



UNIVERSITÉ LIBRE DE BRUXELLES

INFO-F410 - EMBEDDED SYSTEMS DESIGN

---

## Piste de décollage de voitures volantes autonomes

---

*Auteurs*

Mourad Akandouch  
Mounir Hafif  
Daoud Yemni (*partie formal*)  
Maurin Verrijdt (*partie formal*)

*Professeurs*

Gilles Geeraerts  
Nicolas Mazzocchi

21 juin 2018

## Avant-propos

Notre projet ayant été conjointement réalisé avec le cours de Formal Verification et celui-ci, certains passages de nos rapports seront donc similaires dans le but de produire des documents indépendants l'un de l'autre. Également, globalement, ce qui est abordé n'est pas pareil.

## Table des matières

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>La modélisation</b>	<b>4</b>
2.1	L'environnement . . . . .	4
2.1.1	Au sol . . . . .	5
2.1.2	Dans les airs . . . . .	5
2.2	Les acteurs . . . . .	6
2.3	Le contrôleur . . . . .	7
<b>3</b>	<b>Environnement de travail</b>	<b>8</b>
3.1	Automates temporisés . . . . .	8
3.2	Outil de modélisation UPPAAL . . . . .	8
3.3	Jeux temporisés . . . . .	8
<b>4</b>	<b>Modélisation et vérification</b>	<b>9</b>
4.1	Première version . . . . .	9
4.1.1	Modélisation . . . . .	9
4.1.2	Propriétés à vérifier . . . . .	16
4.1.3	Raisons de l'itération suivante . . . . .	16
4.2	Deuxième version . . . . .	17
4.2.1	Modélisation . . . . .	17
4.2.2	Propriétés à vérifier . . . . .	21
4.2.3	Raisons de l'itération suivante . . . . .	22
4.3	Troisième version . . . . .	23
4.3.1	Modélisation . . . . .	23
4.3.2	Propriétés vérifiées . . . . .	23
4.3.3	Raisons de l'itération suivante . . . . .	24
4.4	Quatrième version . . . . .	25
4.4.1	Modélisation . . . . .	25
4.4.2	Propriétés vérifiées . . . . .	26
<b>5</b>	<b>Implémentation</b>	<b>26</b>
<b>6</b>	<b>Synthèse</b>	<b>28</b>
<b>7</b>	<b>Comparaison : synthèse VS contrôleur itératif</b>	<b>32</b>
<b>8</b>	<b>Conclusion</b>	<b>33</b>

## 1 Introduction

Nous sommes en l'an 2050. Elon Musk a réussi à produire des voitures autonomes et volantes. La civilisation a évolué et désormais, nous avons des autoroutes aériennes. Personne n'a le droit de conduire dans ces autoroutes selon le code de la route aérien. Elles sont exclusivement réservées aux voitures autonomes. Une telle loi permet de réduire drastiquement le taux d'accident qui peut survenir puisque nous retirons le facteur humain qui, de nature, est sujet à commettre des erreurs.

L'entreprise SkyWalkers<sup>TM</sup>, une filiale de Tesla, gère le bon fonctionnement des autoroutes aériennes en Belgique. Notre rôle sera de modéliser, vérifier et implémenter un sous-ensemble de ce système que représente cette autoroute aérienne.

Également, pour le cours de Formal Verification, nous nous sommes arrêté à la modélisation et vérification de notre projet. Ici, en plus d'avoir un modèle vérifié, nous synthétisons également un nouveau contrôleur sur base d'une stratégie gagnante garantissant une propriété sûreté qui sera discuté ultérieurement dans ce rapport. De plus, nous comparons l'efficacité de notre contrôleur vérifié et celui synthétisé en les soumettant à des tests. Enfin, nous implementons notre modèle vérifié en langage JavaScript afin de visualiser au mieux le système et voir qu'il fonctionne effectivement.

## 2 La modélisation

Notre modélisation s'articulera autour du décollage de voitures afin qu'elles puissent s'insérer convenablement dans l'autoroute. Voici une vision schématique de l'autoroute aérienne et des voitures qui désirent s'y insérer :

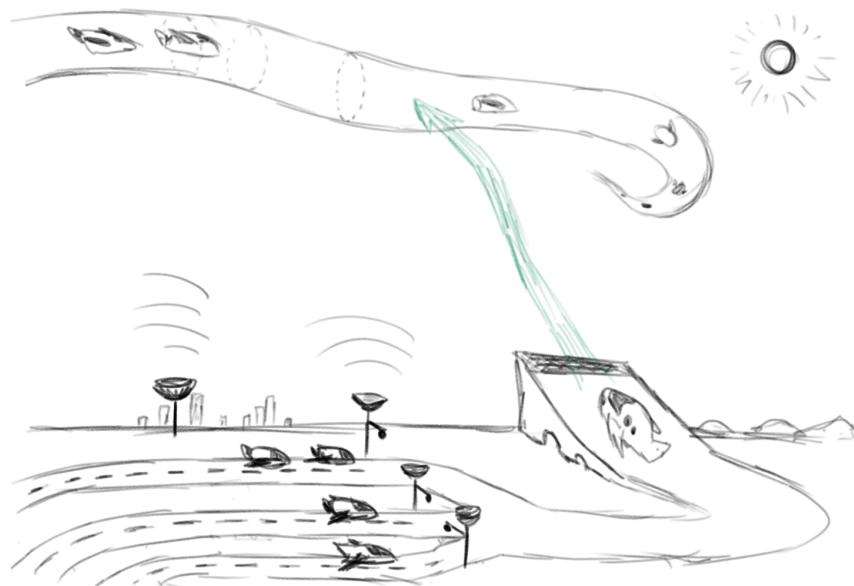


FIGURE 1 – Système de décollage de voitures volantes leur permettant de s'insérer dans une autoroute aérienne.

Sur l'image ci-dessus, nous pouvons constater plusieurs capteurs et autres composants. Premièrement, nous constatons, au sol, qu'il y a trois routes menant à la piste de décollage. Chacune de ces routes possède un composant ressemblant à une parabole munie d'une fine barre horizontale au bout de laquelle est placé un composant électronique (les détails de leur utilités seront donnés dans la section suivante). Aussi, chacune de ces trois routes mènent à une unique piste sur laquelle une voiture ira décoller afin de s'insérer dans l'autoroute aérienne comme dessiné en haut de l'image. Notons que l'espèce de cylindre n'est là que pour se représenter figurativement l'autoroute ; il n'y a que de l'air.

### 2.1 L'environnement

Notre modélisation de l'environnement peut se subdiviser en deux parties : l'environnement au sol et celui dans les airs.

### 2.1.1 Au sol

Concernant l'environnement au sol, il faut modéliser le flux de voitures arrivant dans chacune des trois routes menant à la piste de décollage. Nous avons décidé de modéliser trois routes où chacune ne peut contenir que certains types de voitures sur celles-ci. La première route ne contient que des voitures pouvant accélérer rapidement. La seconde route est réservée aux voitures ayant une accélération moyenne et la dernière est pour celles dont l'accélération n'est pas élevée. La raison pour laquelle nous séparons les types voitures selon leur accélération viendra dans la section suivante.

Donc, nous avons créé trois variables  $\in \{0, \dots, MAX\_NB\_CARS\}$  représentant chacune le nombre de voitures qui attendent devant le composant électronique dans leur route. Ce composant agit tel un feu rouge avec deux valeurs possibles : rouge ou vert. Évidemment, dans notre contexte, il ne s'agit pas réellement de feu de lumière rouge et verte car les voitures sont autonomes. Mais par soucis de simplicité, nous appellerons ces composants des feux rouges.

Nous pouvons immédiatement remarquer que notre modélisation au sol implique la création d'un mutex. En effet, nous avons trois types de voitures tentant d'accéder à une ressource unique : la piste de décollage. Nous devons donc faire en sorte que deux feux de signalisation ne permettent pas, en même temps, de laisser passer leur véhicule.

### 2.1.2 Dans les airs

Dans les airs, l'environnement est le flux de voitures qui voyage sur l'autoroute aérienne. Afin d'insérer une voiture sans encombre, il est de rigueur de ne pas permettre à une voiture de décoller si, en arrivant dans l'autoroute, celle-ci pourrait entrer en collision avec une voiture présente dans l'autoroute et qui arrive au même endroit que la voiture qui s'apprête à s'insérer.

Pour ce faire, nous avons introduit trois capteurs permettant de détecter la présence de véhicules à certains endroits précis de l'autoroute. Cela permet ainsi de savoir lorsqu'une voiture risque d'entrer en collision avec une voiture qui voudrait décoller. En effet, supposons qu'une voiture rapide mette  $x$  secondes pour décoller et s'insérer dans l'autoroute à un endroit précis. Il est nécessaire, dans le but de ne pas créer de collision, de vérifier qu'il n'y ait pas déjà une voiture dans l'autoroute aérienne

à une distance équivalente à  $x$  secondes de vol par rapport à l'endroit d'insertion. Ci-dessous, nous entourons en rouge les capteurs de présence de véhicules.

