit{)} \\
\text{iff} & \pi_1^0 \models \neg p \text{ or } \pi_1^0 \models q & \textit{(semantics of} \vee\textit{)} \\
\text{iff} & \pi_1^0 \not\models p \text{ or } \pi_1^0 \models q & \textit{(semantics of} \neg\textit{)} \\
\text{iff} & p \notin L(s_0) \text{ or } q \in L(s_0) & \textit{(semantics of} \models\textit{)} \\
\text{iff} & p \notin \{p, q\} \text{ or } q \in \{p, q\} & \textit{(definition of } L\textit{)}
\end{array}$$

Because $q \in \{p, q\}$ holds, it follows that $M_a \models \mathbf{G}\,(p \Rightarrow q)$ holds.

b) Let $M_b = (S_b, s_b^0, R_b, L_b)$, where

$$\begin{aligned} S_b &= \{s_0, s_1\}, \\ s_b^0 &= s_0, \\ R_b &= \big\{(s_0, s_1)\big\}, \\ L(s_0) &= \{p, q\}, \text{ and} \\ L(s_1) &= \{q\}. \end{aligned}$$



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

$$\begin{array}{lll} & M_b \not\models \mathbf{G}\, p & \\ \text{iff} & \text{not } (M_b \models \mathbf{G}\, p) & \textit{(semantics of } \models\textit{)} \\ \text{iff} & \text{not } (\pi \models \mathbf{G}\, p \text{ for all execution paths } \pi \text{ in } M_b) & \textit{(–)} \\ \text{iff} & \pi \not\models \mathbf{G}\, p \text{ for some execution path } \pi \text{ in } M_b & \textit{(–)} \end{array}$$

In this case, we see that $\pi_2 \not\models \mathbf{G}\, p$:

$$\begin{array}{lll} & \pi_2 \not\models \mathbf{G}\, p & \\ \text{iff} & \text{not } (\pi_2 \models \mathbf{G}\, p) & \textit{(semantics of } \models\textit{)} \\ \text{iff} & \text{not } (\pi_2^i \models p \text{ for all } 0 \leq i < |\pi_2|) & \textit{(semantics of } \mathbf{G}\textit{)} \\ \text{iff} & \pi_2^i \not\models p \text{ for some } 0 \leq i < |\pi_2| & \textit{(semantics of } \models\textit{)} \\ \text{iff} & \pi_2^0 \not\models p \text{ or } \pi_2^1 \not\models p & \textit{(}|\pi_2| = 2\textit{)} \\ \text{iff} & p \notin L(s_0) \text{ or } p \notin L(s_1) & \textit{(semantics of } \models\textit{)} \\ \text{iff} & p \notin \{p, q\} \text{ or } p \notin \{q\} & \textit{(definition of } L\textit{)} \end{array}$$

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

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

$$\begin{array}{lll} & \pi_1 \models \mathbf{G}\,(p \vee \mathbf{Y}\, q) & \\ \text{iff} & \pi_1^i \models p \vee \mathbf{Y}\, q \text{ for all } 0 \leq i < |\pi_1| & \textit{(semantics of } \mathbf{G}\textit{)} \\ \text{iff} & \pi_1^0 \models p \vee \mathbf{Y}\, q & \textit{(}|\pi_1| = 1\textit{)} \\ \text{iff} & \pi_1^0 \models p \text{ or } \pi_1^0 \models \mathbf{Y}\, q & \textit{(semantics of } \vee\textit{)} \end{array}$$

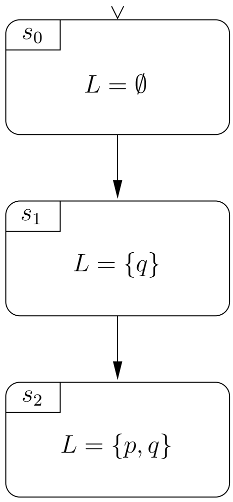
Because $p \in L(s_0) = \{p, q\}$ (i.e., $\pi_1^0 \models p$) holds, it follows that $\pi_1 \models \mathbf{G}\,(p \vee \mathbf{Y}\, q)$.

