Dassault Systèmes
120 rue René Descartes,
28290 Plouzané
Tuteur : Arthur Gauthier

ENSTA Bretagne
2 rue François Verny
29200 Brest

Rapport de projet de fin d'études :

Etude des standards émergents nécessaires à la voiture connectée.

Bichat Clément

Stage de 3ème année – Option Robotique

16/03/2020 – 03/08/2020

Table des matières

Remerciements 4
1 L'entreprise Dassault Systèmes 5
1.1 Présentation et histoire 5
1.2 Activités et produits 5
1.3 Environnement économique 6
2 Contexte et sujet de stage 7
3 Création de logiciels de développement de systèmes embarqués 8
3.1 Enjeux du développement des systèmes embarqués 8
3.2 Les produits développés 8
3.3 Le partenariat AUTOSAR 9
4 Etude technique de ROS 2 9
4.1 Présentation 9
4.2 ROS 2 et l'industrie automobile 12
4.3 Concepts de ROS 2 13
4.4 DDS : un middleware industriel dans ROS 2 14
4.5 Simulation avec ROS 2 15
4.6 Outils supplémentaires 16
5 Réalisation d'un prototype d'ADAS 16
5.1 Objectifs 16
5.2 Développement d'un démonstrateur 17
5.3 Génération de code 23
6 Communication avec une application AUTOSAR Adaptive 29
6.1 Contexte 29
6.2 Exécution d'une application AUTOSAR Adaptive dans un environnement de test 29
6.3 DOCKER et QEMU 29
6.4 Communication avec une application Adaptive 30
7 Retour d'expérience 31
7.1 L'ingénieur en systèmes embarqués 31
7.2 Bilan personnel 31
Glossaire 32
Bibliographie 32
Gestion des tâches : diagramme de Gantt 33
Code source du simulateur 33

Table des figures

Figure 1 : Les 11 secteurs d'activités ciblés 5
Figure 2 : Chiffre d'affaires des ventes de logiciel par secteur d'activité en 2018 6
Figure 4 : Chiffres d'affaires par région, en million d'euros 6
Figure 4 : Chiffre d'affaires par produit, en millions d'euros 6
Figure 5 : Le robot PR2, le premier fonctionnant sous ROS 9
Figure 6 : "l'équation ROS" 10
Figure 8 : Bras robotique Fanuc M710 10
Figure 7 : Humanoid Robonaut 2 10
Figure 9 : Robot d'enseignement TurtleBot3 11
Figure 10 : Véhicule de développement Apex 11
Figure 11 : Architecture logicielle de la conduite autonome avec ROS par BMW 12
Figure 12 : Le computation graph level 13
Figure 13 : Architecture ROS 2 14
Figure 14 : Fonctionnement de l'espace partagé 15
Figure 15 : Ville de Brest virtualisé avec OSM Importer 16
Figure 16 : Environnement de simulation automobile 16
Figure 17 : Scénario de la simulation 17
Figure 18 : Environnement de simulation dans Webots 17
Figure 19 : Architecture ROS 2 18
Figure 20 : Processus de prise de décision 21
Figure 19 : Les différentes phases d'un dépassement + le retour caméra avec la détection de lignes 22
Figure 20 : Détection d'objets avec l'algorithme de deep learning Yolo 23
Figure 21 : diagramme de définition des blocs du système « véhicule » 24
Figure 17 : Diagramme de bloc interne : ADAS 25
Figure 18 : Machine à états du block Decision 26
Figure 19 : Machine à états du bloc Overtaking management 26
Figure 25 : Environnement d'exécution d'une application Adaptive sous Windows 10 29

Remerciements

Ce stage ne se serait pas aussi bien déroulé sans l'accueil, la gentillesse et l'aide de l'équipe de AUTOSAR Builder de Dassault Systèmes.

Mes premiers remerciements s'adressent à Arthur GAUTHIER, responsable de l'équipe pour son accueil et son encadrement quotidien.

Je tiens aussi à remercier Nolween MADEC qui m'a également beaucoup apporté et avec qui il a été très plaisant de travailler au quotidien.

Je remercie également l'ensemble de l'équipe avec qui j'ai pu travailler, Samuel DEVULDER, Luc FOREST et Fabien AILLERIE pour leur expertise technique.

Travailler malgré les conditions particulières imposées par le confinement au côté de ces personnes a été particulièrement enrichissant.

Enfin je tiens à remercier mon professeur référent Luc JAULIN qui m'a suivi tout au long du stage.

1 L'entreprise Dassault Systèmes

1.1 Présentation et histoire

Dassault Systèmes est un éditeur de logiciels créé en 1981 par une équipe de Dassault Aviation ayant l'idée d'informatiser la conception 3D d'avions. A sa création, Dassault systèmes lance CATIA un logiciel de conception assistée par ordinateur.

Un partenariat avec la société IBM lui permet ensuite de conquérir de nombreux clients dans différents secteurs industriels : l'aviation, l'automobile, l'industrie des biens de consommation, l'industrie navale …

Elle développe et rachète ensuite de nombreux logiciels de conception 3D, de maquettisme et de gestion de projet afin d'enrichir son produit phare CATIA.

En 2019, Dassault systèmes est le deuxième éditeur de logiciel européen en terme de chiffre d'affaire, compte plus de 19 000 employés dans le monde répartis dans plus de 180 agences et 117 pays.

1.2 Activités et produits

Les produits de Dassault systèmes se sont diversifiés depuis le lancement de CATIA pour répondre aux besoins des clients industriels. Chaque produit est lié à un des onze secteurs industriels visés par la stratégie de l'entreprise. Les plus important en terme de ventes sont :

- CATIA (conception assisté par ordinateur)
- SolidWorks (conception assisté par ordinateur)
- DELMIA (conception, modélisation et simulation d'une entreprise manufacturière)
- SIMULIA (simulation du comportement d'un produit)
- ENOVIA (gestion du cycle de vie des produits et gestion de projets)

Figure 1 : Les 11 secteurs d'activités ciblés

En 2012, l'entreprise lance une plateforme : la 3DExperience regroupant l'ensemble de ces logiciels disponibles en version « Cloud » ou « On premise » (serveurs installés chez le client). Elle offre ainsi un outil unique et personnalisable pour l'ensemble des corps de métiers de l'entreprise : des différents secteurs de l'ingénierie aux services ventes, comptabilité et marketing.

On observe néanmoins sur la figure 2, que Dassault Système dépend de manière très importante de ses secteurs historiques : l'industrie des transports et le secteur aéronautique.

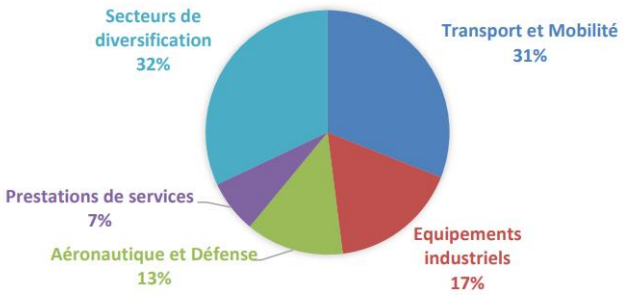


Figure 2 : Chiffre d'affaires des ventes de logiciel par secteur d'activité en 2018

Les revenus de l'entreprise ont augmenté de 15% avec un chiffre d'affaire qui se répartit entre les différentes parties du monde. La croissance du chiffre d'affaire a été portée par les ventes de licences de la plateforme 3DExperience, avec une progression assez faible des ventes des produits phares CATIA et SOLIDWORKS.

in MEUR	1Q19	1Q18	Growth	Growth ex FX
Americas	264.9	209.6	+26%	+17%
Europe	364.9	325.0	+12%	+10%
Asia	225.5	200.6	+12%	+9%
Software revenue	855.3	735.1	+16%	+12%

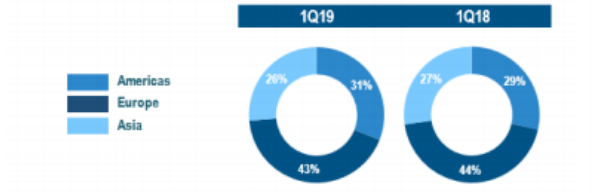


Figure 4 : Chiffres d'affaires par région, en million d'euros

in MEUR	1Q19	1Q18	Growth	Growth ex FX
CATIA SW	270.1	250.7	+8%	+6%
ENOVIA SW	92.3	74.7	+24%	+19%
SOLIDWORKS SW	191.4	169.9	+13%	+5%
Other SW	301.5	239.8	+26%	+21%
Services	103.6	83.5	+24%	+19%
Total revenue	958.9	818.7	+17%	+13%

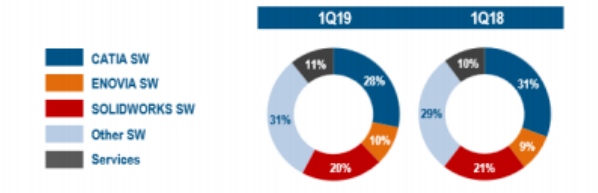


Figure 4 : Chiffre d'affaires par produit, en millions d'euros

1.3 Environnement économique

Dassault Systèmes a réalisé de nombreux partenariats avec de grandes sociétés tels que IBM, ABB (leader mondiale des technologies de l'automatisation), BHP (groupe minier). Elle a acquis et pris des parts dans plusieurs entreprises dans des domaines variées afin d'étoffer son portefeuille de produits avec notamment Exalead (société qui développe des logiciels à base de moteur de recherche), Outscale (société proposant des solutions clouds) , Medidata (société qui développe des logiciels médicaux)…

2 Contexte et sujet de stage

J'ai réalisé mon stage sur le site de Dassault Systèmes basé à Plouzané (29) dans l'équipe AUTOSAR Builder, en télétravail en raison du contexte de la crise sanitaire liée au COVID-19. Cette équipe est composée de neuf ingénieurs spécialisés en informatique embarquée automobile et particulièrement sur le standard AUTOSAR (Automotive Open System Architecture). AUTOSAR est une architecture logicielle standardisée et ouverte pour les unités de contrôle électronique des véhicules.

Cette équipe répartie sur 3 sites (Plouzané, Nantes et Nancy) est en charge du développement des logiciels AUTOSAR Builder et de la solution Embedded Software Producer.

Le secteur de l'informatique et de l'électronique embarqué automobile est en plein développement avec l'impulsion des sujets tels que le véhicule autonome et l'électrification des motorisations. Des nombreux standards et protocoles émergent entre les différents acteurs industriels pour accélérer et fiabiliser les développements. Dans ce cadre, je devais évaluer les différents standards et technologies émergentes afin de proposer et étudier leur intégration aux logiciels développés par Dassault Systèmes.

J'ai ainsi fait l'étude de la deuxième version de ROS (Robot Operating System). J'ai ensuite réalisé un démonstrateur en m'appuyant sur cette technologie afin d'avoir une base de travail pour son intégration dans les produits développés par Dassault Systèmes. Sur la dernière période de mon stage, j'ai pu appréhender et travailler sur les communications d'une application basée sur le standard AUTOSAR Adaptive.

Tout au long de ces six mois, j'ai été intégré à l'équipe. Je participais aux réunions d'avancement hebdomadaires. J'ai réalisé une présentation de mon démonstrateur et d'une étude technique de ROS 2 aux ingénieurs. J'étais également en discussions régulière avec mon tuteur afin d'ajuster les développements et leur pertinence. Je pouvais profiter des connaissances et de l'expérience des autres membres de l'équipe lorsque j'avais des questions ou des problèmes dans mon avancement.

.

3 Création de logiciels de développement de systèmes embarqués

3.1 Enjeux du développement des systèmes embarqués

Les produits embarqués dans des systèmes industriels critiques doivent répondre à de nombreuses exigences de :

- Temps réel
- Sureté de fonctionnement : fiabilité, disponibilité…
- Respect de protocoles et normes relatives au domaine d'application

Ces systèmes deviennent de plus en plus complexes à développer afin de répondre à l'ensemble de ces contraintes. Ainsi, la maitrise de l'ensemble de l'architecture logicielle d'un système embarqué industriel demande de nombreuses connaissances, nécessite un temps et un coût de développement importants.

Afin d'optimiser le développement de leurs produits, de nombreuses entreprises travaillant à la création de systèmes embarqués « finis » font le choix d'utiliser des outils spécifiques afin de s'abstraire de la complexité de la programmation embarquée, de la diversité des plateformes et des protocoles.

Ces outils logiciels vont leur permettre de modéliser des systèmes avec une approche Model-based design, d'utiliser un environnement de développement, de test et d'une chaine de cross-compilation pour des cibles embarquées spécifiques. Cela permettra à des ingénieurs « Domain experts » avec une vision de plus haut niveau de créer et développer des produits innovants en se concentrant sur le côté applicatif de leur solution sans avoir l'expertise embarquée habituellement requise.

3.2 Les produits développés

L'équipe avec laquelle je travaillais développe des logiciels permettant la modélisation, la définition, la simulation et le déploiement de systèmes embarqués. Elle est spécialisée dans la conception de logiciels de développement de systèmes et de logiciels destinés à l'industrie automobile basées sur l'architecture AUTOSAR avec AUTOSAR Builder. AUTOSAR Builder est une suite ouverte et évolutive d'outils basée sur Eclipse permettant de concevoir et de développer des systèmes et des logiciels AUTOSAR.

