Exercise 1: Mutual Exclusion without Request  

The goal of this exercise is to help you understand in detail the SOS-rules for parallel compositions.  

The transition systems below describe a mutual-exclusion protocol with an arbiter. In contrast to the system discussed in the lecture, we omit the request action.

(a) Draw the transition system for the pure interleaving $TS_1 || TS_2$. There must be no synchronization between the two transition systems.  

For every transition in the interleaving, justify why it must exist using one of the two SOS-rules for pure interleaving.

Example: The interleaving must contain the transition $⟨idle, idle⟩ \xrightarrow{enter} ⟨crit, idle⟩$ due to the SOS-rule

$$\frac{idle \xrightarrow{enter} crit}{⟨idle, idle⟩ \xrightarrow{enter} ⟨crit, idle⟩}$$

where $→_1$ is the transition relation for $TS_1$. This is an instance of the first of the two SOS-rules,

$$\frac{s_1 \xrightarrow{α_1} s'_1}{⟨s_1, s_2⟩ \xrightarrow{α} ⟨s'_1, s_2⟩}$$

where we set $s_1 = idle$, $α = enter$, $s'_1 = crit$ and $s_2 = idle$.

(b) Draw the transition system for the parallel composition $(TS_1 || TS_2) || Arbiter$ of the transition system from (a) with the arbiter. The transition systems must synchronize (“handshake”) on the actions $\{enter, exit\}$.

For every transition in the composition, justify why it must exist using one of the three SOS-rules for the synchronization operator.
Exercise 2: Parallelism - Interleaving

The goal of this exercise is to construct the interleaving of two program graphs and the corresponding transition system.

Consider the following parallel system consisting of two processes. The programs for each process are given in a low-level programming language.

Algorithm 1:
\[ r_1 := x + 1; \]
\[ x := r_1; \]

Algorithm 2:
\[ r_2 := 3 \times x; \]
\[ x := r_2; \]

(a) Draw the program graphs \( P_1 \) and \( P_2 \) for each process.

(b) Draw the interleaving of the program graphs \( P_1 \parallel P_2 \).

(c) Draw the reachable part of the transition system \( T_{P_1 \parallel P_2} \). We assume that the initial value of \( x \) is 1 and the initial values of \( r_1 \) and \( r_2 \) are both 0. Use it to determine which values can finally be stored in \( x \).

It is not strictly needed for solving the exercise above, but, for those who are interested in the motivation for the two exercises:

The above program can be seen as a model for an assembler program, namely the assembler program to which the program you have seen in the lecture (two processes consisting of the single statements \( x := x + 1 \) and \( x := 3 \times x \)) is compiled.

This serves to explain that one cannot assume that an update statement in a high-level language is an atomic action.

In the exercise we just used a simplified version of the assembler code by merging the first two statements of the processes to reduce the size of the program graph.

Algorithm 1:
\[
\begin{align*}
\text{LOAD } x; \\
\text{ADDI } 1; \\
\text{STORE } x;
\end{align*}
\]

Algorithm 2:
\[
\begin{align*}
\text{LOAD } x; \\
\text{MULTI } 3; \\
\text{STORE } x;
\end{align*}
\]

Semantics of the assembler commands:

- \textbf{LOAD } i : Load the content of address \( i \) of the memory into an accumulator register \( ACC \).

- \textbf{ADDI } i : Adds \( i \) to the content of register \( ACC \) and stores the result in \( ACC \).

- \textbf{MULTI } i : Multiplies the content of register \( ACC \) with \( i \) and stores the result in \( ACC \).

- \textbf{STORE } i : Store the content of register \( ACC \) at memory address \( i \).

Note: A variable is here represented by a memory address which is a typical abstraction.