Université de Nantes
Faculté des sciences et Techniques
M2 ALMA - Architecture Logicielle
Option Génie Logiciel

Formal Software Engineering

Projet
Aux Feux

Julien OUVRARD
Damien MAUSSION
Thibaut PICHAUD
Tristan JARRY

Octobre - Novembre 2015

Table des matières

1 Introduction

Dans le cadre du module *Formal Software Engineering*, nous devions modéliser deux système de feux différents. L'un comportant deux feux face à face (voir partie 2 et partie 3), l'autre comportant une intersection en T régularisé par 3 feux. Pour les deux feux, nous avons vu leur fonctionnement sans synchronisation (partie 2) et avec synchronisation (partie 3). L'étude des feux de l'intersection en T s'est faite avec et sans contraintes de temps (partie 4). Suite à cette étude, nous avons réalisé son implémentation en java (partie 5).

2 Un système à deux feux, sans temps

Cette partie correspond à la modélisation d'une route avec deux feux sans notion de temps.

2.1 Question 1

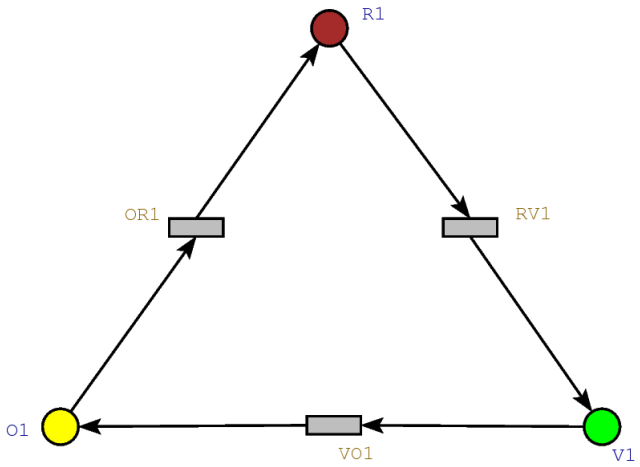


FIGURE 1 – Simple feu tricolore sous Roméo

Un feu simple tricolore, se compose de 3 états : *Vert*, *Orange*, *Rouge*. Ainsi que 3 transitions, une entre chaque état.

2.2 Question 2

Ce graphe de marquage est composé des 8 états possibles, ainsi que toutes les transitions qu'il peut exister entre eux.

2.3 Question 3

2.4 Question 4

Le graphe de marquage proposé ici est plus simple que le précédent. En effet, il n'y a que 6 états possibles, cela est assuré grâce aux places supplémentaires ajoutées.

