

UNIVERSITÉ D'ÉVRY – VAL D'ESSONNE
Département d'informatique



RAPPORT DE STAGE

de 3^e année de la licence d'informatique
parcours INFO(CILS)
soutenu par

Valentin COTTE

le 26 Juin 2017

**Réalisation d'un outil de type front-end pilotant un
analyseur de contrôleurs de véhicules qui repose sur
UPPAAL**

Directeur de stage

Hanna KLAUDEL, Johan ARCILE

Établissement d'accueil

UNIVERSITÉ D'ÉVRY VAL D'ESSONNE (Laboratoire IBISC)

Année universitaire 2016-2017

Table des matières

1	Contexte	2
2	Le sujet et les objectifs	3
2.1	Problématiques	3
2.2	Objectifs	3
3	Conception et implémentation	4
3.1	Ressources exploitables	4
3.2	Architecture	4
3.3	Choix des technologies	7
3.4	Solutions techniques	7
3.5	Exemple	11
4	Limites et extensions	12
5	Ce que le stage m’a apporté	13
A	Annexe	i
B	Annexe	ii
C	Annexe	iii

1 Contexte

Lors de mon stage, j’ai été amené à travailler avec une équipe effectuant des recherches sur les véhicules autonomes et plus précisément leurs prises de décisions.

À l’aide de l’outil UPPAAL¹, l’équipe a modélisé le comportement de véhicules autonomes avec des automates temporisés. Les automates temporisés sont souvent utilisés pour modéliser (à un certain niveau d’abstraction) des systèmes temps réels et tester leurs propriétés temporelles. Une fois la situation initiale défini par un nombre de paramètres variables $n > 40$ (dimensions de la route, taille des véhicules, leur vitesse initiale, leur accélération, leurs limites, etc), UPPAAL permet la vérification formelle de cet ensemble d’automates temporisés pour une liste de requêtes données. On parlera dès lors d’un analyseur de contrôleurs de véhicules. Par exemple, pour une situation donnée avec plusieurs véhicules ayant des objectifs différents, on cherche à savoir s’il existe des cas où un des véhicules est amené à dépasser la limitation de vitesse. UPPAAL va parcourir de façon exhaustive tous les états du système d’automates pour déterminer si la requête est satisfaite ou non, et générer une trace visible par l’utilisateur. On peut voir ainsi les chemins de décisions des véhicules à éviter. Cette vérification de modèle permet d’éviter les erreurs éventuelles pour la situation donnée. De façon général, on peut prévenir certains dangers, comme une collision, et autres prises de risques.

Le sujet du stage consiste à concevoir un outil de type front-end, lequel implémentera cet analyseur de contrôleurs de véhicules. Il s’agit finalement de développer une interface, indépendante de celle d’UPPAAL qui est propre à la modélisation et vérification de systèmes en général, pour se concentrer sur notre cas spécifique d’analyse de contrôleurs véhicules. Le type “front-end” de l’outil à implémenter suggère la création de la partie immergée de l’iceberg², mais cela implique aussi la conception de fonctions plus complexe et non-visible par l’utilisateur.

1. UPPAAL est un outil de modélisation, de validation et de vérification de système en temps réel, tels que des automates temporisés et étendus avec des types de données

2. L’iceberg représente l’analyseur de contrôleurs de véhicules

2 Le sujet et les objectifs

2.1 Problématiques

Pour définir l'état d'un système de voitures autonomes, il faut d'une part procéder manuellement. Ensuite, l'exploration des chemins partants cet l'état initial est certes exhaustive, cependant on ne peut pas considérer cela comme satisfaisant en termes de recherche pour plusieurs situations distinctes, voir un ensemble de situations. Il y a donc une première problématique : il n'y a qu'un unique état totalement couvert par l'analyseur de contrôleurs de véhicules.

Pour être plus précis sur l'utilisation d'UPPAAL, changer d'état initial consiste à modifier chaque paramètre déclaré dans le code, à l'endroit où les variables et constantes utilisées par les automates sont définis. Une grande quantité de paramètre à initialiser pour définir un état du système est nécessaire, afin de définir de nouvelles instances. La mise en place de ces changements manuels se fait au prix d'un coût en temps qui se doit d'être pris en compte.

De plus, le lancement de la vérification ainsi que la récupération des résultats sont tout autant manuelles, et le temps de calcul entre chaque changement est non-négligeable.

En conclusion, la méthode ne permet pas de façon satisfaisante d'effectuer une série de tests manuels, ce qui construit une deuxième problématique.

2.2 Objectifs

Afin de remédier au problème de situation initiale unique, le principal objectif du stage est de trouver un moyen d'analyser un ensemble d'états initiaux. Il faut éviter à l'utilisateur de définir les paramètres un par un de chaque état initial de l'ensemble à analyser. Pour utiliser une méthode de balayage des états initiaux de cet ensemble, l'utilisateur doit avoir la possibilité de définir les paramètres, qui l'intéresse, sous forme d'intervalle.

Pour un paramètre de valeur min , l'utilisateur définit aussi un pas $p > 0$ et une valeur $max > min$. Son intervalle est défini par l'ensemble des valeurs de $v < max$, avec $\forall v, v = min + p \times k$ (soit $k \in \mathbb{N}$).

Pour atteindre le premier objectif, il faut tout d'abord automatiser le processus de vérification pour une situation initiale donnée. En effet, il ne peut s'agir de balayer un ensemble d'états initiaux si l'on est obligé de modifier manuellement chacun de ces états. C'est pourquoi, dans un premier temps, il est nécessaire d'automatiser le processus de modélisation et les requêtes qui sont vérifiées, puis récupérer les résultats.

