Projet de quatrième année département Informatique 2019-2020

Rapport de conception

ABMVis3000

BOURNAC Guénaël
CELLIER Antoine
MEVOLLON Matéo
POLLET Christophe
ZHENG Xiaoang

Tutrice : ROZE Laurence
Tuteur : MARTIN Gregory

12 mai 2020

Table des matières

I. Compte rendu des phases de test

Pour s'assurer que le code de l'application est stable et de bonne qualité, différentes procédures de test ont été utilisées. D'autre part, la mise en pratique de l'intégration continue a permis de s'assurer que le développement n'entraînait pas de régressions.

I.1. Test du modèle

I.1.a. Intégration continue et vérification Checkstyle

Nous avons mis en place un système d'intégration continue via GitLab pour permettre de lancer notre suite de tests à chaque fois que des *commits* sont poussés vers le serveur, ce qui est appelé un pipeline. Nous avons aussi paramétré GitLab pour interdire les *merge request* si le dernier Pipeline de la branche n'a pas réussi à passer.

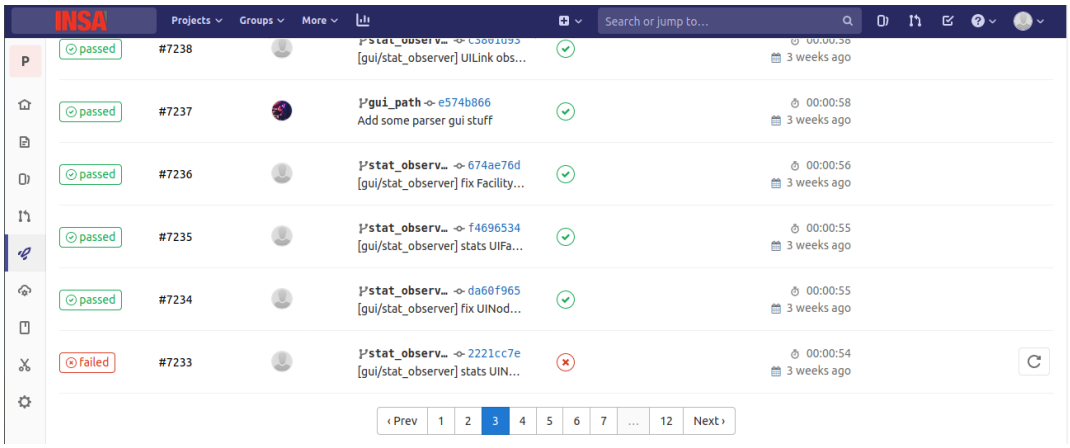


FIGURE 1 – Exemple de l'interface des pipelines GitLab

L'image en Figure 1 montre l'interface de GitLab pour les pipelines avec un qui n'est pas passé. L'avantage de cette méthode est que la personne qui a poussé la branche reçoit un mail pour l'en notifier. Notre politique est de tester au maximum ce qui est testable sauf les getter et setter évidents (car auto-générés avec IntellIJ). Cela nous a amené à écrire plus de 200 tests unitaires.

Pour des raisons de qualité de code, nous avons aussi paramétré un outil nommé checkStyle pour vérifier que la documentation soit bien écrite. Seuls les getter et setter ne sont pas vérifiés par cet outil.

I.1.b. Modèle statique

Le modèle statique est certainement la partie de l'architecture qui contient le moins de tests, ce qui s'explique par le fait que tous les éléments qui y sont contenus sont complètement fixes (les éléments évolutifs étant stockés dans le modèle dynamique). Ainsi la plupart des classes du modèle statique sont constituées exclusivement de getters, setters et constructeurs, qui ne sont donc pas très pertinents pour

les tests. Le résultat de tous les tests unitaires pour le modèle statique est disponible en Figure 2 avec seulement 12 tests sur les 200 au total de notre application.

```
[INFO] Running AbmModel.PersonTest
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.005 s - in AbmModel.PersonTest
[INFO] Running AbmModel.TypeCollectorTest
[INFO] Tests run: 5, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0 s - in AbmModel.TypeCollectorTest
[INFO] Running AbmModel.TypeTest
[INFO] Tests run: 3, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.001 s - in AbmModel.TypeTest
[INFO] Running AbmModel.VehicleTest
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0 s - in AbmModel.VehicleTest
```

FIGURE 2 – Affichage console de l'exécution des tests pour le modèle statique

L'intégration de ces quelques tests unitaires dans le modèle statique a cependant été cruciale pour deux éléments :

- D'abord pour les véhicules et les personnes qui possèdent chacune deux méthodes assez critiques leur permettant de retrouver l'événement le plus proche d'un timestamp donnée dans le futur ou le passé. Ces méthodes sont utilisées par le parseur pour calculer les coordonnées de certains événements. En effet pour connaître les coordonnées d'un événements du type "une personne qui rentre dans un véhicule", il faut retrouver l'événement "véhicule qui rentre dans un lien" le plus proche temporellement pour recopier ses coordonnées. Le calcul des coordonnées pour les événements étant crucial pour assurer le bon fonctionnement de la visualisation, on comprend rapidement à quel point il était indispensable de bien tester ces méthodes. D'autant plus que celles-ci sont directement disponibles dans le modèle abstrait de base dont hérite tous les autres modèles concrets.
- L'autre élément crucial à tester était la gestion des types. En effet quand le parseur appelle la méthode "addTypeEntry" d'un objet du modèle, ce dernier ajoute une entrée parmi ses propres types, mais aussi dans le TypeCollector global utilisé par toute l'application notamment dans le cadre des filtres et des analyses. Là encore, il était nécessaire de s'assurer du bon fonctionnement de cette gestion dès le départ.

I.1.c. Modèle dynamique

Le modèle dynamique est le modèle qui permet l'affichage des objets et le calcul des positions en temps réel. L'objectif des tests sur cette partie est de préparer correctement les données pour l'affichage. Notamment la partie gestion des événements qui est testée jusqu'à 100% de couverture des lignes. En effet les événements sont une partie critique de l'application. Les événements sont toutes les choses qui peuvent arriver pendant la simulation, par exemple rentrer sur un lien, rentrer dans une voiture. Plus d'une dizaine d'évènements peuvent s'exécuter. De plus, il y a toute la partie calcul des positions des véhicules et des personnes à partir des événements qui est complètement testée pour nous assurer de son bon fonctionnement. La classe de translation des coordonnées a été développée en TDD (Test Driven Development) donc avec l'écriture des tests avant le développement de la fonctionnalité. Cette classe a été réalisé ainsi car les coordonnées de MatSim sont données dans un repère cartésien avec l'axe des ordonnées vers le haut. Dans JavaFx il est orienté vers

le bas, Il fallait donc les translater pour éviter l'effet miroir. Il était difficile de faire la modification. Nous avons donc développé le test pour permettre d'en avoir une meilleur intuition puis la fonction.

I.2. Parseur

Le parseur aussi a subi un bon nombre de tests unitaires pour s'assurer de la bonne lecture des fichiers d'entrée, la bonne instanciation de tous les objets du modèle en accord avec les balises du fichier xml, le bon remplissage de tous les attributs mais aussi pour vérifier la bonne exécution de toutes les opérations post-traduction assurées par le "ScenarioProcessor". Parmi elles on retrouve le remplissage des dictionnaires reliant les id aux instances, les associations entre les objets et surtout le calcul de la position de tous les événements mentionnés précédemment. On peut voir le résultats des 48 tests unitaires consacrés au parseur en Figure 3.

Un parseur mal testé aurait conduit à perdre du temps pour le développement, car les erreurs auraient été beaucoup plus compliquées à détecter et à corriger si ces dernières avaient été repérées depuis l'interface utilisateur. En effet, le parseur étant le point d'entrée de l'application, un défaut de ce dernier propagerait une erreur dans toutes les autres parties de l'architecture. En plus des tests unitaires, il a aussi fallu créer un certains nombre de fichiers d'entrées différents afin de tester quelques cas limites et s'assurer que ces derniers étaient correctement gérés en toutes circonstances. Toutes ces précautions se sont avérées être utiles, car certaines erreurs et défauts ont pu être repérés dès le début, et ont donc été rapidement corrigés.

```
[INFO] Tests run: 22, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 9.987 s - in Parsers.TestMSMandatoryParser
[INFO] Running Parsers.TestMSOptionalParser
[INFO] Tests run: 5, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 1.902 s - in Parsers.TestMSOptionalParser
[INFO] Running Parsers.TestMSScenarioProcessor
[INFO] Tests run: 8, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 3.038 s - in Parsers.TestMSScenarioProcessor
[INFO] Running Parsers.TestParserConfigurer
[INFO] Tests run: 6, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 2.36 s - in Parsers.TestParserConfigurer
[INFO] Running Parsers.TestTypesParser
[INFO] Tests run: 7, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 2.675 s - in Parsers.TestTypesParser
```

FIGURE 3 – Affichage console de l'exécution des tests pour le parseur

I.3. Test de l'interface

Pour tester les aspects visuels de l'application nous avons généré la simulation par défaut de MatSim. MatSim est un logiciel qui permet de simuler les mobilité urbaine. Ce logiciel utilise des fichiers de description des personnes et du réseaux routier pour créer différents fichiers notamment un pour les événements. L'ensemble de tous ces fichiers est une simulation. Nous avons ensuite visualisé cette simulation avec VIA pour vérifier la chronologie des événements. Ce qui nous as permis de vérifier que nous obtenions la même avec notre logiciel. VIA est un logiciel qui permet la visualisation d'une simulation MatSim à conditions qu'elle soit suffisamment petite sinon il faut une licence payante. Pour vérifier la bonne implémentation de chaque fonctionnalité nous avons utilisé cette simulation et avons observé les effets de nos modifications de l'interface sur ces données par exemple en ajoutant des filtres qui changent la taille de certains éléments.

I.4. Ville de Rennes

I.4.a. Comment créer les fichiers

Les fichiers de la simulation de la ville de Rennes sont constitués de deux parties principales : le réseau et la population.

Pour la population, nous sommes partis d'un fichier d'une base de donnée publique nationale qui maille le territoire en carré d'un kilomètre de côté. Dans ces carrés sont indiqués divers paramètres tel que le nombre d'habitant, leurs âges... Afin de reconstituer plus précisément leurs habitudes nous avons aussi constitué une base de points d'intérêts (école, lycée, zone artisanale, parking urbain...) à partir de divers sources telles que des bases de données publiques où des services d'urbanismes. Les diverses tranches d'âge de population ont ensuite été associées à des points d'intérêts pertinents (les moins de 10 ans avec les écoles par exemple). Nous avons considéré que 80% de la population se déplaçait à des horaires standards (le matin à 9h pour les écoles par exemple) et dans le point d'intérêt le plus proche de chez eux. Le reste de la population ayant un comportement moins standard (travailleur à mi-temps, écolier allant dans une école loin du domicile...)

Pour le réseau, nous sommes partis d'une subdivision de la base de données d'Open Street Map englobant la ville de Rennes ainsi que les villes proches pouvant avoir une influence sur les déplacements (Cesson-Sévigné, Saint-Grégoire, Bruz...).

I.4.b. Bug sur les identifiants

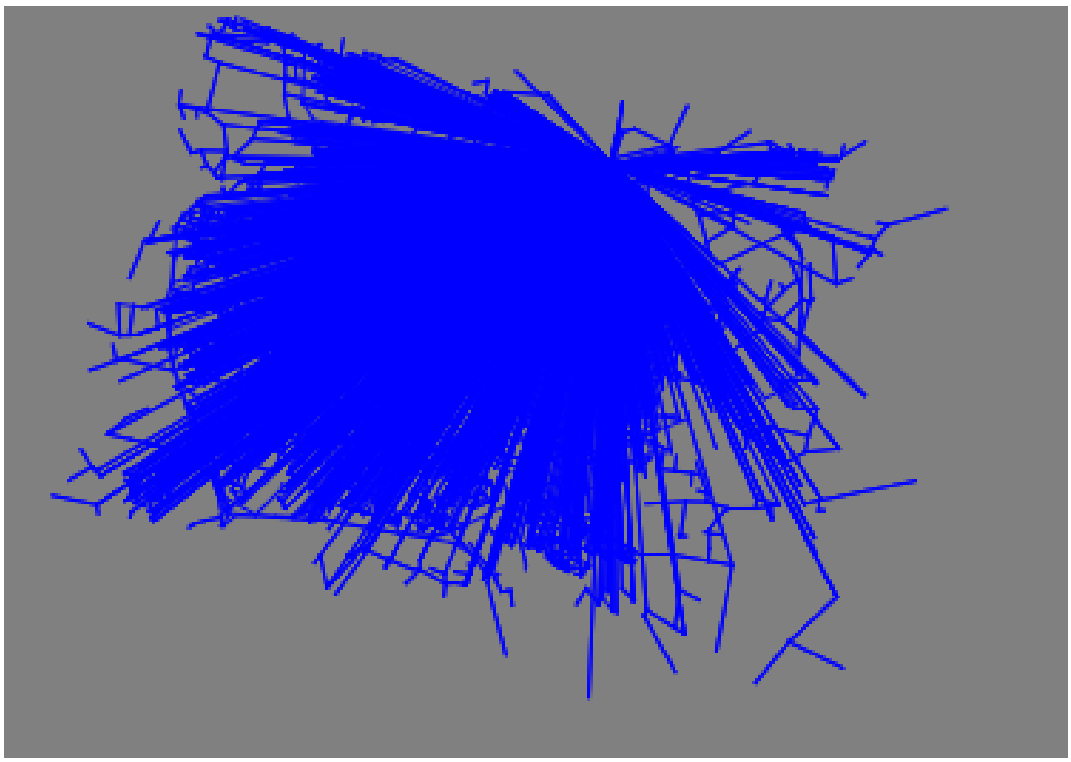


FIGURE 4 – Affichage incorrecte du réseau

La Figure 4 représente l'affichage du réseau de Rennes créé avec une première méthode. Nous pouvons voir qu'il ne ressemble pas à la ville de Rennes. Il semble y avoir un point qui concentre tous les liens et une partie du réseau semble se dessiner correctement. Nous savions que le problème ne venais pas du fichier car MatSim l'acceptait d'où la détection d'un problème. En ajoutant des lignes pour déboguer l'application nous avons remarqué que le noeud qui rassemble tous les traits en rassemblait plus de 19000 et qu'il avait un ID = -1. Cette valeur est la valeur par défaut des ID avant que la valeur lue dans le fichier lui soit ajoutée. Nous avons donc cherché plus particulièrement dans cette partie notamment les Exceptions qui y sont lancées mais que l'on cachait à l'utilisateur. Comme attendue il y avait un numberFormatException ce qui signifie que le programme n'arrivait pas à transformer en entier les ID. La génération ne permettait pas d'avoir des entiers mais des Long. Nous avons donc modifié l'application pour que les identifiants soient des long. Le nouvel affichage du réseau après la résolution du bug se trouve en Figure 5.

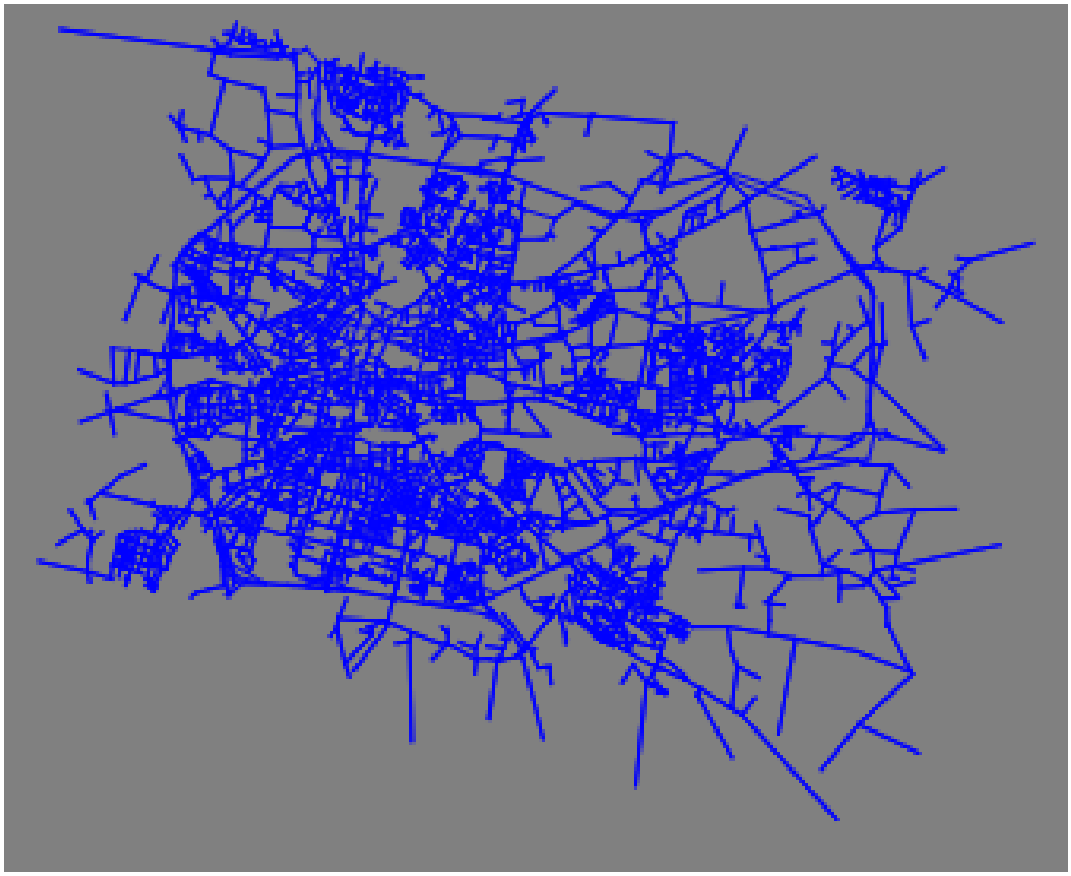


FIGURE 5 – Affichage correcte du réseau

I.4.c. Problème de taille mémoire

Avec les fichiers qui devaient servir pour faire les démonstrations nous avons découvert un nouveau bug. Il s'agit de l'explosion du tas lors de la lecture des fichiers XML. En effet la bibliothèque xerces charge tout le fichier en mémoire avant de créer l'objet sous forme d'arbre qui permet le parcours du fichier XML. Ce n'est pas très

grave pour des petits fichiers mais pas pour des fichiers de 650Mo. Le parser à était testé avec ces fichiers sur une machine de AWS (Amazon Web Services) qui a plus de ressource qu'une machine personnelle et tout a bien fonctionné. Pour régler ce problème la meilleur solution serait d'utiliser une bibliothèque qui ne charge pas tous le fichier dans le tas, certaines ont été trouvée en java mais nous n'avions pas le temps de changer toute l'implémentation du parseur. Ce qui pose des questions de rapidité aussi car le fichier sera chargé plusieurs fois en mémoire. Une seconde solution a été implémentée il s'agit d'un script qui découpe le fichier en plusieurs tout en respectant la forme du fichier XML. L'implémentation du parseur permettait déjà de prendre plusieurs fichiers pour une même donnée par exemple pour avoir les liens. Il a donc était possible de visualiser le réseau final de la ville de Rennes avec cette technique. Mais les données à dessiner sont tellement grandes qu'il y a trop de temps entre chaque image affichée. Nous pouvons visualiser en Figure 6 cette affichage.