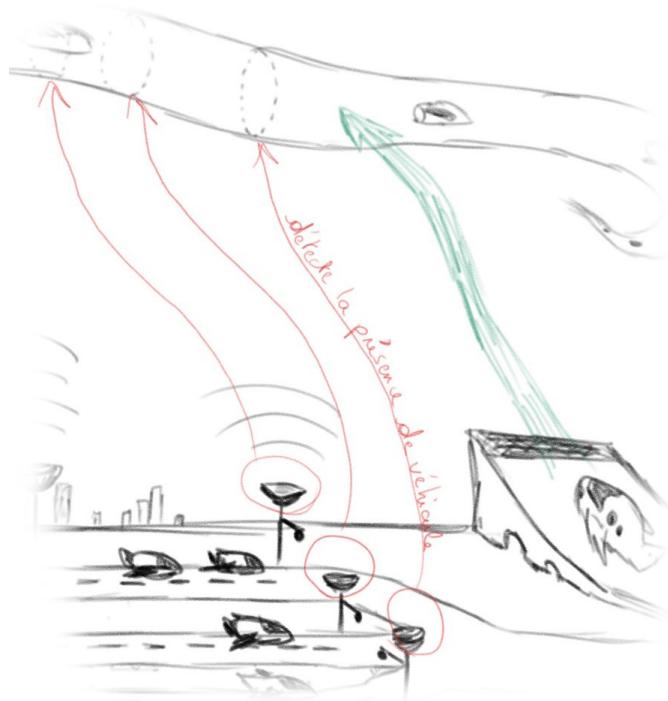


FIGURE 2 – En rouge, sont entourés les capteurs de présence de véhicule à un endroit précis de l'autoroute.

## 2.2 Les acteurs

Ce système autoroutier implique la création d'un système de feu de signalisation pour permettre aux voitures au sol de décoller sans encombre, c'est-à-dire sans entrer en collision avec une autre voiture que ce soit au sol ou dans les airs. Nous avons donc modélisé trois feux au sol : le premier permettant d'autoriser les voitures rapides à décoller, le second pour autoriser les voitures normales et le dernier pour les lentes. Nous apprendrons également plus loin dans ce rapport que ces feux de signalisation au sol ne sont pas les seuls que nous avons mis en place. Nous en reparlons de manière plus détaillée dans la section 4.1. Sur l'illustration ci-dessous, nous entourons en rouge les feux de signalisation terrestres afin de mieux visualiser le système.

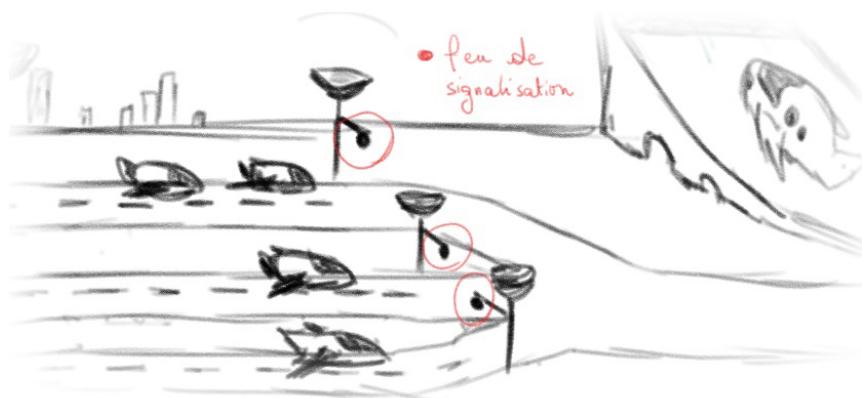


FIGURE 3 – En rouge, sont entourés les composants électroniques ayant le même comportement qu'un feu de signalisation mais pour voiture autonome.

### 2.3 Le contrôleur

Concernant le contrôleur, nous en avons produit deux. Le premier a été construit sur base de stratégies permettant de contrer l'environnement et le second a été synthétisé grâce à UPPAAL TiGa et l'avons également construit.

Concernant le premier contrôleur, au départ, nous avions nos acteurs et l'environnement qui s'exécutaient en parallèle mais rien n'allait. Avec UPPAAL, nous avons tout d'abord demander de vérifier qu'il n'y ait aucun deadlock dans le système. Ensuite, nous avons introduis différentes propriétés qui, pour qu'elles soient satisfaites, ont demandé la construction d'un contrôleur permettant de faire interagir nos acteurs avec l'environnement tout en vérifiant les différentes propriétés introduites. Procéder ainsi à mené à une construction du contrôleur de manière itérative.

Quant au second contrôleur, celui-ci est de prime abord bien moins complet car il a été synthétisé pour vérifier une propriété de sûreté mais au détriment de plusieurs autres fonctionnalités. Également, l'espace d'état généré par notre modélisation étant gargantuesque, nous avons été contraint de le réduire drastiquement afin de pouvoir synthétiser un contrôleur constructible à la main. Nous détaillons ceci plus loin dans la section consacrée à la synthèse.

## 3 Environnement de travail

Afin de modéliser notre système de décollage et d'insertion, nous avons opté pour l'utilisation d'automates temporisés. L'avantage avec de tels modèles est que nous pouvons ainsi facilement exprimer des notions liées à l'attente, l'urgence, etc.

### 3.1 Automates temporisés

Les automates temporisés ayant été vus en cours, nous donnons ici une brève explication de ce modèle sans toutefois faire l'affront formaliser le concept et entrer dans tous les détails, nos lecteurs étant très probablement des chercheurs dans le domaine. Ce sont donc des automates finis munis d'un ensemble d'horloges avançant toutes à la même cadence. Nous pouvons effectuer des opérations sur les horloges comme tester leur valeur ou les remettre à zéro. Nous avons également la notion de *guard* qui permet d'activer si une expression booléenne est satisfaite. Grâce à l'outil de modélisation utilisé dans notre projet, nous avons également droit à plusieurs autres fonctionnalités qui nous ont été extrêmement utiles.

### 3.2 Outil de modélisation UPPAAL

Concernant l'outil avec lequel nous avons modélisé notre système, nous avons choisi d'utiliser UPPAAL pour sa facilité d'utilisation et de vérification de propriétés. De plus, le fait que notre projet ait de fortes contraintes liées au temps nous a de fait enjoint à choisir UUPAAL.

### 3.3 Jeux temporisés

Le système peut également être vu comme un jeu temporisé (Timed game) où nous avons deux joueurs : l'environnement et le contrôleur. L'environnement étant contrôlé par le joueur adverse et son but étant de nous faire perdre à tout prix. Le contrôleur représente notre joueur. Son rôle étant de trouver une stratégie gagnante permettant de parer toutes les tentatives de l'adversaire de nous faire perdre. Afin de représenter au mieux un tel jeu, nous utilisons l'extension "TiGa" d'UPPAAL qui permet de représenter des jeux et de synthétiser des stratégies gagnantes. Elle permet également de créer des transitions non contrôlable ; celles-ci représentant les coups que peut jouer le joueur adverse.

## 4 Modélisation et vérification

Notre modélisation s'est déroulée en plusieurs phases. Quatre versions qui représentent l'évolution du projet et nous les décrirons en détails dans les sections qui suivent. Également, une cinquième version assez a été implémentée où cette fois-ci, le contrôleur a été synthétisé par UPPAAL TiGa.

### 4.1 Première version

Dans cette section, nous expliquons comment la première version a été mise en œuvre, les problèmes rencontrés, les hypothèses prises et les raisons de notre passage vers la seconde version.

#### 4.1.1 Modélisation

Pour la première version, nous avons modélisé le projet comme imaginé initialement. Nous avons tout d'abord créé trois capteurs qui détectent lorsqu'il y a du trafic sur l'autoroute aérienne et notre contrôleur agissait en fonction de l'état de ces capteurs afin de permettre aux voitures au sol en attente de pouvoir décoller. Sur la figure 4 ci-dessous, nous pouvons voir comment est modélisé le premier capteur.

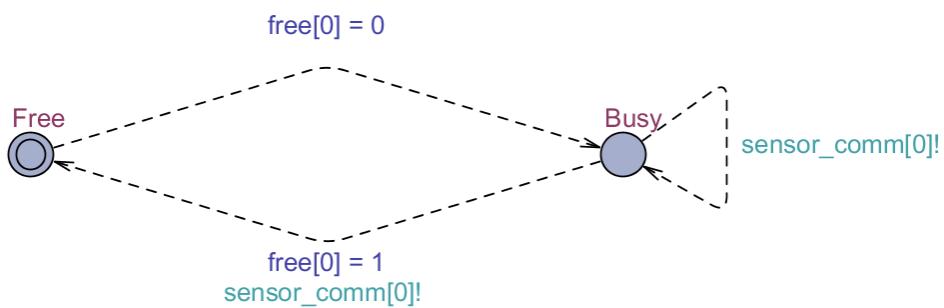


FIGURE 4 – Premier capteur sur autoroute aérienne.

L'état initial est celui de gauche, il représente le fait qu'aucune voiture n'est actuellement présente dans le tronçon d'autoroute correspondant. Comme le trafic

n'est pas déterministe, ces trois capteurs sont donc contrôlés par le joueur adverse : l'environnement. La transition non-contrôlée vers l'état nommé *Busy* représente le fait qu'une voiture est présente dans le tronçon correspondant. Aussi, nous pouvons remarquer que sur la transition *Free* → *Busy*, nous mettons à 0 le premier indice d'un tableau de booléen nommé *free*. Le mettre à zéro signifie donc que le premier capteur n'est pas libre. Ce tableau représente l'état des trois capteurs. Ainsi, à chaque capteur est attribué un indice dans ce tableau qui représente son état actuel. Le contrôleur consulte les valeurs de ce tableau.