3 Conception et implémentation

3.1 Ressources exploitables

L'outil UPPAAL possède une interface graphique et n'est donc pas automatisable, cependant il est possible :

- D'exporter le code modélisant le système sur UPPAAL, en un fichier XML.
- Puis d'utiliser l'outil "Verifyta" (fourni par UPPAAL), grâce à une commande dans un terminal. Verifyta prend en entrées le fichier XML et un autre fichier contenant les requêtes (fichier avec l'extension .q) puis renvoie en sortie les résultats de la vérification.

Dans la figure de l'Annexe A, la requête détecte si pour tous les états possibles du système, il en existe un où il y a une collision.

La requête n'est pas satisfaite, donc il n'y a pas de collision, et donc aucun risque (pour cette unique situation initiale).

3.2 Architecture

Automatisation Avec ces ressources en mains, les premiers objectifs furent très clairs :

- Créer un logiciel avec interface graphique où l'utilisateur avait simplement à remplir les paramètres sous forme de questionnaire, puis qui écrirait le fichier XML en remplaçant les paramètres par les nouveaux (l'utilisateur aura aussi la possibilité de modifier les requêtes).
- Écrire un script qui lancerait ce nouveau logiciel, puis qui lancerai Verifyta avec ce nouveau fichier XML.

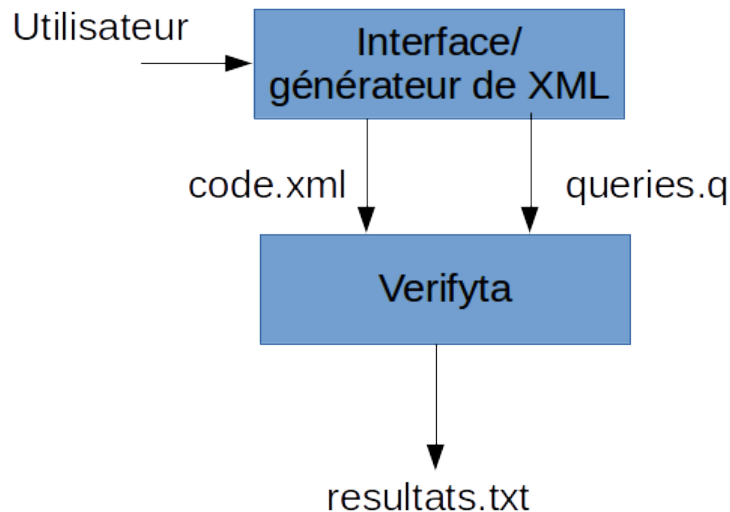


FIGURE 1 – Architecture du script d'automatisation d'une seule instance

Le nouveau logiciel s'appelle Generator.jar (écrit en java), en suivant cette architecture 1 , un premier script a été écrit :

```
#!/bin/sh
java -jar Generator.jar
echo File VerifCar.xml uploaded
echo File queries.q uploaded
./verifyta -q -s VerifCar.xml queries.q > Tests.txt
echo File Tests.txt done|
```

Dans ce script, Generator.jar est lancé, ouvrant le questionnaire où l'utilisateur remplit les paramètres. Une fois terminé, le fichier XML (VerifCar.xml) est créé, ainsi le fichier contenant les requêtes (queries.q). Ensuite, Verifyta est exécuté, et on récupère le résultat de la vérification dans un fichier texte (Tests.txt). Ainsi le processus que l'on a pour objectif de répéter pour un ensemble d'états initiaux, est exécuté de façon semi-automatique³.

On voit ci-dessus quels sont les choix des technologies utilisés, ils seront justifiés un peu plus tard.

Balayage Maintenant que le processus est automatisé, pour balayer un ensemble d'états initiaux, il s'agit de le répéter en boucle en faisant varier les paramètres voulus. Cependant, on ne veut pas remplir de nouveau le questionnaire de paramètres à chaque boucle, l'interface et le générateur vont donc être séparés en deux fonctions d'instincts.

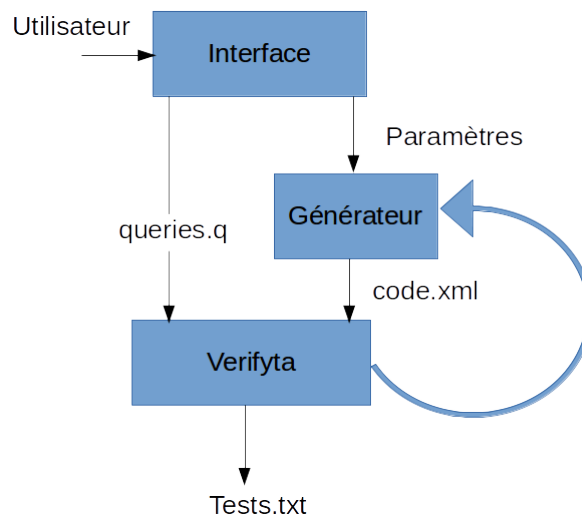


FIGURE 2 – Architecture souhaité

Dans cette nouvelle architecture, l'interface est lancée une seule fois pour retourner en sortie les requêtes (valables pour toutes les vérifications à venir) et les paramètres de chacune des situations initiales. Les paramètres vont être par la suite encodés, par le générateur, en états initiaux qui sont automatiquement vérifiés par Verifyta.