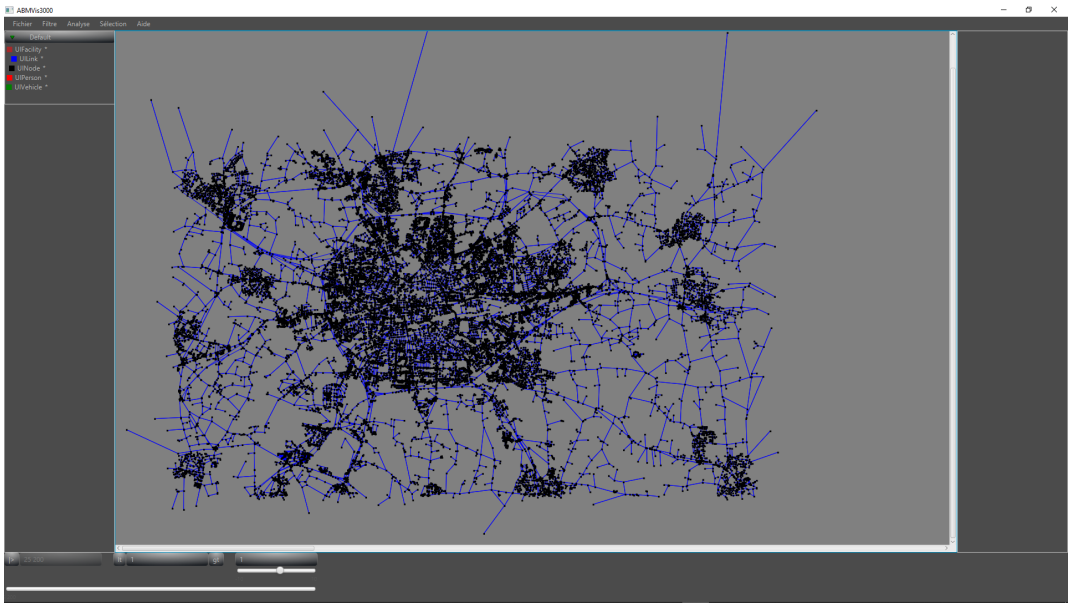


FIGURE 6 – Affichage du réseau de le ville de rennes

II. Documentation technique

Cette partie a pour but d'étudier les changements, corrections et rectifications qui ont été apportées par rapport aux précédents rapports de spécification et de conception. Nous allons d'abord voir les changements liés au rapport de spécification, décrivant les fonctionnalités, puis au rapport de conception décrivant l'implémentation. C'est aussi dans la partie conception que seront décrit tous les aspects techniques.

II.1. Spécifications

Le rapport de spécification avait pour but de présenter et de détailler les fonctionnalités que nous envisagions pour notre application. Ils pouvait s'agir de fonctionnalités demandées et imposées par notre cahier des charges ou encore de fonctionnalités additionnelles et facultatives qui pouvaient éventuellement être implémentées si le temps le permettait. Penser à toutes ces fonctionnalités envisageables dès le départ nous a permis de mettre en place une implémentation et une architecture facilitant la reprise du code pour que ces dernières puissent être développé à l'avenir.

II.1.a. GUI

En ce qui concerne la partie GUI, le rapport de spécification présentait essentiellement les fonctionnalités envisagées ainsi que des maquettes illustrant le visuel attendu pour notre interface utilisateur. Pour ce qui est des fonctionnalités, le fait de les avoir implémentées ou pas sera décrit dans la partie du rapport dédiée aux avancements III.. Ici nous allons surtout nous concentrer sur les différences entre les fonctionnalités implémentées et comment elles ont été pensées à la base. Et comme ces différences sont essentiellement graphiques, cette partie sera surtout constituée de photos de comparaisons entre les maquettes et notre application actuelle.

Comme on peut le voir en Figure 7, on retrouve bien les 5 parties prévues dans l'interface utilisateur, placées comme annoncées dans le rapport de spécification : la zone de visualisation (au centre), la timeline (en bas), les détails des éléments suivis (à droite), les filtres (à gauche) et enfin la barre de menu (en haut). Les changements sont essentiellement graphiques et c'est que nous allons voir en comparant les photos pour chaque partie.

Zone de visualisation Comme on peut le voir en Figure 8 les changements sont essentiellement graphiques. Les différences notables que l'on peut relever cependant sont les suivantes :

- L'horloge, l'icône de pleine écran et l'échelle n'apparaissent pas dans les coins de notre visualisation car ces derniers n'ont tout simplement pas été implémentés (voir Partie III.1.)
- Dans la maquette, il n'était pas prévu d'afficher les personnes en déplacement mais uniquement leurs véhicules (flèches vertes). Comme cela nous paraissait pertinent et que notre implémentation nous le permettait, nous avons choisi de représenter aussi les personnes en déplacement par des carrés rouges qui sont à l'intérieur de leur véhicules représentés par des carrés verts plus large.

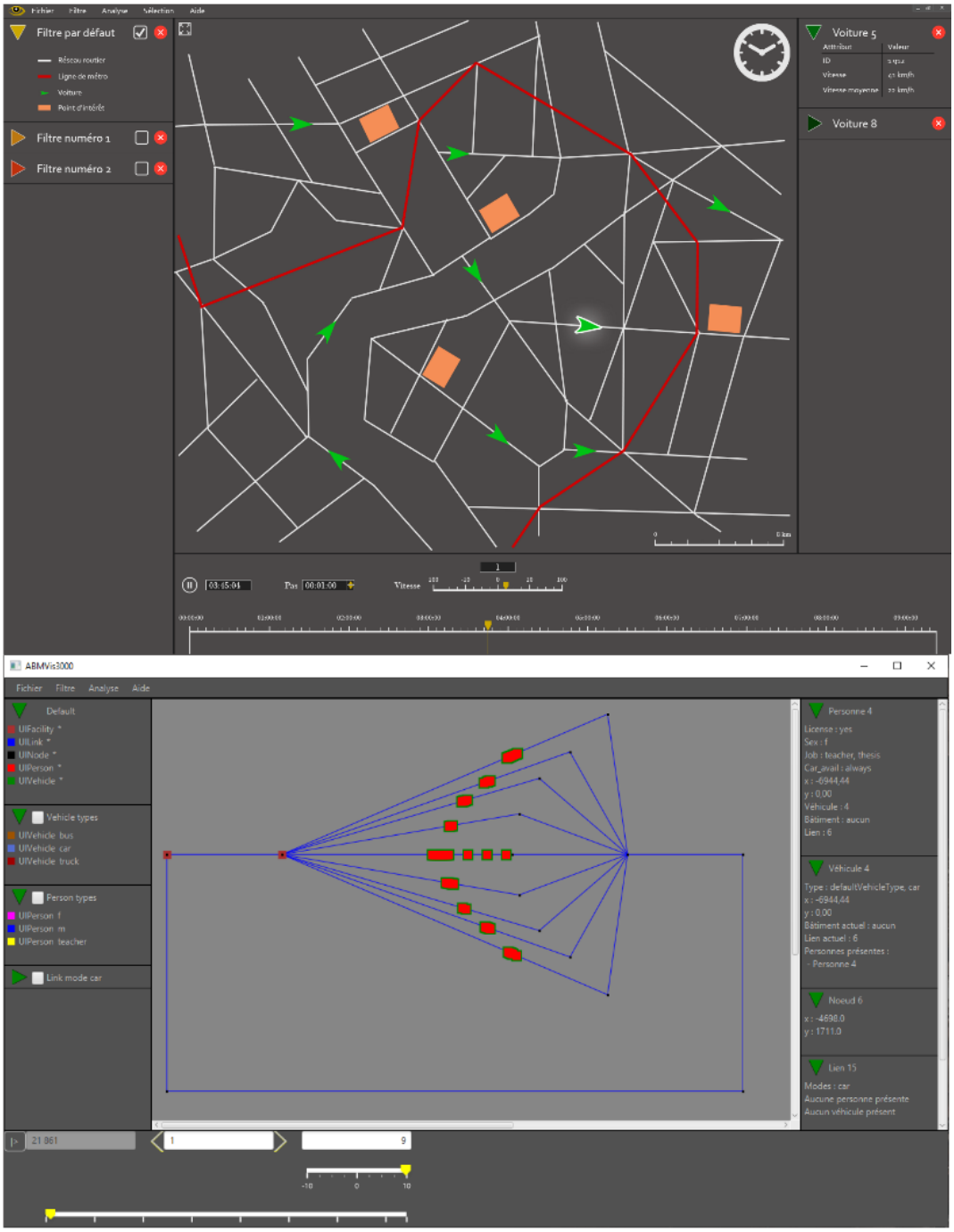


FIGURE 7 – Comparaison entre la maquette (en haut) et l'application (en bas) pour la vue principale de l'interface utilisateur

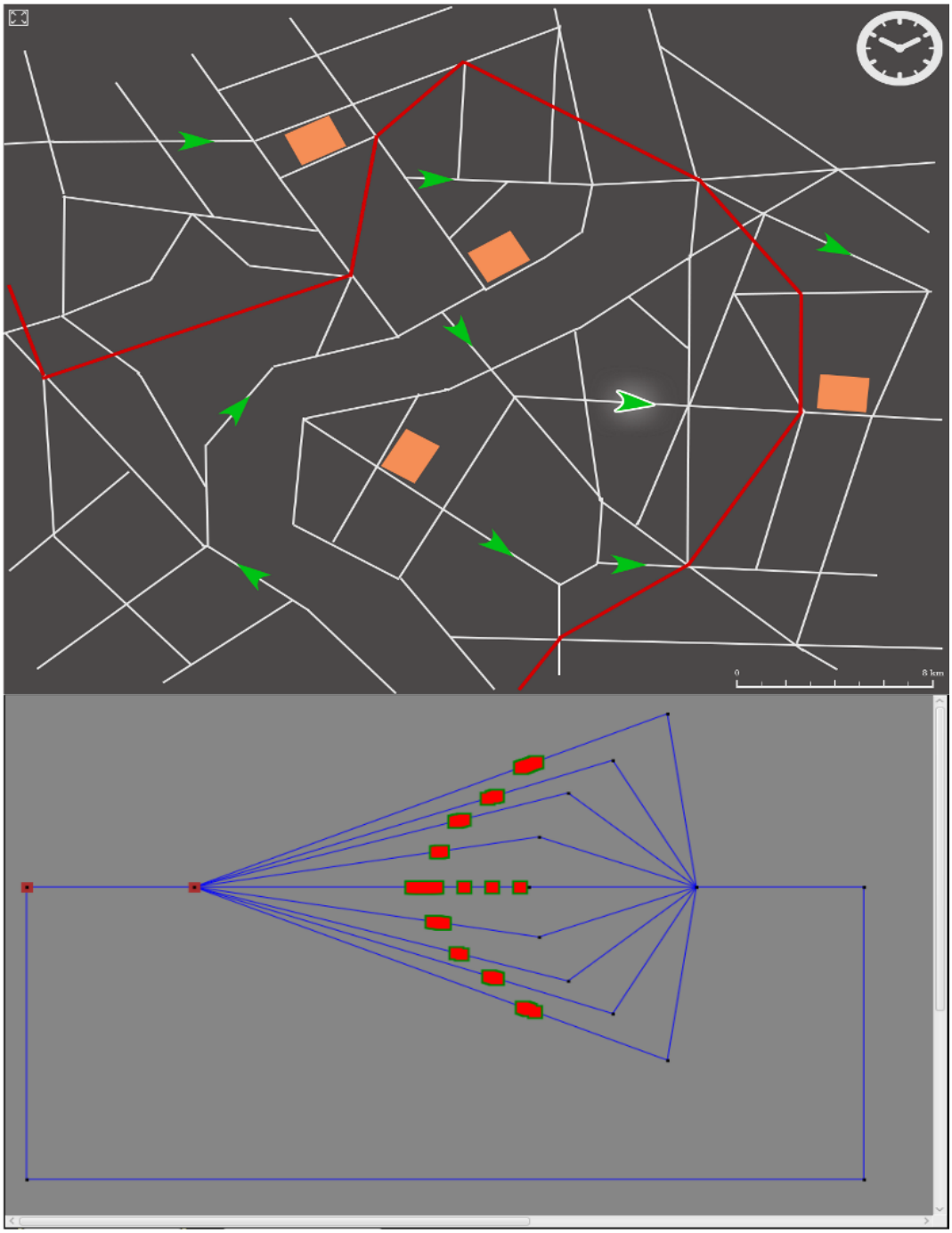


FIGURE 8 – Comparaison de la zone de visualisation entre la maquette (en haut) et l'application (en bas)

D'où le choix d'utiliser des carrés plutôt que des flèches pour représenter les imbrications.

- Les bâtiments sont beaucoup plus discrets (on en voit deux à gauche de l'image) car MatSim les place systématiquement sur les liens et les afficher en trop gros pourrait masquer le reste de la visualisation.
- On ne peut pas déplacer la carte en maintenant le clic gauche enfoncé et en bougeant la souris, pour cela il faut utiliser les ascenseurs verticaux et horizontaux.
- Pour zoomer et dézoomer il faut utiliser des touches de clavier (u et z respectivement) et non le curseur de la souris qui lui interagit avec l'ascenseur vertical.

À savoir que même si les couleurs et les tailles par défaut ont changées, ces dernières peuvent tout de même être modifiées grâce aux filtres.

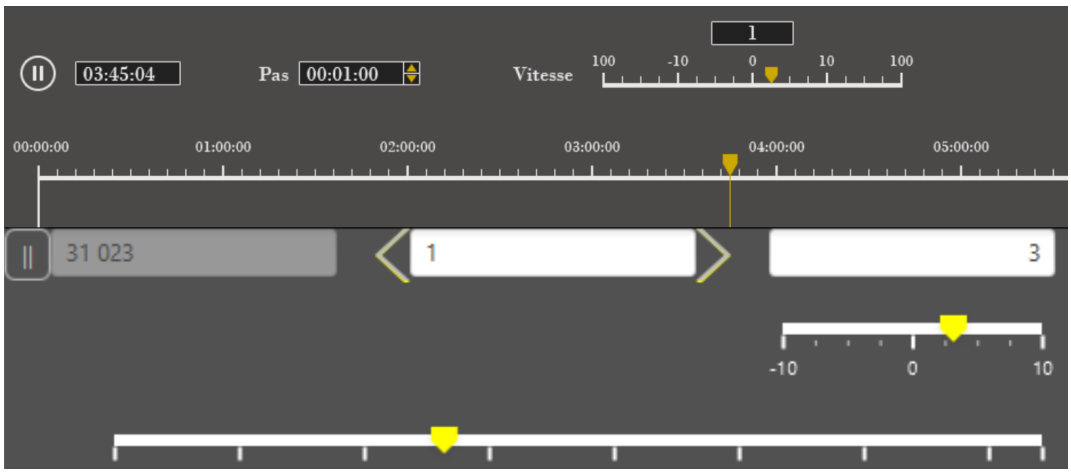


FIGURE 9 – Comparaison de la timeline entre la maquette (en haut) et l'application (en bas)

Zone de gestion temporelle En analysant la Figure 9, on voit bien que toute la gestion temporelle prévue pour la visualisation a été implémentée comme annoncé. Là encore les changements sont essentiellement graphiques (couleurs, positions, affichage ....). On retrouve bien l'affichage du temps, le pas pour l'avancement manuel, la vitesse pour l'avancement automatique ou encore la ligne temporelle. La seule chose que l'on peut remarquer, c'est que notre application gère et affiche les unités de temps en décimaux (double) alors que la maquette représente des "vrai" unités temporelles en minutes et en secondes. Ce changement s'explique surtout par des facilités d'implémentation et par le fait que MatSim gère aussi le temps avec des décimaux pour indiquer le timestamp des événements.

Zone des éléments suivis Une fois de plus, les différences sont essentiellement graphiques d'après la Figure 10. On voit bien les détails des éléments qui s'affichent avec la possibilité de rétracter ou de déplier. Cependant l'affichage ne se fait pas dans un tableau sur notre application, et pour supprimer un élément il faut faire clic "droit > fermer" contrairement à la maquette qui prévoyait un bouton de suppression.