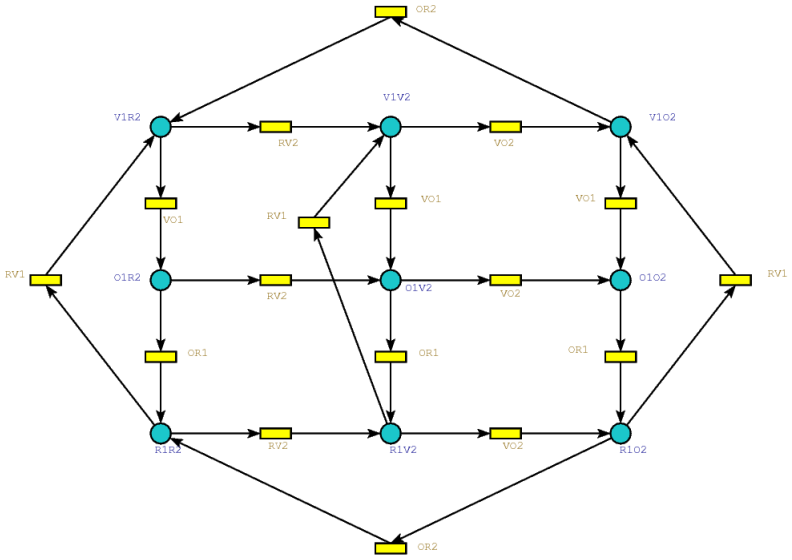


FIGURE 2 – Graphe de marquage avec 2 feux

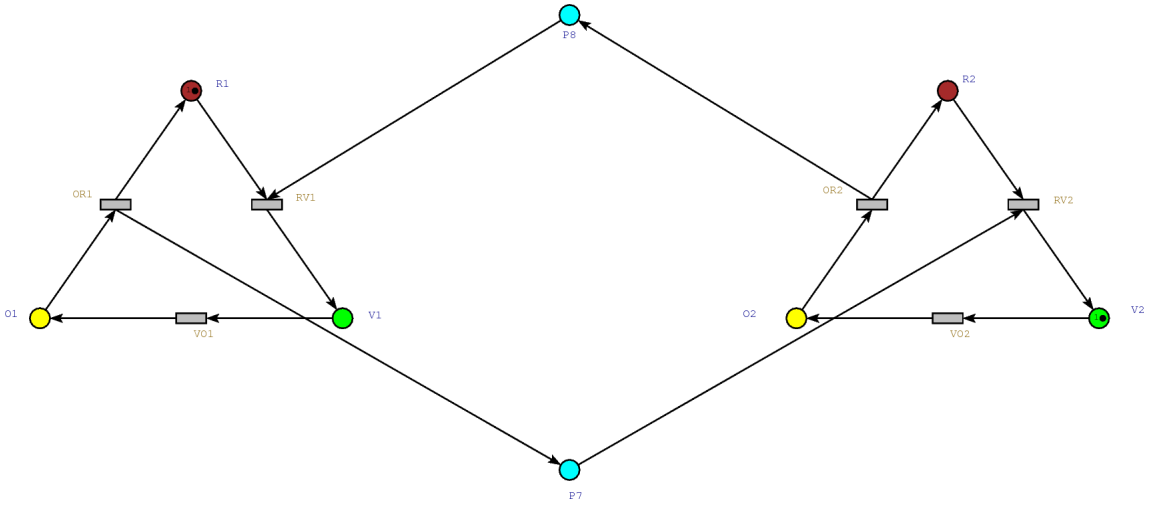


FIGURE 3 – Réseau de Petri avec deux feux

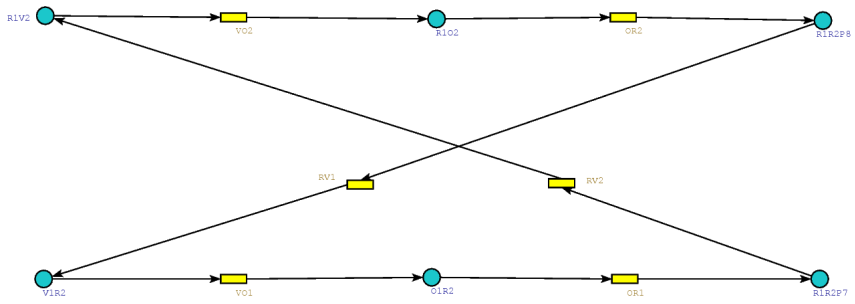


FIGURE 4 – Graphe de marquage des feux synchronisés

2.5 Question 5

2.5.1 Propriété de sûreté

Cette propriété doit assurer que jamais les deux feux ne doivent au vert ou à l'orange simultanément. Ce qui correspond à avoir les deux feux au rouge.

Syntaxe du Model-checker de Roméo : $AG[0, inf](R1 + R2 >= 1)$
Syntaxe LTL : $\forall\Box(R1 \wedge R2)$

2.5.2 Propriété de vivacité

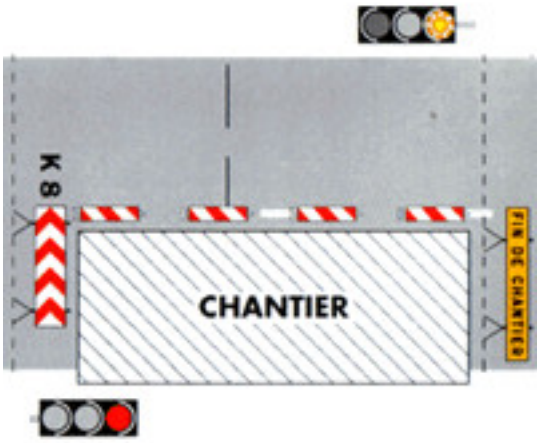
Cette propriété doit permettre aux voitures de circuler, c'est à dire ne pas avoir de feu au rouge indéfiniment (ni au vert).
Syntaxe du Model-checker de Roméo : $(R1 = 1) - - > [0, inf](R1 = 0)$
Syntaxe LTL : $R1 \longrightarrow \neg R1$

2.6 Question 6

Les deux propriétés ont été testées via le Model-checker et elles renvoient toutes les deux *true*. Nous pouvons donc dire qu'elles sont assurées par note modèle.

3 Un système à deux feux, avec temps

Dans cette partie, nous allons étudier 2 feux face à face comme le dessin ci-dessous :



Contrairement à la partie précédente, ici les feux seront munis d'horloges. Dans un premier temps nous étudierons le cas où les deux feux ne sont pas synchronisés (question 7), ils possèdent chacun une horloge qui leur est propre. Le second cas sera réalisé avec une synchronisation au niveau des feux (question 8) grâce à un contrôleur.

3.1 Question 7

Dans cette question nous étudions les feux de chantier sans synchronisation. Les deux feux sont indépendants l'un de l'autre, ils possèdent chacun leur propre horloge. Nous réalisons la modélisation de ces feux avec le logiciel **U**PPAAL. Les deux patrons des feux ont pratiquement la même implémentation. Les deux seules différences sont le nom des horloges et l'état initial. Le premier est représenté de la manière suivante :

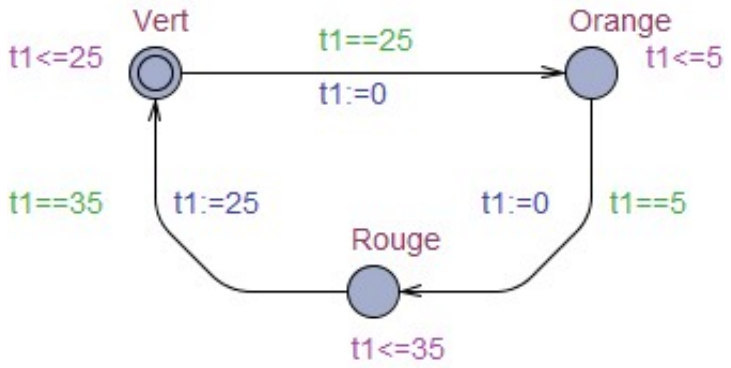


FIGURE 5 – Modélisation du Feu1

On remarque ici que l'état initial est l'état Vert et que le nom de l'horloge est t1. Afin de ne pas se tromper à calculer le temps complet du cycle, à chaque nouvel état on réinitialise (*mise à jour* sous **U**PPAAL) l'horloge.