Cette architecture pose un petit problème, c'est la transmission des paramètres de l'interface au générateur. En effet, le générateur prend en arguments les paramètres pour créer le fichier XML, si on l'exécute avec une commande, on doit écrire par exemple :

```
java -jar Generator.jar 100 10 50000 1050 1000 4000 -500 300 100 100
100 500 200 20000 40000 3 350 750 2 11 11 3 3 4 4 600 0 525 525 2000
2000 -100 0 2 0 1 1 50000 1 1 0 1 1 50000 1 1 200 50 1000 100 500|
```

3. On ne peut pas considérer le processus comme complètement automatisé car l'interface est indissociable de celui-ci.

Concrètement, il est difficile d'exécuter de façon récursive, chacun des générateurs avec des paramètres différents. La solution trouvée pour résoudre ce petit problème est la suivante : l'interface ne va pas transmettre les paramètres, elle va écrire un par un chaque processus de vérification de chaque état initial. L'architecture final résolvant le problème de transmission est défini comme tel :

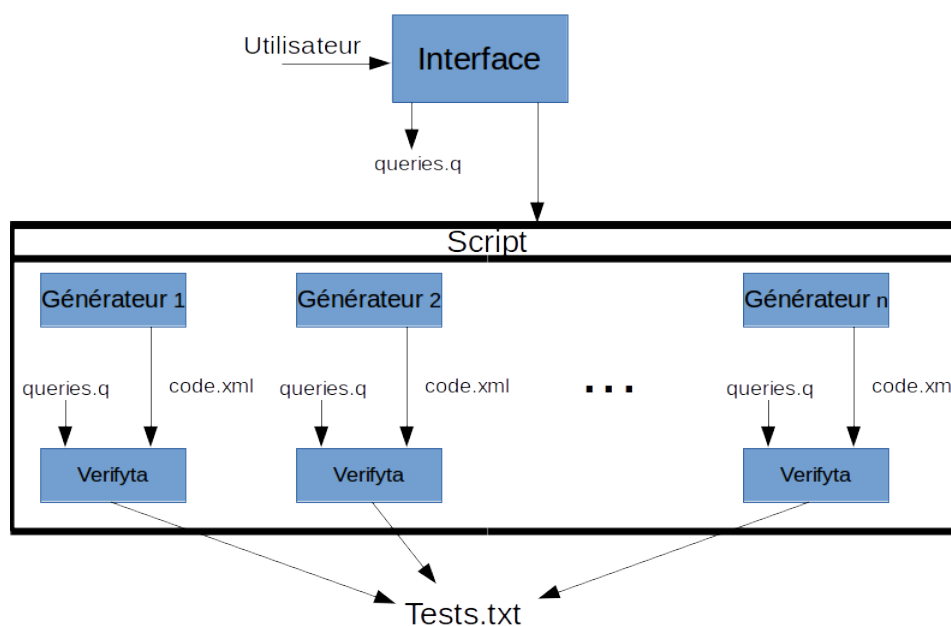


FIGURE 3 – Architecture nécessaire pour résoudre le problème

Pour finir, dans cette dernière architecture, une fois l'interface remplie, elle retourne toujours le fichier de requête "queries.q" qui sera utilisé pour chacune des vérifications. Par ailleurs, au lieu de retourner aussi les paramètres, elle écrit un script dans lequel chaque générateur sera exécuté avec ses propres paramètres. La sortie de chacun de ces générateurs sera vérifiée par Verifyta pour les requêtes données, et les résultats seront tous écrits à la suite dans le même fichier texte.

3.3 Choix des technologies

Script Shell L'utilisation d'un script Shell est nécessaire pour exécuter la commande `Verifyta` fourni par UPPAAL. De plus, il va permettre de bien séparer l'interface graphique et la génération de fichier XML automatique.

Étant donné la simplicité des commandes (principalement des exécutions de fichiers et quelques "echo") l'interpréteur shell importe peu, ici on utilise `/bin/sh`.

Java Les consignes du stage impliquant l'utilisation de matières vu au cours de l'année, il a été naturel de se tourner vers Java pour l'interface graphique. Cependant ce choix est discutable :

- Dans un premier temps, la puissance de java réside dans sa portabilité. Or, l'outil réalisé est exclusif aux systèmes Linux étant donnée l'utilisation du shell. Ce n'est pas très important mais cela ne respecte pas le slogan "Write once, run anywhere" qui accompagne Java.
- Dans un second temps, plus à cause de mon manque expérience que la technologie, j'ai réalisé l'interface en java Swing. Swing est d'ores et déjà considéré par certains professionnels comme obsolète, et pensent qu'il faut passer à JavaFx. J'ai donc appris les bases de JavaFx au cours de mon stage mais cela prendra beaucoup trop de temps (que je n'ai pas) pour remplacer l'interface swing, d'autant plus que ce n'est qu'un problème cosmétique et donc très peu important pour un logiciel d'utilité privé.

J'ai aussi choisi java pour faire le générateur de fichier XML parce qu'au début il était confondu avec l'interface. Cependant, maintenant qu'il est exécuté en boucle dans un problème avec une complexité en temps exponentiel, il peut être intéressant de le coder avec un langage plus proche de la machine (en C par exemple) si l'on veut optimiser le temps un maximum, mais ce n'est pas une priorité car il s'agit simplement d'écrire un fichier (ce qui est relativement rapide), et ce n'est qu'une infime partie du processus.

3.4 Solutions techniques

Quelques-unes des solutions techniques ont déjà été survolées (comme la transmission des paramètres au générateur), il est important d'évoquer quelques résolutions de problèmes techniques avant de passer à un exemple concret.

Récupération des paramètres D'un point de vue technique, les paramètres sont des entiers ou des tableaux d'entiers à récupérer, la programmation des fonctions est orientée Objets, les paramètres seront donc tous récupérés dans un objet représentant un ensemble de paramètres.

La taille des tableaux dépend de certains paramètres, ce qui implique :

- 1) Le questionnaire ne peut pas être statique, le nombres de paramètres à remplir dépend directement de certains paramètres.
- 2) Les paramètres en question doivent être remplis avant, et s'ils sont modifiés après, cela invalide les paramètres dépendants.