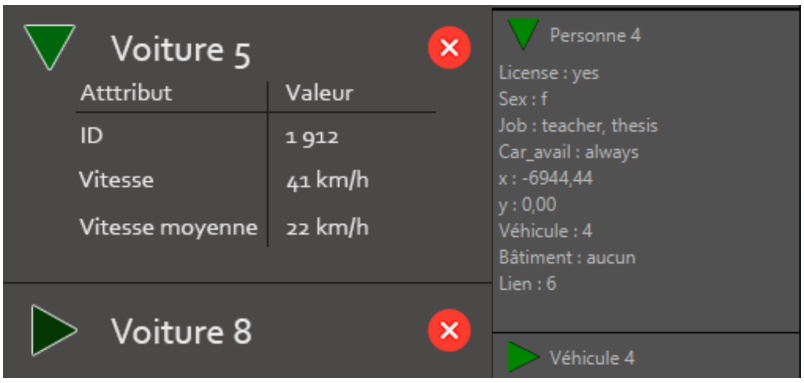


FIGURE 10 – Comparaison du détail des éléments suivis qui s'affichent lors d'un clic sur l'élément ciblé avec la maquette (à gauche) et notre application (à droite)

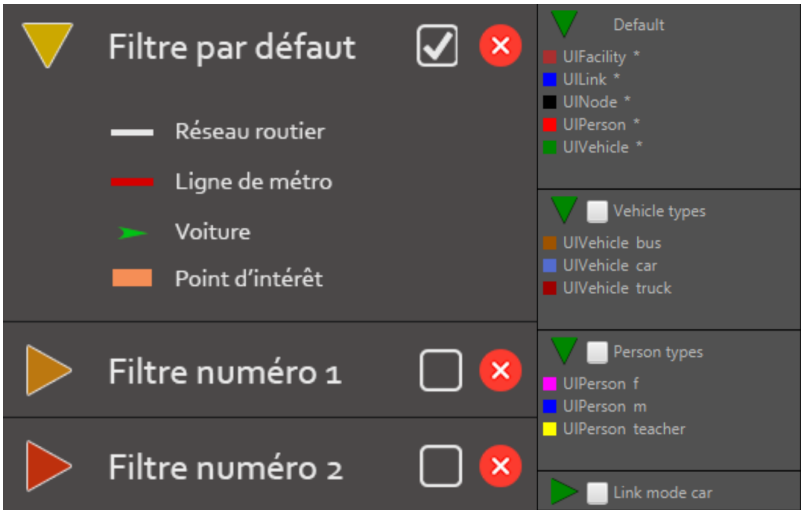


FIGURE 11 – Comparaison de la gestion des filtres entre la maquette (à gauche) et l'application (à droite)

Zone des filtres Toujours à quelques détails graphiques prêt visualisable en Figure 11, la gestion des filtres dans notre application est sensiblement la même que celle imaginée à la phase de spécification. Grâce aux filtres on peut attribuer des tailles et des couleurs précises sur des types particuliers, ou choisir d'en cacher certains. Voici cependant quelques différences d'utilisation :

- Il n'est pas possible d'appliquer un motif sur un filtre, uniquement une taille et une couleur
- Pour supprimer un filtre il faut faire "clic droit > Remove", comme pour le détail des éléments suivis il n'y a pas de bouton de suppression implémenté
- il est possible de réordonner les filtres en faisant "clic droit > Take Down"
- Il n'est pas possible de modifier un filtre en double cliquant dessus comme annoncé dans la spécification. Si on veut modifier un filtre, il n'y a pas d'autres choix que de le supprimer et de le recréer pour appliquer les modifications souhaitées

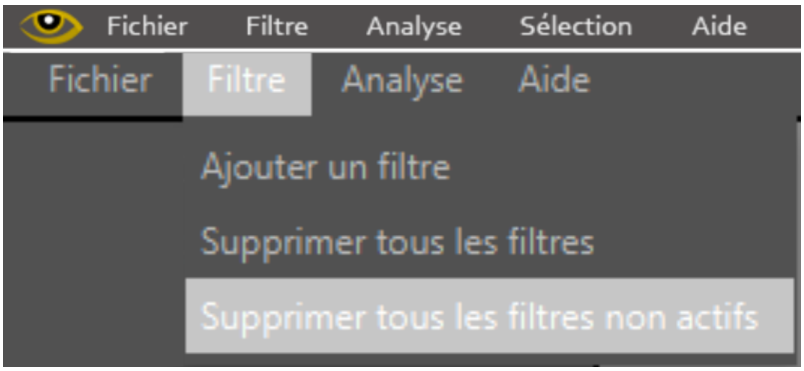


FIGURE 12 – Comparaison de la barre des menus avec la maquette (en haut) et notre application (en bas)

Barre de menu Aucun changement au niveau de la barre des menus et même si la maquette ne le montre pas, toutes les options pensées à la spécification sont dans l'application. Cependant toutes les fonctionnalités listées dans le menu n'ont pas été implémentées, c'est ce que nous verrons dans Partie III. sur l'avancement du projet.

Même chose pour les analyses, il n'y avait pas de maquettes montrant à quoi elle ressembleraient, mais ces dernières étaient imaginées sous forme de diagrammes montrant l'évolution d'une caractéristique au cours du temps, et c'est exactement de cette façon qu'elles ont été implémentées. Certaines analyses possibles avaient été pensées comme exemple, nous verrons dans la Partie III. lesquelles ont réellement étaient développées.

II.1.b. Modèle

Dans le rapport de spécification, nous avions défini des éléments obligatoires qui devaient être renseignés pour pouvoir visualiser la simulation, ainsi que des éléments facultatifs qui devaient permettre de l'enrichir dans le cadre des filtres et des analyses.

Pour ce qui des spécifications obligatoires, ces dernières ont été très bien respectées. En effet, on retrouve le réseau avec tous ses noeuds localisés par coordonnées ainsi que les liens reliant ces noeuds. La gestion des déclencheurs d'actions par événement a aussi été respectée et on retrouve des événements de différents types tel que les embarcations, le trafic et les activités. Ils sont bien constitués d'un horodatage pour situer la date de l'événement, ainsi que de coordonnées pour le localiser sur la carte (par l'intermédiaire du modèle dynamique). Cependant chaque événement ne possède qu'un seul acteur (une seule personne ou un seul véhicule) et non pas une liste d'acteurs impliqués contrairement à ce qui avait été envisagé. Ce choix a été décidé dès la conception et permet d'avoir une gestion plus simple et plus stricte dans le déclenchement des événements.

Pour les éléments facultatifs, plusieurs pistes ont été envisagées et bon nombre d'entre elles ont été conservées mais avec des adaptations. Dans tous les cas, la plupart des éléments additionnels venant enrichir la simulation passe par la "gestion en type" dont les choix de conception ont beaucoup évolués durant le développement (voir Partie II.2.b.). Pour revenir aux spécifications voici les différences entre les prévisions et la réalisation :

- Il est effectivement possible d'enrichir les éléments déjà présents en leur ajoutant des attributs comme la profession d'une personne ou encore le modèle d'une voiture. Pour cela il faut passer par la gestion en type ou la définition d'un nouveau modèle héritant du modèle de base.
- Le regroupement de certains éléments est possible mais pas de la même façon que celle annoncée. Il n'y a pas d'élément "Entreprise" pour regrouper des acteurs ou encore des éléments "Relation" pour regrouper des liens contrairement à ce qui était envisagé, mais tout passe, une fois de plus, par la gestion en type.
- La catégorisation des éléments passe aussi par cette gestion en type et c'est aussi ce qu'annonçait le rapport de spécification. Pour rappel la catégorisation permet de définir le type d'un véhicule (camion, voiture, bus...) ou encore le type d'une route (autoroute, chemin, ligne de bus...).

Cette gestion en type paraît encore flou à ce stade mais une fois de plus, celle-ci sera correctement expliquée dans la Partie II.2.b. car les choix d'implémentation ne sont pas tout à fait les même que ceux annoncés lors de la conception. En revanche, la mise en place d'une échelle est le seul élément qui a été abandonné lors de la conception et du développement.

II.1.c. Parseur

La phase de spécification pour le parser avait surtout pour but de définir les objectifs de ce dernier, à savoir son rôle, ses attentes, ses spécificités ainsi qu'un rapide aperçu de son fonctionnement en vue de la phase de conception. Que ce soit par rapport à la spécification ou encore à la conception, le parseur est sûrement la partie de l'architecture de l'application qui subit le moins de modifications lors de son implémentation. Et comme il n'y a eu quasiment aucun changement par rapport à ce qui était prévu dans le rapport de spécification, nous allons seulement faire un rapide résumé des points qui y ont été abordés :

- Le rôle du parseur est de lire les fichiers d'entrée fournis par l'utilisateur afin de générer un modèle qui pourra être exploité en vue de la visualisation de la simulation côté GUI. C'est exactement ce que fait le parser actuel de notre application en créant un objet scénario contenant tous les éléments traduits pour la simulation.
- Le parser doit générer des éléments obligatoires (qui doivent absolument être renseignés par l'utilisateur ou bien complétés avec des valeurs par défaut quand cela est possible) ainsi que d'éventuels éléments facultatifs qui viennent enrichir la simulation. Là encore, notre parser est divisé en deux parties pour gérer ces deux types d'éléments.
- Comme cela fut annoncé, le parser n'effectue aucune vérification, c'est-à-dire que s'il y a une problème de cohérence dans les données d'entrées, une erreur apparaîtra, par la suite, dans le traitement du modèle généré.
- Une instance de traducteur différente est créé pour chaque type d'élément traduit, par exemple un traducteur pour les véhicules ou encore un traducteur pour les événements.
- Un fichier de configuration au format JSON permet à l'application de retrouver l'emplacement des fichiers d'entrée, les éléments du modèle qui doivent être générés ainsi que les parsers utilisés pour chacun de ces éléments.

- L'utilisateur a la possibilité d'implémenter et d'utiliser ses propres parseurs pour s'adapter à diverses formats de données d'entrée. Ses derniers devront être fournis au format .jar lors de l'import d'un nouveau projet de simulation dans l'interface utilisateur.
- Enfin notre application dispose d'un traducteur par défaut, utilisé pour traduire les fichiers de données de MatSim au format xml. La lecture de ces fichiers se fait alors avec le langage de requête XPath.

Tous les points cités ci-dessus ont été abordés plus en détail dans partie dédiée au parser dans le rapport de spécification, et ils ont tous été respectés et implémentés dans notre application. On peut retrouver en Figure 13 une ébauche réalisé lors de la phase de spécification pour représenter ce à quoi pouvait ressembler un fichier de configuration du parser en json à ce stade du projet. On retrouve bien dans notre version actuelle la découpe des éléments obligatoires et non obligatoires avec les différentes parties du modèle à générer. Cependant l'intégration des type "TypeOfVehicule" ou encore "TypeOfRoad" n'a pas du tout été gérée de cette façon et c'est ce que nous allons voir en Partie II.2.c..

FIGURE 13 – Ébauche réalisée lors de la phase de spécification pour illustrer un fichier de configuration du parser en json

II.2. Conception

Le rapport de conception avait pour but de décrire comment l'application ainsi que toutes les fonctionnalités pensées à la spécification, seraient être implémentées dans le cadre du développement. Nous allons voir parmis ces choix, lesquels ont dûs être revus mais aussi ceux qui se sont avérés être très pertinents dès le départ et qui ont été scrupuleusement respectés.

II.2.a. GUI

Globalement, les choix d'implémentation choisis lors de la phase de conception pour la mise en place de l'interface utilisateur ont été très bien respectés.

Parmi ces choix on retrouve la mise en place d'un modèle dynamique, seul modèle directement accessible par le GUI et ses contrôleurs. Pour rappel, ce modèle dynamique avait pour but de stocker tous les objets dessinables par l'interface graphique ainsi que tous leurs attributs évolutifs comme leur taille, leur couleur ou encore leurs coordonnées sur la carte afin de suivre le côté stateful des éléments de l'application. Ce modèle dynamique avait directement accès au modèle statique de base, traduit du parseur dans le cadre d'une architecture MVVM. Ce choix d'implémentation ainsi que les diagrammes de classe associés, tous décrits dans le rapport de conception ont été très bien respectés et ont même grandement contribués au développement efficace de l'interface utilisateur, preuve de la pertinence d'une telle architecture.

L'implémentation des analyses en revanche, abordé dans les spécifications, n'avait pas été décrite dans la conception car il s'agissait de fonctionnalités facultatives, et donc par choix de priorité, nous ne nous étions pas penché dessus à ce moment là. Elles ont cependant été implémenté à ce stade et le diagramme de classe décrivant leur conception est disponible en Figure 14. Comme on peut le voir, toutes les analyses disponibles sont représentés par une classe dans le modèle dynamique et chacune d'entre elles héritent d'une classe abstraite de base "Analyse" qui rassemble toutes les propriétés communes. Parmis elles on retrouve le titre de l'analyse, l'abscisse de départ et de fin ainsi que le pas pour représenter le graphe ou encore la classe analysée. Mais surtout, toutes les analyses doivent implémenter la méthode "computeAnalysis" qui calcule une liste de points en fonction du type passé en paramètre. Par exemple, pour l'analyse "VehiculesInTrafficAnalysis", la méthode "computeAnalysis" va calculer une liste de points en fonction des véhicules qui rentrent et qui sortent des liens au cours du temps de la simulation. C'est pour cela qu'on retrouve le scénario dynamique en paramètre du constructeur, l'analyse en a besoin pour explorer la simulation est calculer la liste des points. Toutes les listes calculées sont stocké dans un dictionnaire qui relis le type (voiture, bus, camion ....) à sa liste de points calculée. Grâce à cette implémentation, on peut facilement ajouter de nouvelles analyses qui héritent de la classe de base, et il n'y a qu'un seul type de contrôleur pour toutes les analyses. En effet à chaque fois qu'une nouvelle analyse est lancée, on appelle le contrôleur "AnalysisController" avec la bonne instance d'analyse, et la méthode "generateChartResult" appelle la méthode "computeAnalysis" de son analyse, pour récupérer la liste des points qui doit être affichée. Le contrôleur n'a alors plus qu'à afficher cette liste de points sous forme d'une courbe sur l'interface utilisateur. Plusieurs courbes peuvent ainsi être affichées en même temps pour chaque type que l'on souhaite analyser.

Pour ce qui est des filtres, qui ont pour but d'appliquer des propriétés d'affichage spécifiques (couleurs, tailles...) sur certains éléments en fonction de leur type, là encore, la conception était fidèle à l'implémentation. On retrouve bien une classe dédiée aux filtres dans le GUI, avec tous leurs attributs ainsi qu'une classe permettant de les appliquer sur les objets dessinables du modèle dynamique. Cependant les groupes de filtres sont directement gérés par des contrôleurs et non par une classe à part entière comme cela avait été annoncée. En effet cette méthode permet plus facilement de gérer l'ordre d'application des filtres que celle prévue initialement.

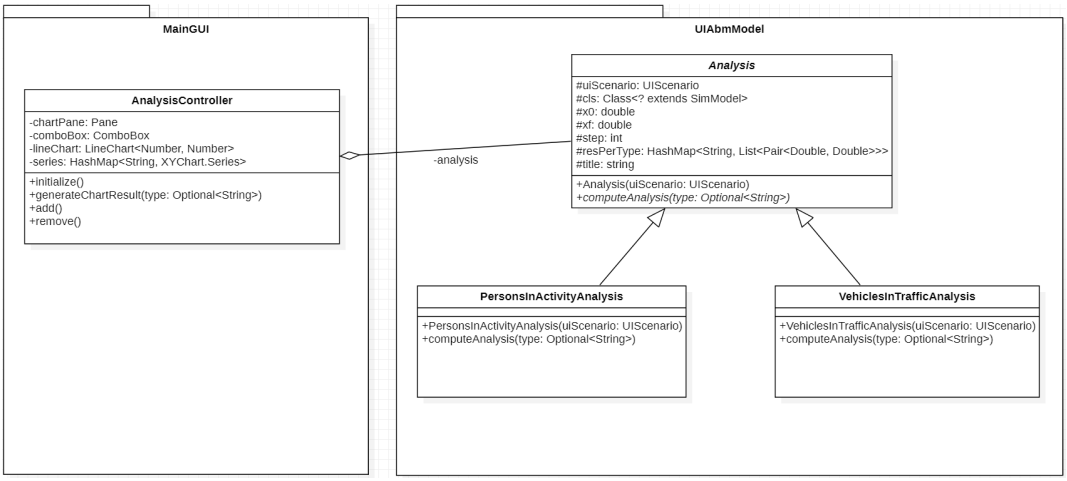


FIGURE 14 – Diagramme UML des analyses

Enfin pour les composants graphiques, le rapport de conception ne faisait que citer une partie des widgets pouvant être utilisés mais il était difficile de dresser un plan de conception précis pour la partie JavaFX. Seul point que l'on peut relever : il était prévu à la base de dessiner la simulation sur un "Canvas", élément de JavaFX spécialisé pour le dessin d'image, ce choix paraissait cohérent car le "Canvas" à de nombreuses méthodes pour la gestion automatique du zoom et des translations d'éléments. Mais en raison de problèmes de taille du Canvas qui ne permet pas de dessiner des images trop grandes nous avons donc dû nous résoudre à utiliser la classe de base "Pane" qui est beaucoup plus compliquée car il faut développer les méthodes de zoom et de translation nous même.

II.2.b. Modèle

Dans le rapport de conception, avant de décrire précisément le contenu du modèle à travers des diagrammes de classes nous avons d'abord expliqué les choix de conception que nous avons fait pour répondre à nos besoins.

Tout d'abord nous avons parlé de l'abstraction du modèle, qui repose sur le fait que tout modèle concret hérite d'un modèle de base abstrait qui contient toutes les propriétés de base du modèle de données que nous souhaitions intégrer à notre application. Ce choix de conception avait surtout pour but de répondre à une des problématiques du cahier des charges qui nous imposait d'avoir une structure de données modulable compte tenu du fait que les données d'entrées de l'utilisateur pouvaient être variées. L'abstraction du modèle devait donc répondre à ce besoin en offrant la possibilité à l'utilisateur d'implémenter son propre modèle issu de celui de base abstrait pour créer sa propre structure de données s'il le désire. Cela lui permet d'ajouter des attributs aux classes déjà présentes ou encore d'ajouter de nouvelles classes pour gérer de nouveaux éléments. Et cela répondait aussi en partie au problème des éléments facultatifs que nous n'avions pas forcément envisagés. Ce choix d'implémentation a été parfaitement respecté et effectivement notre modèle (dans sa partie statique) est bien constitué d'un modèle de base abstrait et tout modèle concret que l'utilisateur veut utiliser pour sa visualisation doit hériter du

modèle de base. C'est ce modèle concret qui sera alimenté par le parseur et qui sera utilisé par le GUI pour la visualisation (par l'intermédiaire du modèle dynamique). Le modèle concret par défaut que nous avons utilisé pour créer notre application est celui de MatSim, qui hérite du modèle de base sans rien y ajouter.

Autre choix de conception que nous avons déjà abordé dans la Partie II.2.a. est celui du MVVM (Model - View - ViewModel) qui consistait à avoir un modèle en deux parties : une partie statique directement alimentée par le parseur et qui contient toutes les propriétés constantes du modèle au cours du temps, et une partie dynamique utilisée par le gui et qui contient toutes les propriétés évolutives permettant à l'interface utilisateur de gérer l'affichage de la simulation. Dans la partie dynamique tous les objets sont dessinables et les événements exécutables et annulables. Une fois de plus ce choix de conception fut parfaitement respecté et il s'est avéré être très pertinent.