Concernant la transition *Busy* → *Free*, nous voyons que nous mettons à 1 la valeur du premier indice du tableau *free*. Cela signifie, évidemment, que le capteur ne détecte pas de voiture et donc que la voie est libre pour les véhicules au sol liés à ce tronçon. Nous remarquons également dans cette même transition que nous envoyons un message via le canal *sensor\_comm[0]*. Ici, il est de rigueur de mentionner que notre modélisation dans cette version s'est articulée autour d'un trafic unidirectionnel comme illustré ci-dessous. C'est-à-dire que lorsque le premier capteur détecte un véhicule, le second capteur **doit** détecter ce véhicule plus tard. L'envoi du message à travers le canal *sensor\_comm[0]* permet justement de simuler ce comportement car le deuxième capteur ne peut prendre sa transition *Free* → *Busy* que si celui-ci reçoit un message à travers le canal susmentionné.

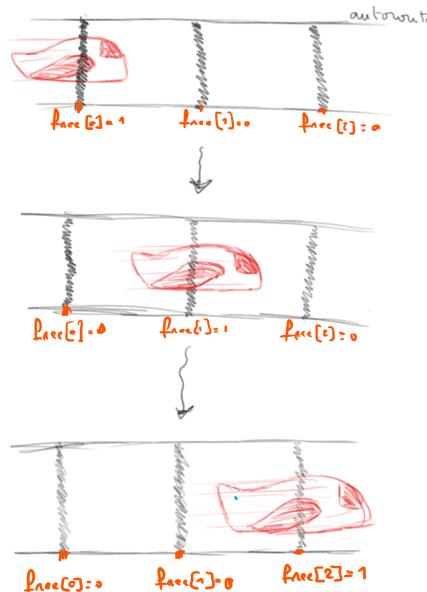
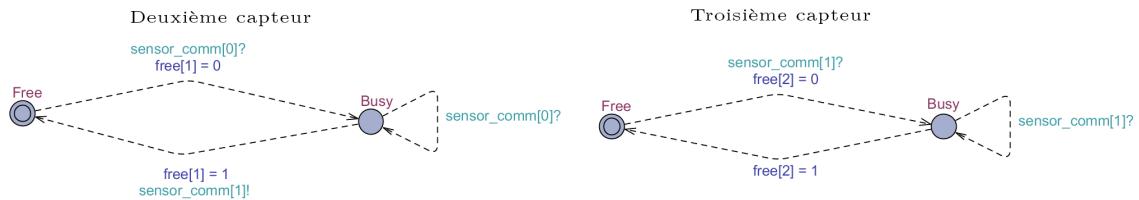


FIGURE 5 – Évolution d'un trafic aérien.

Enfin, le premier capteur possède également une transition  $Busy \rightarrow Busy$  permettant de simuler le fait que plusieurs voitures se suivent l'une à la suite de l'autre. Cette transition envoie également un message à travers le même canal que précédemment pour faire évoluer le trafic aérien. Une conséquence d'un tel comportement est qu'il existe des exécutions où le joueur adverse fasse en sorte que l'autoroute aérienne soit éternellement dense et qu'aucune voiture ne puisse décoller. Sachant tout cela, voici comment ont été modélisés les deux autres capteurs sous UPPAAL :



Nous remarquons que les transitions  $Free \rightarrow Busy$  ne sont prises que si ces capteurs ont effectivement reçus un message du précédent capteur leur indiquant que c'est à leur tour de s'allumer. Notons ici que dès qu'un capteur envoie un message, le suivant passe à  $Busy$ , cela sous-entendrait que les voitures voyagent de manière "instantané" d'un tronçon à un autre. Cela fait parti de nos hypothèses. Nous avons décidé de ne pas créer de délai entre la fin de l'état  $Busy$  du capteur  $i$  et le début dudit état dans le capteur  $i + 1$ . La raison en est que la modélisation serait devenue beaucoup trop difficile à mettre en œuvre car nous aurions dû prendre en charge des voitures pouvant arriver à n'importe quel moment et donc avoir une mémoire mémorisant l'instant de chaque voiture détectée par le premier capteur. Également, l'espace des états à vérifier exploserait. Nous avons donc décidé de nous abstraire d'une telle contrainte.

## Feux de signalisations au sol

Passons désormais au trafic terrestre. Au sol, nous avons créé trois "feux de signalisations" permettant d'autoriser ou pas aux voitures de démarrer. Voici ci-dessous notre modélisation d'un feu de signalisation. Les trois étant exactement les mêmes modulo une variable, nous ne mettrons ci-dessous que le premier.

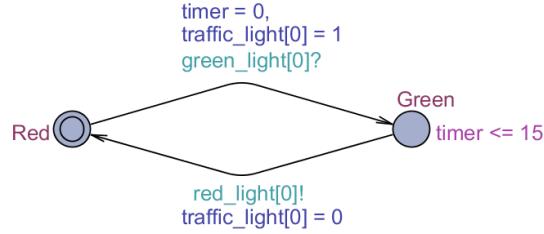


FIGURE 6 – Modélisation du premier feu de signalisation au sol. Les deux autres sont exactement les mêmes exceptés les indices utilisés dans les transition.

Considérons la transition  $Red \rightarrow Green$ . Elle modélise le fait que le feu ”passe au vert”.<sup>1</sup> Nous voyons que la transition met une horloge nommée  $timer$  à 0. Cette variable fait en sorte qu’on ne puisse pas durer trop longtemps dans l’état  $Green$  grâce à l’invariant que nous y avons introduis. Aussi, nous voyons que nous mettons à 1 le booléen au premier indice du tableau  $traffic\_light$ . Ce tableau indique quel feu est au vert et lequel ne l’est pas. Le  $i$ ème feu correspond à l’indice  $i$  du tableau. Enfin, nous constatons que cette transition n’est prise que si nous recevons un message à travers le canal  $green\_light[0]$ . Notre contrôleur se charge de le faire au moment opportun.

Concernant la transition  $Green \rightarrow Red$ , elle est prise endéans les 15 unité de temps de l’horloge  $timer$ . Lorsque le feu prend cette transition, elle remet à 0 le booléen adéquat tout en notifiant le contrôleur qu’il est revenu au rouge. Le choix d’avoir pris 15 unités de temps est tout à fait arbitraire. Nous n’avions pas testé avec d’autres valeurs au moment de la modélisation de cette version. Notons également que cette transition peut être prise immédiatement, donc qu’il existe des exécutions où la transition est prise alors que trop peu de temps s’est écoulé dans le vert. Nous remédions à ce problème dans la quatrième version.

### Modélisation du trafic au sol

En plus d’avoir un environnement aérien, nous en avons également un au sol. Le joueur adverse peut donc manipuler le trafic aérien et terrestre. Nous avons modélisé le fait que des voitures arrivent sur la piste adéquate et s’arrêtent en attente d’une

---

1. Rappelons que dans notre contexte, nous parlons de feu que par analogie et simplicité. En réalité, il s’agit d’un appareil électronique émettant une fréquence indiquant à la voiture sur la bande qu’elle a le droit d’avancer.

autorisation par le feu correspondant à la piste. Ci-dessous, nous montrons le modèle représentant cet environnement.

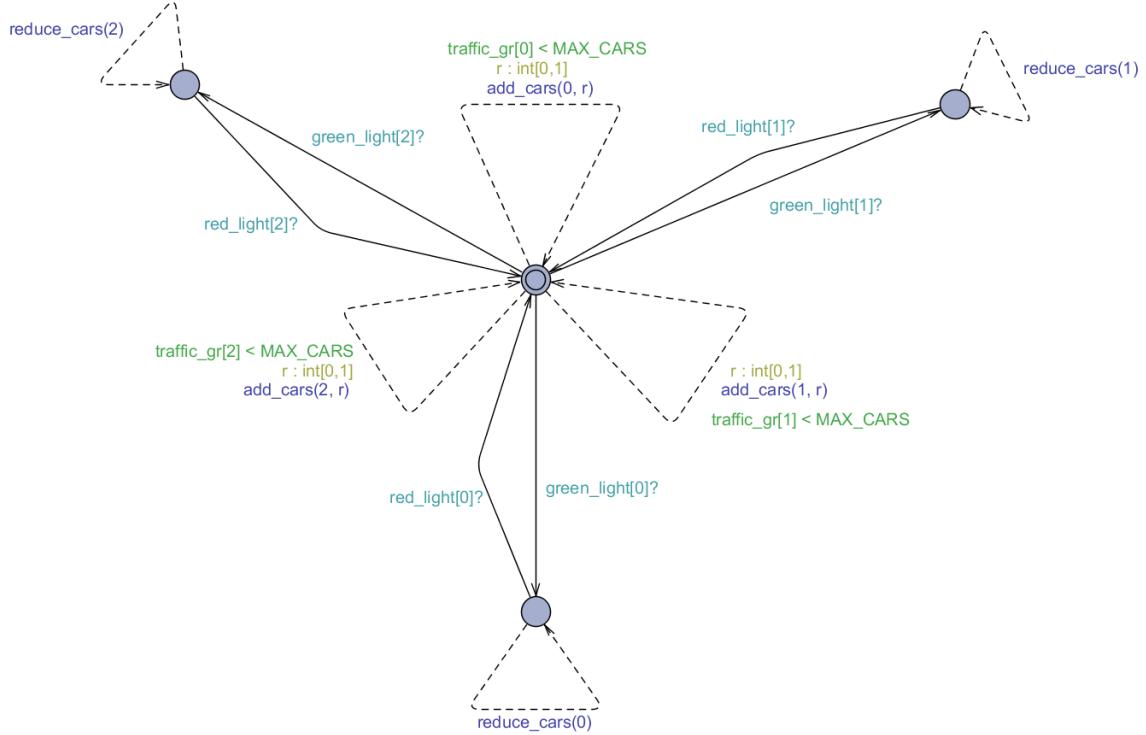


FIGURE 7 – Modélisation du trafic au sol.