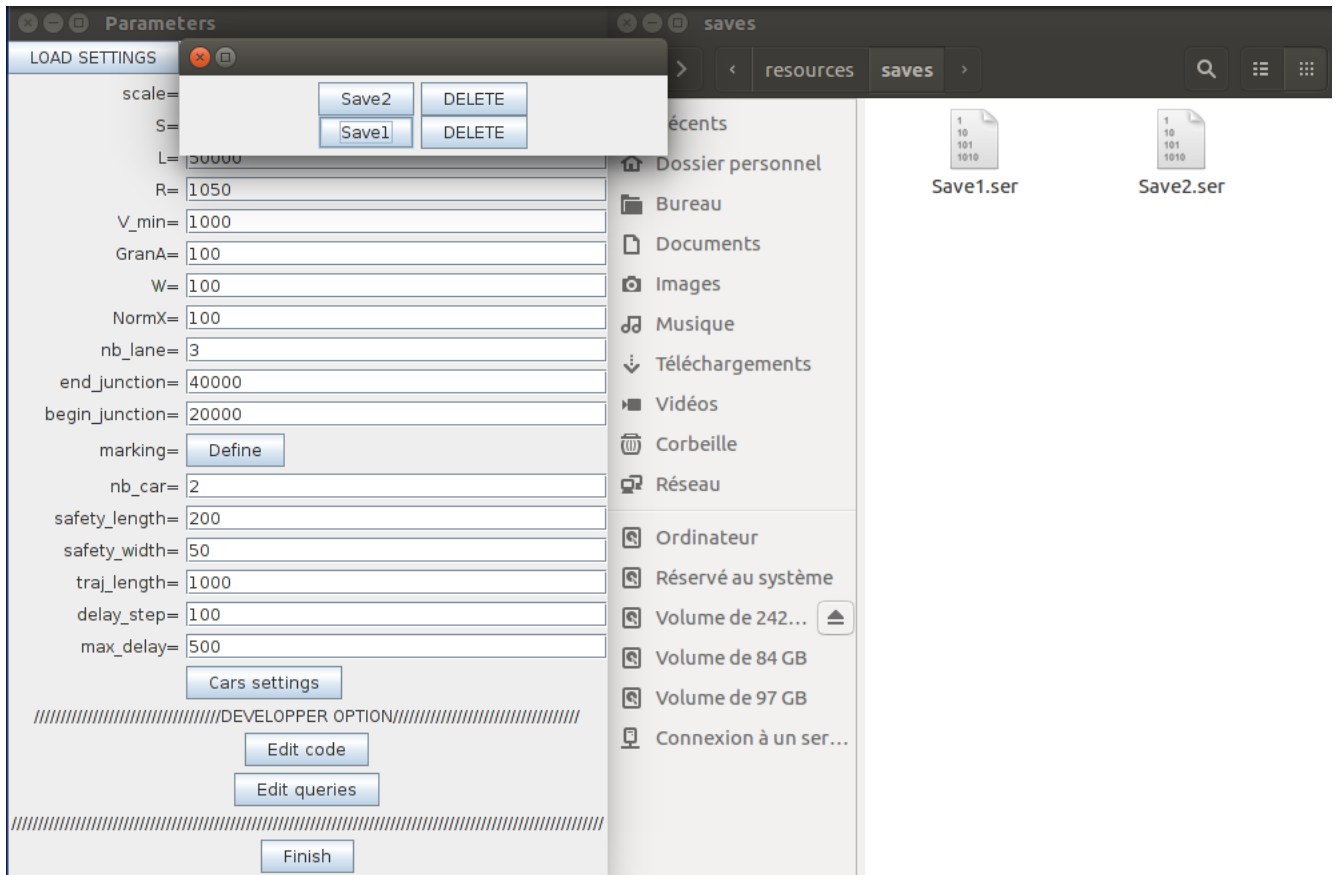
Pour y remédié, les tableaux auront leur propre fenêtre avec leur propre questionnaire à remplir.

En ce qui concerne les paramètres des voitures, ils sont contenues dans d'autres objets représentant les ensembles de paramètres des voitures. Ainsi, la modélisation est bien plus intuitif, et l'ensemble de paramètres principal contient une liste de toutes les voitures.

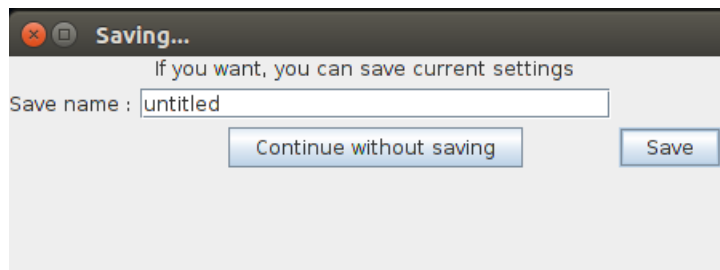
Du côté de l'interface graphique, les voitures ont chacune leur fenêtre avec un questionnaire.

Dans la figure de l'Annexe B, on voit les différentes fenêtres de chaque questionnaire de l'ensemble de paramètres, des tableaux et des ensembles de paramètres des voitures. On notera que l'unité des paramètres dépend de la valeurs "scale", par exemple, la vitesse sera exprimé en $\frac{m/s}{scale}$ (si $scale = 100$ alors $init_V = 2000 \rightarrow 20m/s$).

Sauvegarde des paramètres Le but étant de simplifié le processus d'analyse de contrôle des véhicules, ce serait tout autant coûteux en temps s'il fallait remplir plusieurs fois les paramètres. Heureusement, java nous offres les outils nécessaires pour sérialiser⁴ un objet, ainsi, on peut facilement enregistrer notre ensemble de paramètres dans un fichier et le réutiliser plus tard. Par ailleurs, c'est le dernier ensemble de paramètres enregistrer que l'on ouvrira au lancement du logiciel.



