Problem 1 (4 Points)

Consider two trains moving on a circular path, one going clockwise and the other counter-clockwise. Normally there are two pairs of rails, so that there is no conflict between the two trains. However, the rails cross at some point, and the intersection is guarded by two signal lights.

module Train 1
state: loc1 : {far, near, in}
output: approach1!, leave1!, dummy1!
input: green1?
initially: loc1 = far
update:
\[ \text{loc1 = far} \xrightarrow{\text{approach1!}} \text{loc1 = near} \]
\[ \text{loc1 = far} \xrightarrow{\text{dummy1!}} \]
\[ \text{loc1 = near} \xrightarrow{\text{green1?}} \text{loc1 = in} \]
\[ \text{loc1 = near} \rightarrow \]
\[ \text{loc1 = in} \xrightarrow{\text{exit1!}} \text{loc1 = far} \]

module Train 2
state: loc2 : {far, near, in}
output: approach2!, leave2!, dummy2!
input: green2?
initially: loc2 = away
update:
\[ \text{loc2 = far} \xrightarrow{\text{approach2!}} \text{loc2 = near} \]
\[ \text{loc2 = far} \xrightarrow{\text{dummy2!}} \]
\[ \text{loc2 = near} \xrightarrow{\text{green2?}} \text{loc2 = in} \]
\[ \text{loc2 = near} \rightarrow \]
\[ \text{loc2 = in} \xrightarrow{\text{exit2!}} \text{loc2 = far} \]
module dummy-receiver
input: dummy1?, dummy2?
update:
[true \rightarrow dummy1?\rightarrow]
[true \rightarrow dummy2?\rightarrow]
[true \rightarrow dummy1?, dummy2?\rightarrow]
[true \rightarrow]

Your task is to give one or two additional modules (controllers) that synchronise the trains. In the lecture two were used, but a solution that uses only one controller for both trains is acceptable. Your solution should fulfill the following properties:

- The state \( loc_1 = \text{in} \land loc_2 = \text{in} \) cannot be reached.
- Every state has a non-trivial successor (liveness).
- It must not happen that one train is near, while the other moves from \text{in} to \text{far} to \text{near} to \text{in} (fairness).

Hints: Keep the semantics in mind: A signal can only be sent if it can be received. A module that does not have the signal in its lists of input does not need to handle the signal. The controller may get several signals in the same round, so one transition can receive several signals. Be sure none of the modules gets stuck (because it tries to send a signal that cannot be received). Lastly if several updates are possible sending a signal has priority.

Due to the last constraint I added the dummy receiver that, so the train may chose to stay in the far state.

Problem 2 (3 Points)

Give both a recursive (functional) and state based (while) program for the \text{gcd} function. \text{gcd} or greatest common divisor of two integers; that is the largest positive integer that divides two numbers without a remainder. For example, the \text{gcd}(12, 15) is 3.

For the functional program the function should take two arguments and give a result. For the while program assume the arguments are in variables \(a\) and \(b\) and when the program finishes the result should be in \(r\).

Problem 3 (3 Points)

a) Assume \(e\) to be an arithmetic expression, \(b\) to be a boolean expression, \(x\) to be a variable and \(p, q\) to be programs. We want to define the following syntax:
\[ p = \text{skip} \mid x := e \mid p ; q \mid \text{if } b \text{ then } p \text{ else } q \text{ fi} \mid \text{for } x \text{ to } e \text{ do } p \text{ od} \]

You should give operational semantics (rules) for this language, keeping in mind the following.
• Take the boolean expressions as God-given, make no rules for them.
• Give the arithmetic expressions for +, -, *, /. The division should be an integer division only. Assume all expressions are fully parenthesised. So this would be a valid expression: \((5 + (5 + (4 \times 3)))\)
• Find a way to deal with the problem of division-by-zero. At the end of the program it should be clear if the a division-by-zero (=error) occurred or not.
• \texttt{od} and \texttt{fi} are used to determine where the statement ends (so when you have a semicolon you know if it is part of the \texttt{else} or a statement after the \texttt{if}).
• The semantics of \texttt{for} is as follows: The variable \(x\) starts with the value it has at the beginning of the execution. It will be increased by 1 after each running of \(p\). The last run occurs with \(x = e\). If \(x > e\) initially the loop does not run at all and the program continues with the next statement after the loop. Keep in mind that the value of \(e\) may be changed by the execution of \(p\).

b) Give a derivation for this program
\[
x := 0; \text{ for } x \text{ to } 100 \text{ do } y := (10/(1-x)) \text{ od}.
\]
\textit{Hint:} The derivation tree for this program should be very small, if you are smart about your semantics. Also give the derivation for the arithmetic expression.
And finally because everyone loves it so much :) 

Semantics of Reactive Modules

We define the language of Simple Reactive Modules (SRM).

Syntax  A module consists of

- a set $A$ of input events,
- a set $B$ of output events,
- a set $X$ of state variables,
- an initial condition on the state variables,
- a guarded command for each input event,
- a guarded command for each output event,
- a complete guarded command called update command.

A guarded command is a set of guarded actions. A guarded action consists of a guard, which is an enabling condition on the state variables, and an action, which is a modification of the state variables. A guarded command is complete if in every state at least one guard is true. A system is a set of modules whose output events are disjoint.

Semantics  A state for a system consists of a state for every module. A state for a module is a function that maps every state variable to a value. A guarded command $g$ changes module state $q$ to module state $q'$ if there exists a guarded action in $g$ whose guard is true in $q$ and whose action modifies $q$ to $q'$. Module state $q'$ is an $e$-successor of module state $q$ if $e$ is an input or output event and the guarded command for $e$ changes $q$ to $q'$. Module state $q'$ is an update successor of module state $q$ if the update command changes $q$ to $q'$. System state $s'$ is an update successor of system state $s$ if every module state in $s'$ is an update successor of the respective module state in $s$. System state $s'$ is an $e$-successor of system state $s$ if for every module $M$, if $e$ is an input or output event of $M$, then the $M$-state in $s'$ is an $e$-successor of the $M$-state in $s$, else the $M$-state in $s'$ is an update successor of the $M$-state in $s$. 