$$
\begin{array}{lll}
& \pi_2 \models \mathbf{G}\,(p \vee \mathbf{Y}\,q) & \\
\text{iff} & \pi_2^i \models p \vee \mathbf{Y}\,q \text{ for all } 0 \le i < |\pi_2| & \textit{(semantics of } \mathbf{G}\textit{)} \\
\text{iff} & \pi_2^0 \models p \vee \mathbf{Y}\,q \text{ and } \pi_2^1 \models p \vee \mathbf{Y}\,q & (|\pi_2| = 2) \\
\text{iff} & (\pi_2^0 \models p \text{ or } \pi_2^0 \models \mathbf{Y}\,q) \text{ and } (\pi_2^1 \models p \text{ or } \pi_2^1 \models \mathbf{Y}\,q) & \textit{(semantics of } \vee\textit{)} \\
\text{iff} & (\pi_2^0 \models p \text{ or } (0 > 0 \text{ and } \pi_2^{0-1} \models q)) \text{ and} & \\
& (\pi_2^1 \models p \text{ or } (1 > 0 \text{ and } \pi_2^{1-1} \models q)) & \textit{(semantics of } \mathbf{Y}\textit{)} \\
\text{iff} & (\pi_2^0 \models p) \text{ and } (\pi_2^1 \models p \text{ or } \pi_2^0 \models q) & (0 \not> 0,\ 1 > 0)
\end{array}
$$

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

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

$$
\begin{aligned}
S_c &= \{s_0, s_1, s_2\}, \\
s_c^0 &= s_0, \\
R_c &= \big\{(s_0, s_1), (s_1, s_2)\big\}, \\
L(s_0) &= \emptyset, \\
L(s_1) &= \{q\}, \text{ and} \\
L(s_2) &= \{p, q\}.
\end{aligned}
$$



The Kripke structure M_c has three executions $\sigma_1 = s_0$, $\sigma_2 = s_0 s_1$ 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.

$$
\begin{array}{lll}
& M_c \models \mathbf{G}\,\big(p \Rightarrow (q\,\mathbf{S}\,\neg p)\big) & \\
\text{iff} & \pi_i \models \mathbf{G}\,\big(p \Rightarrow (q\,\mathbf{S}\,\neg p)\big) \text{ for all } i \in \{1,2,3\} & \textit{(semantics of } \models\textit{)} \\
\text{iff} & \pi_i^j \models p \Rightarrow (q\,\mathbf{S}\,\neg p) \text{ for all } i \in \{1,2,3\} \text{ and } 0 \le j < |\pi_i| & \\
& & \textit{(semantics of } \mathbf{G}\textit{)} \\
\text{iff} & \pi_i^j \models (\neg p) \vee (q\,\mathbf{S}\,\neg p) \text{ for all } i \in \{1,2,3\} \text{ and } 0 \le j < |\pi_i| & \\
& & \textit{(semantics of } \Rightarrow\textit{)} \\
\text{iff} & \pi_i^j \models \neg p \text{ or } \pi_i^j \models q\,\mathbf{S}\,\neg p \text{ for all } i \in \{1,2,3\} \text{ and } 0 \le j < |\pi_i| & \\
& & \textit{(semantics of } \vee\textit{)} \\
\text{iff} & \pi_i^j \not\models p \text{ or } \pi_i^j \models q\,\mathbf{S}\,\neg p \text{ for all } i \in \{1,2,3\} \text{ and } 0 \le j < |\pi_i| & \\
& & \textit{(semantics of } \neg\textit{)}
\end{array}
$$