En s'appuyant sur son expertise acquise avec le développement d'AUTOSAR Builder, elle a développé une solution web ESP (Embedded Software Producer) pour la production de logiciels embarqués multi-disciplines. Cette solution web – webapp - s'intègre à la plate-forme 3DEXPERIENCE qui est l'offre web based des applications commercialisées par Dassault Systèmes.

Des travaux sont également menés sur d'autres logiciels du portefeuille de Dassault Systèmes lorsque des compétences sur le développement de systèmes embarqués sont nécessaires.

3.3 Le partenariat AUTOSAR

Pour répondre au besoin d'harmonisation dans la conception d'application embarquées et à la complexité des architectures, le partenariat AUTOSAR est créé en 2003 entre différents industriels du secteur automobile dont notamment BMW, Bosch, Continental, Peugeot-Citroën et Volkswagen. Dassault systèmes est un acteur du consortium AUTOSAR et prend part aux discussions sur les orientations de différents projets en cours.

Dans l'objectif d'établir des standards, AUTOSAR fournit :

- Les spécifications des couches basses logiciels sous forme de modules afin de s'affranchir du matériel lors de la conception d'application. Ces spécifications s'appuient sur un découpage en trois couches : une couche applicative, une couche basse de drivers et un environnement d'exécution (runtime environment), faisant le lien entre les deux.
- Des interfaces de communication entre les composants.
- Une méthodologie de développement.

Le standard AUTOSAR Classic est utilisé depuis une dizaine d'années. Depuis 2017, une nouvelle plateforme AUTOSAR Adaptive est en cours de spécification permettant de répondre aux nouveaux enjeux des véhicules communicants et autonomes.

4 Etude technique de ROS 2

4.1 Présentation

4.1.1 Contexte

Le développement logiciel d'un système robotique est complexe et nécessite de nombreuses connaissances dans de nombreux domaines. Selon le système, il est en effet nécessaire de développer des drivers pour la gestion des actionneurs et des capteurs, de travailler sur des algorithmes de navigation, de traitement d'images, d'intelligence artificielle... Il faut également implémenter des protocoles de communication afin de faire communiquer l'ensemble des éléments du système.

Afin d'optimiser ces tâches et d'éviter de développer l'ensemble des modules à chaque nouveau projet, des chercheurs du laboratoire Willow Garage associés à l'université de Stanford, lors de leur travaux de recherche sur le robot PR2, introduisent une première version de ROS (Robot Operating System) en 2007. Ils définissent ROS comme un méta-système d'exploitation, un système entre un système d'exploitation et un middleware, qu'ils veulent réutilisable ensuite pour de nombreux robots.

Figure 5 : Le robot PR2, le premier fonctionnant sous ROS

ROS fournit concrètement :

- Un middleware créant un système de communication inter-processus
- Une API* afin de développer plus aisément
- Différents outils de simulation et de visualisation nécessaire au développement de systèmes robotiques
- Une plateforme communautaire où les chercheurs et industriels peuvent partager leurs travaux et connaissances

Figure 6 : "l'équation ROS"

Grâce à une abstraction des problématiques liées au développement matériel de plus bas-niveau, une gestion de la mémoire et des processus, ROS a permis d'accélérer et de standardiser le développement des robots.

Voici quelques robots fonctionnant sous ROS :

Figure 8 : Humanoid Robonaut 2

Figure 7 : Bras robotique Fanuc M710

Figure 9 : Robot d'enseignement TurtleBot3

Figure 10 : Véhicule de développement Apex

4.1.2 Historique : de ROS 1 à ROS 2 une version pour l'industrie

La première version de ROS connaît beaucoup de succès et est utilisée dans de nombreuses applications robotiques. Néanmoins certaines limites liées au contexte de son développement initial sont apparues :

- Une connectivité réseau de bonne qualité est nécessaire pour l'échange de données avec l'environnement extérieur au robot.
- Elle est destiné principalement à des applications de recherche.
- Le protocole de communication basé sur une architecture peer to peer centralisée n'est pas optimal et fiable. L'ensemble du système robotique peut être mis en défaut si ce point central connaît une défaillance.
- Peu de règles de bonnes pratiques et de normes ont été définies. Cela peut avoir certaines limites dans des développements d'applications plus critiques.
- Des plateformes matérielles assez puissantes sont souvent nécessaires pour faire fonctionner l'ensemble des systèmes des robots

De plus, les robots développés pour répondre à de nouveaux besoins doivent respecter de nouvelles exigences concernant :

- Les systèmes temps-réel
- Les systèmes industriels critiques et sécuritaires
- Les systèmes distribués

Afin de dépasser ces limitations et répondre à ces nouveaux besoins, une seconde version, ROS 2, est ainsi lancée en 2014 en parallèle de ROS en s'appuyant sur les même concepts de ROS.

ROS 2 se développe sans interférer avec ROS. Cette version est maintenant disponible pour différentes plateformes :

Architecture	Ubuntu Bionic	MacOS Sierra	Windows 10	Debian Stretch	OpenEmbedded
amd64	X	X	X	X	
arm64	X			X	
arm32	X			X	X

4.2 ROS 2 et l'industrie automobile

Les différents atouts de ROS 1 et plus récemment de ROS 2 ont intéressé différents acteurs industriels automobiles tels que BMW et BOSCH. De nombreux démonstrateurs ROS 1 / ROS 2 d'aide à la conduite (suivi de ligne, détection de piétons, régulateur de vitesse adaptatif, stationnement automatique) ont été développés par des constructeurs en lien avec des instituts de recherche. Ils ont pu s'appuyer sur la base de connaissance de la communauté ROS en reprenant des algorithmes de navigation, de détection d'environnement, de fusion de donnée… ainsi que l'ensemble des outils de visualisation et de simulation.

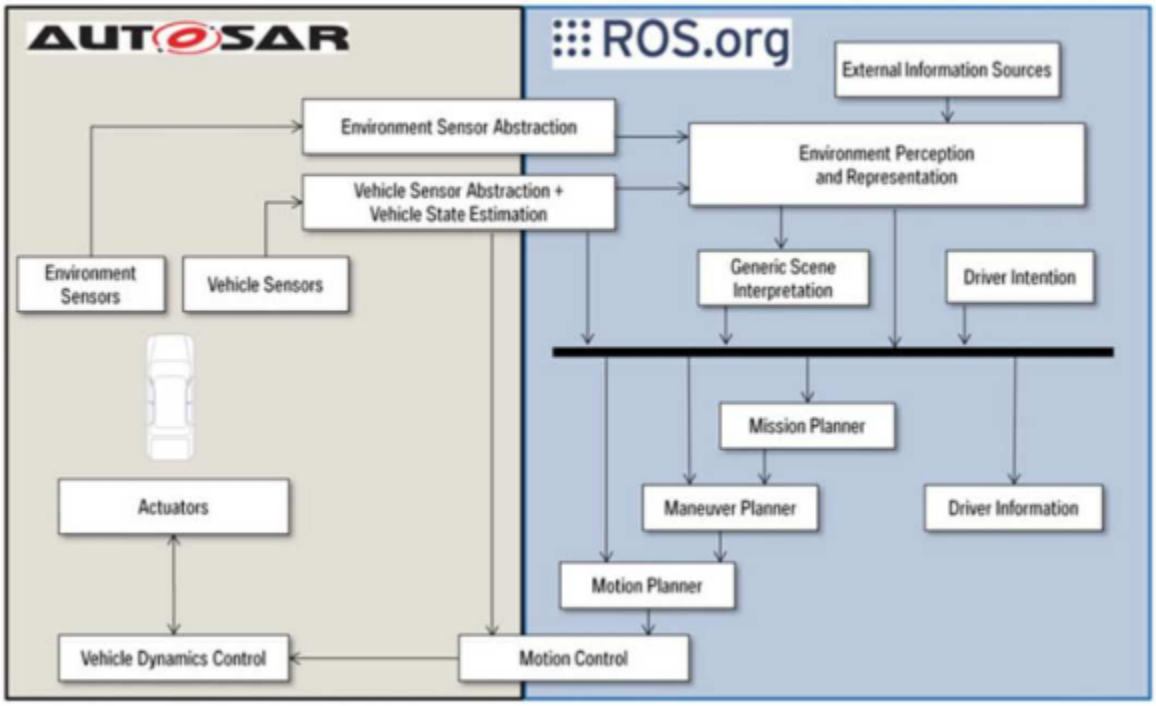


Figure 11 : Architecture logicielle de la conduite autonome avec ROS par BMW

Différents projets sont en cours afin de délivrer une version de ROS 2 modifiée ainsi que des briques applicatives répondant aux normes de sécurité et de fiabilité automobile (ISO 26262 notamment). Ces travaux permettront ainsi l'intégration d'applications ROS 2 dans des véhicules autonomes autorisés à circuler sur voies libres.
L'organisation à but non lucratif Autoware et la société APEX.AI spécialisées dans le développent logiciel de système de mobilité autonome communiquent beaucoup sur leurs travaux basés sur ROS 2.

4.3 Concepts de ROS 2

4.3.1 Le computation graph level

ROS 2 crée un réseau de nœuds qui vont effectuer ensemble des actions sur les données. Les nœuds vont échanger entre eux des messages par l'intermédiaire de topics ou de services.

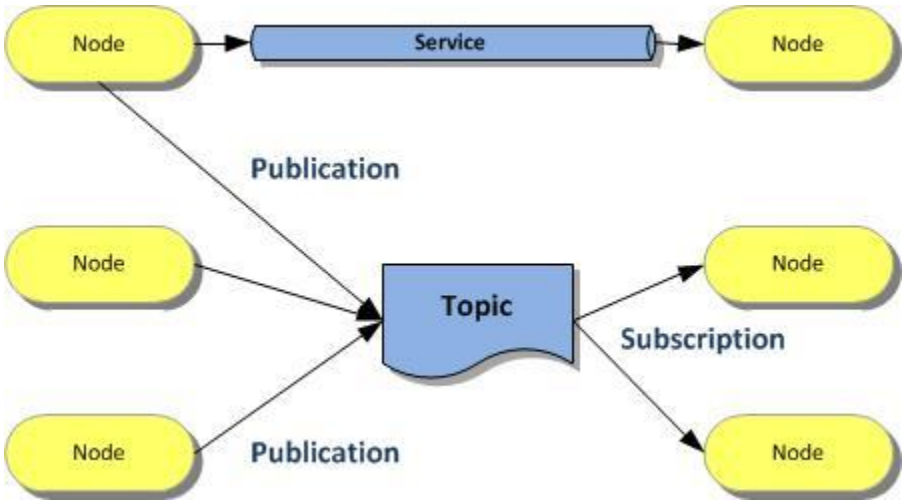


Figure 12 : Le computation graph level

4.3.2 Nœud

Les nœuds sont les algorithmes qui vont effectuer les opérations sur les données. On définit un nœud à petite échelle, on le dédit à une action particulière. Les nœuds sont isolés spatialement et temporellement. Un nœud peut être en charge du traitement des informations d'un capteur, de contrôler un moteur, de gérer l'affichage d'une interface graphique… Cela permet d'avoir une architecture logicielle modulaire et d'avoir des tâches séparées en isolant également les possibles défaillances. ROS 2 permet de lancer plusieurs nœuds dans un unique processus.

Afin d'avoir un plus grand contrôle sur l'ensemble des éléments composants le système robotique et ainsi améliorer la fiabilité, ROS 2 introduit le concept de managed nodes. Ces nœuds vont s'exécuter suivant une machine à états et se signaler à différents moments de leur exécution. Un composant extérieur va vérifier la bonne coordination des nœuds du système.

Les nœuds ainsi que certains fichiers de configurations sont répertoriées par fonctionnalités dans des packages.

4.3.3 Communication

La communication se réalise via des topics ou des services. Un topic est un bus de communication asynchrone où des nœuds publient des données tandis que d'autres s'abonnent à la lecture de ces données (fonctionnement *publish/subscribe*). Un service est un mécanisme de communication synchrone entre deux nœuds (fonctionnement *request/reply*). Chaque canal de communication est typé, les topics et les services échangent des données respectant une structure. Il est aisé de définir sa propre structure de message pour une utilisation spécifique.

ROS 2 introduit différentes qualités de service pour affiner la granularité des communications. Il est possible ainsi de les paramétrer afin d'avoir des échanges d'information fiables et sans perte de données, de privilégier plutôt la rapidité avec une tolérance aux erreurs de transmission ou de trouver un état entre les deux répondant aux besoins de l'application.

4.3.4 Compilation

Un projet ROS 2 est composé de plusieurs packages interdépendant entre eux. Il faut donc des outils pour gérer l'ordre de compilation des packages puis la compilation elle-même. ROS 2 fournit deux outils :

- **Un « Build tool » :** *Colcon*
  Cet outil va déterminer le graph de dépendance et invoquer le build system pour chaque package dans un ordre topologique.
- **Un « Build system »** : *Ament_cmake* pour un package développé en C++ et *Python Setup tools* pour un package développé en Python
  Cet outil va configurer, compiler et installer chaque package.

4.4 DDS : un middleware industriel dans ROS 2

Pour faire communiquer les différentes applications entre elles, il a été choisi lors de la spécification du système de communication de ROS 2, de se baser sur un protocole de middleware normalisé existant : DDS (Data Distribution Service).

Ce middleware est un composant logiciel qui va créer un réseau d'échange d'informations entre différentes applications quelques soient les caractéristiques des réseaux, des protocoles et des systèmes d'exploitation impliqués.