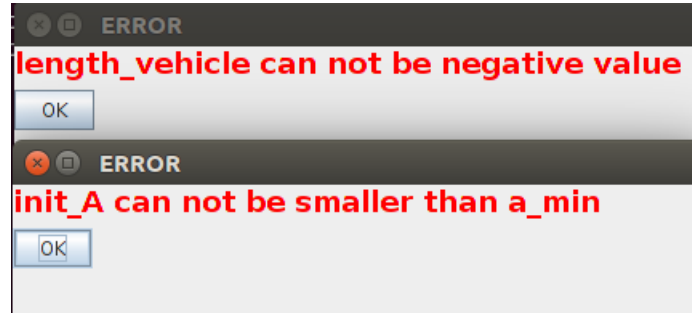
On voit dans la figure ci-dessus, à droite l'interface utilisateur pour sauvegarder, et à gauche le dossier contenant les sauvegardes (les objets sérialisés). Actuellement, la possibilité de sauvegarder les paramètres est offert lorsque l'utilisateur clic sur "Finish", mais très prochainement, il aura aussi la possibilité de le faire aussi en cliquant sur un bouton "Save" tout simplement.



4. On enregistre l'objet en question dans un fichier sous forme de flux binaire, son processus de décodage s'appelle la désérialisation, ce qui nous permet de le récupérer lorsque le logiciel est relancé.

Gestion des erreurs Ces nombreux paramètres ont des contraintes, par exemple, il serait absurde que la vitesse initiale d'une voiture soit plus grande que sa vitesse maximale. Ou encore si l'on définit 2 voitures, puis que l'on augmente le nombre de voitures à 3, si l'utilisateur n'a pas défini la troisième, alors il y a une erreur.

Même si pour l'instant le logiciel est destiné à aider les recherches de l'équipe, et par conséquent l'utilisateur connaît toutes ces contraintes, le logiciel se doit un minimum d'empêcher la mise en marche du processus dans le cas où des erreurs sont commises, et d'indiquer ce qu'il faut corriger (simple affichage de message d'erreur).



Encore une fois, notre objectif est aussi de simplifier le processus.

Concrètement, il s'agit simplement de placer des instructions de branchements conditionnels⁵ et de relever des exceptions.

Générer le modèle Comme il a été précisé précédemment dans l'architecture, c'est une fonction à part entière (Generator.jar) qui prend tous les paramètres en arguments et renvoie le fichier XML.

En réalité, la fonction prend aussi comme ressources un fichier XML qui lui sert de modèle, et le logiciel se contente de réécrire seulement des zones prédéfinies où se trouvent les paramètres à définir. Ainsi, on modifie le modèle de base seulement dans ces zones, tant que les paramètres déclarés restent les mêmes, car c'est l'initialisation uniquement qui change par rapport au modèle de base. Bien évidemment, ce n'est pas le modèle de base pris en ressources qui est réellement modifié, on crée un nouveau fichier.

D'un point de vue programmation, on commence à copier le modèle de base dans le nouveau fichier, puis une fois arrivé dans la zone d'initialisation des paramètres, on écrit une chaîne de caractères identique à cette zone, en remplaçant les valeurs des paramètres par ceux rentrés en argument de la fonction. Et pour finir on copie le reste du modèle, une fois sortie de la zone.

Pour éviter des problèmes de complexité en espace, une fois la vérification terminée, le nouveau fichier XML créé sera écrasé par les processus suivants, dans le cas d'un balayage.

5. On parle d'ici tout simplement d'une longue séquence de "if(contrainte_non_respecté){afficher l'erreur}".

Paramétrer les intervalles Afin de définir un ensemble d'états initiaux, l'utilisateur va pouvoir faire varier les paramètres qui l'intéressent.

Pour récupérer l'intervalle, on rajoute d'autres paramètres secondaires : la limite et le pas. Pour un intervalle $[n, m]$ avec un pas de x , il faut dans l'interface une case pour la valeur de n , x et m

init_V=	<input type="text" value="2000"/>	<input type="text" value="50"/>	<input type="text" value="3000"/>
init_A=	<input type="text" value="-500"/>	<input type="text" value="1"/>	<input type="text" value="0"/>

Sur l'image ci-dessus, `init_V` est défini par l'intervalle $[2000, 3000]$ avec un pas de 50. Soit `init_V` défini par l'ensemble `init_V = {2000, 2050, 2100, ..., 3000}`.

Il y aura un processus pour chaque pas de chaque vitesse et accélération initiales de chaque voiture, de façon exhaustive, Dès lors, la complexité en temps de la fonction est exponentielle, ce qui peut rapidement créer un nombre non-négligeable d'états initiaux pour lesquels chaque chemin existant sera vérifié. Sachant qu'un seul processus prend lui aussi un temps non-négligeable, l'utilisateur peut être amené à alléger son nombre de tests à effectuer, c'est pourquoi il est demandé de définir le pas.

Pour l'instant, les intervalles sont disponibles pour les deux paramètres les plus importants à tester, la vitesse et l'accélération initiales de la voiture.

Traitement des résultats Le traitement des résultats consiste en l'écriture des résultats donnés par `Verifyta`, concaténés sous forme de liste dans un fichier texte (Voir Annexe C).

Il n'y a encore aucun lien entre les résultats et les états initiaux, l'utilisateur ne peut pas encore savoir directement quel résultat correspond à quel état initial (à part en comparant manuellement). C'est donc peu exploitable pour le moment.

Lancer les vérifications En suivant l'architecture, l'utilisateur lance un script exécutant l'interface, puis l'ensemble des processus en boucle.

Rappelons cependant que c'est l'interface qui génère l'ensemble des processus. Concrètement, l'interface génère un script temporaire qui va lancer tout les processus d'affilés. Ce script temporaire sera exécuté dans le script principal, juste après que l'interface ai été fermée par l'utilisateur.

Pour résumer :

- 1) L'utilisateur lance le script
- 2) L'interface se lance
- 3) L'utilisateur rempli les paramètres
- 4) L'interface crée un autre script temporaire
- 5) Le script lance le script temporaire
- 6) Le script temporaire écrit les résultats dans le fichier texte

Dans la figure de l'Annexe C, il y a trois fichiers d'ouverts,

Le script principal : il lance l'interface créant le script secondaire, puis lance celui-ci.