L'état initial est au centre. Nous constatons trois transitions vers l'état initial lui-même. Ces transitions ne sont pas contrôlées et lorsque l'environnement prend l'une d'elles, cela permet de rajouter 0 ou 1 voiture en attente sur la piste correspondant au premier argument de la fonction *add\_cars(num\_piste, nb\_voitures)*. Également, nous contraignons l'environnement à ne plus pouvoir ajouter de voiture si la piste correspondante est déjà pleine.

En réalité, cette contrainte sert également à réduire l'espace d'état. Nous réduisons donc le nombre maximum de voitures à un petit entier (ex. 5 ou 10) pour rester raisonnable dans le temps pris par UPPAAL pour vérifier des propriétés. Le tableau *traffic\_gr* représente le nombre de voitures en attente sur la route *i*.

Les autres transitions ayant comme prédécesseur l'état initial sont prises lorsque le *i*ème feu de signalisation passe au vert. Tant que nous sommes dans l'un de ces

état, les voitures concernées peuvent décoller mais de manière non contrôlée. Nous verrons à la fin de cette section qu'il s'agit là d'un problème.

Enfin, l'une des transitions ayant pour successeur l'état initial est prise lorsque le feu correspondant passe au rouge.

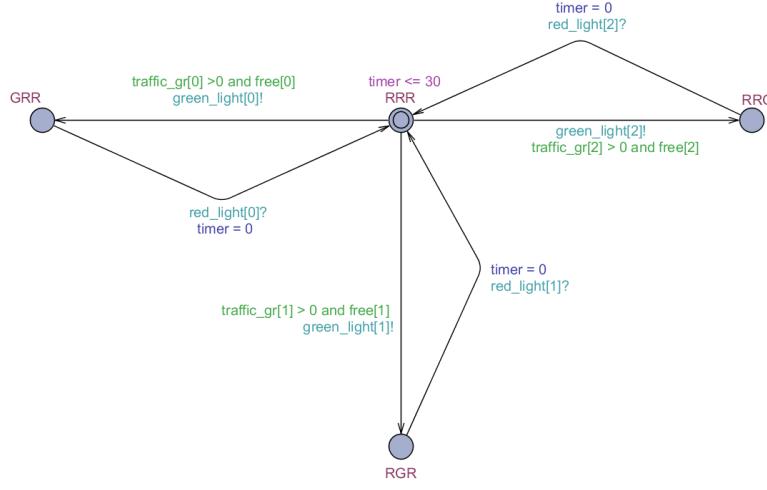
### Le contrôleur

Notre contrôleur se charge de faire passer les feux adéquats au vert lorsque les conditions sont respectées. Pour rappel, nous avons trois types de voitures qui attendent au sol sur des routes qui leurs sont réservées. Une route pour les voitures ayant une forte accélération, une pour celles avec une accélération moyenne et une dernière pour celles qui en ont une lente. Également, à chaque route est associé un capteur qui détecte si une voiture est présente dans un certain tronçon de l'autoroute. Le tronçon considéré est à une distance telle qu'une voiture dans ce tronçon entrerait en collision avec une voiture au sol<sup>2</sup> si jamais celle au sol démarrait à l'instant précis où la voiture sur autoroute était détectée. Le but de notre contrôleur est donc de tenter d'éviter ce genre de collision de se produire. Ci-dessous, nous représentons schématiquement ce que le contrôleur désire éviter :



Pour ce faire, nous avions modélisé le contrôleur ainsi :

- 
- 2. une voiture au sol attendant sur la piste associée au capteur considéré



L'état initial se nomme *RRR*, c'est-à-dire que les trois feux sont rouges. À cet état, nous avons également associé un invariant indiquant qu'il doit prendre une décision endéans les 30 unités de temps de l'horloge *timer*. Les transitions sortantes de l'état initial permettent chacune de faire passer le feu à vert s'il y a des voitures au sol qui attendent sur la piste concernée et que la voie est libre sur l'autoroute aérienne, donc qu'aucune voiture n'ait été détectée sur le tronçon correspondant. Aussi, nous revenons à l'état initial lorsque le feu est revenu au rouge. Nous réinitialisons l'horloge *timer* par la même occasion.

Chose intéressante à noter ici : il s'agit du fait que ce contrôleur devrait créer un deadlock dans le système si les capteurs n'agissaient pas de manière totalement indépendante. En effet, si le contrôleur avait la mainmise sur quand les capteurs peuvent fonctionner, alors nous aurions eu un deadlock si jamais aucune voiture n'était en attente passé les 30 unités de temps ou que la voie n'est jamais libre sur l'autoroute aérienne.

#### 4.1.2 Propriétés à vérifier

Dans cette première version, la modélisation était assez immature et nous n'avons pas pu vérifier grand chose. Néanmoins, voici celles que nous avons vérifié.

- `A[] not deadlock` (cette propriété **doit** être satisfaite) : cette propriété permet de vérifier que notre système ne sera jamais bloqué. La propriété est satisfaite dans cette version mais cela n'est dû qu'au fait que l'environnement a toujours le droit de prendre une transition. Si ce n'était pas le cas, il y aurait eu un deadlock lorsque l'horloge *timer* de notre contrôleur dépasse 30 unités de temps sans pouvoir prendre de transition à cause des guards. Nous avons remarqué cette erreur dans la version suivante.
- `traffic_gr[0] >= 1 and traffic_gr[1] >= 1 and traffic_gr[2] >= 1 -> slow.Green or normal.Green or high.Green` (cette propriété **doit** être satisfaite) : Il s'agit de notre propriété de liveness telle que nous l'avons défini. C'est-à-dire qu'une voiture sera toujours capable décoller. Cette propriété n'a pas pu être satisfaite dans cette première version assez naïve. Elle le sera dans la version suivante.

#### 4.1.3 Raisons de l'itération suivante

Comme expliqué précédemment, ce système comporte plusieurs désavantages qui nous ont bloqué lors de la phase de vérification. En effet, comme les capteurs peuvent prendre des transitions indépendamment de notre volonté, il existe toujours des runs illogiques qui rendaient nos propriétés fausses. Par exemple, les capteurs tournaient éternellement sans jamais laisser le contrôleur prendre une transition.

Aussi, un soucis que nous avons voulu corriger est le fait qu'en cas de trafic toujours dense, aucune voiture ne peut décoller. Il ne s'agit pas réellement d'un problème en soi car cela est représentatif de la réalité : on ne traverse pas une rue si la voie n'est jamais libre au risque de se retrouver aux urgences ou six pieds sous terre. Nous avons voulu corriger ce problème de famine dans la version suivante en contraignant davantage l'environnement.

Enfin, comme nous pouvons le constater dans l'automate représentant le trafic terrestre, lorsqu'un feu passe au vert, les transitions pouvant faire décoller les voitures ne sont pas contrôlées. Ceci est une erreur car cela peut permettre de ne jamais faire décoller de voiture malgré que celles-ci en ont reçues l'autorisation par le feu vert.

## 4.2 Deuxième version

Dans cette version, nous avons choisi d'introduire un nouvel automate dans notre système. Il représente un feu de signalisation mais pour les voitures se trouvant sur l'autoroute. Ce feu permet d'arrêter le trafic aérien afin de donner la possibilité aux voitures au sol de décoller, évitant ainsi une famine. Également, nous sommes devenus moins permissifs quant à la liberté allouée aux capteurs. Ceux-ci ne tournent plus de manière indépendante, nous permettant ainsi d'éviter des runs illogiques où notre contrôleur n'a jamais la main. *Ce serait comme insinuer que je ne peux plus regarder un feu de signalisation au carrefour près de chez moi parce que le feu est en train de fonctionner.* Ce que nous faisons, c'est permettre au contrôleur de consulter l'état des capteurs et agir en fonction de leur valeur.

### 4.2.1 Modélisation

Tout d'abord, introduisons le nouvel automate faisant office de feu de signalisation pour les voitures sur l'autoroute aérienne :

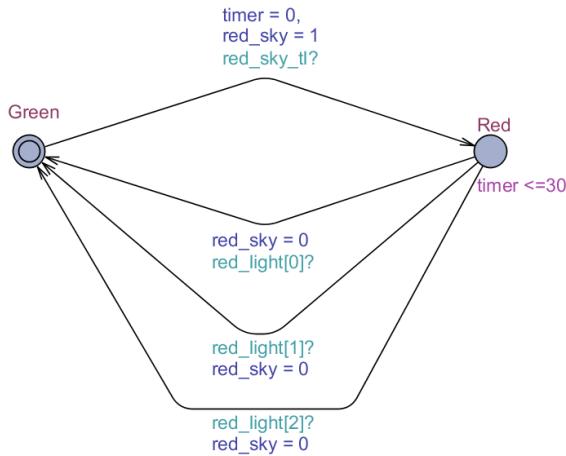


FIGURE 8 – Feu de signalisation aérien.