DDS se base sur un modèle **Data-Centric Publish-Subscribe**. Les différents composants du système publient et souscrivent à des données dans un espace partagé totalement distribué. Cette gestion évite ainsi les goulots d'étranglement et les points de défaillance unique.

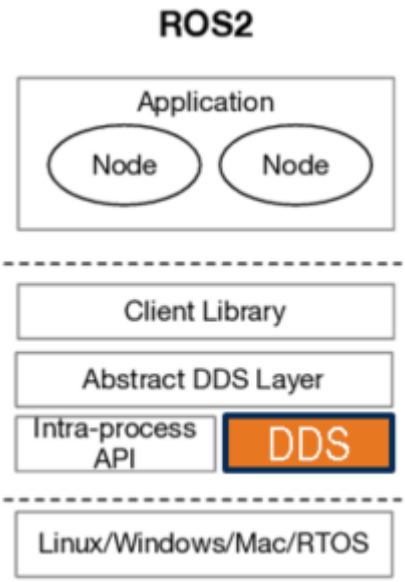


Figure 13 : Architecture ROS 2

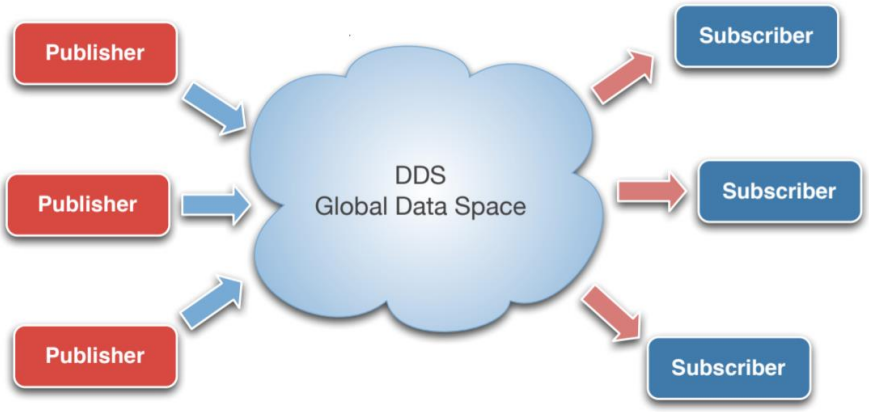


Figure 14 : Fonctionnement de l'espace partagé

Ce protocole de middleware introduit en 2004 par l'Object Management Group est derrière de nombreuses applications distribuées soumises à des contraintes de fiabilité, de rapidité et de déterminisme. Il est utilisé par les industries automobiles, aérospatiale, financière… Il existe différents versions open source et commerciales.

4.5 Simulation avec ROS 2

ROS 2 peut être interfacé avec plusieurs simulateurs. Ces simulateurs fournissent des environnements complets permettant de modéliser, de programmer, tester et simuler des robots. Certains sont open-source et la plupart ne sont pas spécifiques à l'environnement ROS. Les plus utilisés et accessibles sont Gazebo, CoppeliaSim (anciennement V-REP) et Webots.

J'ai choisi d'utiliser Webots pour sa compatibilité avec Windows 10. Webots est un simulateur open-source. Les robots peuvent être programmés via ROS et ROS2 à travers une API. En plus d'environnements de simulation robotique assez classiques, Webots offre des environnements et des véhicules dédiés à la simulation automobile.

Webots permet de créer des modèles réalistes de véhicules en se basant sur les paramètres d'Ackermann (pour la géométrie de la direction) et permet la définition précise de nombreux éléments tels le moteur, les transmissions… L'API est également assez riche, cela nous permet d'avoir un contrôle assez réaliste sur l'ensemble des éléments de la voiture.

De nombreux environnements pour simuler ces véhicules sont déjà développés : des centres urbains, des autoroutes... Des plugins supplémentaires ont été rajoutés pour améliorer les scénarios (OpenStreetMap Importer : création d'environnement à partir du monde réel, SUMO (Simulation of Urban Mobility) : simulation de trafic routier).

Figure 16 : Environnement de simulation automobile

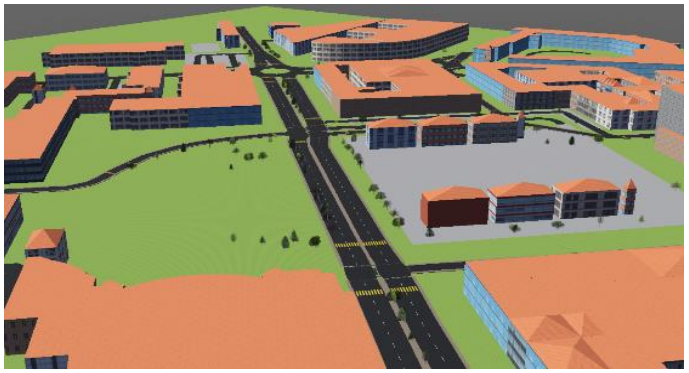


Figure 15 : Ville de Brest virtualisé avec OSM Importer

4.6 Outils supplémentaires

ROS 2 fournit un ensemble d'outils facilitant le développement, les plus utilisés sont :

- ROS2 Visualizer : outil permettant de visualiser les données des topics (données capteurs, vidéo, les modèles des robots, les coordonnées des transformés entre les repères…)
- ROS2 QT : outil graphique sous forme de plugins permettant la visualisation d'informations, l'introspection de l'architecture développée, l'envoi de commandes sur des topics… On peut également développer ses propres plugins pour les besoins de son application.
- ROS Command-line tools : il est aisé via une console de réaliser de nombreuses actions tels que l'exécution de nœuds, de retrouver des informations sur les topics, de vérifier l'état du système…

5 Réalisation d'un prototype d'ADAS

5.1 Objectifs

Afin d'appliquer mes recherches sur ROS 2, ma première tâche a été de proposer et de développer un prototype d'aide à la conduite automobile. J'ai utilisé le simulateur Webots, pour simuler une voiture sur une autoroute en ligne droite à 3 voies.

Ce démonstrateur devait servir de base de travail pour valider le fonctionnement des prototypes des plugins développés par l'équipe dans le logiciel Cameo. Il devait également me permettre de réfléchir à une proposition d'intégration d'un module dans le générateur de fichiers sources afin de prendre en compte une cible ROS 2.

Enfin, cette simulation a été l'occasion de tester le générateur de code en cours de développement pour des plateforme AUTOSAR Adaptive.

Figure 17 : Scénario de la simulation

Figure 18 : Environnement de simulation dans Webots

5.2 Développement d'un démonstrateur

5.2.1 Spécification

Cette aide à la conduite doit réaliser :

- Un suivi de ligne afin que la voiture reste au centre de sa voie
- Le dépassement des véhicules

Ce système a comme entrées :

- Les données des capteurs de distances placés à l'avant et sur les côtés du véhicule
- Les données de la centrale inertielle (angles de tangage, roulis et lacet)
- L'image de la caméra placée à l'avant

Et comme sorties :

- Une commande de vitesse
- Une commande d'angle de braquage des roues

J'ai réalisé l'ensemble de ce démonstrateur sur Windows 10, système d'exploitation privilégié pour les développements chez Dassault Systèmes.

5.2.2 Développements réalisés

5.2.2.1 Architecture ROS 2

J'ai ainsi développé une architecture logicielle ROS 2. Elle comprend 3 nœuds réalisant chacun une tâche précise :

- Interface Node : nœud passerelle entre Webots et l'architecture ROS 2 développée.
- Image processing Node : nœud de traitement d'images
- Decision Node : Un nœud de traitement de l'information et de décision

Ils échangent des informations entre eux via différents topics :

- /Image : flux d'images de la caméra
- /Yaw : données de la centrale inertielle de l'angle de lacet
- /right_distance_sensor : données de distance du capteur situé sur la droite du véhicule
- /left_distance_sensor : données de distance du capteur situé sur la gauche du véhicule
- /front_distance_sensor : données de distance du capteur situé à l'avant du véhicule
- /lane_follow_heading : consigne de l'angle de braquage en sortie du bloc de traitement d'images
- /cmd : consigne de la vitesse et de l'angle de braquage des roues à destination du véhicule

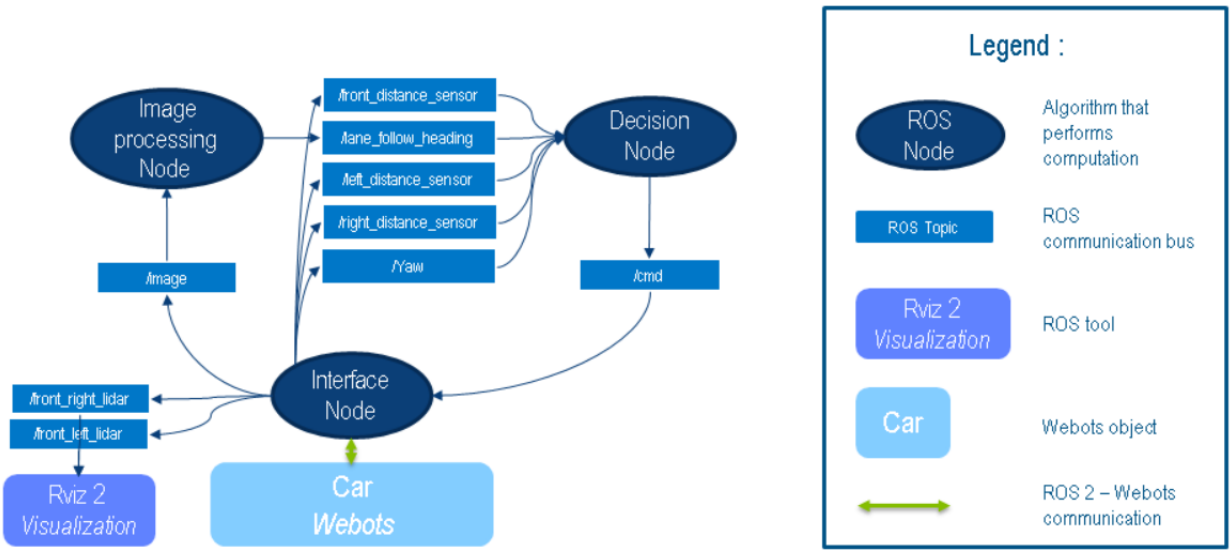


Figure 19 : Architecture ROS 2

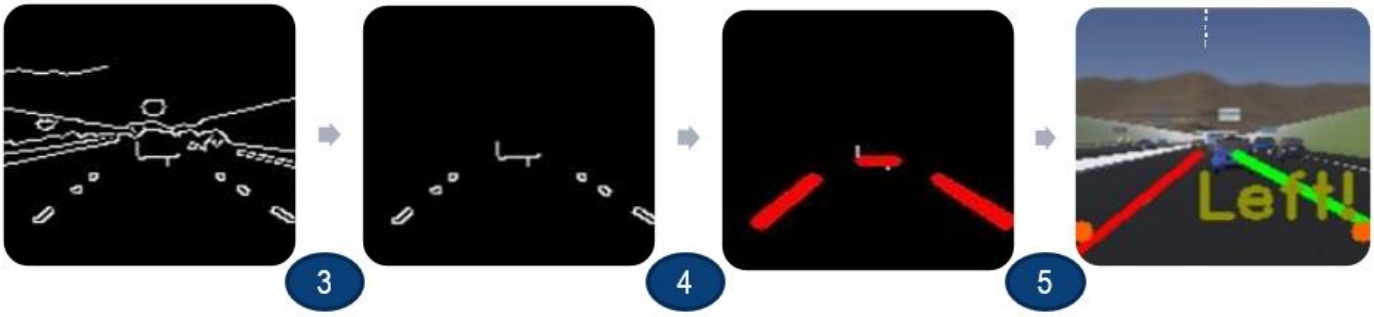
5.2.2.2 *Interface entre Webots et ROS 2*

Le développement d'un nœud est nécessaire pour faire l'interface entre les topics ROS 2 et les données du simulateur. En utilisant l'API* Python de Webots, on peut lire les données renvoyées par les capteurs du véhicule et lui envoyer des commandes.

5.2.2.3 *Traitement d'images : détection de lignes*

Le traitement d'image est réalisé à l'aide de la librairie OpenCV. Différentes étapes sont nécessaires pour détecter les lignes de manières performantes :

1. Passage de l'image en niveau de gris.
2. Application du filtre de Canny pour détecter les contours dans l'image.
   Le filtre de Canny permet de réduire le bruit grâce à un filtre Gaussien et retourne l'intensité des contours en appliquant un gradient en X et Y. Un seuillage permet ensuite de garder les contours ayant l'intensité la plus élevée.
3. Application d'un masque pour ne garder que la région d'intérêt.
   La région d'intérêt est la voie où se trouvent les lignes.
4. Application d'une transformée de Hough afin de détecter les droites.
   La transformée de Hough repose sur le paramétrage d'une droite par ses coordonnées polaires, un angle θ et une distance ρ. Pour chaque point de coordonnées (x, y) des contours, l'algorithme de Hough s'appuie sur la projection dans un plan des coordonnées polaires (dans une matrice accumulatrice) de toutes les droites passant par ce point. Chaque droite incrémente l'élément correspondant de la matrice. Ainsi, les points de la matrice dont la valeur est la plus élevée correspondent à un alignement de points dans l'image. L'algorithme en sortie renvoie les droites à partir d'un seuil ajustable.
5. Filtrage des droites.
   On ne garde que les lignes qui pourraient correspondre à celles de la route. Un filtrage sur les coefficients directeurs des droites dans l'image permet de supprimer les droites horizontales et de séparer les lignes droites et gauches. En faisant la moyenne des abscisses des droites retenues, on peut tracer la ligne de droite et de gauche de la voie.