Le script secondaire : il contient tout les processus de vérification. À la fin de chacun de ces processus, le résultat est écrit à la suite des autres.

Les résultats : contient la liste des résultats.

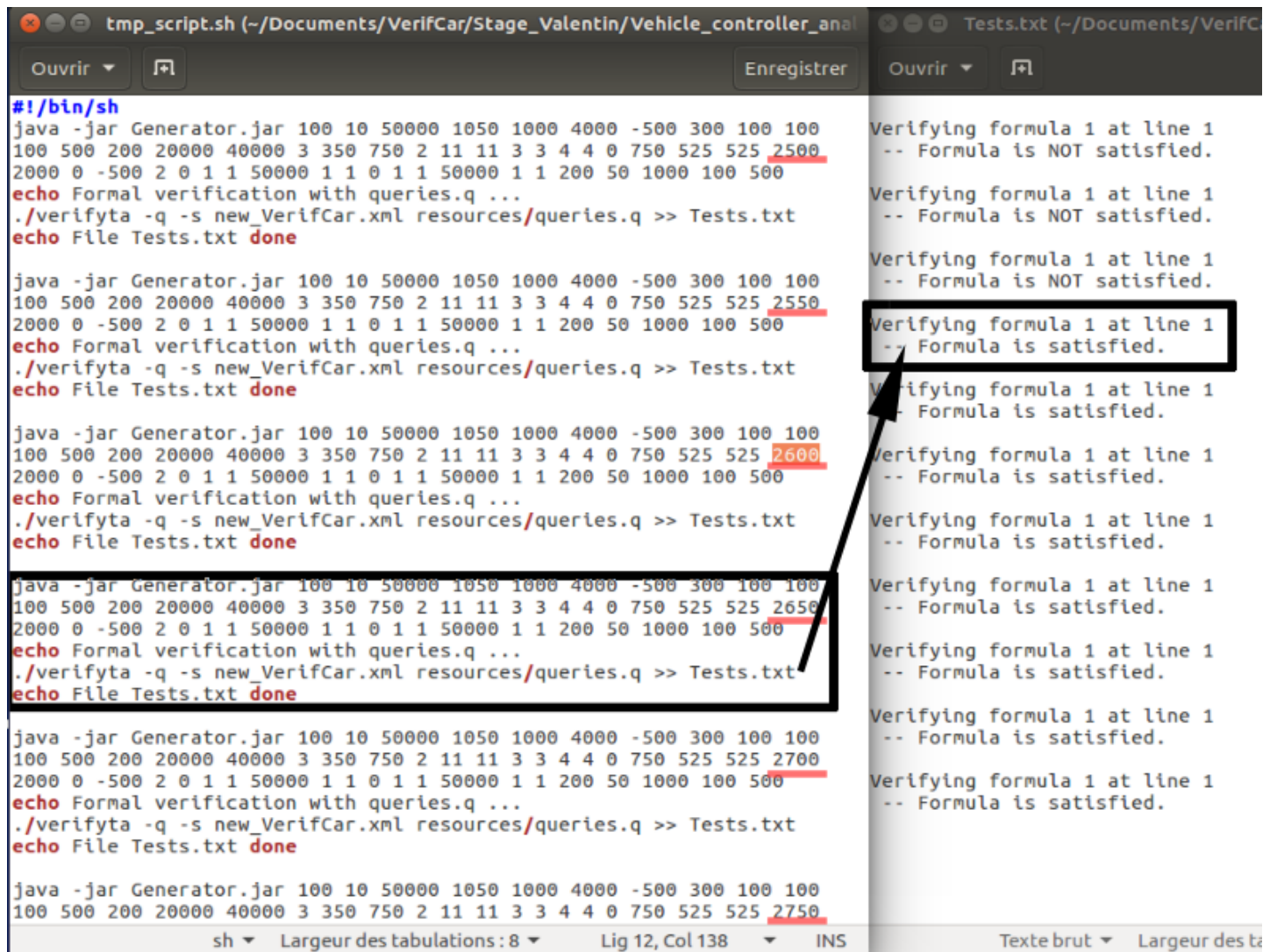
3.5 Exemple

Imaginons la situation suivante où deux voitures se suivent :

- La *voiture1* est très proche de la *voiture2* (7,5m d'écart)
- La *voiture2* freine brusquement (accélération de $-5m/s^2$)

On utilise une requête existante permettant de vérifier l'existence de collision. On a défini les paramètres, tel que la vitesse initiale de la *voiture2* est de 20m/s et la vitesse initiale de la *voiture1* est un intervalle 25m/s à 30m/s avec un pas de 0.5m/s.