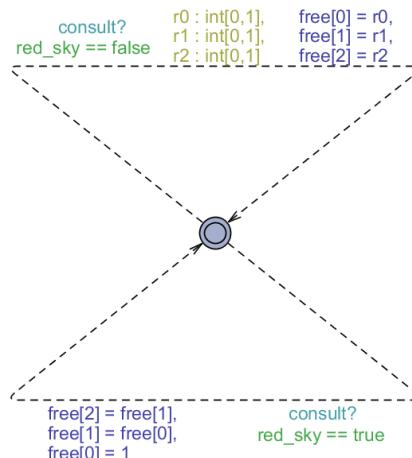
Sur l'automate ci-dessus, l'état initial est *Green*. En effet, il est davantage logique que le feu soit vert par défaut si son rôle est d'arrêter un trafic qui se déplace à grande vitesse. Concentrons-nous maintenant sur la transition *Green* → *Red*. Celle-ci permet d'arrêter le trafic aérien. Nous remarquons qu'elle ne peut être prise que si un message est réceptionné à travers le canal *red\_sky\_tl*.

Par la même occasion, nous remettons l'horloge *timer* à zéro. Enfin, nous mettons à 1 la variable booléenne représentant l'état de ce feu. Son état est consulté dans notre contrôleur.

Aussi, dans l'état *Red*, nous constatons un invariant permettant de ne pas faire durer trop longtemps le feu à rouge. Enfin, les transitions ayant pour successeur l'état initial sont prises lorsqu'une voiture au sol a pu décoller. En effet, si le feu aérien s'est mis au rouge, c'est parce que le trafic de l'autoroute n'a pas pu permettre un décollage durant 30 unités de temps. Notre contrôleur passe donc le feu aérien au rouge et ensuite passe au vert un feu terrestre.

## Les capteurs

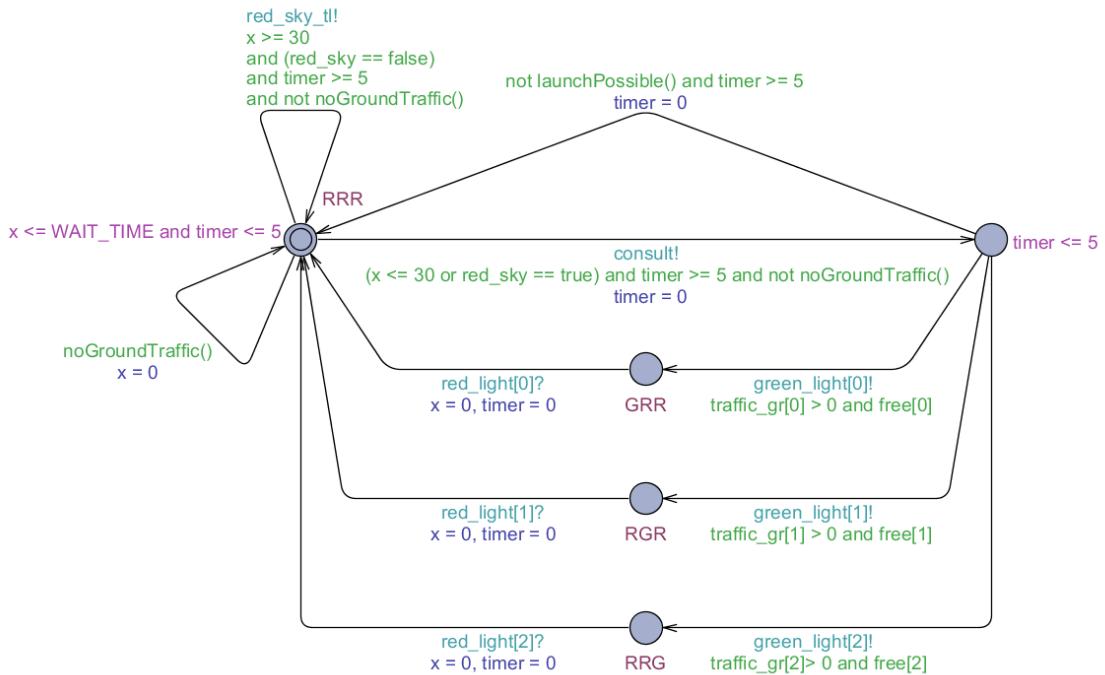
Concernant les capteurs, ceux-ci ont été simplifiés en un seul automate dans cette version. Désormais, à chaque fois qu'ils sont consultés, ceux-ci retournent chacun un état non-déterministe.<sup>3</sup> Lorsque le feu aérien est allumé (c'est-à-dire rouge), ils effectuent cependant leur comportement habituel, modélisant ainsi le fait que l'autoroute se vide. Il faut se visualiser un feu aérien placé *avant* les trois capteurs. Lorsque ce feu passe au rouge, il est donc naturel que les capteurs se vident et ne détectent plus aucun véhicule.



<sup>3</sup>. Ceci signifie que le joueur adverse peut encore se permettre de créer un trafic toujours dense tant que le feu aérien est vert.

## Le contrôleur

Le contrôleur a également changé et est désormais fonctionnel pour ce que nous désirions modéliser. Voici ci-dessous à quoi il ressemble :



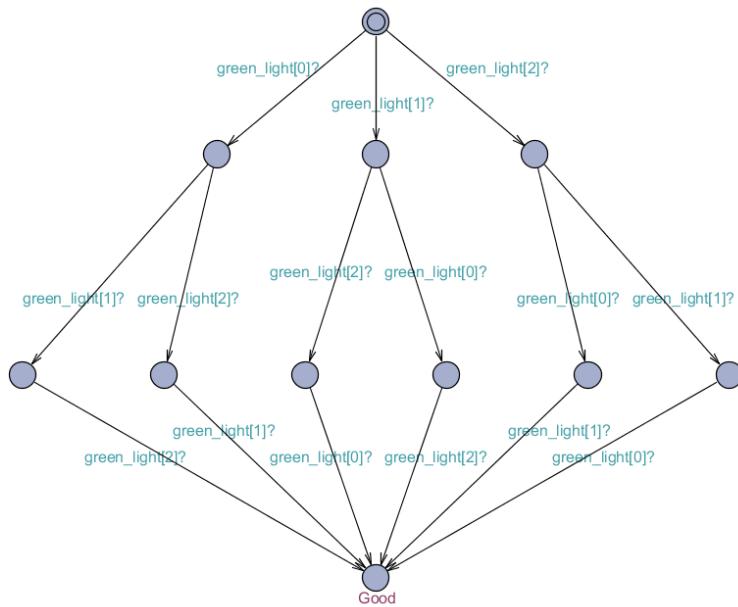
Dans cette deuxième version, le contrôleur consulte l'état des capteurs à intervalle presque régulier<sup>4</sup> tant qu'il y a du trafic en attente au sol. À l'état initial, nous avons également la possibilité d'allumer le feu aérien au-delà de 30 unités de temps de l'horloge  $x$ . À part cela, le comportement du contrôleur reste globalement pareil. Les fonctions `noGroundTraffic()` et `launchPossible()` permettent respectivement de savoir s'il y a des véhicules au sol et si le décollage d'une voiture est possible. ces fonctions ne sont qu'un raccourci vers une expression booléenne.

## Un observateur

Nous avons également créé un observateur nous permettant de tester qu'au cours d'une exécution, les trois feux de signalisation au sol aient pu s'allumer.

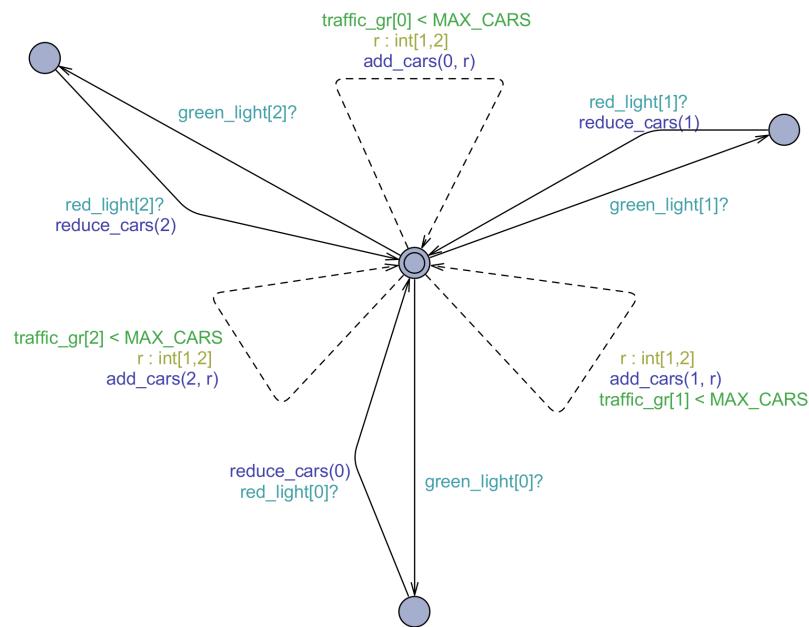
4. Au plus 10 unités de temps entre chaque consultation.

