1 Reactive Modules - Two Train System

In the last lecture, we introduced reactive modules, a language used to describe state transition systems, which becomes particularly useful when the corresponding automaton is too complicated to be depicted through a diagram, or when multiple computing entities coexist in a system, and communicate through the exchange of signals. We considered a modeling scenario wherein a single train approaches a gate, and a controller responds by opening a gate, letting the train pass and then closing it. We analyzed various behaviors of the resulting abstracted system and looked at properties such as safety and liveness. In this lecture, we will look at a slightly more complex system as follows.

Consider two trains moving on a circular path, one going clockwise and the other counterclockwise. 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, controlled by two controllers, whose job is to ensure that there is always at most one train in the intersection.

We first look at the train modules. The location of each train is either far, near or on the bridge in a given state. While going from far to near, each train emits an a! (for approach) signal, and while going from on to far, it emits an e! (for exit) signal. Thus, we have the base train modules as follows.

```
module Train1
  state: loc1 : {far, near, on}
  output: a1!, e1!
  initially: loc1 = far
  update:
  [] loc1 = far a1! → loc1 = near
  [] loc1 = near → loc1 = in
  [] loc1 = in e1! → loc1 = far
  [] true →

module Train2
  state: loc2 : {far, near, on}
  output: a2!, e2!
  initially: loc2 = far
  update:
  [] loc2 = far a2! → loc2 = near
  [] loc2 = near → loc2 = in
  [] loc2 = in e2! → loc2 = far
  [] true →
```

It is easy to see how this system gets into trouble without any controllers. Consider the trajectory

\[ f, f \xrightarrow{a1!} n, f \xrightarrow{a2!} n, n \rightarrow o, o \]

Clearly, this is an undesirable state where both trains are on the bridge at once. We were able to find a finite trajectory which violates the safety property. In general, the problem of checking safety can be framed rigorously as checking the language emptiness of an automaton.

Let us now add controller modules which will regulate the system. The controllers will be traffic
signals which will be either red or green for each train, and each train in the near state will wait for a g? (for green) signal from the corresponding controller to transit to the on state. Each controller will decide what state to be in on the basis of the approach and exit signals of the opposite train. More precisely, if the other train is approaching, the signal will turn red, and will turn green on receiving an exit signal. Otherwise, it will stay in the green state and send a green signal for its own paired train. The four modules of the new system are given below.

**module Train1**

state: loc1 : {far, near, on}
input: a1!, e1!
output: g1!
initially: loc1 = far
update:
- loc1 = far a1! → loc1 = near
- loc1 = near g1! → loc1 = in
- loc1 = in e1! → loc1 = far

**module Train2**

state: loc2 : {far, near, on}
input: g2?
output: a2!, e2!
initially: loc2 = far
update:
- loc2 = far a2? → loc2 = near
- loc2 = near g2? → loc2 = in
- loc2 = in e2? → loc2 = far

**module Light1**

state: l1 : {green, red}
input: a2?, e2?
output: g1!
initially: l1 = green
update:
- l1 = green a2? → l1 = red
- l1 = green g1! → l1 = red
- l1 = red e2? → l1 = green

**module Light2**

state: l2 : {green, red}
input: a1?, e1?
output: g2!
initially: l2 = green
update:
- l2 = green a1? → l2 = red
- l2 = green g2! → l2 = red
- l2 = red e1? → l2 = green

The corresponding automata for the first train are drawn below. The ones for the second train are symmetric.

The state transition diagram of the entire system will have $3 \times 3 \times 2 \times 2 = 36$ states which are members of the cross product of the states of each of the automata. Note that some of these states might not be reachable. We will show a portion of the diagram below which reaches a deadlock condition. The system we have constructed is in fact perfectly safe, but not live, i.e.
it gets stuck in a loop where nothing significant happens. Note that denying liveness requires an infinite sequence of moves, unlike for safety.

\[ f, f, g, g \xrightarrow{a_1} n, f, g, r \xrightarrow{a_2} n, n, r, r \]

We summarize the following three desirable properties that a good system should have.

- No state where both gates are open is reachable. (Safety)
- Every reachable state has a non-trivial successor. (Liveness)
- While one train is near, the other must not be allowed to move from on to far to near to on again. (Fairness)

A safety property can be checked using a monitor automaton, because only need to check a finite reachability problem to any undesirable state. For the fairness condition above, for instance, we have the following module. This automaton can be used to raise a warning flag whenever something unwanted occurs.

module Monitor1
state: alert : \{0, 1, 2, 3\}
input: a1?, e1?, e2?
initially: alert = 0
update:
  [] alert = 0 \xrightarrow{a_1?} alert = 1
  [] alert = 1 \xrightarrow{e_2?} alert = 2
  [] alert = 2 \xrightarrow{e_2?} alert = 3
  [] alert = 1 \xrightarrow{e_1?} alert = 0
  [] alert = 2 \xrightarrow{e_1?} alert = 0
  [] true →

The corresponding monitor automaton is shown below.

---

\(^1\)in fact also a safety property, because it can be violated by a finite trace
Now, the safety property simply becomes 

\[
\text{never (alert = 3)}
\]

This monitor essentially verifies that the composite system (with 144*25 states) has no trajectory to a state with alert=3.

# 2 Functions

So far we have looked at following semantic objects:

- Numbers (decimal number, binary numbers, roman numbers)
- Languages (how to define a language using rules or automaton)
- Interacting automaton via reactive modules

Now we turn to functions.

- simple (textual expressions, graphical circuits)
- complex (state, which brings us to state transition systems and recursion)

Textual expressions, graphical circuits, state and recursion are the *syntaxes for defining functions*. State and recursion brings us to programs; state will lead us to imperative (while) programs, and recursion will lead us to functional programs.

From mathematician’s point of view, a function is a set, ex.:

\[
square \{ (0,0), (1,1), (2,4), (3,9), (4,16) \ldots (11,11^2) \ldots \}
\]

Function \( f \) is a set of pairs \((n, f(n))\), where \( n \) is a value in domain and \( f(n) \) is a result of function. For computer scientists this is not the way to write function, because we want everything to be finite, to deal with finite sets. So the notation above is *inconvenient*. For mathematicians the notion \( f(n) = n^2 \) is good for writing down function, but it leaves the question open: what is \( f \)?

In computer science we commonly write the function using lambda notation:

\[
f = \lambda n. \ n^2
\]

This notation means that the function \( f \) takes an input \( n \) and produces an output \( n^2 \). Using this notation it is possible to write arbitrary complicated things. Example:

\[
g = \lambda n. \ \text{if } n \text{ is even then } n/2 \text{ else } n - 1
\]

\[
g(0) = 0, \\
g(1) = 0, \\
g(2) = 1, \\
g(3) = 2 \\
\vdots
\]

Recursive formulations fall under the paradigm of *Functional programming*. Thus, we write the factorial function as

\[
\text{factorial}(n) = \lambda n. \ \text{if } n \leq 1 \text{ then } 1 \text{ else } n \cdot \text{factorial}(n - 1)
\]
In contrast, in the *imperative Programming* paradigm programs are seen as functions that map states to states, where a state itself is a function from variables to values.

**While algorithm for computing factorial**

input $n \in \mathbb{N}$

$x := 1$

$y := 1$

while $x \leq n$

$x := x + 1$

$y := x \ast y$

end

$z := y$

This program will map any initial state to the state which satisfies \( \{ z = n! \} \). We will look at imperative programming in more detail in the next lecture, where we will define different semantics (operational, axiomatic and denotational), which capture different levels of abstraction.