Voyons maintenant la représentation du 2ème feu :

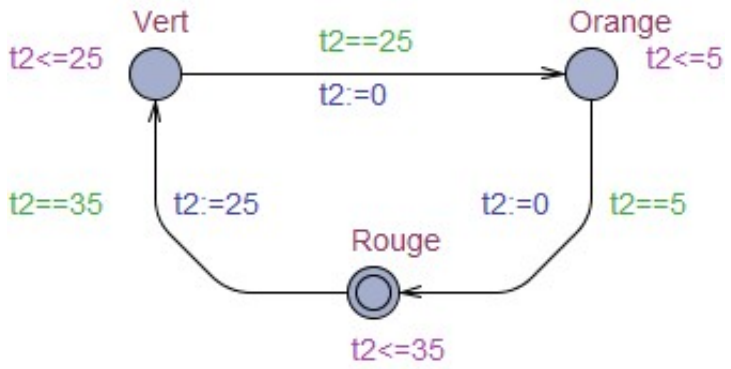


FIGURE 6 – Modélisation du Feu2

Nous pouvons observer ici que le nom de l'horloge a changé ainsi que l'état initial. En effet comme le feu1 possède un état initial sur l'état vert, pour que les deux feux ne soient pas verts en même temps il faut mettre l'état initial du feu2 sur l'état rouge.
Suite à la réalisation de ces deux feux, ils faut réaliser des vérifications. Ces vérifications sont les mêmes que pour la questions suivante, c'est pour ça que nous allons voir un première écriture possible de ces vérification pour cette question :

1. Vérifier s'il est probable que le feu1 soit à vert et le feu2 à vert :
   E<>F1.Vert and F2.Vert
   la propriété n'est pas satisfaite, les deux feux ne sont jamais verts en même temps
2. Vérifier s'il est probable que le feu1 soit à orange et le feu2 à orange :
   E<>F1.Orange and F2.Orange
   la propriété n'est pas satisfaite, les deux feux ne sont jamais oranges en même temps
3. Vérifier s'il est probable que le feu1 soit à rouge et le feu2 à rouge :
   E<>F1.Rouge and F2.Rouge
   la propriété est satisfaite, les deux feux peuvent être rouges en même temps
4. Vérifier s'il est probable que le feu1 soit à vert et le feu2 à orange :
   E<>F1.Vert and F2.Orange
   la propriété n'est pas satisfaite, le feu2 ne peut pas être à orange quand le feu1 est à vert
5. Vérifier s'il est probable que le feu1 soit à vert et le feu2 à rouge :
   E<>F1.Vert and F2.Rouge
   la propriété satisfaite est le feu2 peut être à rouge quand le feu1 est à vert
6. Vérifier s'il est probable que le feu1 soit à orange et le feu2 à vert :
   E<>F1.Orange and F2.Vert
   la propriété n'est pas satisfaite, le feu1 ne peut pas être à orange quand le feu2 est à vert
7. Vérifier s'il est probable que le feu1 soit à rouge et le feu2 à vert :
   E<>F1.Rouge and F2.Vert
   la propriété est satisfaite, le feu1 peut être à rouge quand le feu2 est à vert
8. Vérifier s'il est probable que le feu1 soit à orange et le feu2 à rouge :
   E<>F1.Orange and F2.Rouge
   la propriété est satisfaite, le feu1 peut être à orange quand le feu2 est à rouge

9. Vérifier s'il est probable que le feu1 soit à rouge et le feu2 à orange :
   E<>F1.Rouge and F2.Orange
   la propriété est satisfaite, le feu1 peut être à rouge quand le feu2 est à orange
10. Vérifier s'il est vrai que lorsque le feu1 est rouge un état suivant possible est que le feu1 n'est pas rouge
    F1.Rouge –> not F1.Rouge
    la propriété est satisfaite, il existe un état où le feu1 n'est pas rouge après que le feu1 soit rouge
11. Vérifier s'il est vrai que lorsque le feu2 est rouge un état suivant possible est que le feu2 n'est pas rouge
    F2.Rouge –> not F2.Rouge
    la propriété est satisfaite, il existe un état où le feu2 n'est pas rouge après que le feu2 soit rouge
12. Vérifier s'il est vrai que lorsque le feu1 est vert un état suivant possible est que le feu1 soit vert
    F1.Vert–> F1.Vert
    la propriété est satisfaite, il existe un état où le feu1 est vert après que le feu1 soit vert
13. Vérifier s'il est vrai que lorsque le feu2 est vert un état suivant possible est que le feu2 soit vert
    F2.Vert–> F2.Vert
    la propriété est satisfaite, il existe un état où le feu2 est vert après que le feu2 soit vert
14. Vérifier qu'il n'y a pas de *deadlock* dans ce sytème
    A[] not deadlock
    la propriété est satisfaite, il n'y a pas de *deadlock*

