HAUTE ÉCOLE
D'INGÉNIERIE ET DE GESTION
DU CANTON DE VAUD
www.heig-vd.ch

RAPPORT DE PROJET

PROGRAMMATION CONCURRENTE (PCONC)

Développement d'une simulation de gestion de carrefour avec réseau de Petri

Auteur :
Léonard BISE

Enseignant :
Eytan ZYSMAN

Informatique Embarquée (ISEC)

8 Juin 2018

Table des matières

Table des figures

Liste des tableaux

1 Introduction

Ce projet consiste a développer une simulation qui permet la gestion d'un carrefour entre deux routes à sens unique avec deux feux de signalisations. L'objectif est de permettre la gestion du système en utilisant des réseau de Petri qui interagit avec le monde réel au moyen d'événement extérieur et d'action déclenché par les places du réseau.

Dans le cadre du projet deux phases ont été réalisées, la première phase consistait à développer le système sans utiliser de timer gérer par réseau de Petri, à la place une simple attente était effectué entre le changement des états des feux. Ce rapport décrit la deuxième phase qui consiste à avoir deux réseau de Petri fonctionnant en parallèle, un pour la gestion du carrefour, et le deuxième pour la gestion d'un timer.

La figure 1.1 montre une vue de l'interface graphique de la simulation. Sur le GUI sont représenté les éléments du système, les voitures évoluant sur la route, les feux de signalisation ainsi que le détecteur de véhicule avant le carrefour (zone en jaune claire quand il n'y a pas de véhicule, et jaune foncé lorsque des véhicules sont détectés). Enfin l'interface dispose d'un bouton (en haut à gauche), qui permet de tuer les Threads de génération de véhicule, ce qui termine la simulation.