## Piste de décollage de voitures volantes autonomes



## Le trafic au sol

Enfin, dans cette version nous corrigeons le fait que le décollage des voitures étaient non contrôlables. Voici le nouvel automate implémentant le trafic au sol.



#### 4.2.2 Propriétés à vérifier

Désormais, nous avons davantage de propriétés à pouvoir vérifier maintenant que nous nous sommes abstrait des problèmes de la version précédente. Les propriétés que nous avons vérifié sont :

- `A[] not deadlock` (cette propriété **doit** être satisfaite) : même explication que pour la version précédente. Cette propriété est satisfaite.
- `E<> high.Green and traffic_gr[2] > 0 and free[2] == 0` (cette propriété **ne doit pas** être satisfaite) : Elle n'est effectivement pas satisfaite. Elle représente une propriété de sûreté. Il s'agit du but même de l'existence de notre contrôleur. Nous demandons à vérifier s'il existe un état où la voie n'est pas libre dans le tronçon d'autoroute lié aux voitures à forte accélération tout en ayant le feu terrestre pour cette route à l'état vert et des voitures en attente de décoller. Si un tel état est accessible, alors il est possible de créer un collision.
- `E<> normal.Green and traffic_gr[1] > 0 and free[1] == 0` (cette propriété **ne doit pas** être satisfaite) : Elle n'est effectivement pas satisfaite. Même propriété que ci-dessus mais concerne les voitures à accélération normale.
- `E<> slow.Green and traffic_gr[0] > 0 and free[0] == 0` (cette propriété **ne doit pas** être satisfaite) : Elle n'est effectivement pas satisfaite. Même propriété que ci-dessus mais concerne les voitures à lente accélération.
- `traffic_gr[0] >= 1 and traffic_gr[1] >= 1 and traffic_gr[2] >= 1 -> slow.Green or normal.Green or high.Green` (cette propriété **doit** être satisfaite) : Elle est effectivement satisfaite. Il s'agit de la même propriété de vivacité (liveness) définie dans la première version.
- `E<> Observer.Good` (cette propriété **doit** être satisfaite) : Elle est effectivement satisfaite. Cette propriété vérifie qu'il existe une exécution dans laquelle les trois feux passeront à vert (pas en même temps mais durant l'exécution). Cette propriété nous permet de confirmer qu'il existe au moins une exécution où une voiture de chaque type a pu décoller. Nous avons créé cette propriété car durant la modélisation, nous nous étions confronté à un soucis durant lequel, le contrôleur ne permettait qu'à un seul type de voiture de décoller.
- `A[] not((slow.Green and normal.Green) or (normal.Green and high.Green) or (slow.Green and high.Green))` (cette propriété **doit** être satisfaite) :

Elle est effectivement satisfaite. Nous vérifions que deux feux ne peuvent pas être allumés en même temps. La piste de décollage étant une ressource non partageable. Il s'agit d'un mutex. Nous verrons dans la dernière version que nous serons davantage permissif à ce niveau.

- $E<> (\text{slow.Green} \text{ and } \text{slow.timer} > 15) \text{ or } (\text{normal.Green} \text{ and } \text{normal.timer} > 15) \text{ or } (\text{high.Green} \text{ and } \text{high.timer} > 15)$  (cette propriété **ne doit pas** être satisfaite) : Elle n'est effectivement pas satisfaite. Nous ne désirons pas qu'un feu terrestre reste vert plus de 15 unités de temps.
- $A[] (\text{not}(x > 80) \text{ or } (\text{slow.Green} \text{ or } \text{normal.Green} \text{ or } \text{high.Green}))$  (cette propriété **doit** être satisfaite) : Elle est effectivement satisfaite. Nous nous assurons qu'un feu sera toujours à vert endéans les 80 unités de temps.

Nous avons donc un système plutôt fonctionnel et permettant d'effectivement vérifier qu'il fonctionne comme attendu.

#### 4.2.3 Raisons de l’itération suivante

Nous nous sommes malgré tout rendus compte d'un problème durant cette version. Il s'agissait du fait que nous n'avions pas d'équité entre les types de voitures qui décollent. Il existe des exécutions où il s'agit toujours du même type de véhicule qui est autorisé à décoller. Nous avons donc développé une troisième version proposant une équité entre les voitures qui décollent.

## 4.3 Troisième version

Cette troisième version implémente une équité entre les décollages de véhicules.

### 4.3.1 Modélisation

Il n'y a pas vraiment eu de changement au niveau des automates à part que, désormais, nous appelons une fonction à chaque consultation des capteurs. Cette fonction permet de créer une équité entre les voitures en instaurant un ordonnancement similaire au Round-robin. *Chacun son tour* en d'autres termes. La fonction est la suivante :

```

1 void updateTurn() {
2     // S'il y a du traffic en attente au sol...
3     if(traffic_gr[0] + traffic_gr[1] + traffic_gr[2] > 0) {
4         // et que le type de voiture possédant actuellement un ticket
5         // a déjà décollé OU qu'il n'y a personne en attente
6         // dans la piste possédant un ticket...
7         if(launched or traffic_gr[turn] == 0) {
8             // alors, on remet à false le fait que la voiture
9             // ayant un ticket ait décollé
10            launched = false;
11            // et on donne le ticket à la piste de décollage suivante
12            turn = (turn + 1) % 3;
13            // on s'assure que le prochain à recevoir un ticket
14            // ait des voitures en attente
15            while(traffic_gr[turn] == 0) {
16                turn = (turn + 1) % 3;
17            }
18        }
19    }
20 }
```

### 4.3.2 Propriétés vérifiées

Les propriétés vérifiées sont les mêmes que pour la version précédente et sont toutes correctement (in)satisfaites. Également, nous en rajoutons trois qui permet de définir une liveness encore plus forte.

- `traffic_gr[2] >= 1 --> high.Green` (cette propriété **doit** être satisfait) :  
Elle est effectivement satisfait. Cette propriété permet de nous assurer que peu importe l'exécution, dès qu'il y aura au moins une voiture en attente sur la piste correspondant aux voitures à forte accélération, alors le feu finira forcément par passer au vert.

- `traffic_gr[1] >= 1 --> normal.Green` (cette propriété **doit** être satisfaite) : Elle est effectivement satisfaite. Pareil qu'au-dessus mais pour les voitures à accélération normale.
- `traffic_gr[0] >= 1 --> slow.Green` (cette propriété **doit** être satisfaite) : Elle est effectivement satisfaite. Pareil qu'au-dessus mais pour les voitures à accélération faible.

#### 4.3.3 Raisons de l’itération suivante

Avec cette version, nous avons obtenu un système assez équitable en plus de ne plus causer de famine. Nous avons cependant remarqué que la piste de décollage pouvait être optimisée. En effet, puisqu'une voiture à forte accélération dépassera forcément une voiture à plus faible accélération, nous nous sommes dit que nous pourrions créer une ultime version dans laquelle nous rendrions les feux de signalisation au sol un peu plus intelligent. Nous avons également voulu optimiser le décollage de voiture en plus d'offrir une certaine équité. C'est-à-dire que si c'est au tour de la voiture  $i$  de décoller, alors nous faisons également décoller les voitures plus lentes si possible.

## 4.4 Quatrième version

Dans cette version, les feux au sol sont plus intelligents. Si une voiture à forte accélération décolle, nous pouvons faire en sorte d'également faire décoller une plus lente si les conditions nous le permette. Aussi, nous optimisons les décollages en faisant décoller des voitures plus lentes que celle dont c'est actuellement le tour de décoller si cela est possible.

### 4.4.1 Modélisation

Afin de mettre en œuvre cette version, nous avons principalement modifié notre contrôleur. Globalement, il reste le même sauf que nous obtenons de nouvelles combinaisons d'états. Par exemple, l'état où le feu des voitures rapides et lentes sont allumés en même temps. Voici à quoi ressemble notre contrôleur pour cette version :