Une comparaison de l'abscisse d'un point de la droite de gauche et celle de droite avec deux valeurs seuils calibrées lors de tests permet de déduire que le véhicule ne se trouve plus au centre de la voie. Le résultat de ces comparaisons permet de lui envoyer une consigne d'angle de braquage afin de le replacer au centre de la voie.

Il a été nécessaire de tester différents jeux de valeurs de seuil et différents paramètres dans les fonctions de traitement d'images avant d'avoir un suivi de voie performant.

5.2.2.4 Dépassement

Lorsqu'un autre véhicule est détecté à l'avant et se trouve à une distance de moins de 15 mètre, le véhicule rentre dans une phase de dépassement. Une consigne d'angle de braquage de 0.2 degrés pendant une seconde à droite ou à gauche selon la situation est envoyée. Un suivi de cap est réalisé avec un correcteur proportionnel. La cap est mesuré par la centrale inertielle placée dans le véhicule. Une consigne d'angle nulle est envoyée ensuite pendant une seconde pour replacer le véhicule droit dans sa ligne.

5.2.2.5 Prise de décision

Le véhicule se trouve par défaut dans un mode de suivi de voie, sa vitesse est supérieure aux autres véhicules d'une quinzaine de kilomètre par heure. Lorsqu'une voiture est détectée devant, on vérifie dans quelle ligne se trouve le véhicule. Selon la ligne, on effectue ensuite une vérification de la présence d'un autre véhicule sur les côtés, puis si aucun véhicule n'est présent un dépassement est effectué par la gauche ou par la droite. Lorsque le dépassement est terminé, le véhicule repasse en mode suivi de voie. Lorsque le dépassement est impossible, le véhicule freine en gardant une distance de sécurité avec celui le précédant.

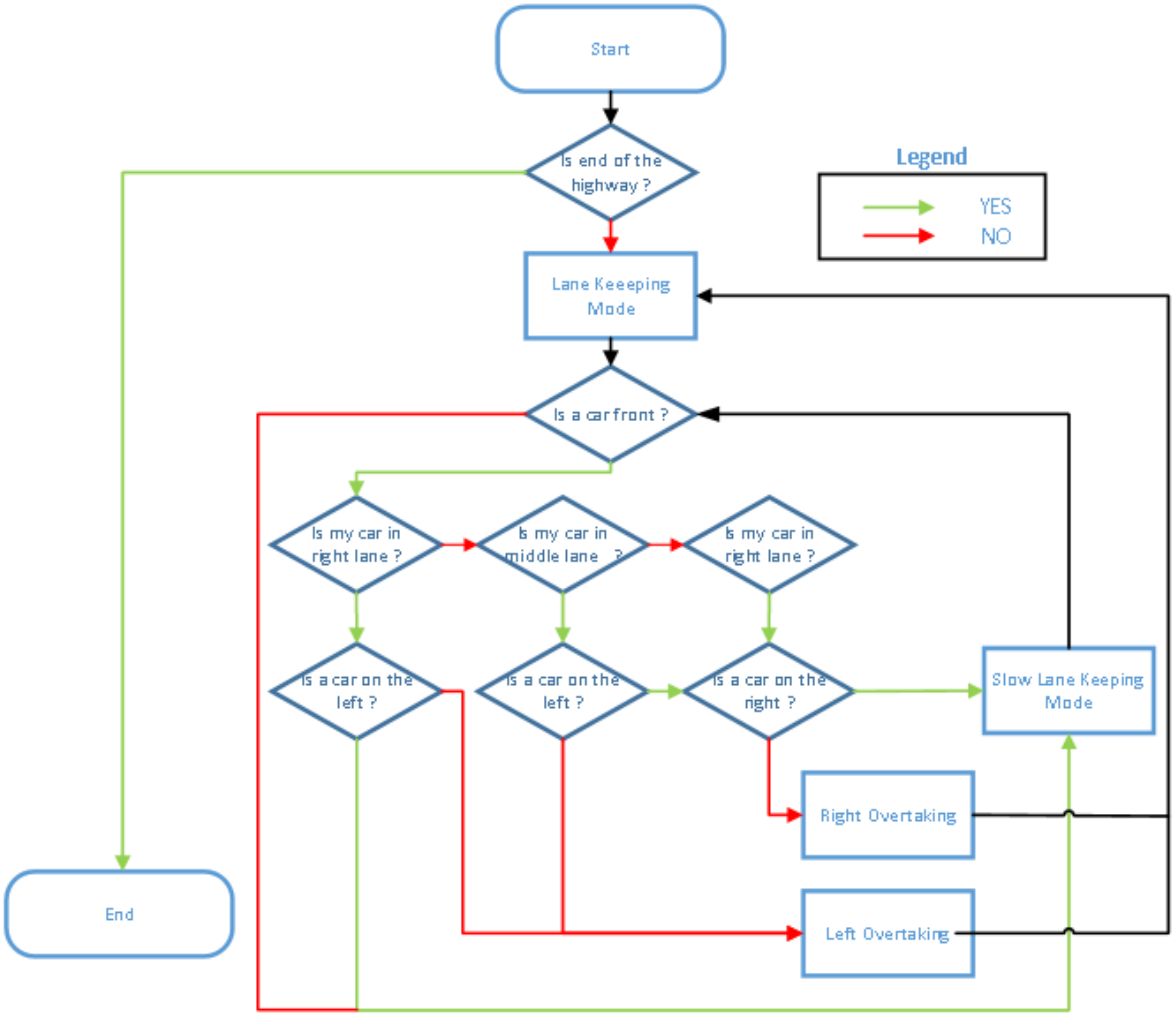


Figure 20 : Processus de prise de décision

5.2.3 Configuration du simulateur

Il a été nécessaire de configurer le simulateur, l'environnement et l'échange de données entre Webots et le nœud d'interface pour avoir une simulation fluide. Les paramètres importants sur lesquels je suis intervenu :

- **Le pas de simulation** : 40 millisecondes.
  Le pas de simulation va fixer la granularité de la simulation, c'est l'intervalle de temps entre deux calculs de l'ensemble des paramètres de la simulation et des paramètres physiques. Il définit le pas du temps simulé, ainsi plus cette valeur est faible, plus la simulation devient précise mais peut rendre la simulation plus lente car nécessite un nombre de calculs importants. Il faut trouver un compromis entre la précision de la simulation et sa vitesse. La valeur du temps simulé peut prendre beaucoup de retard par rapport au temps réel et la fluidité peut se dégrader rapidement. Il a été nécessaire de synchroniser ce pas de simulation avec les différents timers ROS 2.
- **Le nombre d'images affichées par seconde** : 25 images par seconde.
- **Taille de l'image** et **fréquence** de la camera : 256 * 128 pixels.

Ces valeurs dépendent de la configuration matérielle de l'ordinateur sur lequel est lancé la simulation et de certains paramètres de l'application. Les performances de la simulation étaient satisfaisantes, des tests avec une meilleure carte graphique et une configuration des paramètres plus précise auraient pu encore améliorer la simulation mais cela n'était pas forcément nécessaire dans le cadre de mon démonstrateur.

5.2.4 Méthodologie et résultats

Cette démonstration avait pour objectifs la mise en place des différents concepts de ROS 2 dans une application automobile et de préparer les étapes suivantes de mon travail. Je l'ai développé en plusieurs phases en travaillant au départ sur chaque nœud séparément. Je les testais à chaque fois en lançant le véhicule dans l'environnement de simulation. Je me suis aidé des différents outils proposés par ROS 2 qui m'ont permis de déboguer plus rapidement. Néanmoins, tous ces outils ne sont pas encore complétement fonctionnels sur le système d'exploitation Windows 10.

Les algorithmes sont performants dans le cadre de la simulation. Le suivi de voies fonctionne bien, les lignes sont bien détectées et le véhicule reste au centre de la voie. De même, dès qu'un véhicule se trouve devant, il est doublé rapidement.

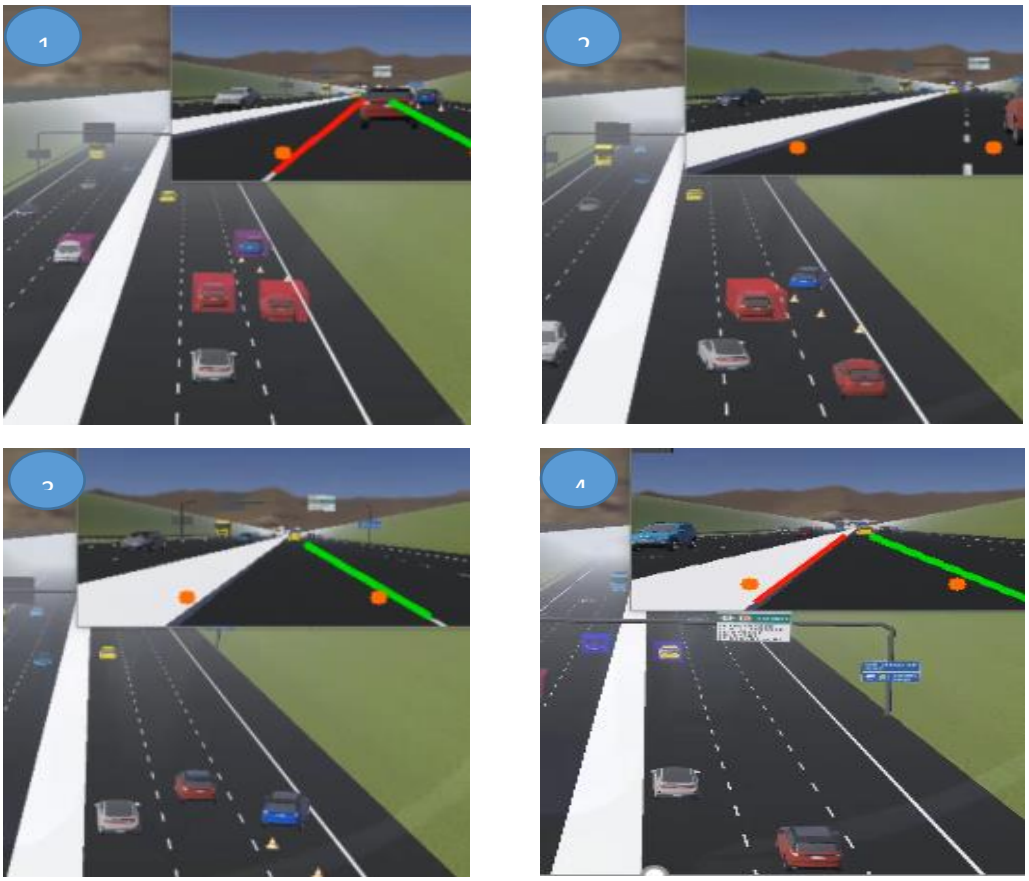


Figure 21 : Les différentes phases d'un dépassement + le retour caméra avec la détection de lignes

Les algorithmes mis en place dans ce démonstrateur restent relativement basiques d'un point de vue robotique. De nombreuses améliorations étaient possibles mais elles dépassaient le cadre du démonstrateur et demandaient un temps plus important de développement. J'aurais pu notamment travailler sur :

- Des d'algorithme de suivi de ligne et de dépassement plus robustes
- Implémentation d'algorithmes d'apprentissage profond pour ajouter une fonction de détection d'objets en temps réel (utilisation des modèle pré-entrainé YOLO)

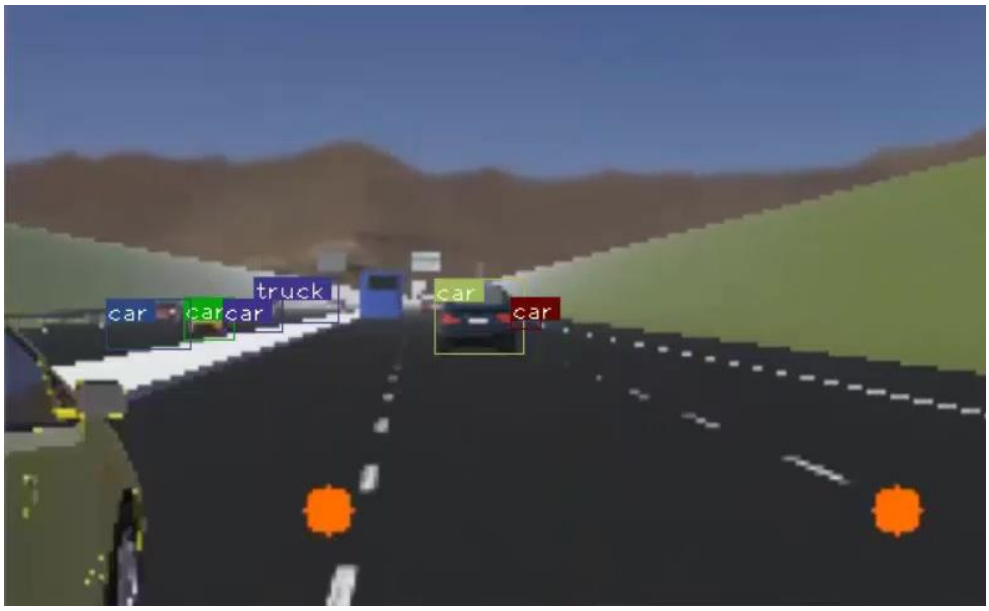


Figure 22 : Détection d'objets avec l'algorithme de deep learning Yolo

5.3 Génération de code

5.3.1 Contexte

Dassault Système a racheté l'entreprise No Magic qui éditait le logiciel Cameo. Cameo est un logiciel de modélisation système. Il permet la définition et la modélisation de systèmes complexes. Il offre des fonctionnalités afin de réaliser de nombreux types de diagrammes et notamment des modèles répondant au standard SysML. Ce logiciel est utilisé par les ingénieurs des bureaux d'études de nombreuses entreprises industrielles telles qu'Airbus, Microsoft, Siemens...