Enfin pour ce qui est du contenu du modèle, là encore, la majorité des prévisions fut respectées et on peut même dire que le temps passé à la conception fut très utile car lors du développement du modèle, il nous suffisait de "recopier" notre conception pour savoir exactement quelle classe coder et ce qu'elles contenaient. Cela a rendu le travail efficace et plus agréable avec très peu de retour en arrière lors du développement. En effet, la conception du modèle fut prise très au sérieux car cette partie conditionne aussi les choix d'implémentation du parseur et du GUI. Le modèle est bien décliné en objets obligatoires implémentant l'interface "MandatorySimModel" et en objets facultatifs de l'interface "OptionalSimModel" le tout implémentant "SimModel". On retrouve parmis les éléments obligatoires le réseau "Network" avec ses noeuds "Node" et ses liens les reliant "Link". Parmis les acteurs sont présents les véhicules "Vehicle" et les personnes "Person". Les événements sont déclinés par des relations d'héritages pour en gérer les différents types de la même façon que celle annoncée. Et les bâtiments aussi sont présents "Facility" en tant qu'élément optionnel. Et le contenu des différentes classes que ce soit au niveau des attributs ou des méthodes est quasiment le même que celui annoncé dans la conception, tout comme pour le modèle dynamique d'ailleurs.

En revanche là où on relève un grand changement dans le développement par rapport à la conception annoncée, c'est au niveau des types, dont l'implémentation, paraissaient encore un peu flou à ce stade du projet. À la base, il était prévu que toutes les classes du modèle (implémentant "SimModel") possède l'attribut suivant : "types : Map<String, Set<String»". Il devait permettre d'associer à tous les objets du modèle un dictionnaire dont les clés représentaient les "types" ("véhicules autorisé", "type de route") et les valeurs représentaient l'ensemble des valeurs prises par la clé ("voiture, bus, moto, camion", "autoroute"). L'implémentation actuelle est similaire, mais celle-ci est gérée à travers une nouvelle classe "Type" que possède toutes les classes du modèle. Le diagramme de classe décrivant cette gestion est disponible en Figure 15. Comme on peut le voir, toutes les instance "SimModel" possède un objet "Type" qui contient un dictionnaire unique "types" reliant tous les attributs de type de cette instance à une liste de valeurs correspondantes. Par exemple, dans le cadre d'une personne, on peut retrouver l'entrée suivante dans ce dictionnaire : "Job -> teacher, thesis". Comme on peut le voir d'après les principales méthodes, on peut ajouter des entrées dans cette structure, en enlever ou encore récupérer tous les types et toutes les valeurs par type répertoriés pour cette instance. Et dans l'interface SimModel, on retrouve la méthode addTypeEntry qui est implémenté par toutes les classes du modèle et qui permet d'ajouter une entrée dans l'objet type en appelant

sa méthode addEntry. Comme on le verra dans la Partie II.2.c., cette méthode est surtout appelé par le parseur lorsqu'il rencontre un type dans la lecture des attributs du fichier xml pour lui permettre de l'ajouter efficacement. Cette structure permet d'avoir une meilleur organisation des types en les gérant dans une classe à part.

Et en ce qui concerne le TypeCollector, aussi disponible en Figure 15, là encore on peut relever quelques changements. Tout d'abord il ne s'agit plus d'une classe générique mais d'une classe utilitaire unique pour tout le modèle. En effet toutes les méthodes qu'elle contient sont statiques comme c'est le cas de la classe Math en Java par exemple. Cette classe ne possède pas une single instance en tant qu'attribut mais plutôt un collecteur unique statique qui consiste en un grand dictionnaire ayant pour but de classer toutes les instances du modèle par valeur de type et par classe. Si on reprend le même exemple, on peut retrouver l'entrée suivante à l'intérieur de ce collecteur : Person.class -> ("Job" -> ("Student" -> person1, person2)) indiquant que person1 et person2 sont des instances de la classe Person qui possède toutes les deux le type Job=Student. Cette classe est surtout utile pour récupérer toutes les instances par type et donc pour appliquer les filtres et les analyses dans l'interface utilisateur. Bien que cette structure puisse paraître complexe, elle s'est avérée être utile et efficace car l'application des filtres et des analyses par type est parfaitement fonctionnelle à ce stade du développement de l'application. De plus, le fait d'utiliser une classe utilitaire, permet d'avoir un collecteur unique qui peut être appelé n'importe où dans l'application. Ainsi, à chaque fois qu'une entrée de type est ajouté dans un l'objet type d'une instance, ce dernier est aussi automatiquement ajouté dans le TypeCollector unique. De ce fait, à chaque fois que le parseur appelle la méthode "addTypeEntry" d'une instance lorsqu'il lit un type dans le fichier xml, ce dernier est ajouté à la fois dans l'objet Type de l'instance et dans le TypeCollector unique, d'où l'efficacité de la méthode.

Ce choix d'implémentation efficace et général explique pourquoi nous n'avons pas eu besoin d'implémenter la classe "Relation" (censé regrouper des liens aux propriétés communes) ou encore la classe "Company" (censé regrouper les acteurs d'une même entreprise). Tous ces besoins sont parfaitement couvert par cette fameuse gestion en type. Nous verrons dans la Partie II.2.c. comment le parseur s'est adapté à ce changement.

En revanche, la classe "TypeSpecification" qui était censé regrouper les caractéristiques d'un type, tel que les horaires ou le lieu de travail d'un enseignant ou encore la vitesse maximale ou la couleur d'une gamme de véhicule, n'a pas été implémentée par manque de temps et aussi car il ne s'agissait pas d'un élément indispensable. Elle aurait permis d'afficher d'éventuelles informations encore plus précises côté interface utilisateur.

II.2.c. Parseur

La conception du parser a été réalisée en effectuant des tests en parallèle pour vérifier la faisabilité des choix d'implémentation. C'est pour cela qu'il y a eu très peu de changements entre la conception annoncée et l'implémentation actuelle. Nous allons donc dans cette partie décrire la structure que nous avons utilisé pour gérer les traducteurs de notre application, ainsi que le fonctionnement global du traducteur par défaut utilisé pour les données de MatSim.

Comme cela a été défini dans le rapport de conception, et selon le même principe que le modèle, tout parser concret doit hériter d'un parser de base abstrait

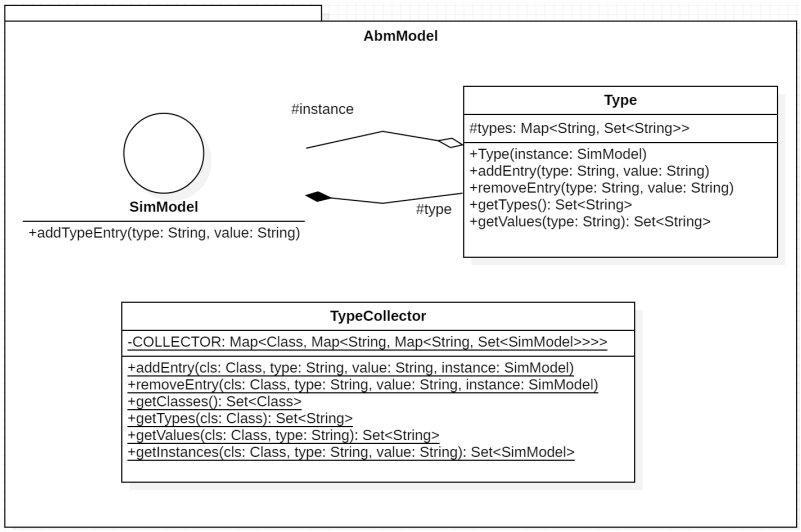


FIGURE 15 – Diagramme de classe décrivant la nouvelle gestion des types dans notre application

définissant les attributs communs ainsi que les méthodes qui doivent être implémentées. Le diagramme de classe du parser purement abstrait est disponible en Figure 16. C'est exactement la même architecture que celle définie à la conception à la seule différence prêt que l'on retrouve un attribut supplémentaire "type : Set<String>" qui permet de gérer les types que l'on évoquera par la suite. Si on analyse cette structure on retrouve le "ParserConfigurer" qui à partir du fichier de configuration json va instancier tous les autres parsers pour chaque type d'objet à traduire (personnes, véhicules, événements...). Un parser est soit un "MandatoryParser", pour les éléments obligatoires, ou un "OptionalParser" pour les éléments facultatifs et les deux héritent de la classe "Parser" rassemblant les propriétés communes de tous les parseurs. Ils sont donc chargés de lire une portion d'un fichier d'entrée qui leur est dédiée pour ensuite instancier les objets du modèle qui en résulte avec leurs attributs de base. Le "ScenarioProcessor" quant à lui est lancé une fois tous les fichiers d'entrée lues et les objets du modèle instanciés pour effectuer des opérations additionnelles sur le scénario qui ne peuvent pas être réalisées directement par les traducteur lors de la lecture des fichiers. Toutes les implémentations de traducteurs concrètes, ainsi que celles par défaut utilisées pour les entrées de MatSim doivent hériter de cette structure de base.

En Figure 17 est disponible un fichier de configuration complet tel qu'ils sont actuellement utilisés dans notre application. Nous allons décrire tous les éléments de ce fichier en commençant par les attributs généraux en haut à gauche :

ScenarioName Le nom du scénario à générer

InputPath Chemin d'accès aux fichiers d'entrée, qui sont les fichiers xml dans le cas de MatSim

PackageName Nom du package du modèle concret utilisé pour la génération du scénario, ici le modèle par défaut de MatSim de notre application

Unit Unité utilisée pour l'échelle (bien que celle-ci n'est pas prise en compte par notre modèle par défaut)

ParserPackage Nom du package de l'implémentation concrète du parser utilisé,

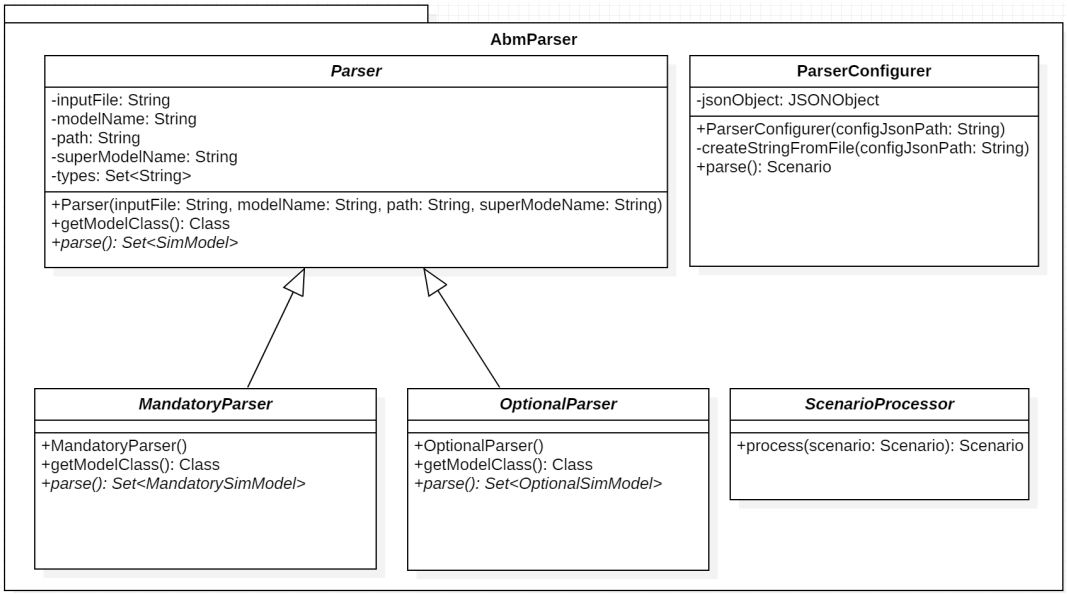


FIGURE 16 – Diagramme de classe du package AbmParser contenant la structure abstraite du parser de base

ici encore, celle par défaut de notre application pour traduire les fichiers de MatSim

ScenarioProcessor Le processeur utilisé pour la phase de post-traduction, une fois de plus, celui par défaut de notre application pour les scénarios de MatSim

Ensuite on a deux grands tableaux, un pour les éléments obligatoires du modèle et un autre pour les éléments facultatifs. Les deux sont organisés de la même façon, chaque élément du tableau contient toutes les informations nécessaires pour traduire une classe du modèle avec les attributs suivants :

InputFile Fichier d'entrée à lire, contenu dans le dossier désigné par "InputPath"

ModelName Classe du modèle à instancier pour cet élément (ici ce ne sont que des classes du modèle de MatSim d'où les initiales "MS")

Path Balise à rechercher dans le fichier "InputFile" et qui servira à instancier la classe "ModelName" associée. Par exemple, dans le cas des "Personnes", la balise "Path" qui doit être recherchés pour l'instanciation dans le fichier disponible en Figure 18, est la balise "person" soulignée en rouge. Les autres balises ne serviront pas dans le cadre de la génération du scénario

SuperModelName Indique le nom de la classe abstraite dont hérite la classe "ModelName" car selon l'abstraction du modèle, tout modèle concret (comme celui de MatSim) doit hériter du modèle de base abstrait

ParserName Le nom du parser qui doit être utilisé dans le package "ParserPackage" et il peut s'agir d'un parser personnalisé. Dans tous les cas il hérite forcément de la classe abstraite "Parser" du diagramme en Figure 16

Types Cet attribut n'était pas mentionné dans le rapport de conception. Il s'agit de la liste des attributs du fichier qui doivent être considérés comme un type pour l'élément traduit, et qui devront donc à la lecture, être ajoutés dans

l'instance "Type" de l'objet instancié ainsi que dans le "TypeCollecteur" général du scénario selon l'organisation décrite dans la Partie II.2.b.

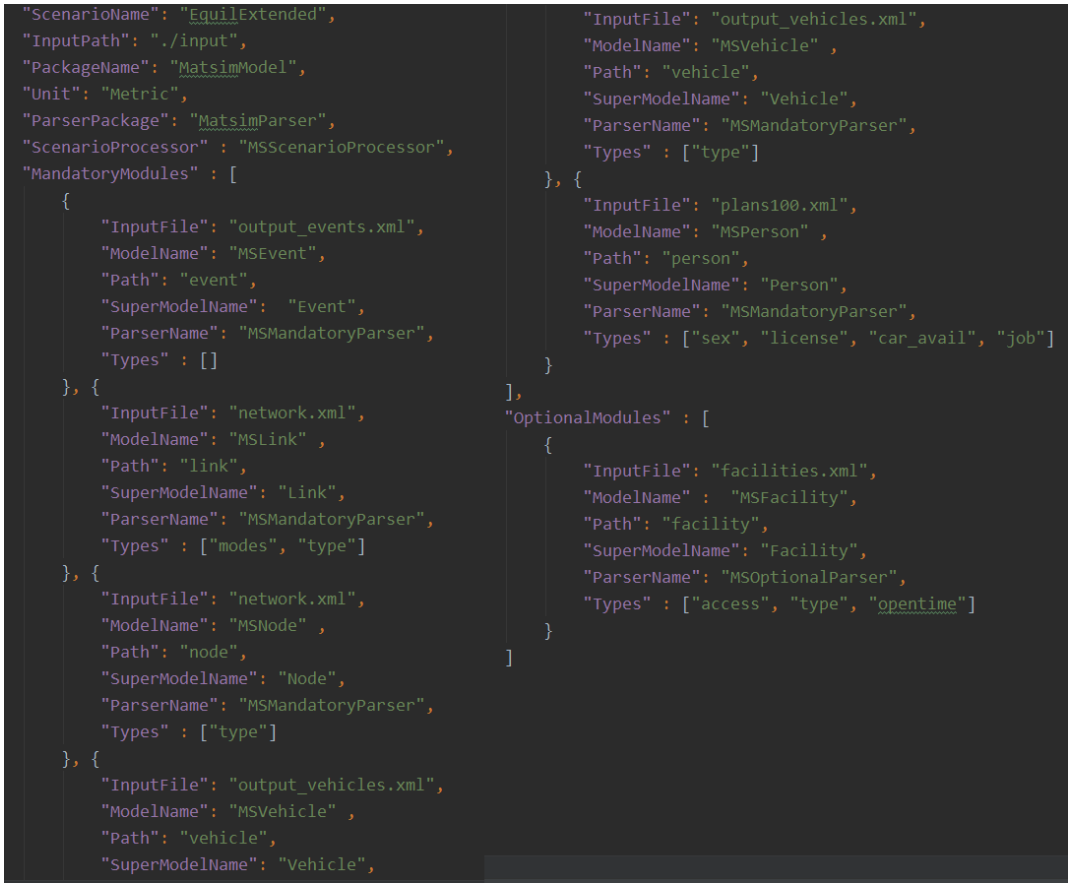


FIGURE 17 – Exemple d'un fichier de configuration json actuel de notre application

Maintenant que tous les éléments du parser ont été décrit voici son mode opératoire résumé dans le cadre de la traduction des données de MatSim. Lors de l'import de la simulation il faut indiquer le chemin d'accès au fichier de configuration JSON. L'application va alors lire le fichier pour connaître l'emplacement des fichiers d'entrées xml de MatSim, le package du modèle à générer ainsi que celui des parseurs à instancier. Les tableaux "MandatoryModules" et "OptionalModules" vont être lus et une instance de parser sera créé avec toutes les indications données pour chacune des classes du modèle qui devront être instanciées. Prenons par exemple, le dernier élément du tableau "MandatoryModules" dans le fichier de configuration en Figure 17 : un "MSMantatoryParser" sera alors instancié pour lire le fichier "plans100.xml". À chaque fois que le parser rencontrera une balise "person", comme celle du fichier en Figure 18, le parser va instancier un nouvel objet de la classe "MSPerson" du modèle héritant de la classe abstraite "Person". Les attributs du fichier seront alors lus un par un, le premier étant l'id d'après la Figure 18, l'attribut id du nouvel objet instancié prendra alors la valeur indiqué dans le fichier. Si le parser rencontre un attribut apparaissant dans la liste "Types" comme c'est le cas de "sex", "license", "car_avail" ou encore "job", il devra appeler la méthode "addTypeEntry" de l'objet instancié, qui est implémenté par toutes les classes du modèle, afin d'ajouter l'entrée de type parmis les

types de l'objet mais aussi dans le "TypeCollector" général. Toute cette organisation des types a été décrite dans la Partie II.2.b.. À savoir que certains attributs peuvent être à la fois renseigné dans les attributs de l'objets et considéré comme un type en même temps. Encore une fois tous les attributs marqués comme "Type" sont utilisés dans le cadre des filtres et des analyses. Si l'attribut lu ne correspond ni à aucun attribut de la classe, ni à aucun attribut marqué comme type, il sera tout simplement ignoré.