En comparant l'ordre des processus dans le script secondaire et l'ordre des résultats dans le fichier texte, on interprétera les résultats manuellement.



```
#!/bin/sh
java -jar Generator.jar 100 10 50000 1050 1000 4000 -500 300 100 100
100 500 200 20000 40000 3 350 750 2 11 11 3 3 4 4 0 750 525 525 2500
2000 0 -500 2 0 1 1 50000 1 1 0 1 1 50000 1 1 200 50 1000 100 500
echo Formal verification with queries.q ...
./verifyta -q -s new_VerifCar.xml resources/queries.q >> Tests.txt
echo File Tests.txt done

java -jar Generator.jar 100 10 50000 1050 1000 4000 -500 300 100 100
100 500 200 20000 40000 3 350 750 2 11 11 3 3 4 4 0 750 525 525 2550
2000 0 -500 2 0 1 1 50000 1 1 0 1 1 50000 1 1 200 50 1000 100 500
echo Formal verification with queries.q ...
./verifyta -q -s new_VerifCar.xml resources/queries.q >> Tests.txt
echo File Tests.txt done

java -jar Generator.jar 100 10 50000 1050 1000 4000 -500 300 100 100
100 500 200 20000 40000 3 350 750 2 11 11 3 3 4 4 0 750 525 525 2600
2000 0 -500 2 0 1 1 50000 1 1 0 1 1 50000 1 1 200 50 1000 100 500
echo Formal verification with queries.q ...
./verifyta -q -s new_VerifCar.xml resources/queries.q >> Tests.txt
echo File Tests.txt done

java -jar Generator.jar 100 10 50000 1050 1000 4000 -500 300 100 100
100 500 200 20000 40000 3 350 750 2 11 11 3 3 4 4 0 750 525 525 2650
2000 0 -500 2 0 1 1 50000 1 1 0 1 1 50000 1 1 200 50 1000 100 500
echo Formal verification with queries.q ...
./verifyta -q -s new_VerifCar.xml resources/queries.q >> Tests.txt
echo File Tests.txt done

java -jar Generator.jar 100 10 50000 1050 1000 4000 -500 300 100 100
100 500 200 20000 40000 3 350 750 2 11 11 3 3 4 4 0 750 525 525 2700
2000 0 -500 2 0 1 1 50000 1 1 0 1 1 50000 1 1 200 50 1000 100 500
echo Formal verification with queries.q ...
./verifyta -q -s new_VerifCar.xml resources/queries.q >> Tests.txt
echo File Tests.txt done

java -jar Generator.jar 100 10 50000 1050 1000 4000 -500 300 100 100
100 500 200 20000 40000 3 350 750 2 11 11 3 3 4 4 0 750 525 525 2750
2000 0 -500 2 0 1 1 50000 1 1 0 1 1 50000 1 1 200 50 1000 100 500
echo Formal verification with queries.q ...
./verifyta -q -s new_VerifCar.xml resources/queries.q >> Tests.txt
echo File Tests.txt done
```

```
Verifying formula 1 at line 1
-- Formula is NOT satisfied.

Verifying formula 1 at line 1
-- Formula is NOT satisfied.

Verifying formula 1 at line 1
-- Formula is NOT satisfied.

Verifying formula 1 at line 1
-- Formula is satisfied.

Verifying formula 1 at line 1
-- Formula is satisfied.

Verifying formula 1 at line 1
-- Formula is satisfied.

Verifying formula 1 at line 1
-- Formula is satisfied.

Verifying formula 1 at line 1
-- Formula is satisfied.

Verifying formula 1 at line 1
-- Formula is satisfied.

Verifying formula 1 at line 1
-- Formula is satisfied.
```

On voit ci-dessus, dans le fichier texte contenant les résultats (à droite), que pour le quatrième état initial, il existe une collision. On notera que l'argument de la vitesse variable est souligné en rouge dans chaque processus (à gauche). Le quatrième résultat correspond au quatrième processus dans le script temporaire, soit celui où la vitesse de la *voiture1* est de 26,5m/s.

On peut donc conclure qu'il y a un risque seulement si la *voiture1* dépasse les 26m/s (le troisième processus).

4 Limites et extensions

On pourra noter plusieurs petites limites à étendre :

- L’aspect de l’interface graphique est très simpliste.
- Toutes les erreurs ne sont probablement pas gérées.
- Il manque la possibilité de sauvegarder à tout moment et d’écraser les anciennes sauvegardes.
- Bien que les deux paramètres les plus intéressants à faire varier peuvent être défini en intervalles, il faudrait pouvoir faire varier les autres.
- Donner la possibilité à l’utilisateur de cacher les intervalles des paramètres qu’il ne souhaite pas faire varier.
- Il est possible qu’il reste quelques bugs.

Remarque : Le stage étant relativement court, je me suis concentré sur la progression plutôt que sur l’amélioration du projet, afin d’en faire un maximum. Par exemple, lorsque j’ai mis en place le système de sauvegarde, je suis rapidement passé à la suite au lieu de m’attarder à faire un autre bouton pour sauvegarde à tout moment. C’est aussi pour cette raison qu’il n’y a seulement deux paramètres que l’utilisateur peut faire varier. Une fois qu’une fonction est implémentée une ou deux fois, il était bien plus intéressant d’avancer et de faire autre chose de nouveau plutôt que de répéter ce qui est déjà fait (et que l’on peut faire plus tard). Ainsi, mon logiciel est certes “inachevé”, mais l’empressement dans la progression du projet m’a permis de le rendre fonctionnelle avant la fin du stage.

Il y a des fonctionnalités plus importantes à implémenter pour faire progresser le projet :

Définir les requêtes Pour l’instant la modification du fichier des requêtes est manuelle et dans un langage propre à la modélisation du système, l’utilisateur doit donc les connaître. Il pourrait être intéressant de faire tout un panel de boutons qui écriraient les requêtes automatiquement, par exemple.

Ou encore, la possibilité de faire varier les requêtes, au même titre que les paramètres, pourrait être mis en place. En revanche, cela impliquerait de revoir complètement l’architecture actuellement, car le fichier “queries.q” ne doit pas changer entre deux vérifications.

L’exploitation des résultats Il manque au logiciel une retranscription des résultats qui soit clair pour l’utilisateur. Cette fonctionnalité est capitale pour faire gagner du temps à l’utilisateur dans le cas d’une courte série de tests et pour qu’il soit capable d’exploiter les résultats pour une série plus longue. Il faudrait, au moins, pouvoir identifier quelle vérification appartient à quel ensemble de paramètres.

De plus, il semblerait suffisamment aisé d’écrire les paramètres qui varient, juste avant le résultat de la requête. C’est donc la prochaine étape dans la progression du projet.