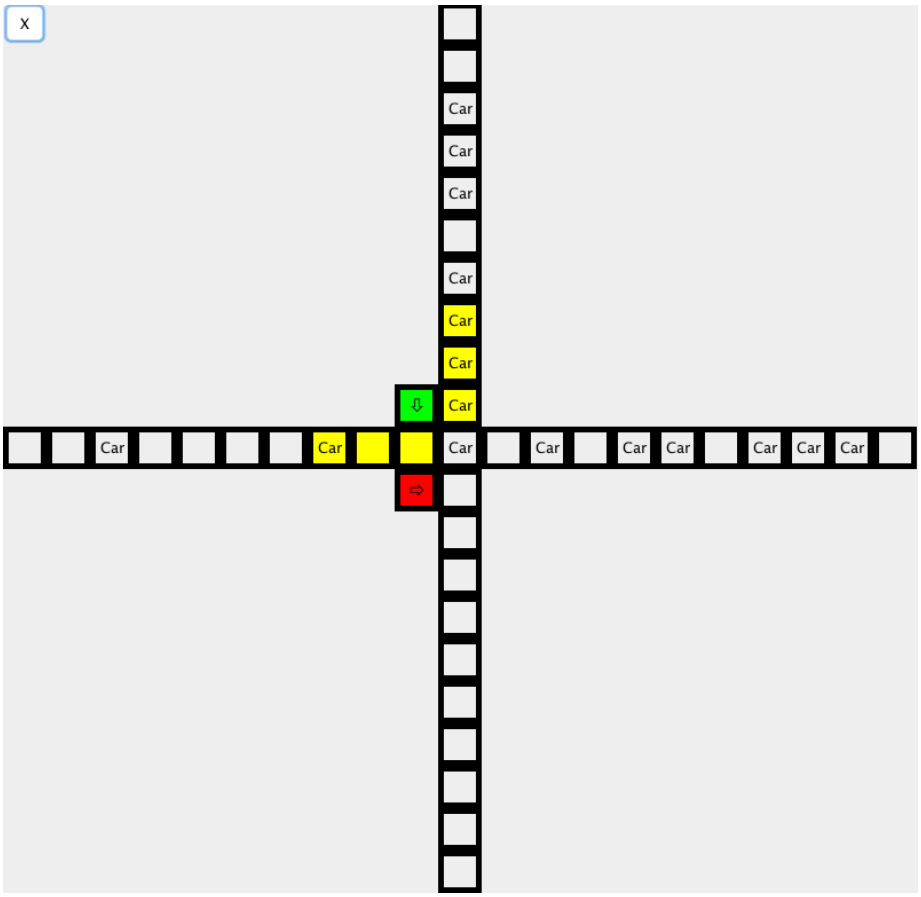

FIGURE 1.1 – Interface graphique de l'application

2 Cahier des Charges

Dans le cadre du projet il nous est demandé de créer un projet codé en java comprenant les éléments suivants :

- Créer un gestionnaire de réseau de petri auquel il est possible de donner un fichier de configuration pour créer un paysage (création des places, transition, arcs de pre et post incidence).
- Lorsque le réseau de Petri est créé, il est possible de lancer la simulation, à chaque tick le gestionnaire va faire évoluer le réseau de Petri en regardant quelle places sont sensibilisée, il va ensuite consommer et produire des jetons aux places requises.
- Lorsqu'un jeton arrive dans une place, si l'utilisateur a lié une action à la place, le gestionnaire va lancer l'exécution de l'action
- L'environnement extérieur doit pouvoir informer le gestionnaire du réseau de Petri des événements qui se sont déclenchés, ceci permettra aux transitions d'être franchies lorsqu'elles sont sensibilisées.
- Utiliser le gestionnaire de réseau de Pétri afin d'implémenter un réseau qui permet la gestion d'un carrefour avec deux flux perpendiculaire de voiture et qui est à sens unique.
- Utiliser le gestionnaire de réseau de Pétri afin d'implémenter un autre réseau qui permet la gestion d'un timer. Ce réseau permettra au réseau de gestion du carrefour de gérer les transitions requises pour le changement d'état des feux de signalisation.

En utilisant le gestionnaire de réseau de Petri, il faudra créer les éléments suivants et les faire interagir avec le réseau pour faire évoluer le système.

- Un objet responsable de la gestion des deux routes, il permettra aux véhicules de savoir s'ils peuvent avancer ou si un véhicule se trouve déjà dans la case devant eux. La gestion se fait grâce à deux vecteurs un par flux, le croisement est donc présent dans chacun des deux flux
- Deux threads, un par flux, qui créeront à des temps aléatoire de nouveaux véhicules, ils seront créés au début de la route
- Un objet véhicule, cet objet, dès sa création, commencera à avancer de case en case en direction de la fin de la route. Lorsqu'il arrivera juste devant le carrefour il vérifiera si le feu est vert avant de pouvoir entrer dans le carrefour. Lorsqu'il atteint la fin de la route il disparait
- Deux détecteurs, un par flux, qui auront comme tâche de déclencher un événement lorsqu'ils détecteront un véhicule présent juste avant le croisement
- Deux détecteurs, un par flux, qui auront comme tâche de déclencher un événement lorsqu'ils détecteront que le croisement est vide
- Deux feu de signalisation, un par flux, qui autoriseront ou non les véhicule du flux à passer dans le croisement

3 Architecture

L'architecture de l'application ainsi que les interactions entre les objets est décrite dans la figure 3.1. Elle montre les interactions principales entre les diverses classes