Une fois que tous les fichiers ont été lus, le "ScenarioProcessor" se lance pour effectuer les opérations post-traduction qui sont les suivantes :

- Remplir les dictionnaires du nouveau scénario reliant les id aux instances du modèle, afin de pouvoir récupérer facilement un objet dans le modèle à partir de son id
- Créer les associations entre les objets, par exemple les liens (qui sont des objets) doivent être reliés à leurs noeuds de départ et d'arrivé (qui sont aussi des objets)
- Les événements doivent être transtypés, en créant une nouvelle instance du bon type pour chaque événement traduit
- Enfin les coordonnées des événements doivent être calculées pour faciliter le travail du GUI, car ces dernières ne sont pas directement fournit dans les fichiers de MatSim mais elles peuvent être déduites

Le processing post-parser ne peut se faire qu'après l'étape de traduction, car lors de la lecture des fichiers, tous les objets du modèle (et donc du scénario) n'ont pas encore été instanciés.

Cette partie résume donc le fonctionnement ainsi que l'implémentation globale du parser dans notre application. Celle-ci est quasi identique à celle décrite dans la phase de conception. Le seul changement, comme cela fut évoqué, c'est que la gestion des types n'était pas clairement défini à ce stade du projet. Et cette gestion des types pour le parser à dû s'adapter à celle utilisée par le modèle qui a évolué durant le développement.

```xml
<?xml version="1.0" ?>
<!DOCTYPE plans SYSTEM "http://www.matsim.org/files/dtd/plans_v4.dtd">
<plans xml:lang="de-CH">
<person id="1" sex="m" license="yes" car_avail="always" job="student">
    <plan>
        <act type="h" x="-25000" y="0" link="1" end_time="06:00" />
        <leg mode="car">
            <route>2 7 12</route>
        </leg>
        <act type="w" x="10000" y="0" link="20" dur="00:10" />
        <leg mode="car">
            <route> </route>
        </leg>
        <act type="w" x="10000" y="0" link="20" dur="03:30" />
        <leg mode="car">
            <route>13 14 15 1</route>
        </leg>
        <act type="h" x="-25000" y="0" link="1" />
    </plan>
</person>

<person id="2" sex="m" license="yes" car_avail="always" job="teacher">
```

FIGURE 18 – Échantillon d'un fichier d'entrée xml au format des données de MatSim permettant de générer les personnes sur notre application. Les balises à lire et à traduire sont soulignées en rouge

III. Avancements

Dans cette partie nous allons décrire l'état de finalisation du projet. C'est-à-dire, parmi les fonctionnalités imaginées, lesquelles ont pu être implémentées et à quel niveau, mais aussi à quel point les critères ont pu être respectés. Nous donnerons aussi quelques pistes sur certains éléments qui n'ont pas pu être traités.

III.1. GUI

Le plus simple pour avoir une vision rapide des fonctionnalités qui ont pu être mises en place dans l'interface utilisateur, est d'analyser le tableau en Table 1. À titre d'indication, la visualisation, la timeline, les filtres, le suivi des éléments et la fonction d'import étaient des éléments obligatoires attendus par le cahier des charges ; les analyses ont été abordées de façon facultative par le cahier des charges et toutes les autres fonctionnalités ont été imaginées pour enrichir l'application sans avoir été mentionné dans le cahier des charges. On peut voir que les attentes sont donc bien respectées. Dans les prochain tableaux CDG représente la présence ou non de la fonctionnalité dans le cahier des charges. Un système de couleur à était mis en place pour mieux visualiser l'avancement des fonctionnalités. Les lignes vertes qui correspondent aux fonctionnalités dévéllopées et dans le cahier des charges. Certaines sont en bleu pour des fonctionnalités non obligatoires mais implémentées. Sont en orange les lignes pour celles qui n'étaient pas dans le cahier des charges et qui n'ont pas étaient implémentées. Et en rouge celles qui étaient dans le cahier des charges mais qui n'ont malheureusement pas étaient implémentées.

Table 1: Tableau récapitulatif de toutes les fonctionnalités spécifiés pour le GUI et leur état d'implémentation à ce stade du projet

CDG	Condition	Respecté	Détails
OUI	Visualisation principale	OUI	Comme on peut le voir sur la partie basse de la Figure 7 et 8 la fonction principale de l'application a bien été réalisée : On peut visualiser les déplacements d'agents et de véhicules sur un réseau routier.
NON	Eléments additionnels sur la visualisation	NON	Comme on peut le voir sur la partie haute de la Figure 8, il était prévu dans les coins de l'espace de visualisation des éléments qui n'ont finalement pas été développés : l'horloge tournante car jugée peu indispensable avec la timeline et peu compatible avec notre gestion du temps en décimales, l'échelle et la possibilité de mettre la visualisation en plein écran par manque de temps.

CDG	Condition	Respecté	Détails
OUI	Timeline	OUI	Comme décrit dans la Partie II.1.a. ainsi que sur la partie basse de la Figure 9, toutes les fonctionnalités prévues pour la timeline ont bien été implémentées : on peut faire avancer la visualisation automatiquement en contrôlant la vitesse ou manuellement en contrôlant le pas, on peut aussi mettre en pause ou encore se placer sur un timestamp précis.
OUI	Détail des éléments suivies	OUI	Comme cela a été évoqué dans la Partie II.1.a. et comme le montre la Figure 10, il est possible d'afficher le détail des attributs d'un élément dynamique en fonction du temps, lorsque l'on clique sur cet élément.
OUI	Filtres	OUI	Comme décrit dans la Partie II.1.a. et visible sur l'exemple de la Figure 19, les filtres ont pu être implémentés : il est possible d'appliquer une couleur, une taille ou un masque sur certains éléments en fonction de leur type. Il est aussi possible de suspendre et de réactiver certains filtres sans avoir à les supprimer (même si la fonction de suppression est aussi parfaitement fonctionnelle). Seul point négatif : il n'est pas possible d'appliquer des motifs (triangles, rond...) dans les filtres.
NON	Analyses	OUI	Les analyses ont été abordées en tant qu'éléments facultatifs sans réellement préciser la nature ni la forme qu'elles devaient prendre. Deux types d'analyses ont cependant pu être implémentées sous forme de courbes graphiques comme on peut le voir en Figure 20. De plus, leur conception décrite dans la Partie II.2.a. permet facilement d'implémenter d'autres types d'analyse au besoin.

CDG	Condition	Respecté	Détails
NON	Sélection	Non implémenté mais alternative	La sélection devait permettre de mettre en surbrillance certains éléments ou de réaliser les analyses en se basant sur des éléments sélectionnés, grâce à une sélection rectangulaire. Cependant les filtres et les analyses répondent quasiment à ces critères mais elles se basent sur les types (renseignés dans les fichiers xml). Si on veut réellement suivre l'évolution ou analyser des éléments précis, il faut modifier les entrées xml et attribuer un type spécifique aux éléments que l'on veut cibler s'ils n'en possèdent pas déjà un. L'application répond donc déjà partiellement à ce besoin sans cette fonctionnalité même si son implémentation aurait rendu la manoeuvre plus pratique pour l'utilisateur.
OUI	Import de projet	OUI	Comme on peut le voir en Figure 21, cette fonction a été complètement implémentée : On peut importer un projet de simulation en indiquant le chemin d'accès au fichier de configuration json (qui indique l'emplacement de toutes les entrées xml ainsi que les règles de parsing). Il est même possible d'importer son propre parseur en .jar pour adopter sa propre stratégie de parsing sur ses entrées xml (voir Partie III.3.). Enfin on peut même générer automatiquement un fichier de configuration à partir d'un modèle.
NON	Import/export d'une image de fond	NON	À ce stade, il n'est pas possible d'importer ou d'exporter une image de fond pour la visualisation. Cette fonctionnalité avait surtout pour but d'afficher une photo satellite derrière le réseau routier dans l'espace de visualisation pour rendre la visualisation plus réaliste.

CDG	Condition	Respecté	Détails
NON	Export d'une vidéo	NON	Il n'est pas possible, depuis l'application, d'enregistrer une vidéo lors de la lecture de la visualisation.
NON	Sauvegarder/Charger la visualisation	NON	Cette fonctionnalité avait surtout pour but de pouvoir retrouver la visualisation dans l'état dans le lequel elle avait été laissée en enregistrant notamment tous les filtres appliqués. Cette fonctionnalité n'a cependant pas été implémentée. Elle demanderait d'ailleurs le choix de patrons de conceptions spécifiques, comme un momento par exemple pour enregistrer l'état actuel de la visualisation.
NON	Autres détails	NON	D'autres fonctionnalités moins importantes ont été pensées comme la réinitialisation de la vue (zoom, filtres,...) qui n'a pas réellement été implémentée mais qui peut quand même se faire en important de nouveau le même fichier de configuration pour réinitialiser le modèle dynamique et donc la vue. Où la possibilité d'afficher la légende dans une fenêtre à part, mais le détail des filtres réalise déjà parfaitement cette fonction.

III.2. Modèle

Comme cela a déjà été évoqué, le modèle constitue l'intermédiaire entre le parseur, chargé de le remplir à partir des fichiers xml, et le GUI chargé de l'exploiter pour afficher la simulation dans l'espace de visualisation. Il n'y a donc pas de réelles fonctionnalités directement visibles comme c'est le cas de l'interface utilisateur. Cependant un certains nombre d'attente du cahier des charges ainsi que d'objectifs que nous nous sommes fixés ont été posés dans le cadre de l'établissement du modèle. Une fois de plus, le remplissage ou non de ces conditions est visible à travers le tableau en Table 2. On peut voir que la quasi totalité des conditions ont bien été respecté, avec 100% de remplissage des conditions venant du cahier des charges.

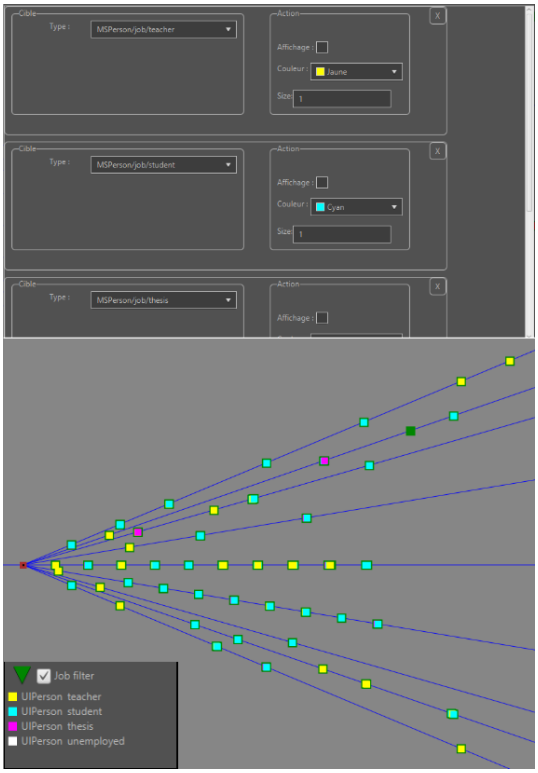


FIGURE 19 – Exemple d'application des filtres sur l'emploie des personnes. Ici on applique une couleur pour chaque profession : en haut la fenêtre d'ajout du filtre et le résultat en bas

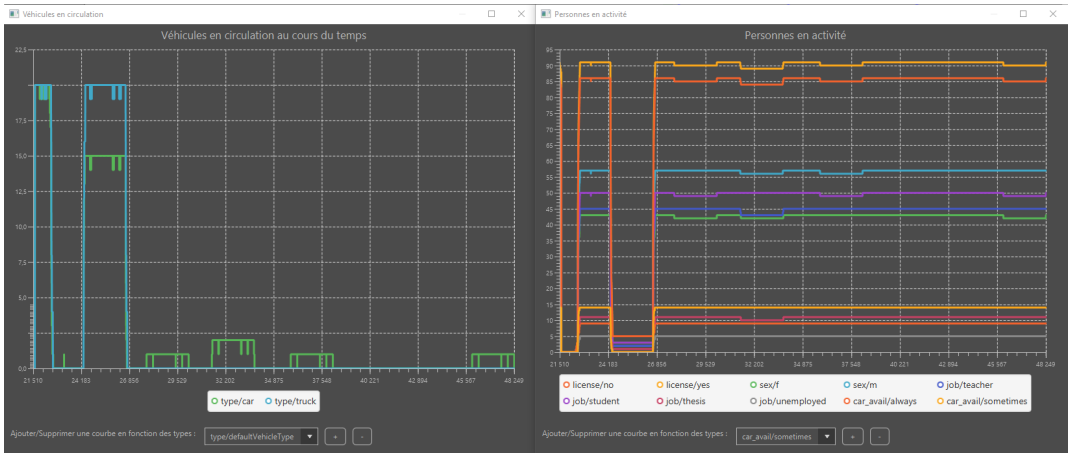


FIGURE 20 – Exemple d'application des deux analyses disponibles : les véhicules en circulation (à gauche) et les personnes en activité (à droite) au cours du temps. L'ordonnée représente l'effectif et l'abscisse le temps. On retrouve une couleur de courbe par type analysé

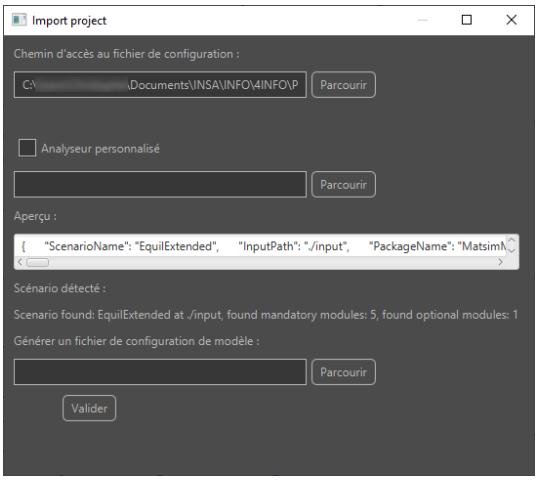


FIGURE 21 – Fenêtre d'import d'une nouvelle simulation : on retrouve, comme décrit, le choix du fichier de configuration json, l'éventuel parser personnalisé .jar, la preview ou encore la génération depuis un modèle

TABLE 2: Tableau résumant le respect des conditions en ce qui concerne le modèle

CDG	Condition	Respecté	Détails
OUI	Contenir les éléments nécessaires pour une visualisation de trafic urbain	OUI	Avec le réseau (noeuds et liens), les acteurs (voitures et agents) et les événements (trafic, embarcations, activités) le modèle contient tout ce qu'il faut pour simuler le trafic urbain.
OUI	Fournir les éléments nécessaires pour appliquer les filtres et établir d'éventuelles analyses	OUI	Grâce à l'intégration de la gestion en type, dont l'implémentation est détaillée dans la Partie II.2.b., on peut appliquer des filtres et des analyses en se basant sur ces derniers.
NON	Laisser la possibilité à l'utilisateur de fournir des descriptions de ses éléments	NON	Comme cela a été évoqué dans la Partie II.2.b., cette condition devait être respectée grâce à l'implémentation d'une classe "TypeSpecification" ou d'une structure similaire rassemblant les propriétés des différents types. Cependant, jugée non indispensable, elle n'a pas été implémentée par manque de temps. Néanmoins, un utilisateur aguerri peut toujours implémenter son propre modèle pour y intégrer des descriptions.

CDG	Condition	Respecté	Détails
OUI	Avoir un modèle à la fois global et modulable qui s'adapte à différents formats d'entrée tout en ayant une structure générale	OUI	L'abstraction du modèle répond à cette problématique : l'utilisateur peut implémenter son propre modèle pour répondre à des besoins précis s'il le désire, et le fait d'avoir un super modèle abstrait permet d'imposer une structure de base.
OUI	Avoir un modèle qui peut être alimenté par un parseur tout en étant exploitable par une interface graphique	OUI	La séparation en modèle statique (propriétés fixes) et en modèle dynamique (propriétés variables et objets dessinables) permet de séparer la partie vue par le parseur et celle exploitée par le modèle pour respecter cette condition par isolement.
NON	Intégrer des éléments obligatoires et facultatifs	OUI	Grâce aux interfaces "MandatorySimModel" et "OptionalSimModel", cette intégration est aussi respectée.

III.3. Parseur

Comme pour le modèle, nous allons étudier les avancements du parser en analysant le tableau en Table 3 qui présente le respect ou non des conditions de départ à ce stade du projet. Hormis les objectifs de hautes performances, toutes les conditions ont été respectées.

TABLE 3: Tableau résumant le respect des conditions en ce qui concerne le Parseur

CDG	Condition	Respecté	Détails
OUI	L'application doit avoir un parseur permettant de traduire les données de sortie d'une simulation MatSim au format xml	OUI	L'application possède un package "MatSimParser" qui correspond au parseur par défaut de l'application pour les données au format de MatSim
OUI	Avoir une structure permettant de s'adapter à différents format de données	OUI	L'application offre à l'utilisateur la possibilité d'utiliser un parseur personnalisé, couplé si besoin avec un modèle spécifique, pour s'adapter à des données particulières. Dans tous les cas le parseur concret hérite des propriétés générales de l'API abstraite.

CDG	Condition	Respecté	Détails
OUI	Possibilité de gérer et traduire de grandes quantité de données	NON	Malheureusement, comme cela a été évoqué dans la partie dédiée aux tests, au-delà d'une grande quantité de données et de taille des fichiers d'entrée, la traduction ne se fait pas dans un temps raisonnable et la mémoire peut être saturée.
OUI	Donner à l'utilisateur, la possibilité de surcharger certains éléments dans les fichiers d'entrée pour pouvoir les filtrer ou les analyser dans l'application lors de la visualisation.	OUI	Effectivement, l'utilisateur peut ajouter ses propres attributs dans les fichiers d'entrée et les marquer comme type potentiel dans le fichier de configuration, afin de pouvoir s'en servir dans le cadre des filtres et des analyses.
NON	Gérer les éléments obligatoires et facultatifs de façon distinctes	OUI	Cette condition est respectée à travers les classes abstraites "MandatoryParser" et "OptionalParser".

Pour conclure, on peut voir que globalement le projet a été mené à terme. À part pour quelques rares fonctionnalités annexes du GUI ou encore la gestion de très gros volumes de données, la totalité des conditions et des attentes émises au départ se retrouvent bien dans la version finale de l'application. De plus, certaines fonctionnalités enrichissantes ont même pues être implémentées comme la gestion en type offrant de grandes possibilités ou encore l'application des analyses.

IV. Bilan de planification

Pendant toute la durée du projet, nous avons effectué un suivi du volume horaire investi par chaque étudiant et dans chaque tâche effectuée. Le projet étant maintenant à son terme, nous allons dresser un bilan de ce suivi et le comparer à la planification faite en amont du deuxième semestre de travail.

IV.1. Données globales

L'anticipation faite en fin janvier visait 700 heures restantes sur le second semestre pour arriver à la fin du projet et de l'année scolaire. Les heures déjà effectuées étaient alors de 500 et la phase de planification en elle-même ayant pris 100 heures, la durée totale du projet a donc été estimée à 1300 heures.