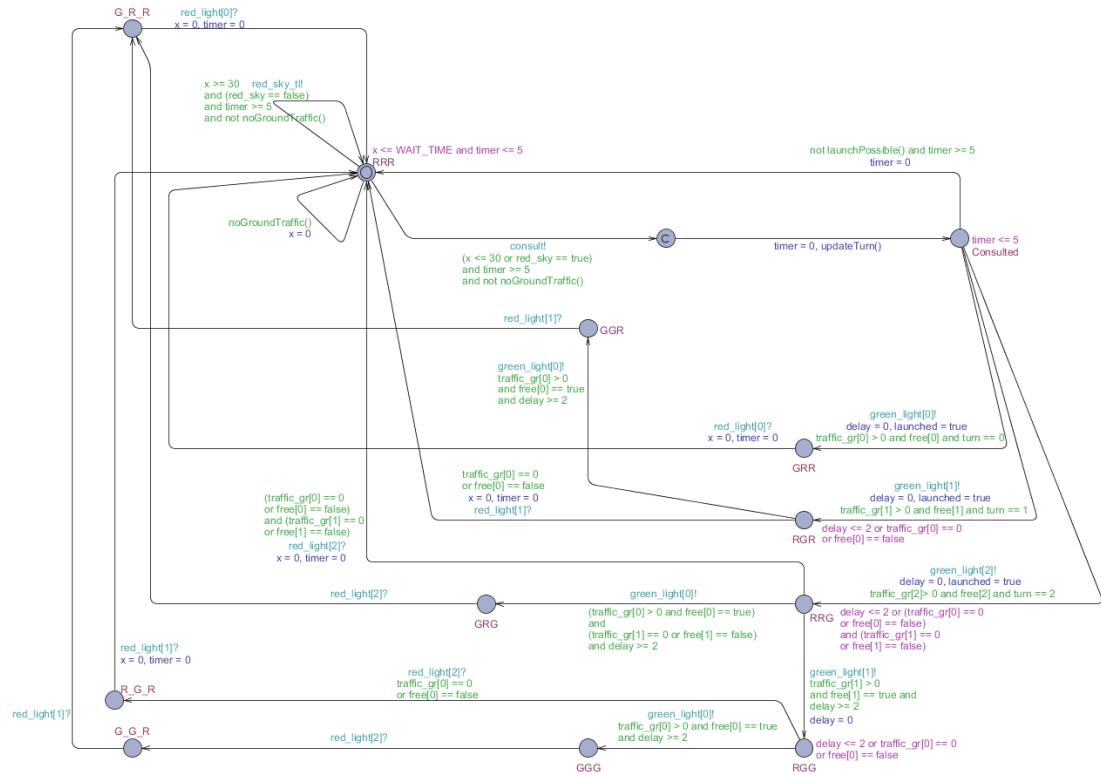


FIGURE 9 – Notre contrôleur permet d'allumer désormais plusieurs feux si les conditions sont respectées.

Dans le contrôleur ci-dessus, tout à droite, nous retrouvons les états où un seul feu est allumé. Ensuite, à partir de ces états, nous avons des transitions munies de guards permettant d'allumer un second feu après un très court délai. Si possible, nous activons un troisième feu avant de les repasser au rouge tout à gauche.

#### 4.4.2 Propriétés vérifiées

En plus des propriétés des versions précédentes, nous en avons encore ajouté. Il est cependant à noter que puisque nous permettons l'allumage de plusieurs feux au sol, la piste de décollage n'est plus un mutex mais une ressource partagée. Concernant les nouvelles propriétés, nous avons :

- `ctrl.Consulted and traffic_gr[0] > 0 and free[0] and traffic_gr[1] > 0 and free[1] and turn == 1 --> ctrl.GGR` (cette propriété **doit** être satisfaite) : Elle est effectivement satisfaite. S'il a la possibilité d'optimiser en lançant deux voitures à la suite (ici, normal -> lente), alors dans le futur il le fera. Ici, la propriété ne concerne que les voitures normales et lentes mais nous l'avons également testé pour les autres combinaisons de feux.

## 5 Implémentation

Nous avons implémenté en langage JavaScript le système modélisé dans la troisième version. Une version peut être trouvée [sur notre dépôt GitHub](#). Pour l'exécuter, il suffit d'ouvrir le fichier *index.html* dans un navigateur Web.<sup>5</sup>

Concernant l'implémentation de l'environnement, des voitures sont ajoutés de manière aléatoires au sol et dans les airs. Le contrôleur agit en fonction de l'état du trafic.

Voici quelques captures d'écran montrant l'implémentation :

---

5. Et par acquis de conscience, un navigateur différent d'Internet Explorer.

## Piste de décollage de voitures volantes autonomes



FIGURE 10 – Ici, nous voyons une voiture "lente" s'apprêtant à s'insérer sur l'autoroute. En pointillé, nous entourons la zone où le véhicule s'insérera.



FIGURE 11 – Ici, nous voyons une voiture "normale" s'apprêtant à s'insérer sur l'autoroute. En pointillé, nous entourons la zone où le véhicule s'insérera.

## 6 Synthèse

Nous avons également tenté de synthétiser un contrôleur sur base d'une stratégie gagnante grâce à UPPAAL TiGa. Nous sommes donc partis sur une version minimalistre de notre projet où tous les automates étaient présent et où, le contrôleur a été réduit afin de ne permettre que d'allumer les feux terrestres. La principale différence avec les autres versions est que le contrôleur n'a aucune notion de ce qu'est un lancement sécurisé.

Après avoir fait cela, nous avons introduit un automate permettant de modéliser un jeu entre notre contrôleur et l'environnement. Cet automate possède deux états qui, dès qu'une collision survient, effectue une transition vers un état nommé "Crash". Le contrôleur décide de quand faire passer le feu terrestre à vert tandis que l'environnement décide laquelle des deux transitions disponibles est active.

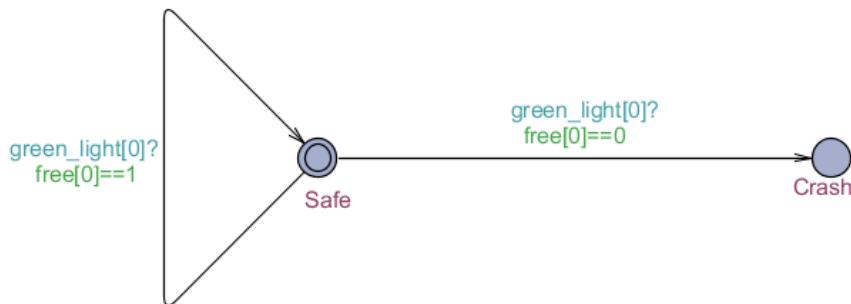


FIGURE 12 – Nouvel automate modélisant le jeu entre l'environnement et le contrôleur

Étant donné que notre objectif était de synthétiser une stratégie gagnante permettant d'éviter des collisions, nous avons introduit la propriété "control: A[] not game.Crash" permettant d'indiquer à TiGa la condition nécessaire pour gagner le jeu. Cette condition permet également à TiGa de synthétiser une stratégie pour le contrôleur qui répond à nos attentes de sécurité, c'est-à-dire de ne jamais produire de collision lors des lancements.

Au début, l'outil nous donnait effectivement des stratégies gagnantes grâce à la commande "./verifytiga -w0 -c0 <path>.xml <path>.q" mais celles-ci étaient

immenses. En effet, TiGa nous retourne une liste de transitions à effectuer selon un contexte précis. Concernant l'immense taille des stratégies fournies, pour avoir un ordre d'idée, TiGa nous retournait une stratégie gagnante mais impliquant de prendre des transitions précises dans plus 6000 contextes très précis. Pour gérer de tels quantités de contextes, il aurait fallu implémenter un outil permettant de parser la stratégie. Désirant implémenter un contrôleur à la main, nous avons donc choisi de réduire l'espace d'état afin de synthétiser une stratégie gagnante pouvant être raisonnablement implémentable à la main. Pour ce faire, nous avons réduit le nombre de feux terrestres à 1, retiré le feu aérien et réduit le nombre de voitures en attente à au plus 5. Voici le contrôleur que nous avions après cette réduction :

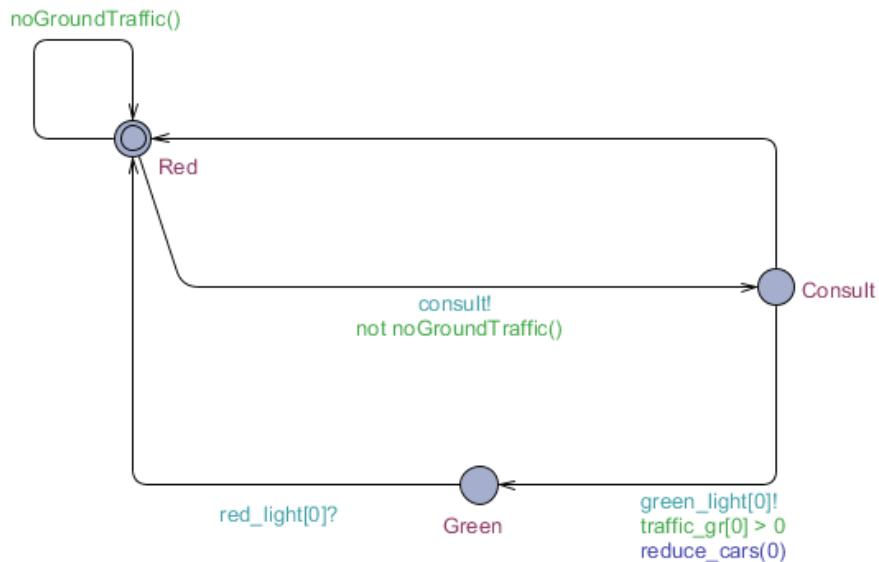


FIGURE 13 – Contrôleur après réduction de l'espace d'état.

Procéder ainsi nous a permis de générer avec succès une stratégie gagnante permettant de s'assurer qu'aucune collision ne puisse avoir lieu. Celle-ci nous a permis de modifier notre contrôleur afin de pouvoir gagner le jeu, et donc d'éviter la production de collision lors des lancements. Ci-dessous, on peut voir que la principale modification du contrôleur s'est faite dans le guard de la transition allant de `Consult` à `Green`. L'ajout du teste sur `free[0]` permet d'obliger des lancements sécurisés uniquement.