3.2 Question 8

Dans cette question nous étudions les feux de chantier avec synchronisation avec un controleur. Les deux feux sont reliés à un controleur. Ce dernier envoie les signaux de passage d'un état à un autre aux deux feux(Feu1 et Feu2). Nous réalisons la modélisation de ces feux avec le logiciel **U**PPAAL. Les deux patrons des feux ont pratiquement la même implémentation. La seule différence est l'état initial.
Avant de réaliser les deux feux et le controleur, il faut créer l'orloge et les signaux dans le fichier *Déclaration*. On crée l'horloge t de la manière suivante : clock t ; On crée les signaux de la manière suivante : chan VtO1, VtO2, OtR1, OtR2, RtV1, RtV2 ;

1. VtO1 : signal pour le Feu1 état Vert vers l'état Orange
2. VtO2 : signal pour le Feu2 état Vert vers l'état Orange
3. OtR1 : signal pour le Feu1 état Orange vers l'état Rouge
4. OtR2 : signal pour le Feu2 état Orange vers l'état Rouge
5. RtV1 : signal pour le Feu1 état Rouge vers l'état Vert
6. RtV2 : signal pour le Feu2 état Rouge vers l'état Vert

Le premier feu est représenté de la manière suivante :
Pour recevoir un signal on ulise le caractère suivant : *?* - exemple : VtO1 ? c'est la reception reception du signal VtO1.

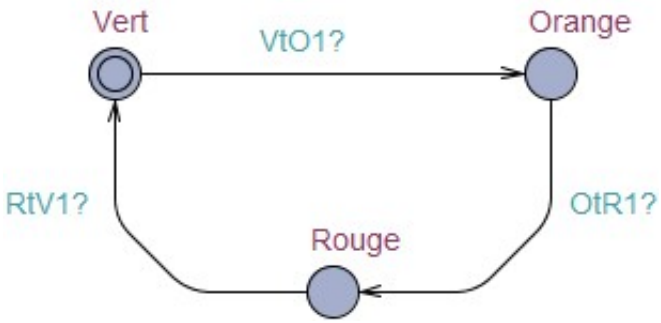


FIGURE 7 – Modélisation du Feu1

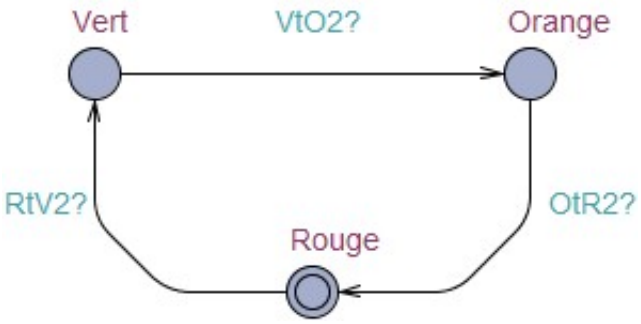


FIGURE 8 – Modélisation du Feu2

Le second feu est représenté de la manière suivante :
Pour recevoir un signal on ulise le caractère suivant : *?* - exemple : OtR2 ? c'est la reception reception du signal OtR2.

Le controleur permet l'envoi des différents signaux. De plus, le controleur est le seul à utiliser l'horloge déclarée au début. La représentation du controleur est la suivante :

Pour envoyer un signal on ulise le caractère suivant : *!* - exemple : RtV2 ! c'est la reception

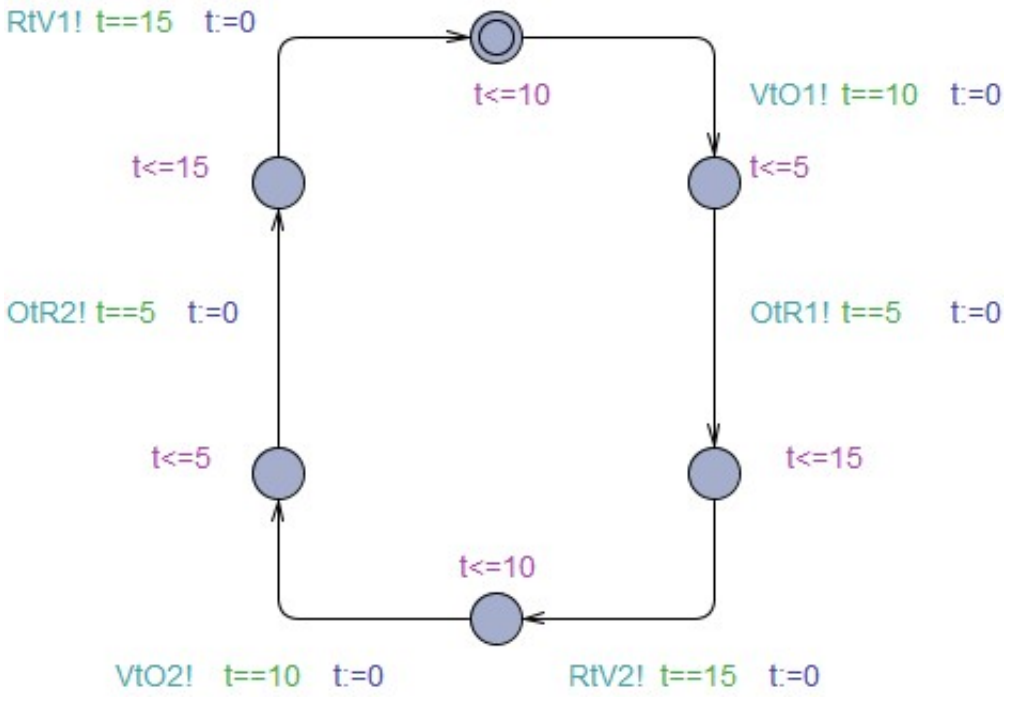


FIGURE 9 – Modélisation du controleur

reception du signal RtV2.
Suite à la réalisation de ces deux feux, ils faut réaliser des vérifications. Ces vérifications sont les mêmes que pour la questions suivante, c'est pour ça que nous allons voir un première écriture possible de ces vérification pour cette question :