Le suivi réel montre que seulement 400 heures ont été nécessaires jusqu'au 5 mai pour terminer toutes les phases du projet sauf celle de soutenance finale. En considérant que 100 heures seront encore nécessaires pour terminer cette dernière phase, on estime la durée totale réelle du projet à 1100 heures.

IV.2. Retour sur la gestion du projet

Il est évident que le projet a demandé un temps total de travail manifestement moins important que le total anticipé en phase de planification. L'équipe du projet se l'explique par l'application qu'ont mis tous les membres durant la phase de conception.

En effet, beaucoup d'efforts ont été faits pour obtenir une conception et une architecture solides qui généreraient le moins d'imprévus possible au cours de la phases de développement et de test. De nombreuses heures ont été investies dans cette première phase, mais beaucoup de temps a été gagné par une phase de développement rapide, comme on peut le voir dans la Figure 22.

On voit que les volumes d'heures pour le développement et la conception ont tous deux été surestimés. Cependant, le volume réel pour la conception reste relativement important comparé au volume réel du développement et on peut penser que ce phénomène a permis un temps de développement plus faible.

IV.3. Répartition du temps investi sur l'année

On peut voir dans la Figure 23 que les deux catégories qui occupent le plus de temps sont les réunions suivies de tous les travaux de formalisme, c'est-à-dire les rapports, soutenances et documentations. Pour cette dernière catégorie, il est normal qu'elle occupe du temps puisque le projet s'inscrit dans un module scolaire et nécessite donc un suivi par les enseignants. Quant aux réunions, elles ont été principalement dédiées à la communication à l'intérieur de l'équipe de travail.

La répartition sur les autres postes s'explique plutôt bien : l'interface utilisateur étant l'enjeu principal du projet, c'est sur ce point que le plus de temps a été investi. Pour le reste, la partie modèle a demandé plus que les autres car il y avait de nombreuses classes à dévelloper.

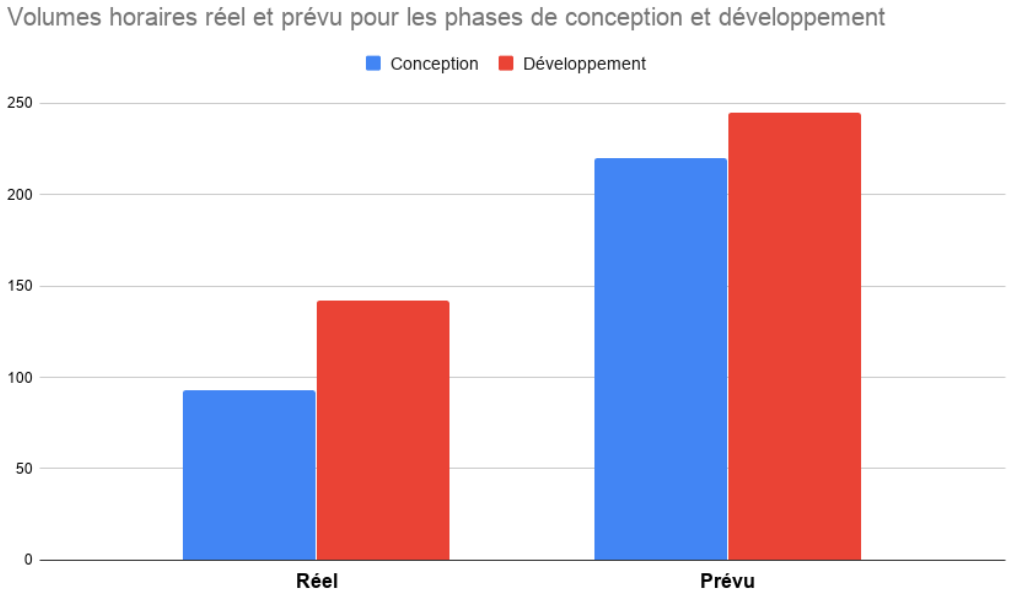


FIGURE 22 – Volumes horaires réel et prévu pour les phases de conception et de développement

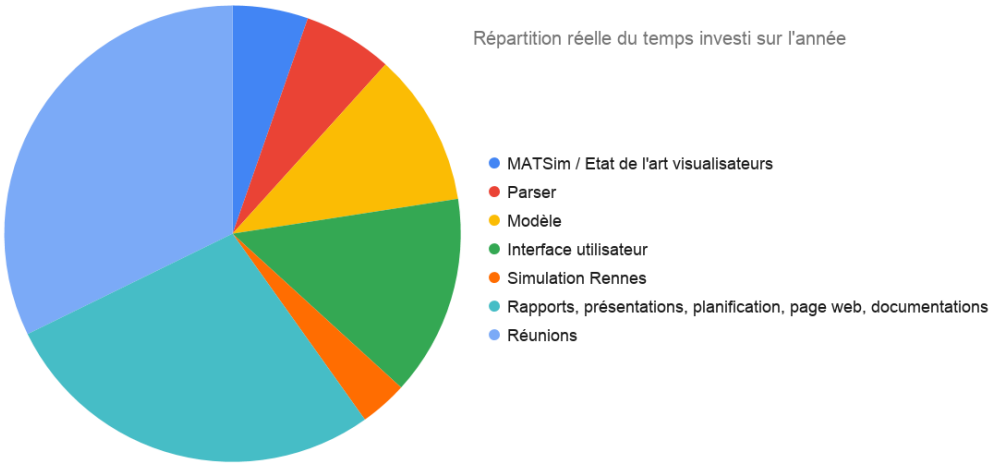


FIGURE 23 – Répartition du temps investi dans chaque type de tâche, sur toute la durée du projet

IV.4. Retour sur la prévision faite sur le second semestre

Au mois de décembre, à la moitié du projet, une phase de planification a permis à l'équipe de dresser une prévision des charges de travail à attribuer à chaque tâche pour le second semestre. Ces prévisions avaient plutôt vocation de limites maximales, les valeurs ont donc été choisies assez larges. On peut maintenant comparer les volumes horaires anticipés aux volumes horaires réellement investis.

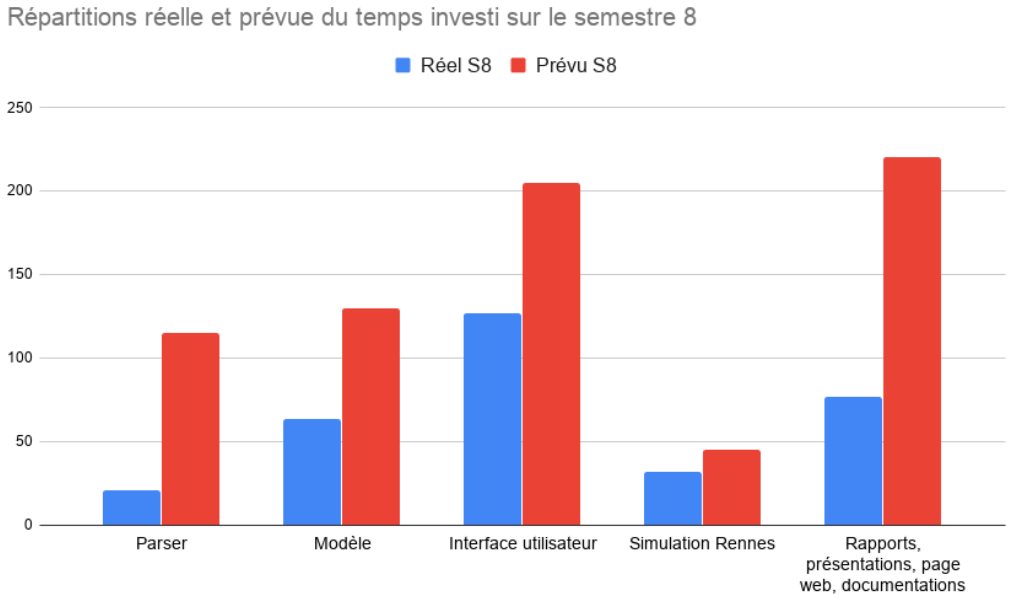


FIGURE 24 – Répartitions réelles et prévues du temps investi sur le semestre 8

La Figure 24 montre que, de manière générale, le volume horaire global nécessaire pour le second semestre de travail a été largement surestimé. Des marges comprises entre 10% et 20% avaient déjà été utilisées, mais l'on voit ici que pour certains postes comme le parser, seulement 20% du temps anticipé a été réellement nécessaire. Cela se doit à un concours de paramètres favorables :

- la conception et le développement se sont déroulés sans difficultés ni imprévus
- les tests n'ont pas demandé de réinvestir du temps dans le développement

Il est également à noter que le temps total investi n'est ici pas complet. Afin de rendre le rapport écrit, les données ont ici été figées au 5 mai pour un total légèrement supérieur à 1000 heures. On peut donc anticiper qu'un certain volume d'heures reste à être investi dans le projet pour la finalisation de ce rapport et la préparation de la soutenance orale.

IV.5. Evolution de la charge de travail par semaine sur le second semestre

La phase de planification a aussi permis d'estimer la charge de travail à prévoir par semaine tout au long du second semestre du projet, comme présenté en Figure

25. On peut maintenant faire un retour sur cette prévision à l'aide du suivi réel effectué présenté en Figure 26.

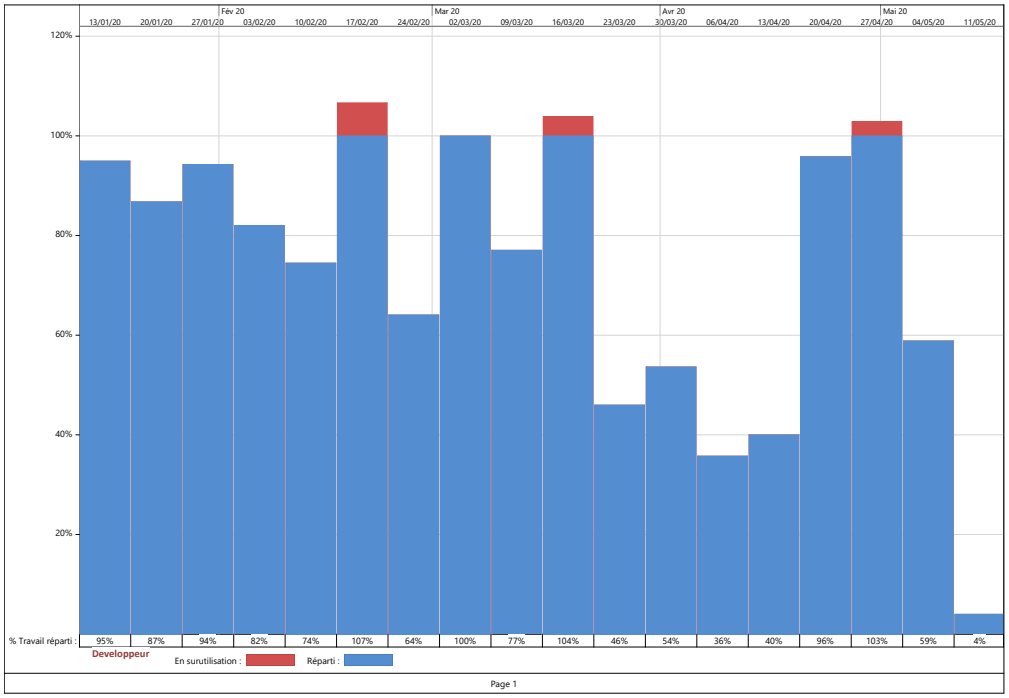


FIGURE 25 – Evolution de la charge prévue par semaine

On voit que le schéma réel diffère du schéma anticipé. Un pic de volume survient vers la fin du projet, pour finaliser la phase de développement, mais celui-ci est survenu début avril alors qu'il était anticipé en fin avril. En effet, la planification prévoyait une forte baisse du temps investi au début du mois d'avril, cependant les membres de l'équipe ont préféré ne pas diminuer leur investissement et prendre de l'avance sur la finalisation du projet. Il est également à noter que le fermeture de l'école et l'annulation de tous les cours présentiels pour une partie du mois de mars et pour le mois d'avril a permis un meilleur investissement sur ces périodes.

IV.6. conclusion

Ce bilan de planification montre que la planification a eu de nombreux différends avec la situation réellement survenue. L'estimation des charges de travail aussi bien que leur répartition sur le semestre se montrent imprécises, mais coïncident au moins en partie.

Il n'était pas question de parvenir à une planification parfaite du mode de travail du semestre, mais plutôt de poser des jalons de sécurité à respecter pour s'assurer que le projet arriverait à son terme. Sous cet angle, cette phase de planification a bien rempli son rôle puisque le modèle a été utilisé au cours de l'année pour estimer si l'avancement était suffisant ou en retard.

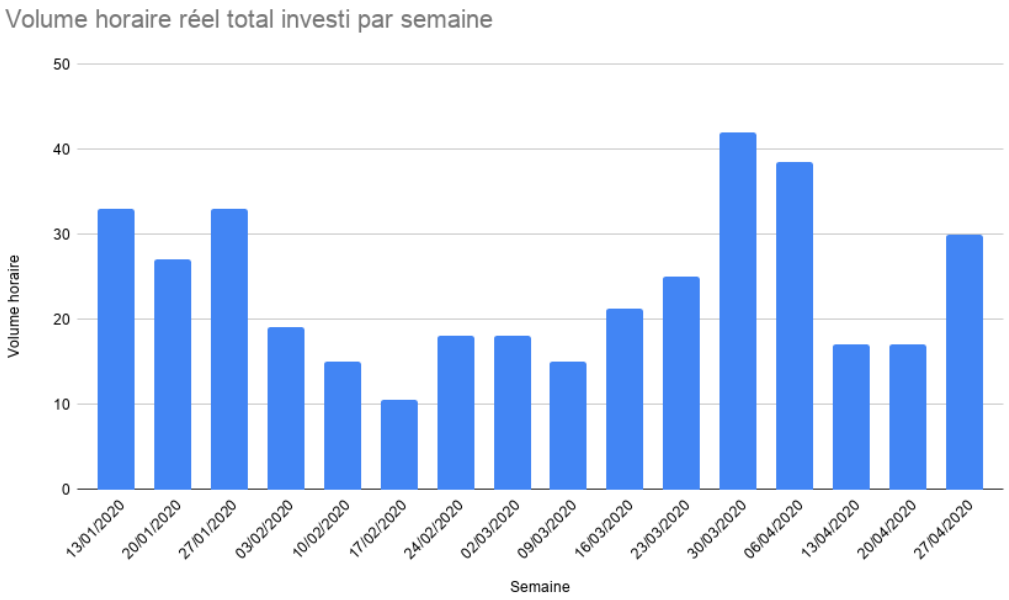


FIGURE 26 – Evolution de la charge réelle par semaine

V. Documentation Utilisateur

V.1. Installation

Pour installer l'application, il faut télécharger le fichier AbmVis-1.0.jar de l'application sur le site : https://guenobournac.github.io/AbmVis/. il s'agit de la dernière version lors de l'écriture de ce rapport.

V.1.a. Installation de Java 11

L'application nécessite Java 11 pour fonctionner. Vous pouvez vérifier la version de Java en exécutant la commande "java -version". Pour installer le JDK 11 :

- Linux Ubuntu : suivre cette documentation, à la section "Installation -> Version 11" : https://doc.ubuntu-fr.org/openjdkversion_11.
- Windows : il faut télécharger le JDK à cette adresse : https://jdk.java.net/java-se-ri/11 et le décompresser.

V.1.b. Installation de JavaFX

Téléchargez le SDK de javaFX correspondant à votre système https://gluonhq.com/products/javafx/ et décompressez-le de manière à connaitre son chemin.

V.1.c. Exécution

Dans cette partie, on notera :

- CHEMIN_JAVA le chemin vers l'exécutable java.exe du dossier bin du JDK téléchargé
- CHEMIN_JFX le chemin vers le dossier lib du SDK javaFX
- CHEMIN_JAR le chemin vers le fichier jar de l'application

Sous Linux Ouvrez un terminal et exécutez :

```
java --module-path CHEMIN_JFX --add-modules
    javafx.controls,javafx.fxml -jar CHEMIN_JAR
```

Sous Windows Ouvrez une invite de commande et exécutez :

```
CHEMIN_JAVA  --module-path CHEMIN_JFX --add-modules
    javafx.controls,javafx.fxml -jar CHEMIN_JAR
```

V.2. Bien démarrer

V.2.a. Aperçu

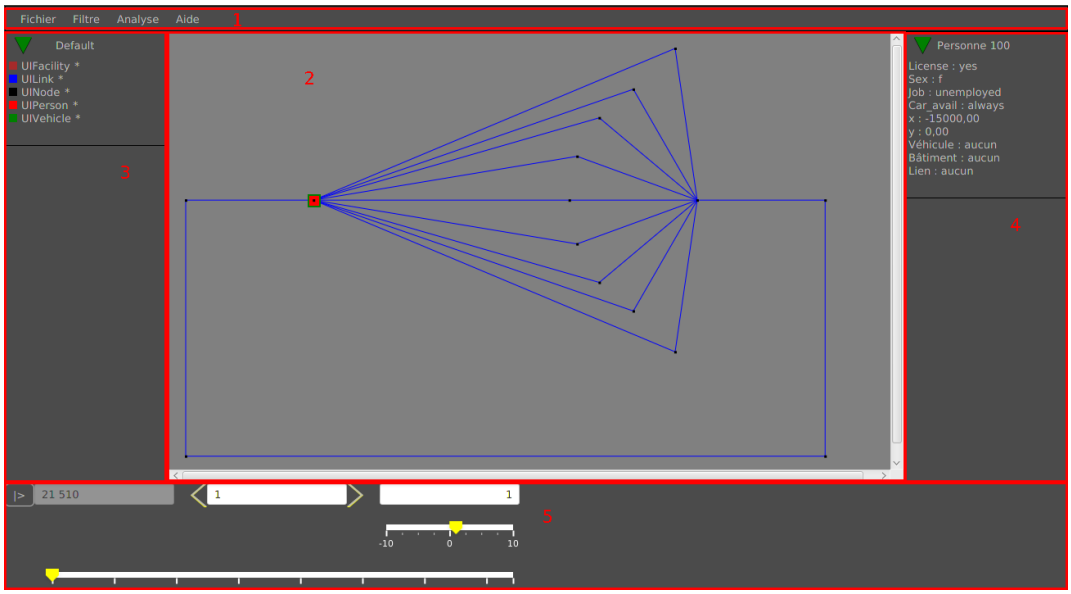


FIGURE 27 – Visuel de l'application

L'application est découpée en plusieurs parties comme visualisé en Figure 27. Tous les panneaux présentés ci-dessous seront détaillés ultérieurement dans des parties dédiées si nécessaire. La partie 1 correspond au menu de l'application. Cette

partie permet de faire des actions ne dépendant pas de la visualisation comme, par exemple, ajouter une nouvelle simulation, des filtres, ou visualiser différentes aides. La partie du milieu en numéro 2 permet de visualiser la simulation. C'est dans ce panneau que seront dessinés le réseau, les véhicules et les personnes. Sur cette figure les liens (route) sont les traits bleus, les noeuds du réseau les points noir, les personnes les carrés rouges et les véhicules des carrés plus grand en vert. Le panneau 3 à gauche est le panneau d'affichage des filtres, c'est un résumé des modifications qui seront appliquées aux éléments du panneau central. Par exemple changer la couleur des véhicule. La Partie V.3.a. détaille mieux cette fonctionnalité qui n'est pas nécessaire pour une première prise en main de l'application. Le panneau 4 permet de visualiser les informations sur certains objets. Par exemple il est possible de visualiser quelle personne est dans quelle voiture ou les coordonnées des véhicules, la Partie V.3.c. sera dédiée à l'explication de cette fonctionnalité. Le curseur du bas en numéro 5 permet la gestion il sera détaillé en Partie V.2.b.. Il permet de connaître et de choisir le moment visualisé dans le panneau centrale.