5 Ce que le stage m'a apporté

Pour commencer, c'est ma première expérience professionnelle. C'est aussi la première fois que je mets mes compétences informatiques au service de quelqu'un d'autre. Je suis plutôt satisfait, j'ai eu l'occasion de montrer ce dont j'étais capable et surtout d'apprendre ce dont je n'étais pas capable.

Au départ, j'ai commencé par ce que j'avais appris, jusqu'à rencontrer un problème, me renseigner pour le résoudre, puis rencontrer un autre problème, etc. C'est la méthode d'acquisition d'expérience la plus efficace que je connaisse.

J'ai donc dans un premier temps acquis une confiance que je n'avais absolument pas envers mes propres compétences informatiques. Et dans un second temps, le savoir que j'ai acquis durant mes 3 années de licences s'est transformé en savoir-faire.

Linux Le système d'exploitation Linux est quelque chose que j'ai vu et revu plusieurs fois au cours de ma scolarité, cependant, je n'avais jamais réellement travaillé avec. Avant le début de mon stage, j'ai installé ubuntu sur mon pc et j'ai travaillé exclusivement avec, durant mes 7 semaines de stage. J'ai donc pu apprendre énormément sur cet OS et voir par moi-même les avantages et inconvénients.

Scripts Shell Là encore je n'avais jamais écrit de script shell. Je suis parti de zéro, et même si je n'ai pas eu besoin d'aller bien loin pour arriver à mes fins, il est évident que cela va de pair avec l'apprentissage de Linux.

Java Le langage de programmation Java est probablement celui que l'on a le plus étudié, et j'ai été ravi de le mettre en pratique et d'en apprendre encore plus.

Durant mon stage j'ai aussi appris JavaFx, la technologie d'implémentation d'interface graphique la plus sollicitée du moment. Même s'il est évident que je n'aurais ni le temps ni l'envie de remplacer l'interface graphique du logiciel que j'ai conçu pendant mon stage, je suis tout autant certain que je vais m'en servir pour le prochain projet que je programmerai en Java.

Premier pas dans la recherche Le sujet de mon stage était au départ quelque chose d'assez concret, puis au fur et à mesure que j'avancais, j'ai dû mettre en application ce j'avais vu dans des cours plus théorique, notamment sur la complexité des algorithmes. En travaillant avec des doctorants et des enseignants chercheurs, j'ai pu apercevoir ce qu'était la recherche en laboratoire.

FIGURE 4 – Utilisation de Verifitya

```
// Scale
const int scale = 100;

// System parameters (define size of the data structure)
const int S := 10; // sample period, in 1/scale seconds
const int L := 50000; // length of the road segment, in 1/scale
meters
const int R := 1050; // width of the road segment, in 1/scale meters
const int V_min := 1000; // min value of longitudinal speed, in
1/scale meters per second
const int V_max := 4000; // max value of longitudinal speed, in
1/scale meters per second
const int A_min := -500; // min value of longitudinal
acceleration, in 1/scale meters per second squared
const int A_max := 300; // max value of longitudinal acceleration, in
1/scale meters per second squared
const int GranA := 100; // granularity of the acceleration expressed
in 1/scale meters per second squared
const int W := 100; // maximal absolute value of the lateral speed
expressed in 1/scale meters per second
const int NormX := 100; // maximal loss of precision during a second
in 1/scale meters (&gt;= (GranA*S/scale)/2)

// Environment constraints
const int length_vehicle := 500; // length of a vehicle in 1/scale
meters
const int width_vehicle := 200; // width of a vehicle in 1/scale
meters

query_collision()
```

```
valentin@Pc-valentin: /tmp
valentingPc-valentin: /tmp$ ./verifitya code_uppaal.xml queries.q
Options for the verification:
Generating no trace
Search order is breadth first
Using conservative space optimisation
Seed is 1497394709
State space representation uses minimal constraint systems

Verifying formula 1 at line 1
-- Formula is NOT satisfied.
valentingPc-valentin: /tmp$
```

- Le cadre jaune représente le fichier XML code_uppaal.xml (on ne voit qu'une partie du code ici)
- Le cadre vert représente la requête présente dans le fichier queries.q
- Le cadre rouge représente la commande (l'option verifitya proposée par UPPAAL), elle prend en argument le code et les requêtes
- Le cadre bleu représente le résultat de la vérification

FIGURE 5 – Questionnaires de paramètres

LOAD SETTINGS

Parameters

scale=100

S=10

L=50000

R=1050

V_min=1000

GranA=100

W=100

NormX=100

nb_lane=4

end_junction=40000

begin_junction=20000

marking=Define

nb_car=3

safety_length=200

safety_width=50

traf_length=1000

delay_step=100

max_delay=500

Cars settings

DEVELOPPER OPTION

Edit code

Edit queries

Finish

Marking

posY1=330

posY2=660

posY3=1000

Save

Navigation points of Car 0

Navigation points of Car 1

Navigation points of Car 2

posX

lane_inf

lane_sup

P1 10000

P2 50000

1

1

1

2

Save

Car 1

Car 2

Car 3

length_vehicle=500

width_vehicle=200

v_max=4000

a_min=-500

a_max=300

ctr_freq=11

max_com=4

min_com=3

init_posX=600

init_posY=525

init_V=2000

init_A=-100

navigation_points=2

navigation_list=Define

Save

length_vehicle=500

width_vehicle=200

v_max=4000

a_min=-500

a_max=300

ctr_freq=11

max_com=4

min_com=3

init_posX=0

init_posY=525

init_V=2000

init_A=0

navigation_points=2

navigation_list=Define

Save

length_vehicle=500

width_vehicle=200

v_max=4000

a_min=-500

a_max=300

ctr_freq=11

max_com=4

min_com=3

init_posX=0

init_posY=0

init_V=0

init_A=0

navigation_points=2

navigation_list=Define

Save

Paramètres principaux

Véhicules

Tableaux

FIGURE 6 – Exemple d'exécution d'un balayage