1. Vérifier que si le feu1 est vert alors le feu2 ne peut jamais être vert ou orange :
   A[] F1.Vert imply not (F2.Vert || F2.Orange)
   la propriété est satisfaite, quand le feu1 est vert, le feu2 ne peut pas être ni vert ni orange, il sera obligatoirement rouge
2. Vérifier que si le feu1 est vert alors le feu2 ne peut jamais être vert ou orange :
   A[] F1.Vert imply not (F2.Vert || F2.Orange)
   la propriété est satisfaite, quand le feu1 est vert, le feu2 ne peut pas être ni vert ni orange, il sera obligatoirement rouge
3. Vérifier que si le feu1 est rouge alors il est possible que le feu2 soit vert ou orange :
   A<> F1.Rouge imply (F2.Vert or F2.Orange)
   la propriété est satisfaite, quand le feu1 est rouge il est possible que le feu2 soit vert ou orange
4. Vérifier que si le feu2 est rouge alors il est possible que le feu1 soit vert ou orange :
   A<> F2.Rouge imply (F1.Vert or F1.Orange)
   la propriété est satisfaite, quand le feu1 est rouge il est possible que le feu2 soit vert ou orange
5. Vérifier s'il est vrai que lorsque le feu1 est rouge un état suivant possible est que le feu1 n'est pas rouge
   F1.Rouge –> not F1.Rouge
   la propriété est satisfaite, il existe un état où le feu1 n'est pas rouge après que le feu1 soit rouge
6. Vérifier s'il est vrai que lorsque le feu2 est rouge un état suivant possible est que le feu2 n'est pas rouge
   F2.Rouge –> not F2.Rouge
   la propriété est satisfaite, il existe un état où le feu2 n'est pas rouge après que le feu2 soit rouge
7. Vérifier s'il est vrai que lorsque le feu1 est vert un état suivant possible est que le feu1 soit vert
   F1.Vert–> F1.Vert
   la propriété est satisfaite, il existe un état où le feu1 est vert après que le feu1 soit vert
8. Vérifier s'il est vrai que lorsque le feu2 est vert un état suivant possible est que le feu2 soit vert
   F2.Vert–> F2.Vert
   la propriété est satisfaite, il existe un état où le feu2 est vert après que le feu2 soit vert
9. Vérifier qu'il n'y a pas de *deadlock* dans ce sytème
   A[] not deadlock
   la propriété est satisfaite, il n'y a pas de *deadlock*

4 Modélisation d'un carrefour en T

On considère la spécification informelle suivante, tirée d'une application réelle :
Un système de contrôle doit assurer la sécurité et le bon fonctionnement d'un ensemble de feux tricolores assigné à une jonction en T (Un feu principal plus un feu secondaire). De base, Le feu principal sera au vert sur la route principale et le feu secondaire sera au rouge sur la route secondaire jusqu'a ce qu'une voiture soit détectée sur cette route. Dans ce cas, les deux feux inverseront leurs états pour permettre aux voitures sur la route secondaire de pouvoir circuler. Après un intervalle de temps défini, ces deux feux ré-inverseront leurs états pour permettre la circulation sur la route principale et ainsi revenir à l'état initial. Un schéma modélise cette application :

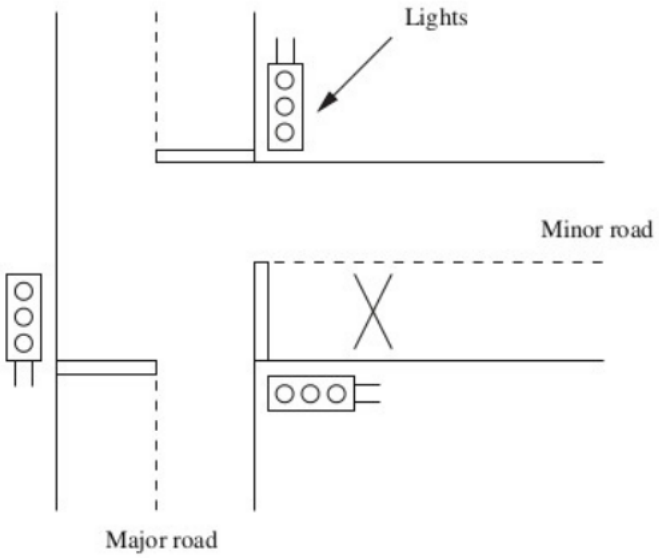


FIGURE 10 – Jonction en T

4.1 Question 9

Pour cette question, nous ignorerons les contraintes de temps. Il nous est demandé de modéliser ce système comme un ensemble d'automates synchronisés. Pour ce faire, nous avons utilisé le logiciel UPPAAL. Premièrement, nous avons commencé par modéliser le capteur de voiture sur le feu secondaire : Ce capteur envoie un simplement un signal quand il détecte une

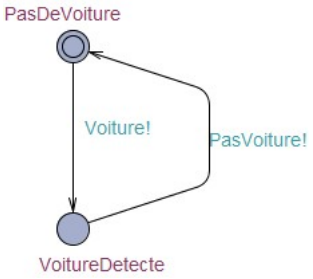


FIGURE 11 – Modélisation du capteur

voiture sur le feu secondaire, et en envoie un autre quand le traffic est dégagé (pas de voiture).