V.2.b. Fonctionnalités de base

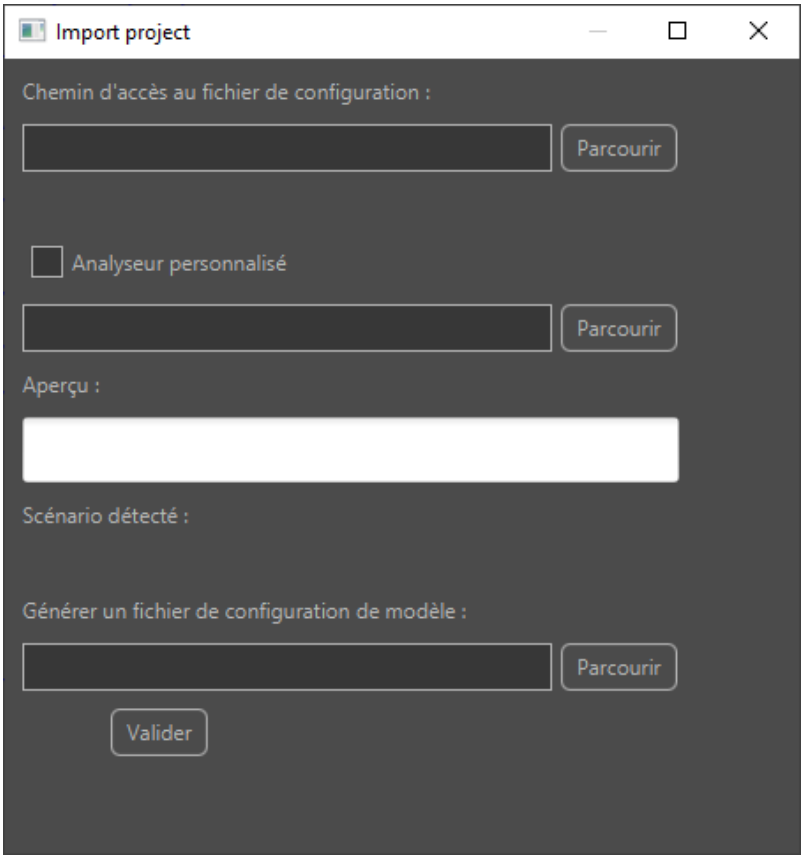


FIGURE 28 – Panneau de choix du fichier de configuration avant avoir fait un choix

Afin de pouvoir commencer à visualiser la simulation il est nécessaire de l'importer via la barre de menu, onglet fichier, importer un projet. La fenêtre d'importation, comme vu dans la Figure 28 permet de choisir entre plusieurs modes d'importations. Un chemin vers le fichier de configuration de base au format JSON,

devra être renseigné dans le premier champ pour permettre le choix des fichiers à importer voir la Partie V.4..

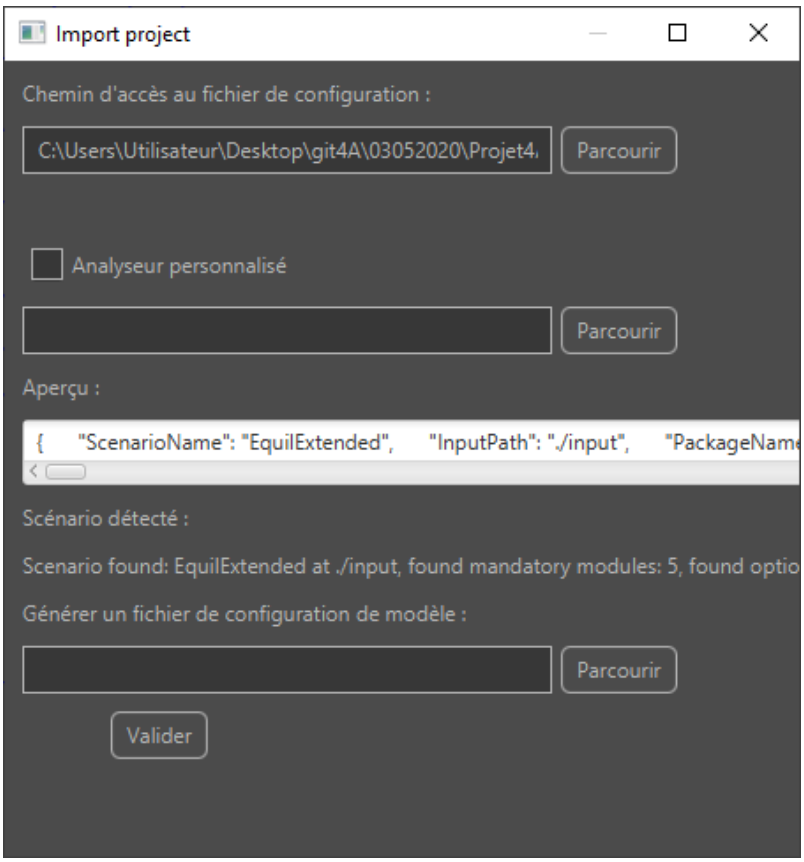


FIGURE 29 – Panneau de choix du fichier de configuration après avoir fait un choix

Si le fichier est conforme, un bref résumé du contenu sera affiché dans les champs inférieurs à des fins de contrôle comme nous pouvons le voir sur la Figure 29.

Après avoir validé l'importation du fichier, l'application se charge de l'affichage de la simulation et d'un filtre de visualisation par défaut qui affiche l'intégralité des informations. On peut ainsi voir sur la Figure 30 le filtre "Default" contenant plusieurs sous catégories pouvant être des points d'intérêt, des personnes, des liens ou des noeuds.

Nous pouvons voir en Figure 31 le panneau de gestion du temps. Il permet d'avancer ou de reculer le temps de la simulation. Le premier élément à visualiser est celui noté 1 qui correspond seulement à l'affichage du moment de la journée en seconde, soit dans l'exemple 24027 secondes. Le widget en 2 permet la gestion manuel du temps la zone de texte du centre permet d'écrire le nombre de seconde dont on veut augmenter ou diminuer à chaque pas de temps. Le bouton à gauche de cette zone permet de reculer dans la visualisation de ce pas alors que le bouton de droite permet d'avancer de ce nombre de secondes. Sachant qu'il est impossible d'avoir une valeur du temps inférieure à la valeur minimale lue dans le fichier, donc si le pas est de "10" alors que le temps est à 3 secondes au dessus de sa valeur minimum il reviendra à ce minimum. La *slide bar* en numéro 3 peut être cliqué pour aller à n'importe quel moment de la simulation. Elle permet de savoir le moment de la visualisation courant

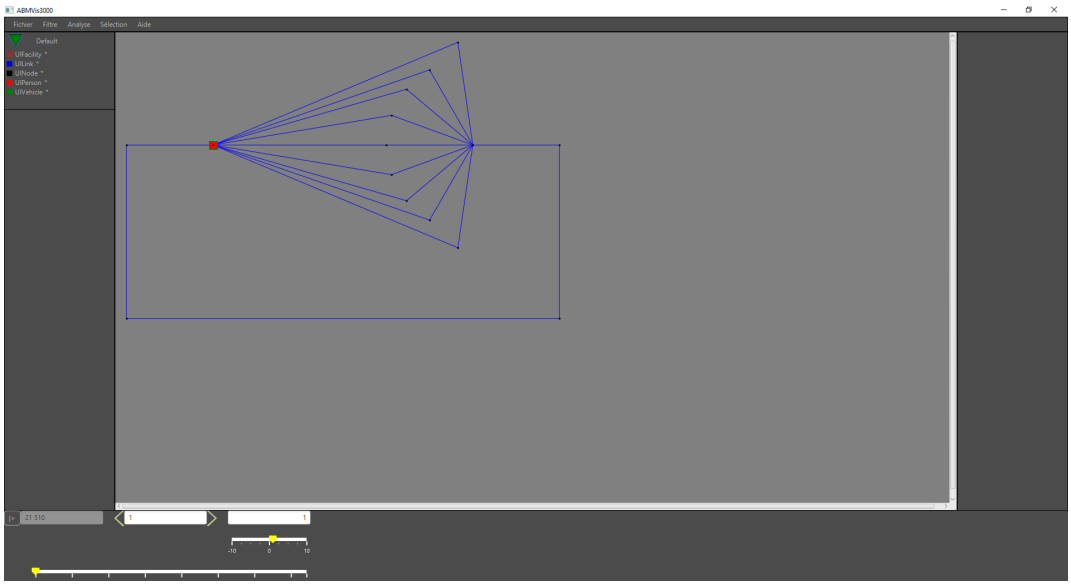


FIGURE 30 – Panneau de choix du fichier de configuration après avoir fait un choix

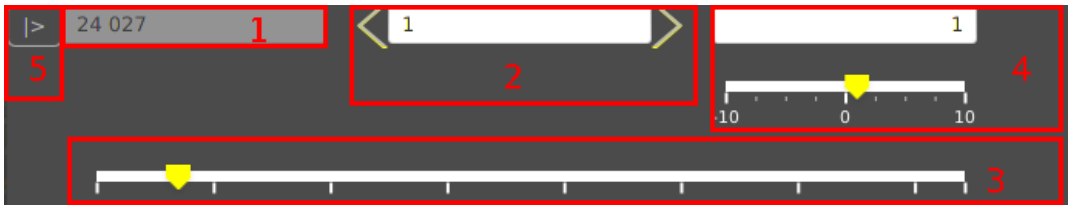


FIGURE 31 – Zone de gestion du temps avec des numéro

par rapport à son minimum et son maximum. Pour permettre une visualisation plus fluide il y a un module de gestion automatique du temps. Le couple *slide bar*, zone de texte numéro 4 qui permet de changer la vitesse. La *slide bar* va seulement de -10 à 10 mais il est possible de mettre des valeurs plus grandes en remplissant directement la zone de texte. Le bouton 5 permet de lancer et d'arrêter le défilement automatique du temps. Lorsque le défilement du temps est automatique entre chaque image le temps est augmenté de la valeur dans la zone de texte du widget 4. Donc si la valeur est 0 c'est une autre manière de mettre en pause, à éviter car continue de calculer des images il vaut mieux utiliser le bouton pour mettre en pause, si la valeur est positive le temps augmente on avance donc dans la simulation et si cette vitesse est négative la visualisation recule dans le temps.

V.2.c. Lancer sa première visualisation

Ceci est le tutoriel pour comprendre les fonctionnalités de base de l'application en facilitant au maximum son utilisation. Pour cela une simulation par défaut est disponible, il suffit de cliquer sur aide puis sur simulation par défaut. Il s'agit de la simulation par défaut de Matsim nommée Equil, le scenario comprend 100 personnes qui partent au travail. Sur le chemin elles se répartissent sur toutes les routes disponibles. Vous pouvez ensuite cliquer sur le bouton qui permet d'enlever la pause dans le panneau de gestion du temps. Il est conseillé de mettre la vitesse à 10 pour ne pas trop attendre. Il est possible de retourner au moment initial en utilisant la bar en bas en bougeant le bouton jusqu'à sa plus faible valeur ou en mettant la vitesse avec une valeur négative. Assurez vous que la visualisation est en pause pour la manipulation suivante. Choisissez une valeur entre 5 et 20 pour le pas, n'importe quelle autre valeur fonctionnerait mais trop grande la visualisation ira trop vite trop petite elle ira trop lentement. Cliquez plusieurs fois sur le bouton à droite du champ pour choisir le pas. Regardez comment évolue la visualisation, le champ qui donne le moment et la *Slide bar*. Vous pouvez faire la même chose avec le bouton de gauche. Pourquoi nous pouvons voir des carrés rouge dans les carrés vert ? C'est parce que les personnes en rouge sont dans les véhicules en vert. Il sera possible de le vérifier grâce au panneau de droite voir la Partie V.3.c. pour plus d'explication

V.3. Pour aller plus loin

V.3.a. Les filtres

Les filtres sont des aides pour l'utilisateur et son analyse du trafic visualisé. Il s'agit de permettre la modification de l'affichage en fonction des besoins de l'utilisateur. Par exemple si la personne souhaite cacher les camions parce qu'il ne font pas partie de ce qu'il cherche à analyser ou si il veut mettre en avant les autoroutes en les changeant de couleur et de taille, il doit en avoir la possibilité. C'est pour résoudre cette problématique que les filtres ont été développé. Par exemple pour PSA avec Gregory MARTIN comme encadrant l'objectif est d'analyser l'utilisation de l'autopartage sur un réseau. Avec les filtres il est possible de changer la couleur et la taille des véhicules en autopartage pour les mettre en évidence.

Panneau des Filtres Sur ce panneau en Figure 32, il est possible de visualiser tous les filtres par ordre d'application. Comme pour le CSS dans le web le plus

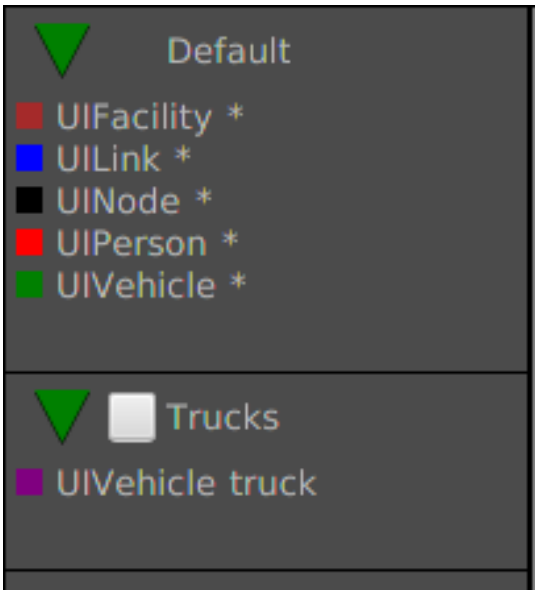


FIGURE 32 – Zone d'affichage des filtres

général est d'abord appliqué puis les propriétés peuvent être spécialisées avec des règles plus précises. Nous avons décidé de les appliquer du haut vers le bas pour que l'utilisateur puisse spécialiser les règles et qu'il connaisse l'ordre pour l'utiliser. En effet si deux règles peuvent s'appliquer pour modifier le visuel d'un objet celui le plus bas s'appliquera. Sur cette même figure il est possible de voir que l'utilisateur peut décider de rétracter un jeu de règles pour ne plus les voir afficher. Il y a aussi une case à cocher qui permet à l'utilisateur de désactiver un jeu complet de règles pour ne plus qu'elles s'appliquent. Pour chaque filtre dans le jeu de filtre nous pouvons visualiser la nouvelle couleur d'affichage, la classe sur laquelle s'applique le filtre et le type de cette classe affectée par le filtre. Nous déconseillons l'utilisation de la couleur blanche car c'est la couleur affiché sur ce panneau quand l'objet doit être enlevé de la simulation, si le blanc est choisis pour un type d'objet il sera bien affiché en blanc comme voulu mais sur ce panneau il sera impossible de différencier ceux qui sont blanc par ce qu'ils cachent ou par ce qu'ils affichent en blanc. En faisant un clique droit sur un jeu de filtres, il est possible de faire deux actions, supprimer ce jeu ou le descendre vers le bas pour en changer l'ordre d'application et le rendre plus prioritaire. Le jeu de filtres par défaut est le plus haut, il ne peut ni être descendu, ni être désactivé, ni être supprimé. Il contient le label "*" après le nom de la classe affecté, ce qui veut dire qu'il s'applique à tous les types de cette classe.

Menu des filtres Le menu relatif aux filtres permet l'ajout de filtres pour que l'utilisateur puisse créer les siens. Il y a deux autres possibilités dans ce menu la première est de supprimer tous les jeux de filtres ajoutés par l'utilisateur, donc le seul restant sera celui par défaut. La seconde est de supprimer tous les filtres non actifs c'est-à-dire tous ceux qui n'ont pas la case cochée.

Ajout de filtres Ce panneau d'ajout de filtres permet comme son nom l'indique d'ajouter un filtre, il est visualisable en Figure 33. Il est possible avec la zone de texte du haut de donner un nom au jeu de filtre ainsi créé pour mieux le reconnaître.