L'équipe de Brest travaille sur de nouveaux modules et plugins permettant à partir d'un modèle SysML de générer du code exécutable pour différentes cibles matérielles. Les différents modules s'appuient sur ceux développés pour le générateur ESP.

5.3.2 Modélisation SysML

La première tâche dans cette partie a été de réaliser un modèle SysML du démonstrateur.

Le SysML (Systems Modeling Language) est un langage de modélisation permettant la spécification, l'analyse, la conception et la validation de systèmes complexes logiciels ou matériels. C'est une extension d'un sous-ensemble du langage UML (Unified Modeling Language). Le référentiel SysML propose neuf diagrammes permettant de modéliser les systèmes à travers les exigences requises, leurs aspects comportementaux et structuraux. Les modèles SysML ne modélisent pas les systèmes avec des classes et des objets comme en UML mais utilisent la notion de blocs. Un bloc va pouvoir englober un concept logiciel ou matériel, une définition de données, un processus…

J'ai commencé à modéliser l'aide à la conduite en réalisant un diagramme de définition des blocs. Il permet de décrire l'architecture du système et d'exprimer la hiérarchisation entre les sous entités notamment.

Le système d'aide à la conduite (ADAS) est décomposé en sous-entités applicatives. Pour chaque bloc, les entrées et les sorties sont également définis. En m'appuyant sur le travail réalisé sur le démonstrateur, j'ai donc défini les blocs :

- Distance_sensor_processing **:** traitement des données des capteurs de distance et signalement de la détection de la présence d'un véhicule par une variable booléenne
- Image_Processing **:** traitement d'images et détection des lignes de la voie
- Lane_follower **:** traitement des lignes de la voie et envoie d'une commande de braquage
- Overtaking_management **:** gestion du dépassement
- Décision **:** traitement des différentes informations et gestion de passage du mode suivi de ligne au mode dépassement

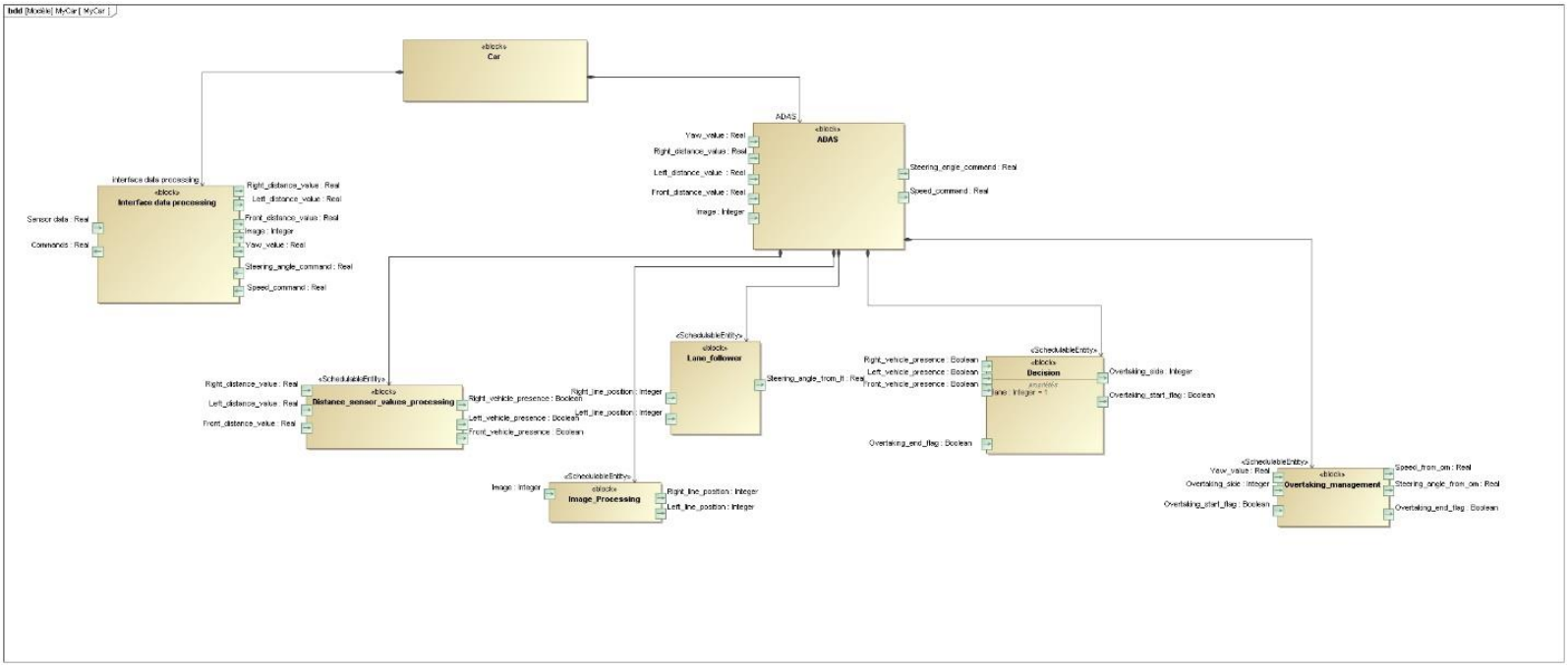


Figure 23 : diagramme de définition des blocs du système « véhicule »

En se basant sur ce diagramme, on peut ensuite réaliser un diagramme de bloc interne (IBD : Internal Block Diagram) afin de décrire la vue interne du bloc ADAS. Ainsi, les blocs qui le composent sont instanciés et les flux qui les relient sont décrits.

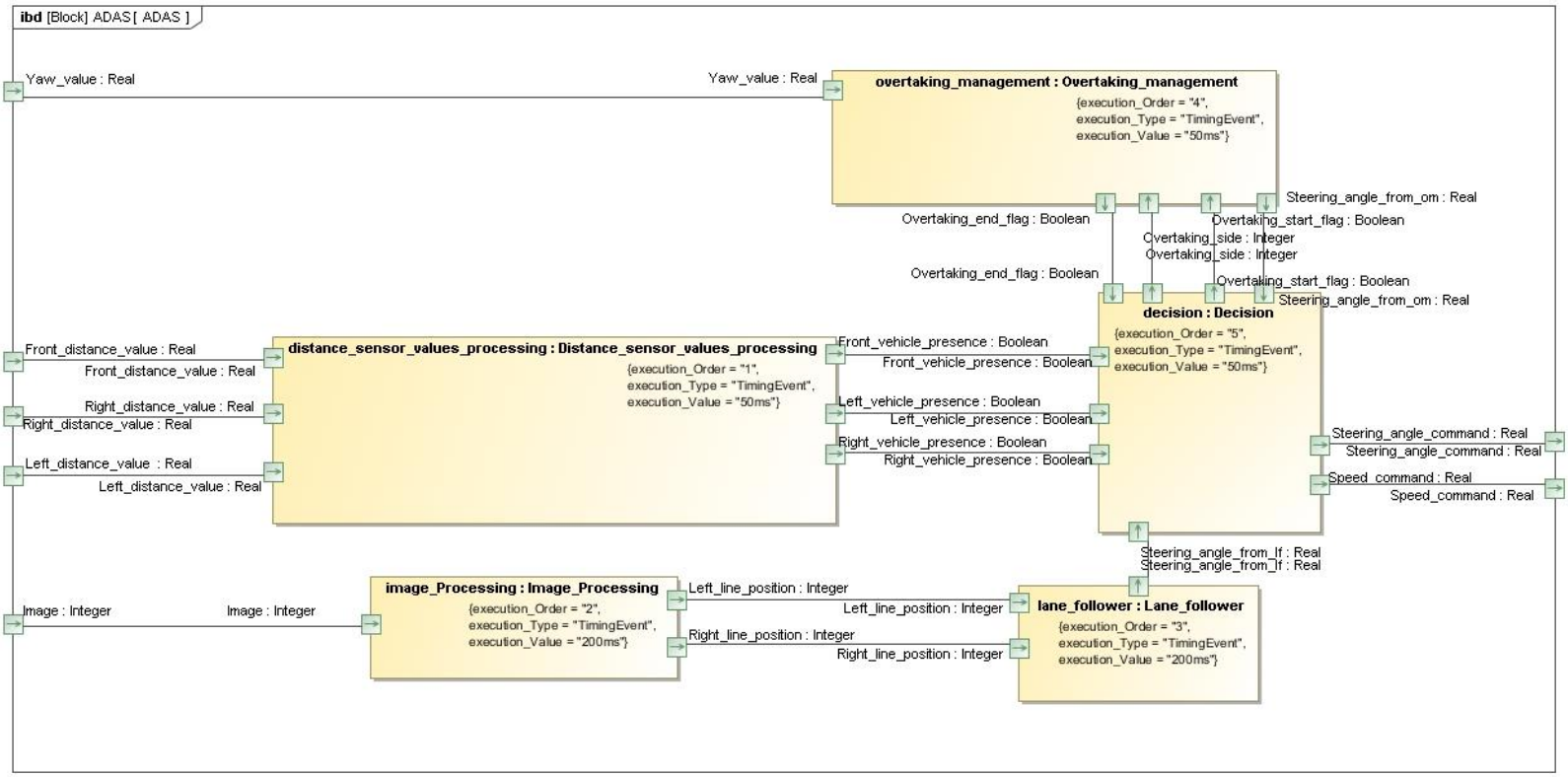


Figure 24 : Diagramme de bloc interne : ADAS

J'ai spécifié ensuite le comportement des bloc en définissant pour chacun soit un diagramme à états, un diagramme d'activité ou directement du code applicatif en C ou C++ (Opaque behavior). Il a été choisi de développer les fonctionnalités de génération de code pour les diagrammes d'états et ceux composés par du code applicatif, j'ai donc retenu ces formalismes pour le comportement de mes blocs.

J'ai modélisé le comportement du bloc Decision et Overtaking_managament avec des diagrammes d'états, implémentant du code pour les autres blocs. Afin de valider les fonctionnalités du générateur, j'ai essayé d'utiliser une grande variété de conventions de modélisation avec des états composites, différentes conditions, spécifiant pour certains blocs une tâche d'entrée (entry behavior) et une tâche de sortie (exit behavior) …