Because $p \notin L(s_0) = \emptyset$ and $p \notin L(s_1) = \{q\}$ hold, it follows that $\pi_i^j \not\models p$ holds for all $i \in \{1,2,3\}$ and $0 \le j < \min\big\{2, |\pi_i|\big\}$. Therefore $\pi_1 \models \mathbf{G}\,\big(p \Rightarrow (q\,\mathbf{S}\,\neg p)\big)$ and $\pi_2 \models \mathbf{G}\,\big(p \Rightarrow (q\,\mathbf{S}\,\neg p)\big)$ hold, and $\pi_3^j \models p \Rightarrow (q\,\mathbf{S}\,\neg p)$ holds for all $j \in \{0, 1\}$. Thus $\pi_3 \models \mathbf{G}\,\big(p \Rightarrow (q\,\mathbf{S}\,\neg p)\big)$ (and therefore, $M_c \models \mathbf{G}\,\big(p \Rightarrow (q\,\mathbf{S}\,\neg p)\big)$) holds iff $\pi_3^2 \models p \Rightarrow (q\,\mathbf{S}\,\neg p)$.

$$
\begin{array}{lll}
& \pi_3^2 \models p \Rightarrow (q \,\mathbf{S}\, \neg p) & \\
\text{iff} & \pi_3^2 \not\models p \text{ or } \pi_3^2 \models q \,\mathbf{S}\, \neg p & \textit{(see above)} \\
\text{iff} & p \notin L(s_2) \text{ or (there exists an index } 0 \leq k \leq 2 \text{ such that } \pi_3^k \models \neg p & \\
& \text{and } \pi_3^n \models q \text{ for all } k < n \leq 2) & \textit{(semantics of } \models, \mathbf{S}\textit{)} \\
\text{iff} & \text{there exists an index } 0 \leq k \leq 2 \text{ such that } \pi_3^k \not\models p \text{ and } \pi_3^n \models q & \\
& \text{for all } k < n \leq 2 & \textit{(}p \in L(s_2) = \{p, q\}\textit{, semantics of } \models\textit{)} \\
\text{iff} & (\pi_3^0 \not\models p \text{ and } \pi_3^1 \models q \text{ and } \pi_3^2 \models q) \text{ or} & \\
& (\pi_3^1 \not\models p \text{ and } \pi_3^2 \models q) \text{ or} & \\
& (\pi_3^2 \not\models p) & \textit{(k is one of } 0, 1, 2\textit{)}
\end{array}
$$

Because $p \notin L(s_0) = \emptyset$ and $p \notin L(s_1) = \{q\}$ (i.e., $\pi_3^0 \not\models p$ and $\pi_3^1 \not\models p$) hold, but $q \in L(s_1)$ and $q \in L(s_2) = \{p, q\}$ (i.e., $\pi_3^1 \models q$ and $\pi_3^2 \models q$) hold, it follows that the above condition is satisfied. Therefore $\pi_3^2 \models p \Rightarrow (q \,\mathbf{S}\, \neg p)$, and it follows that $M_c \models \mathbf{G}\big(p \Rightarrow (q \,\mathbf{S}\, \neg p)\big)$.

As above, because $p \notin L(s_0)$ and $p \notin L(s_1)$, it is easy to check that $\pi_i^j \models p \Rightarrow \mathbf{Y}\,\mathbf{Y}\,\neg p$ holds for all $i \in \{1, 2, 3\}$ and $0 \leq j < \min\big\{2, |\pi_i|\big\}$. Therefore, $M_c \models \mathbf{G}\,(p \Rightarrow \mathbf{Y}\,\mathbf{Y}\,\neg p)$ holds iff $\pi_3^2 \models p \Rightarrow \mathbf{Y}\,\mathbf{Y}\,\neg p$.