Après avoir modélisé le capteur, nous nous sommes occupés du Petit Feu :

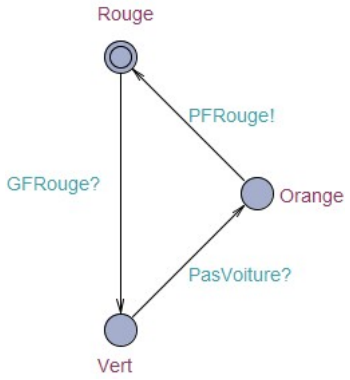


FIGURE 12 – Modélisation du petit feu

On peut voir que le feu a trois états :
- Rouge
- Orange
- Vert

Quand le feu secondaire est au rouge, il attend un signal du grand feu pour l'avertir et pour pouvoir passer au vert. Une fois ce signal reçu, il passe au vert jusqu'à ce qu'il reçoive le signal "plus de voiture" du capteur. Une fois reçu, il passe au orange, envoie un signal au Grand Feu pour l'avertir, et passe au rouge.
Pour finir cette modélisation, nous avons crée le feu principal :

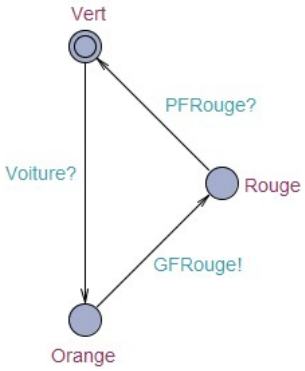


FIGURE 13 – Modélisation du grand feu

Ce feu a les mêmes états que le feu secondaire. A l'état initial, au vert, il interroge le capteur pour savoir si une voiture est détectée. Si cette condition est vraie, il passe au orange. Dans cet etat, il envoie un signal au petit feu pour qu'il puisse commencer son cycle. Une fois passé au rouge, il attend que le petit feu envoie un signal lui disant qu'il est lui-même au rouge pour pouvoir finir son cycle et repartir au vert (principe de sécurité : on attends qu'un feu soit rouge pour que l'autre soit au vert).
En fonctionnant ensemble, ces trois modèles permettent de traiter le sujet de la jonction en T.

4.2 Question 10

La question traite du même sujet que la question 9, mais nous allons incorporer des contraintes de temps. Ces contraintes sont :

1. L'état vert du feu secondaire dure 30 secondes.
2. L'état orange des deux feux dure 5 secondes.
3. La transition entre deux états d'un feu (ex : entre vert et orange) dure une seconde.
4. Dans chaque cycle, l'état vert du feu principal dure au moins 30 secondes.

Pour ce faire, nous avons rajouté ces contraintes dans les modèles. Le capteur n'a pas changé, car il n'est pas affecté par les nouvelles contraintes (son fonctionnement est indépendant de celui des feux, mais pas l'inverse !).
Le premier modèle que nous avons changé a été le petit feu : Du fait des contraintes, nous

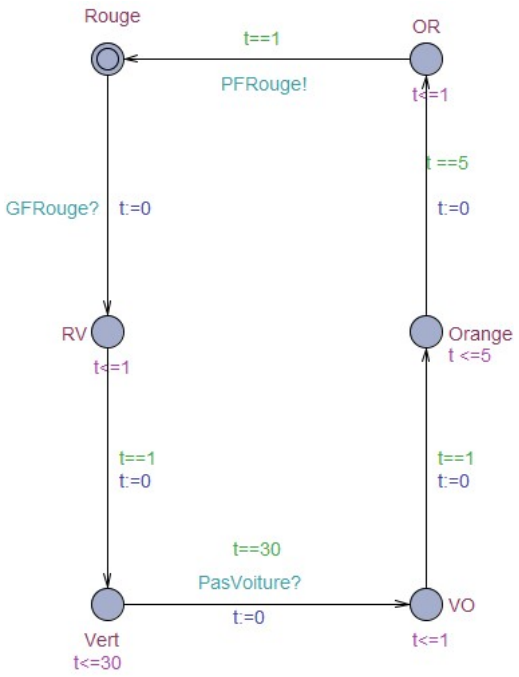


FIGURE 14 – Modélisation du petit feu avec contraintes de temps

sommes passés de 3 à 6 états :

— Rouge
— Rouge au Vert (RV)
— Vert
— Vert au Orange (VO)
— Orange
— Orange au Rouge (OR)

On peut voir que le timer est réinitialisé à tous les états (pour simplifier l'écriture des contraintes). On remarque que les états intermédiaires ont une garde d'une seconde, pour respecter la condition 3. On peut aussi voir que l'état vert a une garde de 30 secondes qui permet de valider la condition 1. Enfin, on remarque que l'état orange a une garde de 5 secondes qui permet de vérifier la condition 2.
Ensuite, nous avons changé le modèle grand feu :
Pour ce feu , nous avons ajouté les mêmes contraintes de temps que sur le secondaire, pour respecter les contraintes qui nous ont étés imposés.