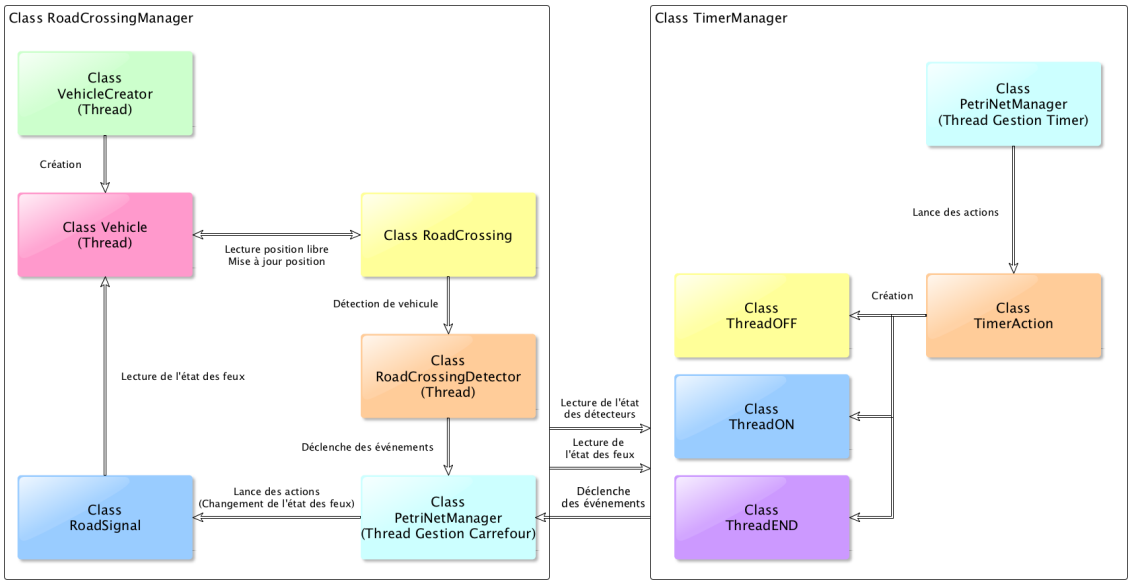

FIGURE 3.1 – Architecture de l'application

4 Gestion des réseaux de Pétri

La classe PetriNetManager est la classe centrale du gestionnaire de réseau de Pétri. Après l'instanciation de la classe, un réseau peut être initialisé en utilisant la méthode loadFromTextFile avec un fichier de configuration. Ceci créera toute les places, transitions et arcs requis pour le réseau décrit.

Une fois le réseau initialisé l'utilisateur à la possibilité de faire les actions suivantes :

- Lier une place du réseau avec une classe implémentant l'interface PetriPlaceInterface avec la méthode setPlaceAction. Lorsque le gestionnaire du réseau détecte l'entrée d'un jeton dans la place, il déclenchera automatiquement l'exécution de l'action
- Lancer le gestionnaire de réseau de Pétri avec la méthode start. Cette action lancera l'exécution du Thread de PetriNetManager qui après chaque tick effectuera la simulation du réseau
- Une fois le gestionnaire lancé, l'utilisateur doit déclencher des événements liés à une certaine transition au moyen de la méthode setEventState ce qui permettra le franchissement de la dite transition si elle est préalablement sensibilisée

Le cœur de la simulation du réseau de Pétri est effectuée par la méthode step. Cette méthode simule un pas de l'évolution du réseau de Pétri, elle est cadencée par la durée d'un tick, c'est à dire un temps entre deux pas.

La liste suivante décrits les opérations effectuées par la méthode step, elle se découpe en quatre phases.

- Phase 0 : Les actions des places qui ont eu de nouveaux jetons sont appelées
- Phase 1 : Le gestionnaire détermine les transitions qui sont sensibilisées, c'est à dire que les places qui y sont liées contiennent le nombre de jeton requis, et les ajoute dans une liste de transition sensibilisée. La liste est ensuite mélangée pour que les transitions aient un ordre aléatoire
- Phase 2 : Pour chaque transition qui sont dans la liste des transitions sensibilisées, le gestionnaire vérifie si l'événement associé a été déclenché, c'est à dire si il est présent dans la queue d'événement. Si c'est le cas, les jetons des places qui sont liées à la transition sont consommés et elle est ajoutée à la liste des transitions franchies. La liste des transitions sensibilisées est à nouveau parcourue pour vérifier si certaines transitions ne le sont plus suite à la consommation de jetons
- Phase 3 : Les jetons nécessaire sont produit dans les places liées aux transitions qui ont été franchies

D'autres classes sont utilisées pour la gestion des réseaux, elles sont décrites ci-dessous.

