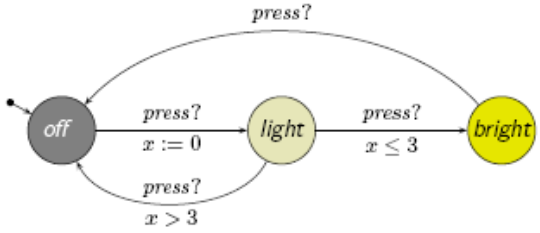
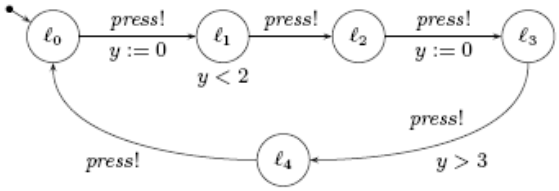
**Exercise 1:**

Let consider the following figure:




1) Explain the signification of this process;
2) Edit it in Uppaal;
3) Propose a second automaton user which can be synchronized with this process;
4) Is the system deadlock-free

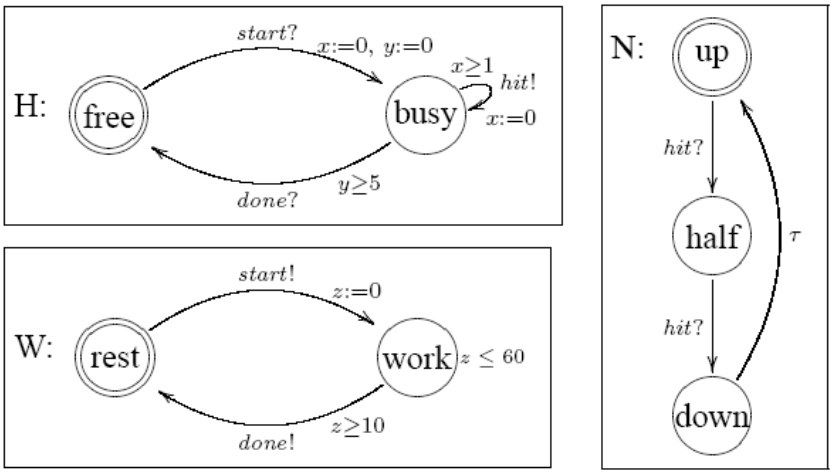
Let consider the following process:




1) Can this system be composed with the previous one?
2) Is the new net deadlock-free?

**Exercise 2:**

Let consider the network of three automata:




1) Edit this network into uppaal;
2) Verify the following formulae:
   – A[] (W.rest or z<= 100)
   – E[] (W.rest and H.busy)
   – A[] W.rest
   – E[] H.busy
   – W.work → □W.rest

**Exercise 3: (Uppaal: Fisher's protocol)**

Fischer's protocol is a well-known mutual exclusion protocol designed for **n** processes. It is a timed protocol where the concurrent processes check for both a **delay** and their **turn** to enter the critical section using a shared variable **id**.

- Starting from the initial location, processes go to a request location, **req**, if id==0 (which checks that it is the turn for no process to enter the critical section).
- Processes stay non-deterministically between 0 and k time units in **req**, and then go to the **wait** location and set **id** to their process ID (**pid**).
- There it must wait at least k time units, x>k, k being a constant, before entering the critical section **CS** (if it is its turn, id==pid).
- The protocol is based on the fact that after (strict) k time units with id different from 0, all the processes that want to enter the critical section are waiting to enter the critical section as well, but only one has the right ID.
- Upon exiting the critical section, processes reset **id** to allow other processes to enter CS.
- When processes are **waiting**, they may retry when another process exits CS by returning to **req**.

**Try to verify the following formulae:**

- **Safety:** A[] P1.cs + P2.cs + P3.cs + P4.cs <= 1
- P1.req --> P1.wait
- A[] not deadlock

**Exercise 4: (Uppaal :** train gate example**)**

In this example, a railway control system which controls access to a **bridge** for **several trains**. The system is defined as a **number of trains** and a **controller**. A train **cannot be stopped instantly** and **restarting also takes time**. Because of these temporal constraints, we assume:

1) When the train is approaching, a train sends the signal: **appr!**;
2) Thereafter, it has 10 time units to receive a stop signal (to stop safely before the bridge);
3) After these 10 time units, the train takes further 10 time units to reach the bridge if the train is not stopped.
4) If a train is stopped, it resumes its course when the controller sends a **go!** signal to it after a previous train has left the bridge and sent a **leave!** signal.