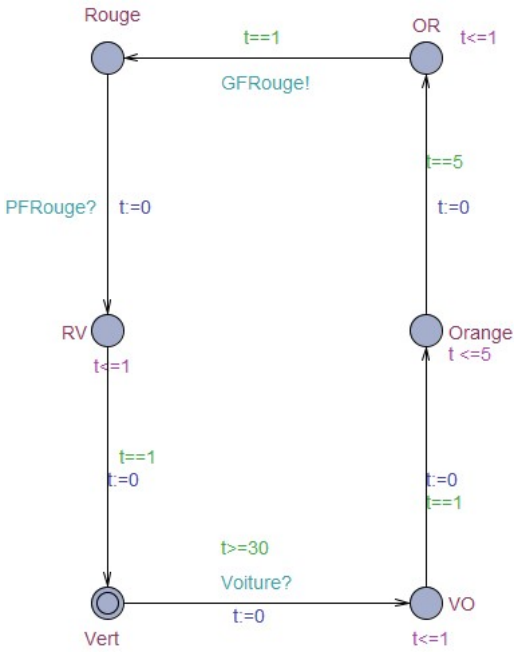


FIGURE 15 – Modélisation du grand feu avec contraintes de temps

4.3 Question 11

Cette question finale sur la jonction en T concerne les principes de surêté et de vivacité émanant des deux feux.Pour ce faire, nous avons imaginé des hypothèses qui casserait le cycle de fonctionnement, et nous avons testé notre modèle à l'aide du vérifieur pour voir si il était protégé de ces cas spéciaux mais défaillants. Nous avons donc testé comme contraintes :

1. Le modèle a t-il des *deadlocks* ?
   E <> deadlock
2. Les deux feux peuvent-ils être au rouge ?
   E<> ProcessPF.Rouge and ProcessGF.Rouge
3. Peut-on avoir le petit feu au orange et le grand feu au vert ?
   E<> ProcessPF.Orange and ProcessGF.Vert
4. Peut-on avoir le petit feu au vert et le grand feu au orange ?
   E<> ProcessPF.Vert and ProcessGF.Orange
5. Peut-on avoir les deux feux au vert ?
   E<> ProcessPF.Vert and ProcessGF.Vert
6. La transition du petit feu entre orange et rouge dure t-il plus d'une seconde ?
   E<> ProcessPF.OR and ProcessPF.t>1
7. Le grand feu passe t-il du rouge au vert ?
   ProcessGF.Rouge –> ProcessGF.Vert
8. Est-ce que le petit feu passe au vert quand une voiture est détectée ?
   Process3.VoitureDetecte –> ProcessPF.Vert
9. Le petit feu orange dure t-il plus de 5 secondes ?
   E<> ProcessPF.Orange and ProcessPF.t>5
10. le grand feu orange dure t-il plus de 5 secondes ?
    E<> ProcessGF.Orange and ProcessGF.t>5

11. La transition du grand feu entre rouge et vert dure t-il plus d'une seconde ?
    E<> ProcessGF.RV and ProcessGF.t>1
12. La transition du grand feu entre orange et rouge dure t-il plus d'une seconde ?
    E<> ProcessGF.OR and ProcessGF.t>1
13. La transition du grand feu entre vert et orange dure t-il plus d'une seconde ?
    E<> ProcessGF.VO and ProcessGF.t>1
14. La transition du petit feu entre rouge et vert dure t-il plus d'une seconde ?
    E<> ProcessPF.RV and ProcessPF.t>1
15. La transition du petit feu entre vert et orange dure t-il plus d'une seconde ?
    E<> ProcessPF.VO and ProcessPF.t>1
16. Le petit feu vert dure t-il plus de 30 secondes ?
    E<> ProcessPF.Vert and ProcessPF.t>30
17. Le grand feu vert dure t-il au moins 30 secondes ?
    E<> ProcessGF.Vert and ProcessGF.t>30

Toutes ces propriétés ont étés ajoutés au vérifieur et ont étés checkés. Notre modélisation d'une jonction en T réponds donc parfaitement aux conditions requises et aux propriétés de vivacité et de sureté.

5 Implémentation

L'implémentation de notre projet est disponible dans le dossier AuFeux. Il s'agit d'un projet Eclipse classique (dans Eclipse, File » Import) qui implémente la question 11. Ce projet se décompose en plusieurs packages qui seront détaillés dans les parties suivantes.

Pour le lancer, la classe main se trouve dans AuFeux/SystemConsoleUI. De plus, une Javadoc peut être générée pour mieux comprendre le code.

5.1 Package AuFeux

5.1.1 Classe Systeme

Cette classe abstraite représente notre croisement. Elle possède donc un attribut *GrandFeu* (p. 18), *PetitFeu* (p. 18) et un *Capteur* (p. 17). Une méthode *run* permet de lancer la simulation. Cette classe implémente l'interface *MyObserver* pour être notifié à chaque changement des feux ou du capteur. L'implémentation de la méthode *update* est laissé à ses classes filles. Enfin, un attribut d'instance nommé `GLOBAL_TIME_UNIT` permet de spécifier l'unité de temps utilisé dans le système (0.1s par défaut).