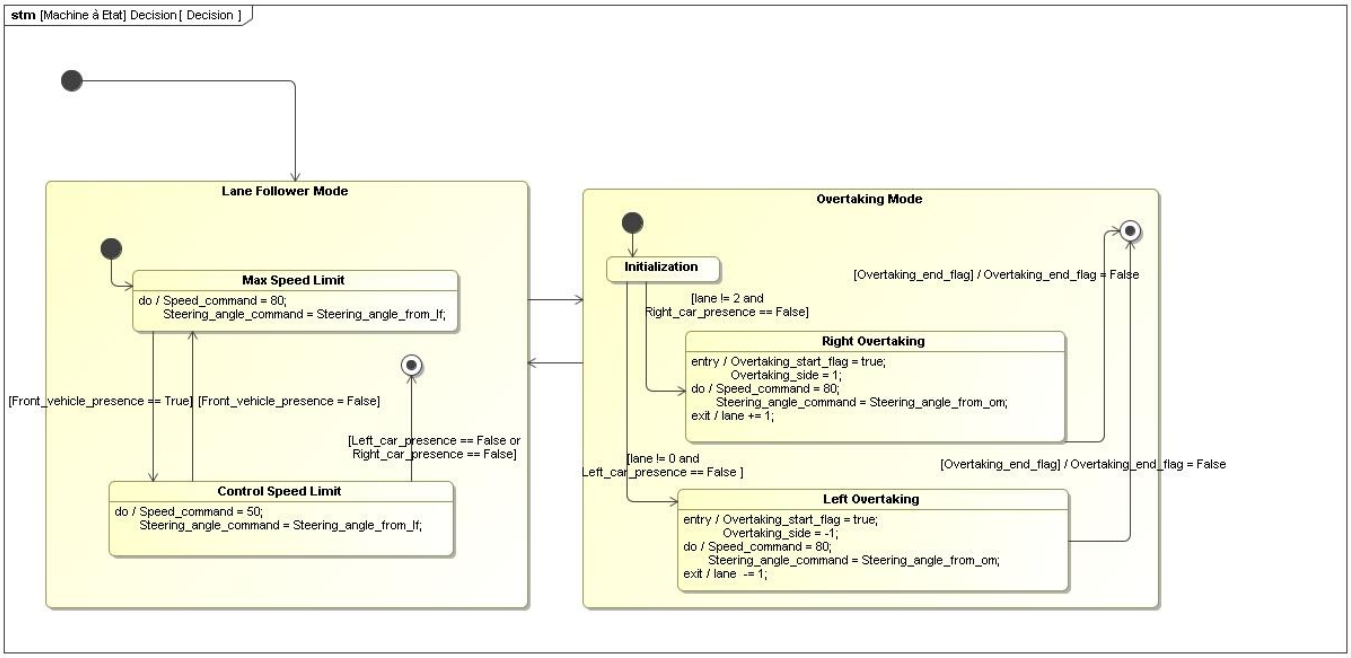


Figure 25 : Machine à états du block Decision

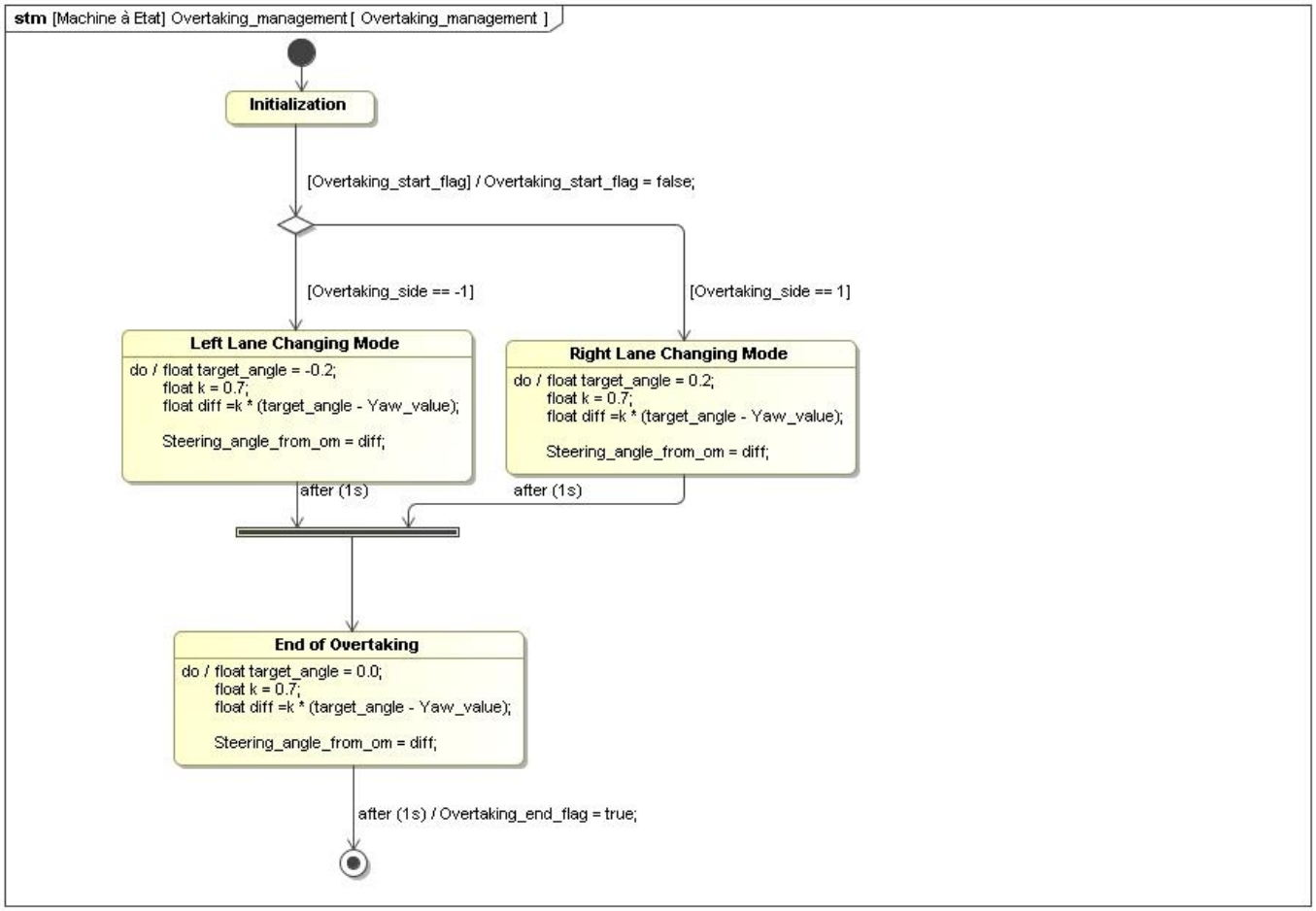


Figure 26 : Machine à états du bloc Overtaking management

5.3.3 Fonctionnement de la génération de code

La génération de code à partir du logiciel CAMEO s'appuie les modules déjà développés pour ESP. L'ensemble des modules de génération sont développés en JAVA et ils permettent dans ESP la génération de fichiers sources pour notamment les plateforme RIOT (un système d'exploitation pour les objets connecté), PLCopen (automate programmable industriel), AUTOSAR Classic et Adaptive.

En sortie du logiciel Cameo, les projets composés des différents diagrammes peuvent être exporté via un fichier XMI. XMI est un standard crée par l'Object Management Group basé sur le langage de description XML pour l'échange de données UML. Ensuite, la génération de code peut être réalisée en local ou à distance sur un serveur d'application Apache Tomcat grâce à une servlet.

Les module de génération des différents fichiers se basent sur un ensemble de « templates », des fichiers modèles ainsi que les fonctions permettant de compléter ces gabaries en fonction des données du modèle SysML.

Les fichiers sources générés respectent une arborescence précise :

- Fichiers sources applicatifs
- Fichiers du RTE (RunTime Environment) faisant le lien entre le code applicatif et la plateforme cible spécifique avec des :
    - Fichiers de configuration dépendants de la cible
    - Fichiers de configuration indépendants de la cible

5.3.4 Génération et compilation de code spécifique à une cible ROS 2

J'ai travaillé sur un module permettant la génération de fichiers sources pour une cible ROS 2. Pour cela je me suis basé sur le module de génération pour une cible RIOT.

Au départ, à partir du modèle SysML, j'ai généré les fichiers pour une cible RIOT et les ai modifiés manuellement pour qu'ils compilent pour une cible ROS 2 afin de bien comprendre la logique, le fonctionnement et l'organisation des fichiers générés. J'ai ensuite développé un module Java permettant la génération des fichiers de configuration ROS 2. J'ai ainsi développé deux fichiers « templates » et différentes fonctions les remplissant en fonction des données d'entrées.

Ainsi, un processus est créé et pour chaque bloc du diagramme de bloc interne, un nœud ROS 2 est lancé avec l'Opaque Behavior ou la machine à états associée.

La gestion des entrées-sorties des différents blocs a été plus complexe. La stratégie de communication n'a pas été fixée tout de suite car il ne semblait pas totalement pertinent de créer des subscribers ou des publishers ROS 2 pour chaque entrée-sortie. D'autres moyens de communication comme la mémoire partagée semblaient plus pertinente pour des nœuds à l'intérieur d'un même processus.

J'ai pu ensuite vérifier le bon déroulement de la compilation et de l'exécution du code généré. Ces développements ont fourni une preuve de fonctionnement et il a été choisi de ne pas pousser tout de suite les développements du générateur de fichiers sources ROS 2.

5.3.5 Génération et compilation de code spécifique à une cible AUTOSAR Adaptive

J’ai ensuite travaillé sur un module permettant la génération de fichiers sources pour une cible AUTOSAR Adaptive. Les fichiers sources générés pour une application Adaptive sont plus complexes que ceux pour une cible ROS 2.

Le générateur pour une application Adaptive est en cours de développement, la génération du démonstrateur a permis de résoudre quelques erreurs.

6 Communication avec une application AUTOSAR Adaptive

6.1 Contexte

J'ai été en charge d'étudier différentes possibilités de communication entre d'une part des applications AUTOSAR Adaptive entre elles et d'autre part une application avec le système hôte hébergeant l'environnement d'exécution. L'objectif était de réaliser différentes simulations et tests du module applicatif.

6.2 Exécution d'une application AUTOSAR Adaptive dans un environnement de test

Une application AUTOSAR Adaptive s'exécute sur un système d'exploitation UNIX, lors de mes tests sur Poky, une distribution Linux du projet Yocto. Lors des développements et des tests des modules applicatifs, QEMU est utilisé pour émuler cette plateforme embarquée. QEMU est lancé à l'intérieur d'un conteneur DOCKER basé sur une image UBUNTU 20.4 (noyau 4.19.76).

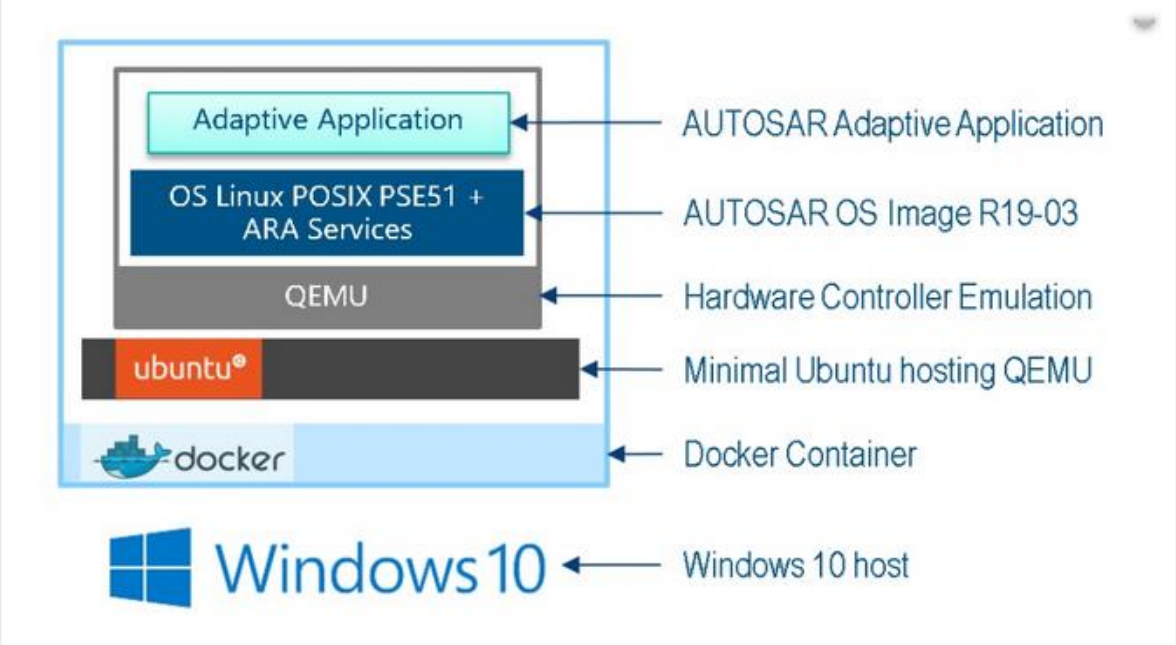


Figure 27 : Environnement d'exécution d'une application Adaptive sous Windows 10

6.3 DOCKER et QEMU

QEMU est une machine virtuelle pouvant émuler une plateforme matérielle, elle peut simuler une architecture de processeurs X86, ARM, Sparc, MIPS… Il permet d'exécuter différents systèmes d'exploitation lors de l'émulation. Les périphériques nécessaires sont également simulés.

Docker est un outil qui permet la conteneurisation d'application. Les conteneurs partagent le noyau de l'hôte mais s'exécutent de manière séparée en isolant les processus de l'application du système hôte ainsi que des autres conteneurs. Un conteneur va virtualiser un système d'exploitation en n'émulant pas la plateforme matérielle, ce qui est beaucoup plus léger qu'une machine virtuelle qui a besoin d'une plus grande puissance de calcul. Depuis peu, Docker fonctionne très bien sur une machine hôte avec un système d'exploitation Windows en virtualisant un noyau Linux pour des conteneurs basés sur une image Linux. L'utilisation de docker permet de standardiser les opérations sur les applications développées en encapsulant toutes les dépendances nécessaires au bon fonctionnement de l'application.

6.4 Communication avec une application Adaptive

Un composant Adaptive, pour communiquer avec d'autres composants, peut utiliser différents protocoles de communication : DDS, SOME/IP, des sockets.

Il a été nécessaire de travailler sur la communication à travers les différentes couches de l'environnement d'exécution afin de pouvoir faire communiquer une application lancée sur le système hôte Windows 10 et l'application Adaptive. La première application Adaptive que j'ai lancée était un serveur HTTP, il a été aisée de la faire communiquer avec un client lancé dans le conteneur.

Néanmoins, je n'ai pas réussi à faire communiquer directement le serveur Adaptive et un client lancé sur l'hôte. Pour contourner ce problème de communication entre les couches, j'ai développé une passerelle composée de deux threads avec un socket client et socket serveur pour rediriger les données à l'intérieur du conteneur Docker sur les bonnes adresses IP et les bons ports.

J'ai manqué de temps pour approfondir ce travail. L'objectif était de faire communiquer une application Adaptive avec un simulateur (Webots par exemple) lancé sur le système hôte Windows.

7 Retour d'expérience

7.1 L'ingénieur en systèmes embarqués

L'ingénieur en systèmes embarqués est au centre d'un ensemble de technologies qu'il doit réussir à maitriser. Il doit avoir de bonnes connaissances en développement informatique dans des langages de bas et haut niveau pour pouvoir programmer les différents composants, des interfaces… et avoir des connaissances sur les différents protocoles de communication, des notions en réseau. Des bases solides en électronique et en instrumentation sont également nécessaires. Le domaine de l'informatique en général ne cesse d'évoluer, il est primordial de continuer à prendre le temps de se former sur de nouvelles technologies.

