Programmation IV — Mini project 4: Simulation — 9 mai 2007

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.

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. 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 $2^n - 1$.

Indication : pensez récursif, et songez 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 rooms where each room is connected to four neighboring 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 50%.

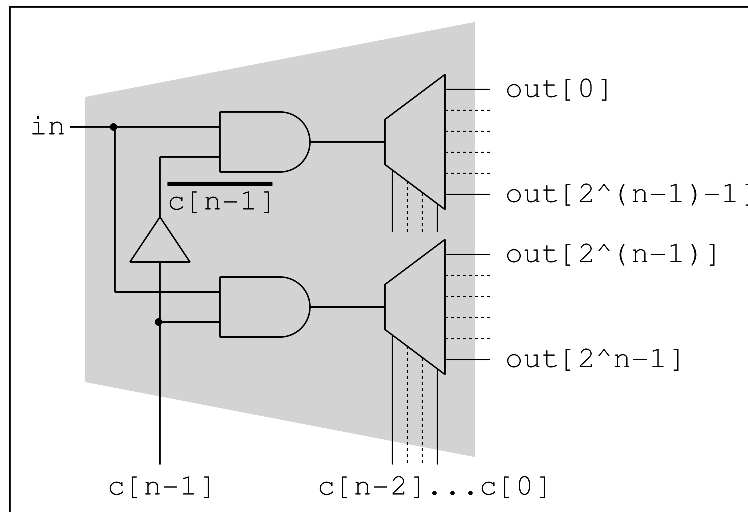

Figure 1: Démultiplexeur à n entrées

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

Rules

1. Persons move to one of their neighboring rooms within the next 5 days (with equally distributed probability).
2. When a person moves into a room with an infected person it might get infected according to the transmissibility rate, unless the person is already infected or immune.
3. Persons avoid rooms with visibly infected (sick or dead) people.
4. After an incubation period of 4 days, infected persons become sick.
5. After 10 days of becoming infected, persons die with a probability of 25%. Dead people do not move, but stay visibly infected.
6. After 12 days of becoming infected, persons become immune, but stay non-visibly infected.
7. After 14 days of becoming infected, persons turn healthy (not infected, not immune).

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 web site). Determine the average number of dead people over 5 runs. Each run simulates 150 days.

2. 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.

3. *Pandemic Response.* To reduce the number of casualties, the government of Scalia (with their new president Scalozy) decides to enforce one of the following health policies:

   (a) *Reduce Mobility Act.* The mobility of people is decreased by half. Furthermore, a visibly infected person is allowed to move only with half its normal speed.

   (b) *The Chosen Few Act.* 5% of the people (probably national leaders, doctors or police) are given vaccines when first created. They never become infected.

   Which of the health policies is more effective?