5.1.2 Classe SystemeConsoleUI

La classe *SystemeConsoleUI* hérite de *Système* et fournit une interface graphique en console. Elle ne fait qu'implémenté la méthode *update*. Un variable booléenne permet de choisir d'afficher ou non les notifications de changement de valeur d'horloge.

La méthode *main* présente dans cette classe permet de choisir l'unité de temps utilisée et l'affichage des horloges puis de lancer la simulation en console.

5.2 Package Annexe

Ce package contient divers interfaces, classes abstraites ou classes utilisées dans divers endroits du code.

5.2.1 Interface MyObserver/MyObservable

Ces deux interfaces implémentent le pattern Observer. Nous n'avons pas utitlisé les classes de java.util car Observable est une classe et non une interface et que Java ne permet pas l'héritage multiple.

5.2.2 Classe Clock

Cette classa modélise une horloge. Elle hérite de *Thread* et implémente *MyObservable*. Il s'agit d'un compteur incrémenté toute les unités de temps et pouvant être réinitialisé.

5.2.3 Classe Feu

Cette classe abstraite modélise un feu de circulation. Elle hérite de la classe Thread et implémente MyObserver (pour observer son horloge) et MyObervable (pour être observé par le système). Un pattern State a été réalisé pour l'état du feu (Vert, Orange...). De plus, une méthode existe pour chaque signal (*GFRouge*, *voiture* ...). L'état courant est utilisé our traiter ces différents signaux.

5.2.4 Classe FeuState

Cette classe abstraite permet de modéliser l'état d'un feu. Elle possède une méthode pour chaque signal qui retourne vrai si l'état a pu traiter. La méthode *reachState* est appellée lorsqu'un feu arrive dans cet état.

5.2.5 Les états utilisés par les deux feux

Certains états sont les mêmes qu'il s'agisse d'un grand feu ou d'un petit feu. Ces états ont étés placés dans le package Annexe. Il s'agit des états suivants :

— VertOrangeState : Etat intermédiaire entre Vert et Orange
— OrangeRougeState : Etat intermédiaire entre Orange et Rouge
— RougeVertState : Etat intermédiaire entre Rouge et Vert
— OrangeState : Feu Vert

Les états intermédiaires sont les mêmes dans les deux feux (attente d'une unité de temps puis passage à l'état suivant). Le feu orange est aussi le même pour les deux feux (attente de 5 unités de temps puis passage à l'état rouge).

5.3 Package Capteur

5.3.1 Classe Capteur

La classe *Capteur* modélise le capteur de véhicule. Elle peut posséder deux états distincts :

— PasVoitureState : aucune voiture détectée. A l'arrivée dans l'état, appelle la méthode *pasVoiture* pour les deux feux. Attend 200 unités de temps pour faire repasser le capteur à l'état *VoitureDetecteState*.

— VoitureDetecteState : voitre détectée. A l'arrivée dans l'état, appelle la méthode *voiture* pour les deux feux puis passe le capteur à l'état *PasVoitureState*.

Lorsque l'état courant change, les observeurs en sont notifiés.

5.4 Package GrandFeu

5.4.1 Classe GrandFeu

Cette classe implémente la classe *Feu* pour modéliser un feu de ciculation sur la grande route. Seuls les états Rouge et Vert sont spécifiques à ce feu. Voici leurs spécificités :

— VertGFState : A l'arrivée dans l'état, l'horloge est réinitialisée. Lorsque sa méthode *voiture* est appellée, il attend que l'horloge ait une valeur supérieur à 30 puis réinitialise l'horloge et passe à l'état vert-orange.

— RougeGFState : A l'arrivée dans l'état, appelle la méthode *GFRouge* du petit feu. Lorsque sa méthode *PFRouge* est appellée, le grand feu reinitialise son horloge puis passe à l'état rouge-vert.

5.5 Package PetitFeu

5.5.1 Classe PetitFeu

Cette classe implémente la classe *Feu* pour modéliser un feu sur la petite route. Comme pour le grand feu, seuls les états Rouge et Vert sont spécifiques à ce feu.

Voici leurs spécificités :

— VertPFState : Lorsque sa méthode *PasVoiture* est appellée, il attend que l'horloge ait une valeur supérieur à 30 puis réinitialise l'horloge et passe à l'état vert-orange.

— RougePFState : A l'arrivée dans l'état, appelle la méthode *PFRouge* du grand feu. Lorsque sa méthode *PFRouge* est appellée, le petit feu reinitialise son horloge puis passe à l'état rouge-vert.

6 Conclusion et suite du travail

Ce projet a été pour nous un premier pas dans le monde de l'ingénierie logicielle formelle. Nous avons consolidé nos acquis théoriques, et nous avons pu utiliser de nouveaux logiciels spécialisés tels que UPPAAL ou ROMEO. Chacun du groupe a pu participer à une partie, donc ce projet a été mené à bien grâce à toute l'équipe. Cependant quand une partie était finie la personne qui travaillait dessus pouvait aider un des autres membres du groupes. Ceci nous a permis de pouvoir travailler sur différentes parties du projet. En complément du travail effectué, nous aurions pu ajouter une GUI un peu plus complexe à l'implémentation. Mais cela ne nous a pas empêcher d'apprécier ce sujet. En effet, le sujet proposé était concret et donc nous pouvions nous imaginer facilement comment devaient fonctionner les feux et quels étaient les différents cas d'utilisation d'un tel système.