7.2 Bilan personnel

Ces six mois de stage qui se sont déroulés dans des conditions particulières avec une grande partie du travail réalisée à distance m'ont permis de progresser sur de nombreux points.

J'ai beaucoup appris techniquement, je connaissais déjà assez bien ROS et mes différents travaux m'ont permis de découvrir et de travailler en profondeur sur ROS 2. J'ai découvert d'autres protocoles comme DDS, Vsomeip et commencé à appréhender le standard AUTOSAR. A travers ces travaux, j'ai pu approfondir mes compétences en programmation C, C++ et JAVA, et sur l'utilisation d'outils très communs dans le développement logiciel comme Docker et Qemu. Ces connaissances me seront utiles par la suite, en effet on retrouve de nombreux concepts similaires dans différentes plateformes embarquées.

Mes travaux sur la modélisation de systèmes sur le logiciel Cameo et sur des outils de génération de code m'ont permis de comprendre les méthodologies et les enjeux du développement des systèmes embarqués en milieu industriel.

J'ai pu ainsi appréhender l'ensemble de la chaîne du design d'un système embarqué, de la spécification et de la modélisation d'une application, à la génération et à l'exécution de fichiers sources en passant par la simulation. Ce fil directeur a été présent à toutes les étapes de mon stage.

Ces six mois m'ont permis d'approfondir mes connaissances sur les méthodologies et les outils utilisés lors de développement de projets informatiques. Il s'agissait de mon troisième stage en tant qu'assistant ingénieur, j'ai réalisé mon premier stage dans une TPE et le second dans une PME composée d'une trentaine de personnes. Cette expérience m'a donné la possibilité d'avoir une vision et une expérience supplémentaire dans une grande entreprise. J'ai pu comparer l'organisation ainsi que les différents processus de développement.

Ces différentes expériences ont confirmé mon attrait pour continuer à travailler dans le domaine des systèmes embarqués.

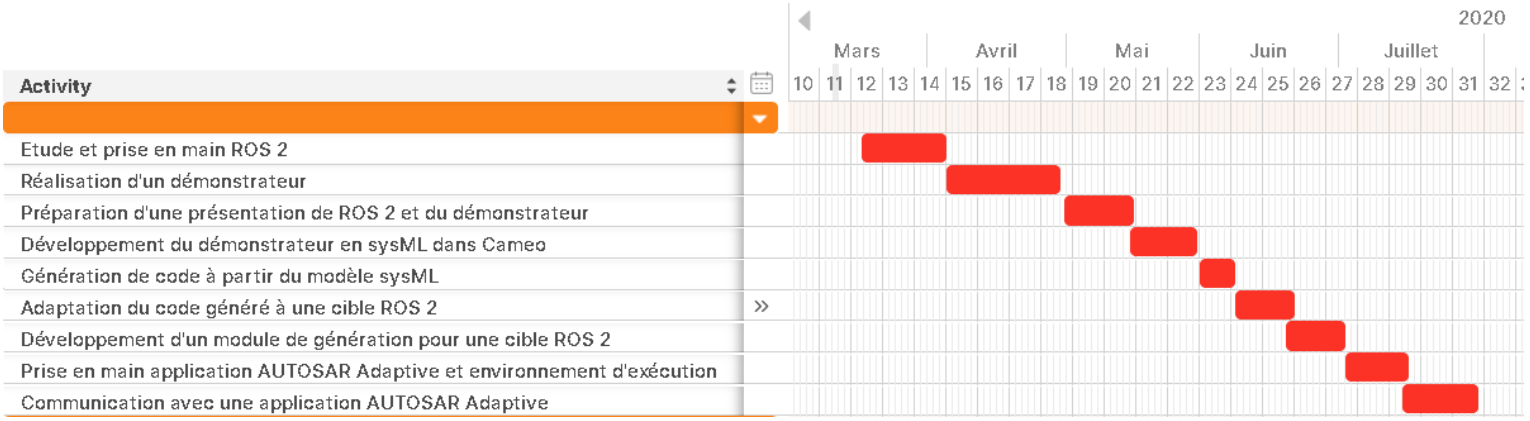
Glossaire

Terme	Signification
ADAS	Advanced driver-assistance systems Aide à la conduite automobile : système de sécurité active information ou assistance du conducteur
API	Interface de programmation applicative (Application Programming interface) est un ensemble normalisé de classe, de méthode et fonction et de constantes qui sert de façade par laquelle un logicielle offre des services à d'autres logiciels.
ESP	Embedded Software Producer. Solution web pour le développement d'applications embarqués multi disciplines de Dassault Systèmes
UML	Langage de modélisation unifié (Unified Modeling Language). C'est un langage de modélisation graphique à base de pictogramme conçu pour fournir une méthode normalisée pour visualiser la conception d'un sytème.

Bibliographie

- Dassault Systèmes. 2020. *Notre histoire.* https://www.3ds.com/fr/a-propos-de-3ds/notre-histoire/
- Wikipedia. 2020. *Dassault Systèmes*. https://fr.wikipedia.org/wiki/Dassault_Syst%C3%A8mes
- L. Nehaoua. 2013. *Mécatronique des véhicules automobiles – La norme AUTOSAR*. http://nehsetl.free.fr/Cours2.pdf
- *ROS 2 Design*. 2013. https://design.ros2.org/
- Emmanual Robotis. 2017. *Turtlebot3*. https://emanual.robotis.com/docs/en/platform/turtlebot3/overview/
- F. Legrand. *Informatique appliquée aux sciences physiques*. https://www.f-legrand.fr/scidoc/docmml/index.html
- Didier Fagnon. 2012. *SysML : les diagrammes*. https://eduscol.education.fr/sti/sites/eduscol.education.fr.sti/files/ressources/techniques/979/979-179-p100.pdf
- Guillaume Finance. 2010. *Modélisation SysML*. https://uml.developpez.com/cours/Modelisation-SysML/#LIV-A

Gestion des tâches : diagramme de Gantt



Code source du simulateur

Decision node

```
#include <chrono>
#include <memory>
#include <string>
#include <algorithm>
#include <vector>
#include <thread>
#include <stdlib.h>

#include "rclcpp/rclcpp.hpp"
#include "std_msgs/msg/string.hpp"
#include "std_msgs/msg/float32.hpp"
# include "sensor_msgs/msg/image.hpp"
# include "sensor_msgs/msg/laser_scan.hpp"
# include "geometry_msgs/msg/twist.hpp"
#include "rosgraph_msgs/msg/clock.hpp"

using std::placeholders::_1;
using namespace std;

class HighLevel : public rclcpp::Node
{
  public :
    HighLevel () : Node ("Decision_node")
    {
      RCLCPP_INFO(this->get_logger(), "Node : Decision_node ");

       // Distance sensors
```

```
      front_sensor_subscriber= this-
>create_subscription<std_msgs::msg::Float32>(
      "front_sensor_distance", 1,
std::bind(&HighLevel::front_sensor_callback, this,_1));

      left_sensor_subscriber= this-
>create_subscription<std_msgs::msg::Float32>(
      "right_sensor_distance", 1, std::bind(&HighLevel::left_sensor_callback,
this,_1));

      left_sensor_subscriber= this-
>create_subscription<std_msgs::msg::Float32>(
      "right_sensor_distance", 1,
std::bind(&HighLevel::right_sensor_callback, this,_1));

      //imu subscriber

      imu_subscriber= this->create_subscription<std_msgs::msg::Float32>(
      "imu_data_yaw", 1, std::bind(&HighLevel::imu_callback, this,_1));

      // Cmd from image_proc Node
      cmd_line_subscriber = this-
>create_subscription<geometry_msgs::msg::Twist>(
      "line_follow",1, std::bind(&HighLevel::cmd_line_callback, this,_1));

      cmd_steering_publisher = this-
>create_publisher<std_msgs::msg::Float32>("/steering_commande",1);

      // Cmd velocity to low_level Node
      cmd_vel_publisher = this-
>create_publisher<geometry_msgs::msg::Twist>("/cmd_vel",1);

      //clock
      clock_subscriber = this-
>create_subscription<rosgraph_msgs::msg::Clock>(
      "clock",1, std::bind(&HighLevel::clock_callback, this,_1));

     // main loop timer
     main_loop_timer = this->create_wall_timer(40ms,
std::bind(&HighLevel::main_loop_callback, this));

      max_speed = 70;
      front_distance = 20;
      is_car_right = false;
      is_car_left = false;
      is_car_front = false;
      cmd_line_z = 0.0 ;
      speed = 0.0;
      line_number = 1;
      sec = 0;
      nanosec = 0;
      flag_timer = true;
      start_overtaking= 0;
      yaw = 0;
      head = 0;
    }


    void front_sensor_callback (const std_msgs::msg::Float32::SharedPtr msg){
```

```
      front_distance = msg->data;
      if (front_distance < 14){
        is_car_front = true;
        //RCLCPP_INFO(this->get_logger(), "Car ahead ");
        if (flag_timer == true){
          start_overtaking = sec * 1000 + nanosec/1000000;
          flag_timer = false;
          head = 0.0;

        }
      }
    }

    void right_sensor_callback (const std_msgs::msg::Float32::SharedPtr msg){
      //cout<<"Distance : "<< msg->data<<endl;
      float right_distance = msg->data;
      if (right_distance < 3){
        is_car_right = true;
        RCLCPP_INFO(this->get_logger(), "Car on the right");}
    }

    void left_sensor_callback (const std_msgs::msg::Float32::SharedPtr msg){
      //cout<<"Distance : "<< msg->data<<endl;
      float left_distance = msg->data;
      if (left_distance < 3){
        is_car_left = true;
        RCLCPP_INFO(this->get_logger(), "Car on the left");}
    }


    void imu_callback (const std_msgs::msg::Float32::SharedPtr msg){
      yaw = (msg->data);
      head -= yaw;
      RCLCPP_INFO(this->get_logger(), "Heading : '%f'", head);

    }

    void cmd_line_callback (const geometry_msgs::msg::Twist::SharedPtr msg){
      if (is_car_front == false){
        geometry_msgs::msg::Twist cmd;
        cmd.linear.x = max_speed;
        cmd.angular.z = msg->angular.z;
        cmd_vel_publisher->publish(cmd);

        RCLCPP_INFO(this->get_logger(), " Not near front vehicule, Distance =
'%f'", front_distance);
        RCLCPP_INFO(this->get_logger(), "Line number : '%i'", line_number);
      }
    }

    void clock_callback (const rosgraph_msgs::msg::Clock::SharedPtr msg){
      sec = msg->clock.sec;
      nanosec = msg->clock.nanosec;
      main_loop_callback();
    }


    float apply_pid(float angle, float targetAngle){
```

```
    float k = 0.7;
    float diff = targetAngle - angle;
    float  consigne = k * diff;
    return consigne;
  }


  void steering (int side){
    float commande_angle;
    float consigne_angle;

    float time_over = 1400;
    float end_1 = start_overtaking + time_over;
    float end_2 = start_overtaking + 2*time_over;


    if (side == 1){//right
      RCLCPP_INFO(this->get_logger(), "Right turn");
      consigne_angle = 0.2;
    }else if (side == -1){ //left
      RCLCPP_INFO(this->get_logger(), "Left Turn");
      consigne_angle = -0.2;
    }

    if ((sec * 1000 +nanosec/1000000) < end_1)
    {
      commande_angle = apply_pid(head, consigne_angle);
      RCLCPP_INFO(this->get_logger(), "Overtake Ongoing : turn ");
    }
    else if (((sec * 1000 +nanosec/1000000) > end_1) && ( (sec * 1000
+nanosec/1000000) < end_2) )
    {
      commande_angle= apply_pid(head, 0);
      RCLCPP_INFO(this->get_logger(), "Overtake Ongoing : steering 0");
    }
    else if ( (sec * 1000 +nanosec/1000000) > end_2) {
      RCLCPP_INFO(this->get_logger(), "End of the overtake");
      commande_angle= apply_pid(head, 0);
      is_car_front = false;
      flag_timer = true;
      if (side == 1){//right
        RCLCPP_INFO(this->get_logger(), "Right turn");
        line_number += 1;
      }else if (side == -1){ //left
        RCLCPP_INFO(this->get_logger(), "Left Turn");
        line_number -= 1;
      }
      }
    geometry_msgs::msg::Twist cmd;
    cmd.linear.x = max_speed;
    cmd.angular.z = commande_angle;
    cmd_vel_publisher->publish(cmd);
    }

    //Right Lane = 0
    //Left lane = 2

  void main_loop_callback ()  {
    float speed_com;
```

```
      float angle_com;
      int side;

      if (is_car_front == true){
        if (line_number == 1){
          if (is_car_left == false){ // left overtaking
            side = -1;
            steering(side);
            speed_com = 60;
          }else if (is_car_right == false){ // rigth overtaking
            side = 1;
            steering(side);
            speed_com = 60;
          }else { //slow and wait
            speed_com = 30;
            angle_com = cmd_line_z;
          }
        }
        else if (line_number == 0){
          if (is_car_right == false){ // right overtaking
            side = 1;
            steering(side);
            speed_com = 60;
          }else{ //slow and wait
            speed_com = 30;
            angle_com = cmd_line_z;
          }

        }else if (line_number == 2){
          if (is_car_left == false){ //left overtaking
            side = -1;
            speed_com = 60;
            steering(side);
          }else{ //slow and wait
            speed_com = 30;
            angle_com = cmd_line_z;
          }
        }
      }else {
        speed_com = max_speed * (front_distance/20);
        speed_com = max_speed;
        angle_com = cmd_line_z;
        RCLCPP_INFO(this->get_logger(), " Not near front vehicule, Distance =
'%f'", front_distance);
        RCLCPP_INFO(this->get_logger(), "Line number : '%i'", line_number);
      }

      cmd.linear.x = speed_com;
      cmd.angular.z = angle_com;
      cmd_vel_publisher->publish(cmd);
    }



 private :
    rclcpp::Subscription<geometry_msgs::msg::Twist>::SharedPtr
cmd_line_subscriber;
    rclcpp::Publisher<geometry_msgs::msg::Twist>::SharedPtr
cmd_vel_publisher;
```

```
    rclcpp::Publisher<std_msgs::msg::Float32>::SharedPtr
cmd_steering_publisher;

    rclcpp::Subscription<std_msgs::msg::Float32>::SharedPtr
front_sensor_subscriber;
    rclcpp::Subscription<std_msgs::msg::Float32>::SharedPtr
left_sensor_subscriber;
    rclcpp::Subscription<std_msgs::msg::Float32>::SharedPtr
right_sensor_subscriber;
    rclcpp::Subscription<rosgraph_msgs::msg::Clock>::SharedPtr
clock_subscriber;
    rclcpp::Subscription<std_msgs::msg::Float32>::SharedPtr imu_subscriber;

    rclcpp::TimerBase::SharedPtr main_loop_timer;


    float max_speed;
    float front_distance;
    bool is_car_right;
    bool is_car_left;
    bool is_car_front;
    float cmd_line_z;
    float speed;
    int line_number;
    int sec;
    int nanosec;
    bool flag_timer;
    int start_overtaking;
    float head;
    float yaw;



};

int main(int argc, char * argv[])
{
  rclcpp::init(argc, argv);
  rclcpp::spin(std::make_shared<HighLevel>());
  rclcpp::shutdown();
  return 0;
}
```