- PetriNetManagerTest : Contient quelques tests du bon fonctionnement du gestionnaire de réseau
- PetriArc : Décrit les caractéristiques lié aux arcs de pré et post incidence comme le type (simple, test, inhibit) et leur poids
- PetriPlace : Une place dans un réseau de Pétri ainsi que sont action liée si elle en as une
- PetriPlaceInterface : Une interface qui permet à la classe qui l'implémente d'être ensuite

déclenchée par le gestionnaire de réseau lorsqu'un jeton entre dans la place qui y est liée.

— PetriTransition : Contient les informations relative à une transition faisant partie du réseau de Pétri

5 Description des acteurs

Le projet de gestion de carrefour demande la création de différents acteurs qui sont listé ci-dessous.

- RoadCrossingManager : Le gestionnaire du carrefour, c'est cette classe qui créer tout les objets nécessaire et qui permet de les lier. Elle créer et configure le réseau de Pétri au travers de la classe PetriNetManager.
- RoadCrossing : La grille sur laquelle les voitures se déplace. Cette classe est passive, elle ne fait que d'être modifié par d'autre classe. Les instance de Vehicle, l'utilise pour savoir si elles peuvent avancer ou si un autre véhicule se trouve déjà devant eux
- RoadCrossingDetector : Les détecteurs utilisés pour déclencher des événements dans le réseau de Pétri. Il y'en a deux par flux. Un placé juste avant le carrefour déclenchera l'événement lié à la détection de flux voulant entrer dans le carrefour, cela permet au réseau de Pétri de faire passer le feu de l'autre flux au rouge et ainsi permettre au premier flux de pouvoir passer. Le deuxième type de détecteur vérifiera que le carrefour est bien vide, ce détecteur ne déclenche pas d'événement dans le réseau mais son état est lu régulièrement par le gestionnaire du réseau de Pétri pour le Timer, ce qui permettra de s'assurer que le carrefour est vide avant de mettre le feu du flux stoppé au vert.
- RoadSignal : Le feu routier, c'est un élément passif qui ne fait que d'avoir un état, soit rouge ou alors vert. Il y a un feu par flux, les véhicules lorsqu'ils arrivent devant le carrefour doivent veiller à vérifier que le feu est vert avant de pouvoir y rentrer. L'état des feux est modifiés au travers de deux actions du réseau de Pétri.
- SignalStateAction : Cette classe implémente l'interface PetriPlaceAction et défini donc l'action à effectuer lorsqu'un feu doit changer d'état. Elle est directement appellé par le gestionnaire de réseau de Petri quand nécessaire et elle contient une réference au feu qu'elle doit gérer.
- Vehicle : Définit le comportement d'un véhicule, après avoir été créé il commence à avancer régulièrement le long de son flux en utilisant la grille définie par la classe RoadCrossing. Juste avant de rentrer dans le carrefour la couleur du feu sera vérifiée pour voir si il est possible d'avancer. Lorsqu'elle arrive à la fin de la route, le véhicule a terminé sa vie et est détruit.
- VehicleCreator : Ce Thread est responsable de la création de véhicule dans un certain flux de manière régulière
- RoadCrossingGUIThread : La tâche qui est responsable de rafraîchir les informations affiché sur l'interface graphique

Le timer définit les acteurs suivant.