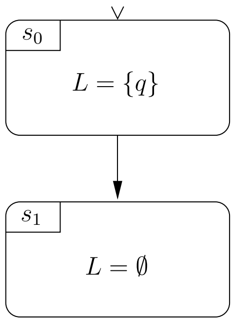
$$
\begin{array}{lll}
& \pi_3^2 \models p \Rightarrow \mathbf{Y}\,\mathbf{Y}\,\neg p & \\
\text{iff} & \pi_3^2 \models (\neg p) \vee \mathbf{Y}\,\mathbf{Y}\,\neg p & \textit{(semantics of } \Rightarrow\textit{)} \\
\text{iff} & \pi_3^2 \models \neg p \text{ or } \pi_3^2 \models \mathbf{Y}\,\mathbf{Y}\,\neg p & \textit{(semantics of } \vee\textit{)} \\
\text{iff} & \pi_3^2 \not\models p \text{ or } (2 > 0 \text{ and } \pi_3^{2-1} \models \mathbf{Y}\,\neg p) & \textit{(semantics of } \neg, \mathbf{Y}\textit{)} \\
\text{iff} & p \notin L(s_2) \text{ or } \pi_3^1 \models \mathbf{Y}\,\neg p & \textit{(}2 > 0\textit{, semantics of } \models\textit{)} \\
\text{iff} & 1 > 0 \text{ and } \pi_3^{1-1} \models \neg p & \textit{(}p \in L(s_2) = \{p, q\}\textit{, semantics of } \mathbf{Y}\textit{)} \\
\text{iff} & \pi_3^0 \not\models p & \textit{(}1 > 0\textit{, semantics of } \neg\textit{)} \\
\text{iff} & p \notin L(s_0) & \textit{(semantics of } \models\textit{)}
\end{array}
$$

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

$$\begin{aligned} S_d &= \{s_0, s_1\}, \\ s_d^0 &= s_0, \\ R_d &= \{(s_0, s_1)\}, \\ L(s_0) &= \{q\}, \text{ and} \\ L(s_1) &= \emptyset. \end{aligned}$$



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

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

On the other hand,

	$M_d \models \mathbf{G}\,\mathbf{O}\,q$	
iff	$\pi \models \mathbf{G}\,\mathbf{O}\,q$ for all execution paths π in M_d	*(semantics of $\models$)*
iff	$\pi_i \models \mathbf{G}\,\mathbf{O}\,q$ for all $i \in \{1,2\}$	*(M_d has the execution paths π_1 and π_2)*
iff	$\pi_i^j \models \mathbf{O}\,q$ for all $i \in \{1,2\}$ and $0 \leq j < \lvert\pi_i\rvert$	*(semantics of $\mathbf{G}$)*
iff	$\pi_i^j \models \top\,\mathbf{S}\,q$ for all $i \in \{1,2\}$ and $0 \leq j < \lvert\pi_i\rvert$	*(semantics of $\mathbf{O}$)*
iff	for all $i \in \{1,2\}$ and $0 \leq j < \lvert\pi_i\rvert$, $\pi_i^k \models q$ for some $0 \leq k \leq j$, and $\pi_i^n \models \top$ for all $k < n \leq j$	*(semantics of $\mathbf{S}$)*
iff	for all $i \in \{1,2\}$ and $0 \leq j < \lvert\pi_i\rvert$, $\pi_i^k \models q$ for some $0 \leq k \leq j$, and $\pi_i^n \models p \vee \neg p$ for all $k < n \leq j$	*(semantics of $\top$)*
iff	for all $i \in \{1,2\}$ and $0 \leq j < \lvert\pi_i\rvert$, $\pi_i^k \models q$ for some $0 \leq k \leq j$, and ($\pi_i^n \models p$ or $\pi_i^n \models \neg p$) for all $k < n \leq j$	*(semantics of $\vee$)*
iff	for all $i \in \{1,2\}$ and $0 \leq j < \lvert\pi_i\rvert$, $\pi_i^k \models q$ for some $0 \leq k \leq j$	*($\pi_i^n \models p$ or $\pi_i^n \models \neg p$ always holds for all $k < n \leq j$)*

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 $M_d \models \mathbf{G}\,\mathbf{O}\,q$ holds.