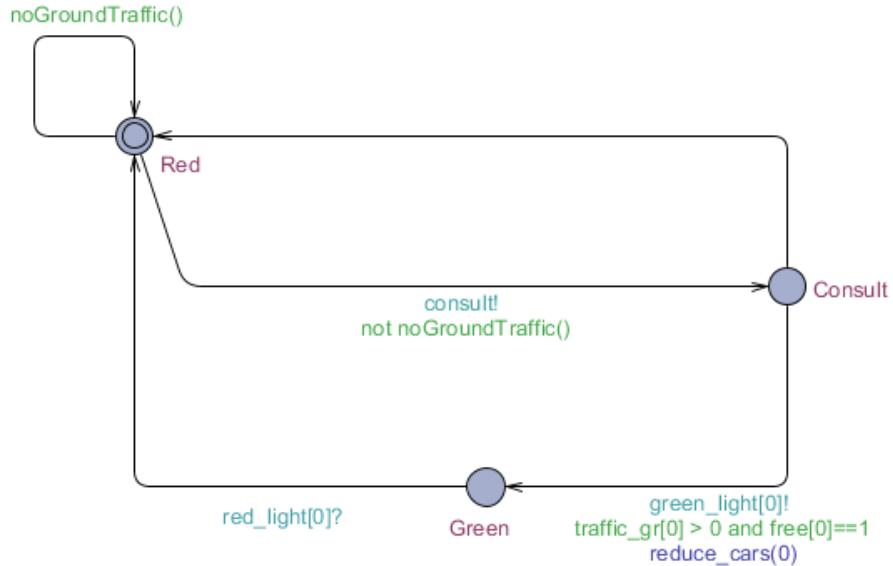


FIGURE 14 – Contrôleur modifié sur base de la stratégie gagnante synthétisée par TiGa.

Ce contrôleur permet effectivement de garantir l'absence de collision. Nous avons introduit la propriété "A[] not game.Crash" à UPPAAL pour lui demander de la vérifier et celle-ci est effectivement satisfaite.

Cependant, comme nous nous en doutions, créer un contrôleur de cette manière nous a fait perdre notre propriété de liveness. Nous ne sommes plus assuré qu'un véhicule sera toujours capable de décoller. En effet, la suppression du feu aérien lors de la phase de réduction d'espace d'état permet à l'environnement de ne jamais laisser de voitures décoller. La propriété "A<> slow.Green" n'est pas satisfaite contrairement aux autres versions.

**Extrait -** Voici un extrait de la stratégie que nous a retourné TiGa :

Initial state:

```
( slow.Red sensors.on ctrl.Red tg.Init game.Safe ) free[0]=1  
traffic_gr[0]=0 traffic_light[0]=0  
(x==slow.timer && slow.timer==ctrl.timer && ctrl.timer==0)
```

Strategy to avoid losing:

```
State: ( slow.Red sensors.on ctrl.Consult tg.Init game.Safe )  
free[0]=1 traffic_gr[0]=0 traffic_light[0]=0  
When you are in (x<=80),  
take transition ctrl.Consult->ctrl.Red { 1, tau, 1 }
```

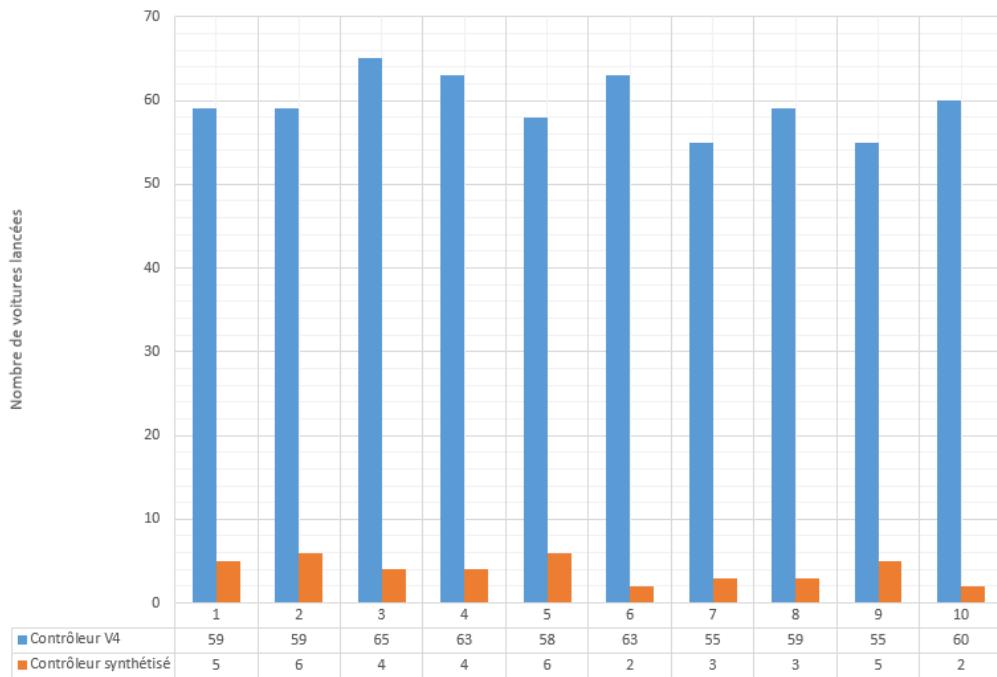
```
State: ( slow.Red sensors.on ctrl.Consult tg.Init game.Safe )  
free[0]=1 traffic_gr[0]=1 traffic_light[0]=0  
When you are in true,  
take transition ctrl.Consult->ctrl.Green  
{ traffic_gr[0] > 0, green_light[0]!, reduce_cars(0) }  
slow.Red->slow.Green { 1, green_light[0]?, timer := 0,  
traffic_light[0] := 1 }  
game.Safe->game.Safe { free[0] == 1, green_light[0]?, 1 }  
<...>
```

(La stratégie complète est disponible sur notre dépôt GitHub, dans le fichier Strategy.txt sous le dossier Synthesis\_version.)

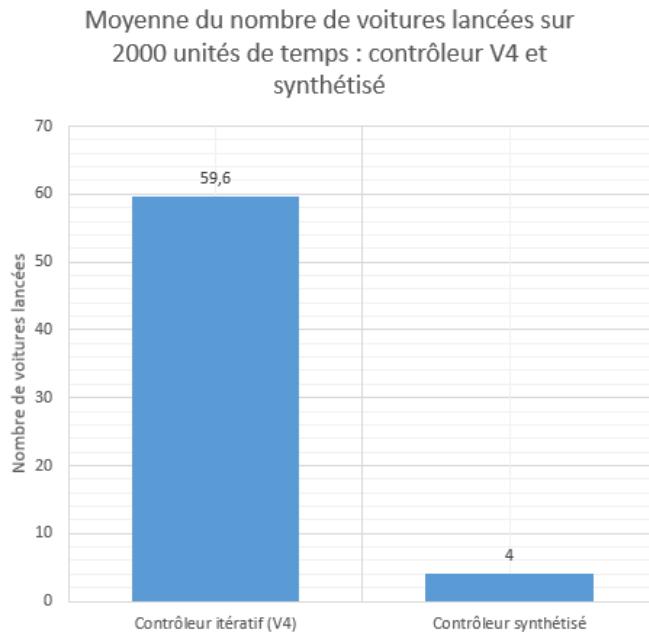
## 7 Comparaison : synthèse VS contrôleur itératif

Afin de comparer l'efficacité de notre contrôleur avec celui synthétisé par UP-PAAL, nous avons effectué des tests. Le test suivant concerne le nombre de voitures ayant décollés sur le même intervalle de temps, sur des exécutions aléatoires. On a répété ces tests une dizaine de fois et il s'est avéré que notre contrôleur (créé de manière itératif) permet de lancer, en moyenne, environ quinze fois plus de voitures que celui généré par TiGa. Voici ci-dessous un graphe représentant ce qui vient d'être mentionné.

Sur 10 runs aléatoires, nombre de voitures lancées sur 2000 unités de temps :  
contrôleur V4 et synthétisé



Ci-dessous nous pouvons observer un graphe représentant la moyenne des résultats pour les deux contrôleurs.



Également, nous avons remarqué que le contrôleur synthétisé ne satisfait plus la plupart de nos propriétés car il a été généré pour satisfaire une unique propriété. Par exemple, le contrôleur généré par TiGa viole nos propriétés de liveness ; il existe des exécutions où on ne lancera jamais de voitures.

## 8 Conclusion

Nous voici arrivé au terme de ce projet. Nous avons tous, grâce à ce projet, pu perfectionner nos compétences en modélisation, vérification et avons appris à synthétiser des automates sur base d'une stratégie gagnante fournie par TiGa.

Également, nous avons pu nous perfectionner dans le domaine des automates et jeux temporisés. Les parties les plus difficiles durant ce projet ont été la détection des erreurs lorsqu'UPPAAL nous montrait un contre exemple pour ne pas satisfaire nos propriétés, surtout lorsqu'il s'agissait d'erreurs due à notre incompréhension.

Après avoir parlé avec nos professeurs dont les conseils nous ont été précieux, nous avons souvent été directement débloqués des situations dans lesquelles nous étions et grâce à M. Geeraerts qui nous a donné l'idée de comparer nos deux contrôleurs dont un a été synthétisé, nous en avons encore davantage appris sur le sujet. La synthèse est certes une technique puissante mais pas aisée à maîtriser.