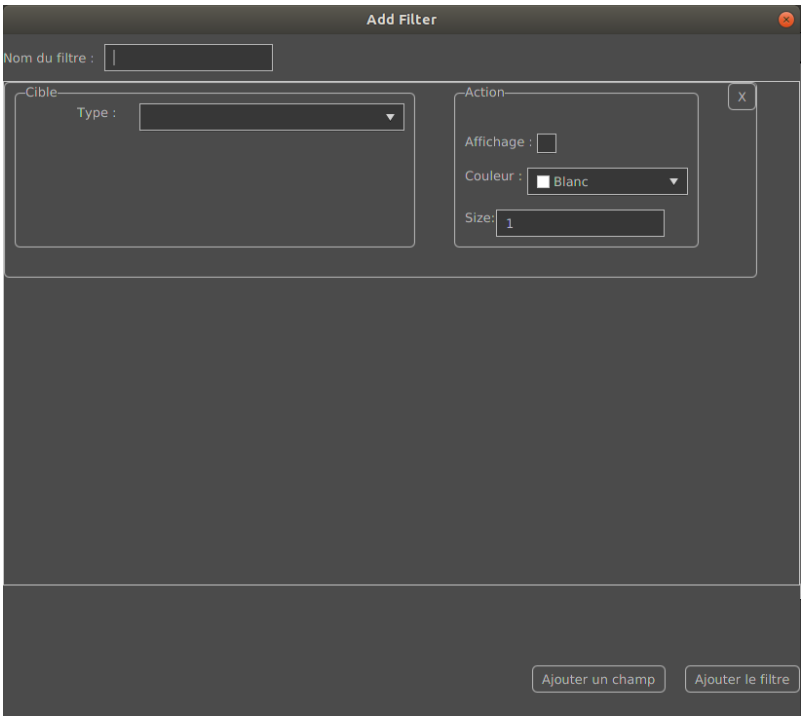


FIGURE 33 – Panneau d'ajout de filtres

La partie nommée type permet de choisir le type ciblé par le filtre. Il est possible d'empêcher l'affichage du type ciblé en cliquant sur la case à cocher ce qui empêche le choix d'une couleur et d'une taille. Il y a possibilité de choisir la taille et la couleur d'affichage. Il est à noter que la taille de 1 correspond à la taille par défaut, une taille de 0,5 un objet deux fois plus petit et 2 un objet deux fois plus grands que la valeur par défaut. Le bouton d'ajout de champ permet d'ajouter une nouvelle ligne pour ajouter plusieurs filtres dans un groupe. Puis le bouton ajouter le filtre permet d'ajouter sur le panneau principal le groupe de filtres.

V.3.b. Les analyses

Lors d'un clique sur le menu analyse il est possible de choisir entre deux analyses : les véhicules en circulation et les personnes en activité. Le premier permet de visualiser un graphique avec en abscisse le temps et en ordonnée le nombre de véhicules en circulation au cours du temps et pour le second l'ordonné correspond au nombre de personnes en activité au cours du temps. Nous pouvons voir en Figure 34 le panneau de l'analyse du nombre de véhicules en circulation. Nous pouvons visualiser le graphique correspondant à toutes les voitures par défaut. Avec les boutons du bas il est possible de cliquer sur "-" pour enlever la courbe et sur "+" pour en ajouter une autre en fonction du type.

Sur l'image en Figure 35 il est possible de visualiser la courbe pour les véhicules dont le type est bus.

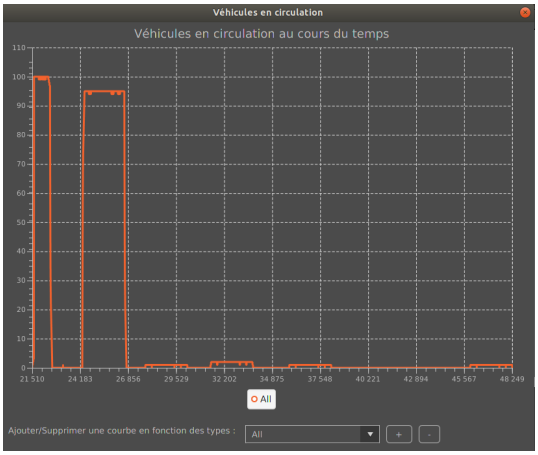


FIGURE 34 – Panneau des analyses avec une seule courbe

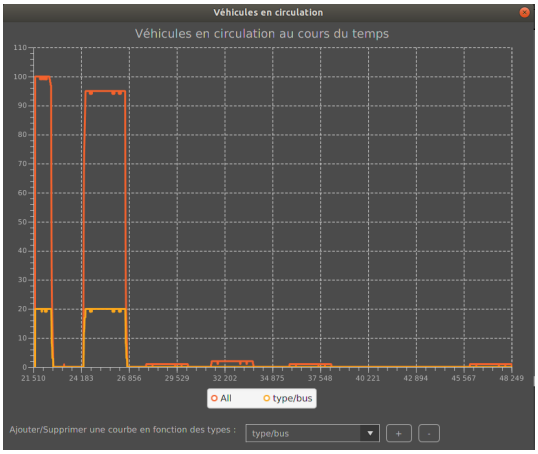


FIGURE 35 – Panneau des analyses avec deux courbes

V.3.c. Les informations détaillées

Le panneau à droite de la fenêtre de visualisation est dédié à l'affichage des caractéristiques des objets de la simulation. Sur le panneau de visualisation, il suffit de cliquer sur une entité (véhicule, agent, centre d'intérêt, axe ou noeud de transport) pour l'ajouter sur la panneau de droite en tant qu'élément observé. Ses principales caractéristiques sont alors affichées et continuellement mises à jour avec la date actuelle de la visualisation. Il suffit ensuite d'effectuer un clic droit et de cliquer sur le bouton fermer dans le menu contextuel pour supprimer un élément du panneau de droite.

V.3.d. Raccourcis clavier

L'application contient des raccourcis clavier. Ces raccourcis permettent de simplifier l'utilisation de l'application en utilisant moins la souris pour ceux qui préfèrent ne pas l'utiliser. Ces raccourcis sont détaillés dans le panneau correspondant dans les aides de l'application. Nous ne pouvons pas détailler quelle touche permet quelle action car les touches peuvent changer en effet l'application est faite pour les changer facilement dans le code. Mais nous pouvons en détailler les fonctionnalités. La première permet de zoomer et dézoomer l'affichage. Une seconde est de pouvoir augmenter le pas manuel de 1 ou de le diminuer. La suivante permet d'augmenter et de diminuer la vitesse de défilement automatique de 1. La quatrième permet de mettre en pause et de l'enlever. Pour connaître la touche ou les touches associées à chaque fonctionnalité se référer à la page correspondante dans l'application(aides/Raccourcis clavier).

V.4. Description des fichiers d'entrées

Comprendre le fonctionnement du parser et en particulier son mode opératoire peut être très utile pour bien maîtriser la création des fichiers de configuration du parser en json. Celle-ci a été expliqué en détail dans la Partie III.3., dédiée aux changements de conception sur le parser, dans laquelle l'exploitation du fichier de configuration a été détaillée. Si vous n'êtes pas familier avec ce fichier, il est toujours possible de générer un fichier de configuration à partir d'un modèle.

Le fichier de configuration d'exemple en Figure 36 va nous servir à détailler chacun des champs qui y sont présents. Tout d'abord, encadré en bleu, on retrouve les champs permettant d'indiquer les propriétés globales ainsi que les emplacements de tous les autres éléments utiles à la traduction :

ScenarioName Le nom du scénario que vous souhaitez importer, purement à titre indicatif.

InputPath Indique le chemin d'accès vers les fichiers d'entrée qui sont censés contenir toutes les données de la simulation. Dans le cas de MatSim il s'agit des fichier xml comme ceux que l'on retrouve en Figure 37 et 38.

PackageName Indique le nom du package du modèle que vous souhaitez utiliser pour la génération des objets du scénario. C'est ce modèle qui sera utilisé par le l'interface graphique pour exploiter les données et les afficher. Si vous souhaitez utiliser votre propre scénario, c'est ici qu'il faut indiquer le package correspondant. En revanche si vous voulez utiliser le modèle de base, indiquer "MatsimModel", modèle de MatSim par défaut.

FIGURE 36 – Exemple de fichier de configuration json pour le parser avec indications couleur

Unit Indique l'unité utilisée pour l'échelle, mais celle-ci est purement indicative, car la gestion métrique n'a pas encore été intégrée dans l'application.

ParserPackage Indique le package du parser que vous souhaitez utiliser. Si vous importez un parser personnalisé, vous devez indiquer son package ici. En revanche si vous voulez utiliser le parser par défaut de l'application, précisez "MatSimParser", parser par défaut pour traduire les fichiers de MatSim.

ScenarioProcessor Indique la classe du processeur chargé d'effectuer les opérations post-traduction. Ce dernier doit se trouver dans le package "ParserPackage", et celui utilisé par défaut est "MSScenarioProcessor" pour les données de MatSim de nouveau.

Ensuite on retrouve deux grands tableaux pour donner des précisions sur chacun des éléments à traduire : on a un tableau pour les éléments obligatoires en jaune et un pour les éléments facultatifs en orange (un seul ici). Comme expliqué plus tôt III.3., une instance de parser est créée pour chaque type d'élément (voitures, personnes, événements...). Il faut donc préciser les modalités de traduction pour chaque élément, et pour l'exemple, nous allons nous concentrer sur l'élément "Person" encadré en rouge avec son fichier xml associé en Figure 37. On retrouve les champs suivants :

InputFile Indique le nom du fichier d'entrée qui doit être lu pour générer les personnes, ici il s'agit du fichier "plans100.xml" que l'on retrouve en Figure 37. Ce fichier doit être contenu dans le dossier indiqué par le chemin "InputPath" que l'on a défini dans l'encadré bleu.

ModelName Indique le nom de la classe du modèle à instancier pour générer chacun des objets de type "Person". Cette classe doit être aussi présente dans le package "PackageName" précisé dans l'encadré bleu.

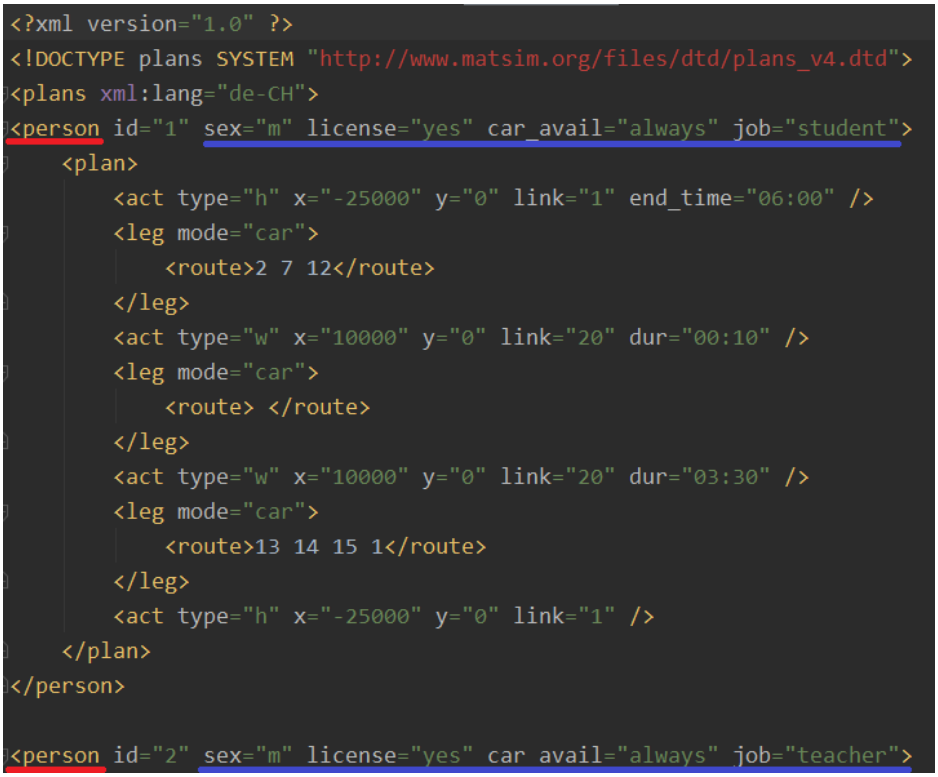
Path Indique le nom de la balise qui doit être identifiée pour instancier l'élément "Person". Ici il s'agit de la balise "person" soulignée en rouge dans le fichier en Figure 37, ce qui signifie que la classe "ModelName=MSPerson" sera instanciée à chaque fois que cette balise sera rencontrée pour créer un nouvel objet dans le modèle.

SuperModelName Indique le nom de la super classe dont hérite la classe concrète "ModelName=MSPerson" selon l'abstraction du modèle détaillée dans la partie des changements de conception consacré au modèle.

ParserName Indique le nom de la classe du parser qui doit être instanciée pour traduire et générer tous les éléments de type "Person" ici. Encore une fois, elle doit être contenue dans le package "ParserPackage" qui lui aussi a été défini dans l'encadré bleu.

Types Permet de préciser tous les attributs contenu dans les balises "Path" du fichier "InputFile" que vous souhaitez gérer comme des types. Ici tous les attributs du tableau que l'on retrouve souligné en bleu dans le fichier xml correspondant, sont des attributs qui seront identifiés comme des types dans la simulation, et sur lesquels vous aurez des leviers d'action pour l'application des filtres et des analyses dans l'interface graphique. Par exemple vous pourrez filtrer les personnes pour n'afficher que celles qui respectent "job=teacher" ou encore établir vos analyses en séparant "sex=f" ou "sex=m" dans notre exemple ici.

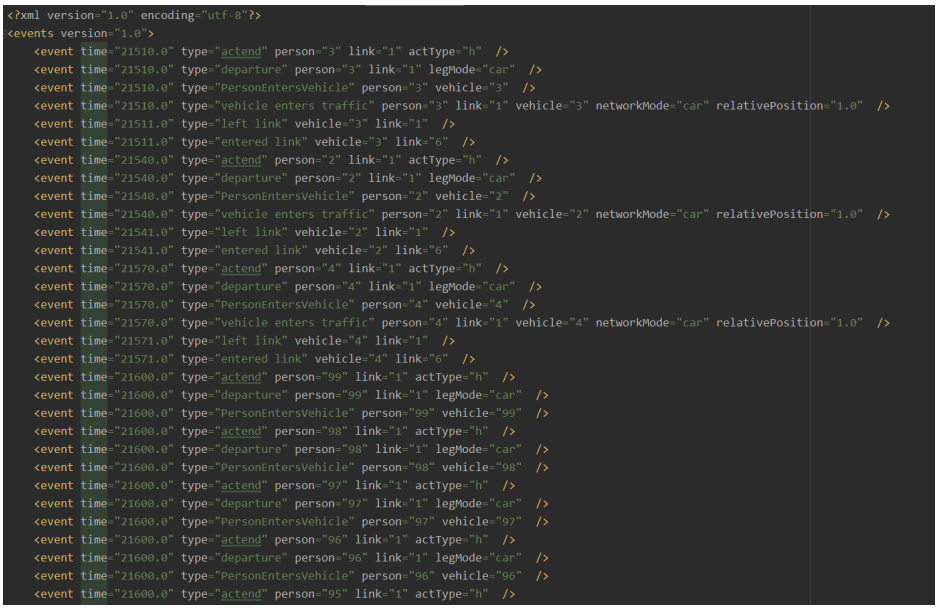
Voici donc toutes les indications pour rédiger correctement votre fichier de configuration. Pour ce qui est des fichiers d'entrée, il y en a un autre exemple en Figure 38, qui représente ici les événements selon MatSim. Il vous est possible de les modifier en respectant et/ou en adaptant les indications du fichier de configuration mais si vous souhaitez utiliser des fichiers d'entrée avec des formats complètement différents, il faudra sûrement envisager d'implémenter votre propre parser.



```xml
<?xml version="1.0" ?>
<!DOCTYPE plans SYSTEM "http://www.matsim.org/files/dtd/plans_v4.dtd">
<plans xml:lang="de-CH">
<person id="1" sex="m" license="yes" car_avail="always" job="student">
    <plan>
        <act type="h" x="-25000" y="0" link="1" end_time="06:00" />
        <leg mode="car">
            <route>2 7 12</route>
        </leg>
        <act type="w" x="10000" y="0" link="20" dur="00:10" />
        <leg mode="car">
            <route> </route>
        </leg>
        <act type="w" x="10000" y="0" link="20" dur="03:30" />
        <leg mode="car">
            <route>13 14 15 1</route>
        </leg>
        <act type="h" x="-25000" y="0" link="1" />
    </plan>
</person>

<person id="2" sex="m" license="yes" car_avail="always" job="teacher">
```

FIGURE 37 – Fichier d'entrée "plan100.xml" mentionné dans l'encadré rouge de la Figure 36, et qui contient toutes les données pour les éléments du type "Person"



```xml
<?xml version="1.0" encoding="utf-8"?>
<events version="1.0">
    <event time="21510.0" type="actend" person="3" link="1" actType="h" />
    <event time="21510.0" type="departure" person="3" link="1" legMode="car" />
    <event time="21510.0" type="PersonEntersVehicle" person="3" vehicle="3" />
    <event time="21510.0" type="vehicle enters traffic" person="3" link="1" vehicle="3" networkMode="car" relativePosition="1.0" />
    <event time="21511.0" type="left link" vehicle="3" link="1" />
    <event time="21511.0" type="entered link" vehicle="3" link="6" />
    <event time="21540.0" type="actend" person="2" link="1" actType="h" />
    <event time="21540.0" type="departure" person="2" link="1" legMode="car" />
    <event time="21540.0" type="PersonEntersVehicle" person="2" vehicle="2" />
    <event time="21540.0" type="vehicle enters traffic" person="2" link="1" vehicle="2" networkMode="car" relativePosition="1.0" />
    <event time="21541.0" type="left link" vehicle="2" link="1" />
    <event time="21541.0" type="entered link" vehicle="2" link="6" />
    <event time="21570.0" type="actend" person="4" link="1" actType="h" />
    <event time="21570.0" type="departure" person="4" link="1" legMode="car" />
    <event time="21570.0" type="PersonEntersVehicle" person="4" vehicle="4" />
    <event time="21570.0" type="vehicle enters traffic" person="4" link="1" vehicle="4" networkMode="car" relativePosition="1.0" />
    <event time="21571.0" type="left link" vehicle="4" link="1" />
    <event time="21571.0" type="entered link" vehicle="4" link="6" />
    <event time="21600.0" type="actend" person="99" link="1" actType="h" />
    <event time="21600.0" type="departure" person="99" link="1" legMode="car" />
    <event time="21600.0" type="PersonEntersVehicle" person="99" vehicle="99" />
    <event time="21600.0" type="actend" person="98" link="1" actType="h" />
    <event time="21600.0" type="departure" person="98" link="1" legMode="car" />
    <event time="21600.0" type="PersonEntersVehicle" person="98" vehicle="98" />
    <event time="21600.0" type="actend" person="97" link="1" actType="h" />
    <event time="21600.0" type="departure" person="97" link="1" legMode="car" />
    <event time="21600.0" type="PersonEntersVehicle" person="97" vehicle="97" />
    <event time="21600.0" type="actend" person="96" link="1" actType="h" />
    <event time="21600.0" type="departure" person="96" link="1" legMode="car" />
    <event time="21600.0" type="PersonEntersVehicle" person="96" vehicle="96" />
    <event time="21600.0" type="actend" person="95" link="1" actType="h" />
```

FIGURE 38 – Un autre exemple de fichier d'entrée xml pour les éléments de type "Event"

A Tables

Table des figures

Liste des tableaux