2. We first characterise the finite words that are counterexamples to the formula φ. Let $\pi = x_0 x_1 \ldots x_n \in (2^{AP})^*$ be a finite word over the

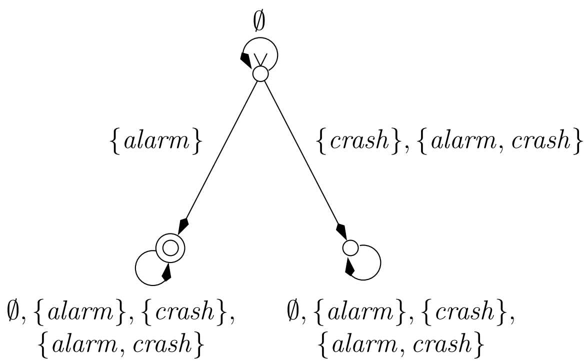
alphabet $2^{AP} = \big\{\emptyset, \{\mathit{alarm}\}, \{\mathit{crash}\}, \{\mathit{alarm}, \mathit{crash}\}\big\}$. The word π is a counterexample to the formula φ, i.e., $\pi \not\models \mathbf{G}\big(\mathit{alarm} \Rightarrow \mathbf{O}(\mathit{crash})\big)$,

iff not $\big(\pi \models \mathbf{G}\big(\mathit{alarm} \Rightarrow \mathbf{O}(\mathit{crash})\big)\big)$ *(semantics of $\models$)*
iff not (for all $0 \le i \le n$: $\pi^i \models \mathit{alarm} \Rightarrow \mathbf{O}(\mathit{crash})$) *(semantics of $\mathbf{G}$)*
iff not (for all $0 \le i \le n$: $\pi^i \models (\neg\mathit{alarm}) \vee \mathbf{O}(\mathit{crash})$) *(semantics of $\Rightarrow$)*
iff not (for all $0 \le i \le n$: ($\pi^i \models \neg\mathit{alarm}$ or $\pi^i \models \mathbf{O}(\mathit{crash})$)) *(semantics of $\vee$)*
iff not (for all $0 \le i \le n$: ($\pi^i \not\models \mathit{alarm}$, or there exists an index $0 \le j \le i$ such that $\pi^j \models \mathit{crash}$)) *(semantics of $\neg, \mathbf{O}$)*
iff there exists an $0 \le i \le n$: ($\pi^i \models \mathit{alarm}$, and $\pi^j \not\models \mathit{crash}$ for all $0 \le j \le i$).

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

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

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 $\{\mathit{alarm}\}$ or not.



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:

```
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]

```
$ 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:

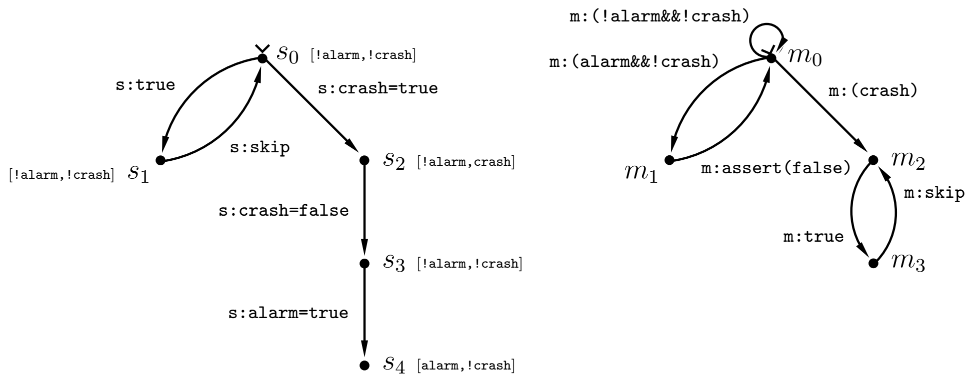
```
$ spin -t -p 3.pml
Starting system with pid 0
Starting monitor with pid 1
  1:    proc  0 (system) line   7 "3.pml" (state 3)     [crash = 1]
  2:    proc  0 (system) line   9 "3.pml" (state 8)     [crash = 0]
  3:    proc  0 (system) line  10 "3.pml" (state 9)     [alarm = 1]
  4:    proc  1 (monitor) 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

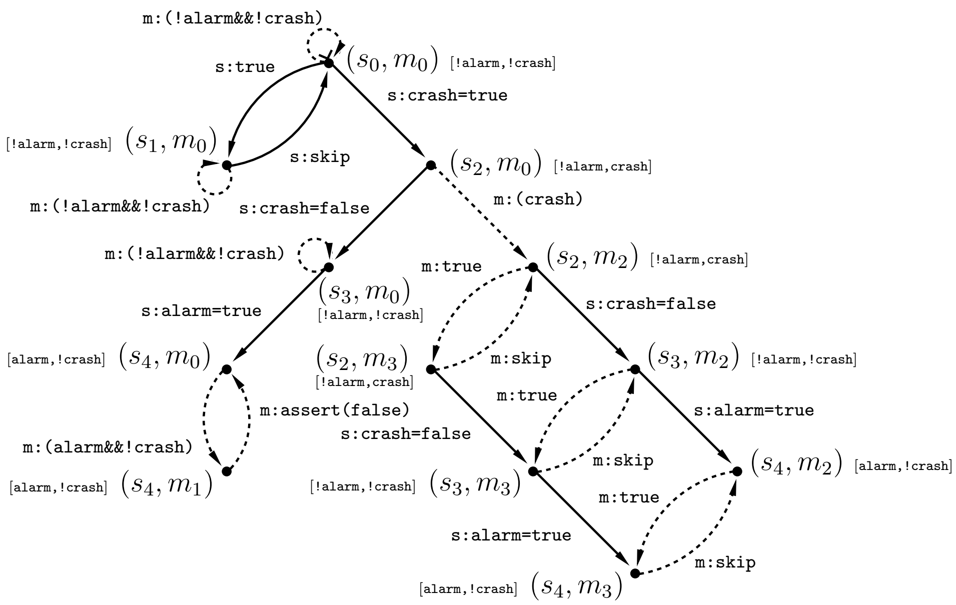
[1] The `-DREACH` option for the compiler and the `-i` option for the verifier are used only to optimize the length of the counterexample. They are not necessary to uncover the error.

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.



$$
\begin{aligned}
\Sigma_{system} &= \{\texttt{s:true},\ \texttt{s:skip},\ \texttt{s:crash=true},\ \texttt{s:crash=false},\ \texttt{s:alarm=true}\}\\
\Sigma_{monitor} &= \{\texttt{m:(!alarm\&\&!crash)},\ \texttt{m:(alarm\&\&!crash)},\ \texttt{m:assert(false)},\\
&\quad\ \texttt{m:(crash)},\ \texttt{m:true},\ \texttt{m:skip}\}
\end{aligned}
$$

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, $(s_0, m_0) \to (s_2, m_0) \to (s_3, m_0) \to (s_4, m_0) \to (s_4, m_1) \to (s_4, m_0)$) 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 `crash` in the state (s_2, m_0) before the system process resets the value of the variable again to `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.

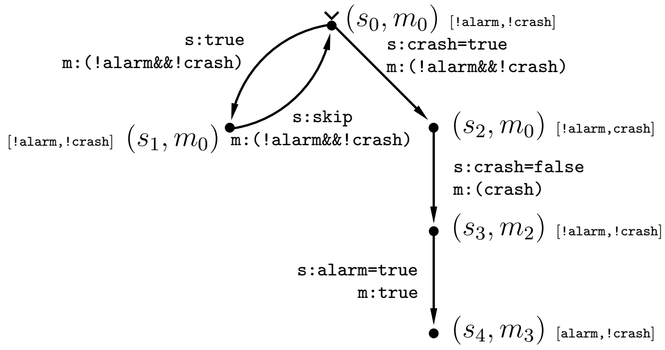
The `never` claim construct of Promela provides a direct way to add to a Promela model a process which is guaranteed to execute synchronously with the rest of the system (only one such process per model is allowed; furthermore, because never claims are intended to be used to observe the behaviour of 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.