**Questions:**

1) How we can consider the bridge in this system?
2) In this first solution, we suppose that the controller uses a queue to save which trains are in which states. We propose a model composed of three automata: Train, Gate, and the Queue.
   - The template Queue: has a parameter **int[0, N] e**. This parameter indicates the train to be added in the queue or the train extracted from the queue;
   - The queue has (necessary) local variables: **int[0,N] list[N]** (elements of the queue), **len** (length of the queue);
   - The queue is synchronized with the gate on the actions: **notempty!**, **empty!**, **add?**, **hd!** (This last one gives the head of the queue).
   - When an element is removed from the queue, the list **int[0, N] list[N]** must be updated in order to make it reliable. The queue must have two states: **start** (where it works ordinary and can be synchronized with the gate) and a second state **shiftdown** (used to do the necessary arrangement once the first element is removed). This rearrangement is required because the removed element had its location at position 0, and after its removing the queue must update its structure using a shift. The queue changes from the state start towards the state shiftdown when it synchronizes with the gate through: **rem!**. You can use a local variable i to do this shiftdwon.
   - The gate has the two known necessary states: free, occupied (You must add other locations if necessary). From the location free, the gate asks the queue to see if it is **empty?** Or **notempty?**. If the queue is notempty, the gate takes the train in the head **hd?**, and it send a **go!** to this train and the gate will be occupied. If the queue is empty, the gate asks for any approaching train **app?**, it will add this train in the queue, then it will be also in occupied state. From the occupied state, the gate can waits for: **leave?** or an **appr?**. In case of a leave, the gate remove this train from the queue: **rem!**. In case of an **appr?**, the gate stops the train, then it adds it in the queue and rejoins the occupied state.
   - A train when it is far from the bridge, it is in a **safe state**, it send **appr!** for the gate to inform it with the necessary identifier (each train has its identifier). Then, if the train receives **stop?** Before 10 time units, it will be stoped (stop state), after it will wait for a **go?** to continue. Once it receives the go, the train will be in the **start** state, from where it can pass to the **cross state** after 7 time unit. From the cross state, it will wait for a **leave?** to rejoin the safe state. In another case, if the train did not receive **stop?** Before 10 time units it will reach the **cross state** automatically.
   - It is necessary to define some global variable known by the three templates, to decide about which train is active in the system at each time.

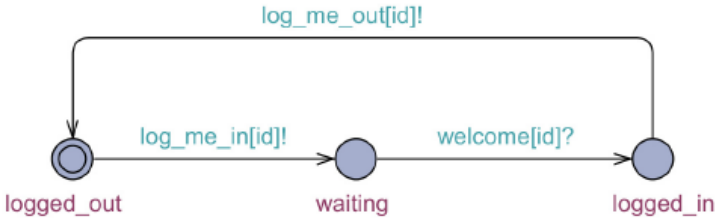
**Exercice 5 :**

Let us model a simple "chat" system consisting of two kinds of programs: a **user application U** and a **server S**. There are **three instances** of user applications: **U(1), U(2) and U(3)** and one **instance of the server: S**. We will model the logging in and out only, where the following property must be satisfied (apart from the fact that the system must work): a user may log in or out to the server.

- We will create two templates: U for the user application and S for the server.
- Write in the project's system declarations the following: ***system U, S;***
- We start by creating a template for a user application. Name it 'U' and write next to the name a parameter:

***const N id***

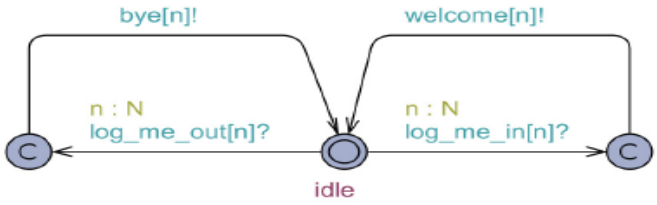
- Now we want to have three user applications numbered from 0 to 2, so we have to write the following in the project's declarations: ***typedef int[0,2] N;***
- The automaton of the user will be as follows:




- To use the channels, write the following in the project's declarations:

***chan log_me_in[N], log_me_out[N];***
***broadcast chan welcome[N], bye[N];***

- To server is as follows:




- Verify the following properties:
  The system cannot be deadlocked: *A[] not deadlock*
  Every user may log in, e.g.: *E<> forall (i : N) U(i).logged_in*