- TimerManager : Le gestionnaire du timer, c'est cette classe qui créer tout les objets nécessaire et qui permet de les lier. Elle créer et configure le réseau de Pétri au travers de la classe PetriNetManager.
- TimerAction : Cette classe est utilisée pour créer les différents Thread qui sont utile pour l'évolution des états du Timer. Les actions sont déclenché par le PetriNetManager lorsque des jetons arrivent dans les places du réseau.
- ThreadOFF : Ce Thread vérifie régulièrement si la condition nécessaire pour enclencher le timer est établit. C'est à dire si un des deux flux est au vert et que des véhicules sont

détectés dans l'autre flux. Une fois sa tâche remplie, le timer passe en mode ON qui est géré par la classe ThreadON.

— ThreadON : Ce Thread déclenche un timer à sa création puis il vérifie si le temps imparti est écoulé. Si le flux qui est au vert se tarit, alors le Thread déclenche l'événement directement ce qui permet de ne pas attendre sans raison. Une fois sa tâche remplie, le timer passe en mode END qui est géré par la classe ThreadEND.

— ThreadEND : Le ThreadEND attend que le carrefour soit vide puis il déclenche l'événement permettant au réseau de Pétri du gestionnaire de carrefour de faire basculer le feu au vert du flux en attente. Une fois sa tâche remplie, le timer passe en mode OFF qui est géré par le classe ThreadOFF.

6 Echanges entre acteurs

Le réseau de Pétri est composé de places et de transitions. Afin de pouvoir faire évoluer le réseau de Pétri, des événements relatif au monde réel, c'est à dire à l'état du carrefour, doivent être détectés. Lorsque le réseau est informé du déclenchement d'un événement, il peut alors faire évoluer le réseau en conséquence, c'est à dire de consommer des jetons et d'en produire.

Dans le réseau de Pétri de la gestion du carrefour il existe 4 transitions, voir le fichier config/-roadCrossingRDP.cfg qui décrit le réseau de Pétri pour la gestion du carrefour.

— CrossingEmptyA : Le carrefour du flux A est vide.
— CrossingEmptyB : le carrefour du flux B est vide.
— ALaneFilled : Des véhicules sont présent dans le flux A.
— BLaneFilled : Des véhicules sont présent dans le flux B.

Lorsqu'un feu est vert, le système vérifie en permanence l'état de l'autre flux afin de savoir quand il devra donner la main à l'autre flux. La vérification de cet état est fait par la classe RoadCrossingDetector, cependant le système ne doit pas donner directement la main à l'autre flux car dans le cas ou beaucoup de véhicules circulent les feux passeraient en permanence du vert au rouge. Pour éviter ce cas de figure, le réseau de Pétri Timer entre en jeu. Ce timer permet d'attendre un certain temps avant de donner la mains à l'autre flux, dans le cas ou le flux qui est actuellement au vert se tarit alors le timer donne la main à l'autre flux immédiatement.

Les événements pour les carrefour vide, c'est à dire CrossingEmptyA et CrossingEmptyB, sont directement déclenché par la classe RoadCrossingDetector.

Les réseaux de Pétri sont informé du déclenchement des événements au travers de la méthode setEventState, qui donne l'état d'un événement du RDP par exemple CrossingEmptyA et son état (true/false). Lorsque l'événement est à true, alors le RDP peut franchir la transition si elle est sensibilisée.

La classe RoadCrossingManager contient toutes les références des instances des objets du système relatif à la gestion du carrefour, une référence à cet objet est passée à tout les acteurs qui ont besoin d'interagir. Par exemple la classe Vehicle, disposera d'une telle référence afin de pouvoir interroger le carrefour pour savoir si il peut avancer ou de savoir l'état du feu de signalisation par exemple.

De manière similaire une référence de la classe TimerManager sera distribuée aux éléments du timer qui doivent interagir.

Afin d'éviter des problèmes liés aux accès concurrents qui sont effectués sur la classe RoadCrossing par les Vehicle créés durant l'exécution de la simulation, les Vehicle accèdent aux informations de la classe RoadCrossing au travers de bloque de code dit "Synchronized", cela permet de laisser la main au Thread de la classe Vehicle jusqu'à ce que l'entièreté du déplacement à été effectué.