Partie I : Simulateur de circuits logiques

Le but de cette série est de réaliser un simulateur de circuits logiques simple. Ce simulateur est un cas particulier de simulateur basé sur des *événements discrets*.

La structure de données centrale du simulateur est *l'agenda*. L'agenda mémorise l'ensemble des *actions* qui auront lieu dans le futur, triées en fonction de leur heure de déclenchement. En plus de l'agenda, le simulateur garde le *temps courant de simulation*.

La simulation s'effectue en extrayant la première action a de l'agenda, en avançant le temps de simulation jusqu'au temps de déclenchement de a, en déclenchant a puis en recommençant jusqu'à ce que l'agenda soit vide.

Les circuits simulés par notre programme sont composés de *fils* et de *composants*. Un fil permet de transporter un signal digital, qui peut être 0 (représenté par `false`) ou 1 (représenté par `true`). Un composant est connecté à des fils d'entrée et de sortie, et la valeur qu'il place sur ses fils de sortie est une fonction des valeurs se trouvant sur ses fils d'entrée. Le changement des valeurs se trouvant sur les fils d'entrée ne produit pas instantanément un changement des valeurs sur les fils de sortie : chaque composant a un *délai* qui lui est propre.

À chaque fil est associé un ensemble de fonctions, nommées également *actions*, qui sont activées à chaque changement de la valeur transportée par le fil.

Pour cette série nous utiliserons trois composants de base : un composant NON, un composant ET et un composant OU. Tous nos circuits seront composés à partir de ces pièces de base.

Exercice 1

Écrivez la méthode `orGate` dans la classe `CircuitSimulator` qui réalise une porte OU. Faites deux versions de cette porte : la première similaire à la porte ET, la seconde définie uniquement en termes des portes ET et NON.

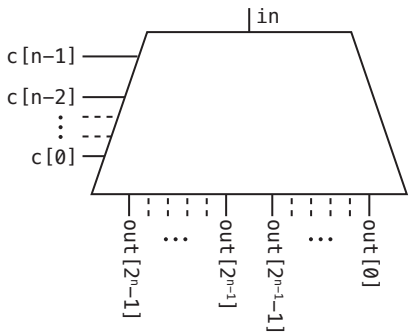

FIGURE 1 – Démultiplexeur à n entrées

Exercice 2

Définissez une fonction `demux` qui réalise un démultiplexeur à n fils de contrôle (et 2^n sorties), tel que celui présenté à la figure 1. Pour mémoire, un démultiplexeur aiguille un signal d'entrée sur une sortie parmi 2^n en fonction de n signaux de contrôle. Tous les autres signaux sont mis à 0. La fonction `demux` a le profil suivant :

```
def demux(in: Wire, c: List[Wire], out: List[Wire]): Unit
```

On admet ici que les listes de fils de contrôle (`c`) et de sortie (`out`) sont triées par index décroissant. C'est-à-dire qu'en tête de la liste `c` se trouve le fil de contrôle d'indice $n-1$ et en tête de la liste `out` se trouve le fil de sortie d'indice 2^n-1.

Votre implantation doit être récursive et doit se baser sur les portes que vous avez implantées précédemment. Pour vous aider, pensez au fait que le démultiplexeur le plus simple qui soit comporte 0 fils de contrôle et 1 fil de sortie.

Part II : Epidemy Simulation

In this part, you are going to write an epidemy simulator using the simulation framework you completed in the first part (the `Simulator` class).

The scenario is as follows : the world ("Scalia") consists of a regular grid of 64 rooms where each room is connected to four neighbouring rooms. Each room may contain an arbitrary number of persons. Scaliosis — a vicious killer virus — rages with a prevalence rate of 1% among the peaceful population of Scalia. It spreads with a transmissibility rate of 40%.

In the beginning, 300 people are equally distributed among the rooms. Each step of the simulation corresponds to one simulated day. The disease and behaviour of people is simulated according to the following rules.

Rules

1. After each move (and also after the beginning of the simulation), a person moves to one of their neighbouring rooms within the next 5 days (with equally distributed probability). Note that the first row is considered to be a neighbour of row eight (and vice versa) ; analogously, the first column is a neighbour of column eight (and vice versa).
2. When a person moves into a room with an infectious person he might get infected according to the transmissibility rate, unless the person is already infected or immune. A person cannot get infected between moves (this is slightly unrealistic, but will simplify your implementation).
3. A person avoids rooms with sick or dead (visibly infectious) people. This means that if a person is surrounded by visibly infectious people, he does not change position ; however, he might change position the next time he tries to move (for example, if a visibly infectious person moved out of one of the neighbouring rooms or became immune).
4. When a person becomes infected, he does not immediately get sick, but enters a phase of incubation in which he is infectious but not sick.
5. After 6 days of becoming infected, a person becomes sick and is therefore visibly infectious.
6. After 14 days of becoming infected, a person dies with a probability of 25%. Dead people do not move, but stay visibly infectious.
7. After 16 days of becoming infected, a person becomes immune and is no longer visibly infectious, but remains infectious.
8. After 18 days of becoming infected, a person turns healthy. He is now in the same state as he was before his infection, which means that he can get infected again.

Graphical display

To make the project more interesting, we provide you with a graphical display class (class `EpidemyDisplay`) that is used to visualize simulations. The display is automatically available to you when you complete the provided (partial) `EpidemySimulator` class. To start the graphical output (and the simulation), run the `simulation.EpidemyDisplay` object.

Healthy persons are painted in green color. Persons that have their `sick` field set to true are painted in red color, suitable to indicate visibly infected people. Immune (i.e. infected but not sick) people are painted in yellow.

Problems

1. Implement an epidemy simulator according to the above rules by completing the partial `EpidemySimulator` class (available on the course web site). Make sure to first inspect the code of `EpidemyDisplay` and `Simulator`, and understand how the display uses the simulator's agenda to perform the simulation. Your task is to add the correct `WorkItem`s to the agenda.
2. Determine the average number of dead people after 150 days over a set of 5 experiment runs.
3. Extend your simulation with air traffic : when a person decides to move, she will choose to take the airplane with a probability of 1%, thereby moving to a random room in the grid (rooms with visibly infected people are not avoided). How does air traffic impact the epidemy ? Determine the average number of dead people over 5 runs.
4. *Pandemic Response.* To reduce the number of casualties, president Scalozy of Scalia decides to enforce one of the following health policies :
   (a) *Reduce Mobility Act.* The mobility of people is decreased by half. The mobility of a visibly infected person is further reduced by half.
   (b) *The Chosen Few Act.* 5% of people (VIPs such as pop singers, football players, etc.) are given vaccines when first created. They never become infected.

   Which of the health policies is more effective (use the version with air traffic for your tests) ? Write your answers down as a comment in the source file.
5. Please add boolean variables to your source code which allow to easily turn on and off the last three features you implemented (air traffic, reduced mobility and vaccination). The version you hand in should have all of these features turned off.