# INF6603 — Travail pratique 1
# Modélisation et vérification à l'aide d'UPPAAL

Automne 2017

## Objectif

Ce sujet vise à vous familiariser avec la modélisation d'un système et les techniques de vérification par model-checking avec l'outil UPPAAL.

Les exercices sont basés sur des jeux (de type puzzle, ou casse tête). Le but est de modéliser ces jeux dans UPPAAL et d'utiliser le model-checker afin de déterminer une solution, si il y en existe une.

Les exercices sont inspirés de http://people.cs.aau.dk/~bnielsen/TOV08/ESV04/exercises

## Exercice 1 : Remplissage de seaux (8/20)

Le problème de remplissage des seaux (*jug filling*) est un casse-tête classique.

On dispose d'un certain nombre de seaux (vides), d'une provision infinie d'eau (un robinet), et d'un évier. Les actions suivantes sont autorisées :

a) Remplir un seau au robinet jusqu'à ce qu'il soit plein.
b) Vider totalement un seau.
c) Transférer le contenu d'un seau dans un autre. On arrête alors de verser dès que le seau de destination est plein ou que le seau de départ est vide.

Le but du jeu est de parvenir à remplir chaque seau avec un volume précis d'eau.

Dans le cas de base de ce jeu, le joueur a deux seaux, numéroté $S1$ et $S2$, de capacités respectives de 3L et 5L. L'objectif est d'atteindre une configuration où le seau $S1$ est vide tandis que le seau $S2$ contient 4L d'eau.

Dans le cas étendu, on considère $k$ seaux, de volumes respectifs $v_1, ..., v_k$. L'objectif est d'atteindre une configuration $c_1, ..., c_k$. Les volumes $v_i$ et $c_i$ sont des entiers naturels.

On notera $(v_1, ..., v_k) \rightarrow (c_1, ..., c_k)$ ce problème.

1) Résoudre à la main le cas de base du jeu.

2) Modéliser le cas étendu du jeu dans UPPAAL, en respectant les contraintes suivantes :

   a) Le fichier des définitions globales ne doit contenir que des définitions de canaux de communication ou de variables globales (qui ne serviront qu'à transférer des valeurs entre les processus).
   b) Les quantités d'eau actuelle et maximale dans un seau doivent être locales.

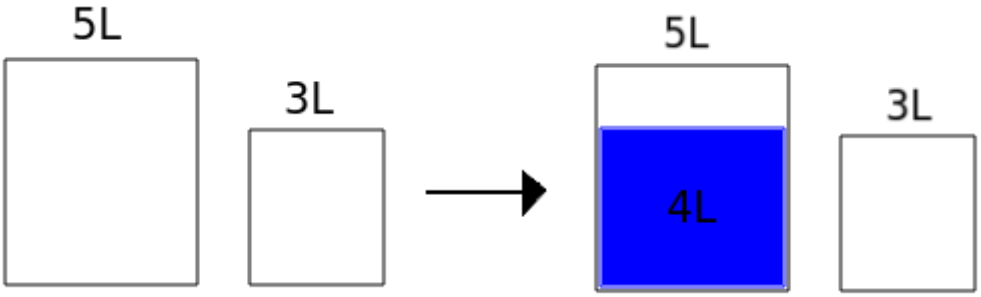


Figure 1: Cas de base du jeu

**Indices :**

- utiliser un template paramétré pour représenter les seaux.
- représenter les actions autorisées par des transitions dans les automates.
- synchroniser les actions de transfert par des canaux de communication.
- utiliser l'ordre des *updates* des actions synchronisés pour transférer une valeur entre des processus (par l'intermédiaire d'une variable globale).

3) Exprimer par une propriété CTL le fait de gagner le jeu. On pourra utiliser une fonction intermédiaire pour déterminer si une configuration est gagnante.

4) Prouver qu'il existe une solution pour la configuration de base du jeu $(5, 3) \rightarrow (4, 0)$. Rechercher la solution nécessitant le moins d'étapes.

5) Déterminer si le modèle vérifie la propriété d'absence de blocage. Ce résultat est-il lié à l'existence ou non d'une solution ?

6) Déterminer si il existe une solution pour les cas suivants, et donner la solution nécessitant le moins d'actions :

   - $(5, 3) \rightarrow (3, 2)$
   - $(9, 11) \rightarrow (8, 0)$
   - $(17, 12, 5) \rightarrow (8, 2, 0)$

## Exercice 2 : Rush Hour (10/20)

Rush Hour est un casse-tête classique. L'objectif est de permettre à la voiture rouge de quitter le plateau (par la sortie située en face d'elle), en déplaçant les autres véhicules pour libérer le passage.

Le jeu se joue sur un plateau de 6x6. Les véhicules sont des blocs de tailles 1x2 (voitures) ou 1x3 (camions), en position verticale ou horizontale. Un véhicule peut avancer ou reculer (tant qu'il n'atteint pas un autre véhicule ou le bord du plateau), mais il ne peut pas tourner ou se déplacer latéralement.

La voiture rouge est situé face à la sortie.

1) Modéliser le jeu dans UPPAAL.


Figure 2: Rush Hour

**Indices :**

- utiliser un tableau à deux dimensions `int board[6][6]` pour représenter les cases occupées ou non.
- définir un type pour représenter les véhicules

```
typedef struct Vehicle {
    int direction;
    int x;
    int y;
    int length;
    ...
}
```

- définir les fonctions suivantes :
  - `is_move_autorized` : indique si un véhicule peut se déplacer d'un certain nombre de case dans un sens donné.
  - `move` : réalise un déplacement valide en mettant à jour la grille.
  - `is_solved` : indique si le chemin est libre pour la voiture rouge.

2) Utiliser UPPAAL pour résoudre les puzzles de la Figure 3, en trouvant la solution la plus *courte* (nécessitant le moins de déplacements).

On considère de plus qu'une voiture à besoin d'une unité de temps pour se déplacer d'une case, alors qu'un camion a besoin de 2 unités de temps pour se déplacer d'une case.

3) Étendre le modèle pour prendre en compte ces contraintes temporelles. Trouver la solution la plus *rapide* (nécessitant le moins de temps) aux puzzles de la Figure 2.

4) Vérifier les propriétés suivantes, en présentant un contre-exemple si elles ne sont pas valides.

   a) La voiture rouge ne quitte jamais la ligne centrale.
   b) Le système n'est pas dans un état de deadlock tant que le jeu n'est pas terminé.
   c) Il est possible de gagner en ne déplaçant aucun camion.


Figure 3: Rush hour — puzzles à résoudre

## Consignes et remises (2 / 20 pour le rapport)

Le travail est à réaliser par équipe de 2. Une seule remise contenant les noms des deux membres du groupe est nécessaire.

Remettez votre travail via moodle, sous la forme d'une archive contenant :

- les fichiers UPPAAL (.xml et .q).
- un rapport expliquant les modèles que vous avez conçus (automate et code), les propriétés, vos résultats et vos éventuelles hypothèses.

La qualité de la langue, du code et des commentaires sera prise en compte (indentation, utilisation de constantes nommées. . . ).

**Date limite de remise** : 27 octobre 2017, 18h.