Image processing node

```
#include <chrono>
#include <memory>
#include <string>
#include <numeric> //for shared_ptr
#include <vector>


#include "rclcpp/rclcpp.hpp"
#include "std_msgs/msg/string.hpp"
#include "sensor_msgs/msg/image.hpp"
```

```
#include "geometry_msgs/msg/twist.hpp"
#include <geometry_msgs/msg/twist.hpp>
#include "rclcpp_components/register_node_macro.hpp"

#include "opencv2/opencv.hpp"
#include <opencv2/core.hpp>
#include <opencv2/imgcodecs.hpp>
#include "opencv2/imgproc/imgproc.hpp"
#include "opencv2/highgui/highgui.hpp"

using namespace cv;
using namespace std;
using std::placeholders::_1;

class ImageProc : public rclcpp::Node
{
  public :
    ImageProc() : Node ("Image_proc_Node")
    {
      RCLCPP_INFO(this->get_logger(), " Node : Image_proc_node ");
      image_subscriber= this->create_subscription<sensor_msgs::msg::Image>(
      "image", 1, std::bind(&ImageProc::image_callback, this,_1));
      twist_pub= this-
>create_publisher<geometry_msgs::msg::Twist>("/line_follow", 10);

    }

    void princ(Mat src){

      Mat src_gray, final, res;
      Mat dst, detected_edges;
      Mat all_lines;
      int lowThreshold = 75;
      const int max_lowThreshold = 100;
      const int ratio = 3;
      const int kernel_size = 3;

      namedWindow ("Car simulation",WINDOW_NORMAL);
      resizeWindow("Car simulation", 512,256);

      Point angle[1][3];
      angle[0][0] = Point(10,src.size().height);
      angle[0][1] = Point((src.size().width)/2-1, (src.size().height)/2);
      angle[0][2] = Point((src.size().width), src.size().height-10);

      const Point* ppt[1] = {angle[0]};
      int npt[] = {3};


      cout<< "Width : "<< src.size().width << endl;
      cout<< "Heigth : "<<src.size().height<<endl;

      src.copyTo(final);
      cvtColor( src, src_gray, COLOR_BGR2GRAY );

      blur( src_gray, detected_edges, Size(3,3) );
      Canny( detected_edges, detected_edges, 75, 150, kernel_size );

      Mat mask = Mat::zeros(detected_edges.size(), detected_edges.type());
```

```
      fillPoly(mask,ppt, npt, 1 ,Scalar(255,255,255),8 );
      bitwise_and(detected_edges,mask, res);

      vector<Vec4i> lines;
      HoughLinesP( res, lines, 1, CV_PI/60, 15,15, 10);

      vector<int> x_start_right;
      int x_start_right_line_avg;
      vector<int> x_start_left;
      int x_start_left_line_avg;

      int x_end_left_avg;
      int x_end_right_avg;

      vector<int> x_end_right;
      vector<int> x_end_left;

      for( size_t i = 0; i < lines.size(); i++ )
      {
        Vec4i l = lines[i];
        float gradient;
        cout<<"ligne"<<lines.size()<<endl;
        cout<<"point_1 : "<<l[0]<<" "<<l[1]<<endl;
        cout<<"point_2 : "<<l[2]<<" "<<l[3]<<endl;
        line( all_lines, Point(l[0],l[1]), Point(l[2],l[3]), Scalar(0,0,255),
4);

        gradient = ((float)l[3]-(float)l[1])/((float)l[2]-(float)l[0]);
        cout<<"gradient :"<<gradient<<endl;

        if ((gradient >0.35) && (gradient < 1.1)){ //right line
          if ((l[1]> 70) && (l[3] > 70)){
            int b = (int)(l[3] - gradient*l[2]);
              x_start_right.push_back((int)((1/gradient)*((120)-b)));//215
              x_end_right.push_back((int)((1/gradient)*((70)-b)));
            if (x_start_right.size()>10){
              x_start_right.erase(x_start_right.begin());
              x_end_right.erase(x_end_right.begin());
            }
          }
        }else if ((gradient >-1.1) && (gradient <-0.35)) { //left line
          if ((l[1] >70) && (l[3] > 70) ){
            int b = (int)(l[1] - gradient*l[0]);
            x_start_left.push_back((int)((1/gradient)*((120)-b)));//215
            x_end_left.push_back((int)((1/gradient)*((70)-b)));
            if (x_start_left.size()>10){
              x_start_left.erase(x_start_left.begin());
              x_end_right.erase(x_start_right.begin());
            }
          }
        }
      }


      if (!x_start_right.empty()){

          x_start_right_line_avg = accumulate(x_start_right.begin(),
x_start_right.end(),0)/x_start_right.size();
```

```
          x_end_right_avg = accumulate(x_end_right.begin(),
x_end_right.end(),0)/x_end_right.size();

          Point start_of_the_line(x_start_right_line_avg,120); //215
          Point end_of_the_line (x_end_right_avg, 70);
          line(final, start_of_the_line, end_of_the_line, Scalar(0,255,0),
4);

          }
      if (!x_start_left.empty()){
          x_start_left_line_avg = accumulate(x_start_left.begin(),
x_start_left.end(),0)/x_start_left.size();
          x_end_left_avg = accumulate(x_end_left.begin(),
x_end_left.end(),0)/x_end_left.size();

          Point start_of_the_line(x_start_left_line_avg,120 );
          Point end_of_the_line (x_end_left_avg, 70);
          line( final, start_of_the_line, end_of_the_line, Scalar(0,0,255),
4);
          }
          5
          Point right_threshold(188, 107 );
          circle(final, right_threshold, 5, Scalar(0,100,255),-1,8,0 );
          Point left_threshold(68,107 );//140,215
          circle(final, left_threshold, 5, Scalar(0,100,255),-1,8,0 );


         if (x_start_right_line_avg > 188 ){ //last //375
            geometry_msgs::msg::Twist cmd;
            //putText(final, " Right !", Point(100,100), FONT_HERSHEY_DUPLEX,
1, Scalar(0,143,143), 2);
            cmd.angular.z = 0.02;
            cmd.linear.x = 60;
            twist_pub->publish(cmd);

          }
         if (x_start_left_line_avg < 68 ){ //last 70
            geometry_msgs::msg::Twist cmd;
            //putText(final, " Left!", Point(100,100), FONT_HERSHEY_DUPLEX,
1, Scalar(0,143,143), 2);
            cmd.angular.z =  - 0.02;
            cmd.linear.x = 60;
            twist_pub->publish(cmd);
          }



      imshow ("Car simulation", final);

      waitKey(1);
    }


    void image_callback(const sensor_msgs::msg::Image::SharedPtr msg)
    {
      const char* window_name = "Edge Map";
      Mat frame(msg->height, msg->width, CV_8UC4,const_cast<unsigned char *>
(msg->data.data()),msg->step);
      this->princ(frame);
```

```
    }

    private :

      rclcpp::Subscription<sensor_msgs::msg::Image>::SharedPtr
image_subscriber;
      rclcpp::Publisher<geometry_msgs::msg::Twist>::SharedPtr twist_pub;

};


int main(int argc, char * argv[])
{
  rclcpp::init(argc, argv);
  rclcpp::spin(std::make_shared<ImageProc>());
  rclcpp::shutdown();
  return 0;
}
```

Interface node

```
"""Interface Node Highway Overtake"""

import rclpy
from webots_ros2_core.laser_publisher import LaserPublisher
from sensor_msgs.msg import Range, Imu, LaserScan, Illuminance, Image,
CameraInfo
from std_msgs.msg import Float32
from geometry_msgs.msg import Twist

from webots_ros2_core.webots_node import WebotsNode

class LowLevel(WebotsNode):

    def __init__(self, args):
        super().__init__('Interface_Node_highway', args)
        self.get_logger().info("Initialisation de la classe :
Interface_Node_highway")
        self.cap = 0

        self.car.setSteeringAngle(0)
        self.car.setCruisingSpeed(60)
        self.car.setHazardFlashers(True)

        self.create_subscription(Twist, '/cmd_vel', self.cmd_vel_callback, 1)


        #Camera
        self.camera = self.car.getCamera("camera")
        self.camera.enable(50)
        self.camera_publisher = self.create_publisher(Image,"/image",10)
        self.camera_timer = self.create_timer(0.01, self.camera_callback)
        #self.camera_info_publisher = self.create_publisher(CameraInfo,
'/camera_info', 10)

        # lidar
```

```
        self.lidar = LaserPublisher (self.car, self)

        #Distance Sensor

        self.right_distance_sensor = self.car.getDistanceSensor("distance
sensor right")
        self.right_distance_sensor.enable(self.timestep)

        self.left_distance_sensor = self.car.getDistanceSensor("distance
sensor left")
        self.left_distance_sensor.enable(self.timestep)

        self.front_distance_sensor = self.car.getDistanceSensor("distance
sensor front")
        self.front_distance_sensor.enable(self.timestep)


        self.right_sensor_publisher =
self.create_publisher(Float32,"/rigth_sensor_distance",1)
        self.left_sensor_publisher =
self.create_publisher(Float32,"/left_sensor_distance",1)
        self.front_sensor_publisher =
self.create_publisher(Float32,"/front_sensor_distance",1)


        self.distance_sensor_timer = self.create_timer(self.timestep / 1000,
self.distance_sensor_callback)

        #Imu
        self.imu = self.car.getInertialUnit("inertial unit")
        self.imu.enable(10)
        self.imu_sensor_timer = self.create_timer(0.01,
self.imu_sensor_callback)
        self.imu_publisher = self.create_publisher(Float32,
"/imu_data_yaw",1)


    def distance_sensor_callback(self):
        right_distance = Float32()
        left_distance = Float32 ()
        front_distance = Float32()
        right_distance.data = self.right_distance_sensor.getValue()
        left_distance.data = self.left_distance_sensor.getValue()
        front_distance.data = self.front_distance_sensor.getValue()

        self.right_sensor_publisher.publish(right_distance)
        self.left_sensor_publisher.publish(left_distance)
        self.front_sensor_publisher.publish(front_distance)

    def imu_sensor_callback(self):
        imu_data = Float32()
        imu_data_ = self.imu.getRollPitchYaw()[0]
        self.cap += imu_data_
        imu_data.data = imu_data_
        self.imu_publisher.publish(imu_data)



    def cmd_vel_callback(self, twist):
```

```
        velocity = twist.linear.x
        angle = twist.angular.z


        self.car.setCruisingSpeed(velocity)
        self.car.setSteeringAngle(angle)
        print ( "Speed : ", self.car.getTargetCruisingSpeed())
        print("Angle : ", angle)

        print ("Imu data : Heading ", -self.cap)


    def camera_callback(self):
    # Image data
        msg = Image()
        msg.height = self.camera.getHeight()
        msg.width = self.camera.getWidth()
        msg.is_bigendian = False
        msg.step = self.camera.getWidth() * 4
        msg.data = self.camera.getImage()
        #first 30 elements : 134 91 72 255 134 91 72 255 134 91 72 255 134 90
72 255 134 90 72
        #[B G R A B G R A B G R A ....
        msg.encoding = 'bgra8'
        self.camera_publisher.publish(msg)

def main(args=None):
    rclpy.init(args=args)
    lowLevel = LowLevel(args=args)
    rclpy.spin(lowLevel)
    rclpy.shutdown()


if __name__ == '__main__':
    main()
```