Génie Electrique et Automatique

Option CDISC

Projet long 2017

Commande et Simulation d'un réseau de transport d'un système de production robotisé

Auteurs :
M. Jean-Baptiste Blanc-Rouchossé
M^{lle} Claire Delage
M. Enrique Maldonado
M. Maxime Maurin
M^{lle}Aurélie Quintana
M^{lle} Céline Tomé

Encadrants :
M^{me} Sandra Ngueveu
M. Cyril Briand

Remerciements

Tout d'abord, nous tenons à remercier Madame Sandra U. Ngueveu et Monsieur Cyril Briand qui nous ont permis de réaliser ce projet et qui nous ont aidés et suivis tout au long de ces 6 semaines en tant qu'encadrants de projet.

Nous remercions également Madame Francine Fugier, secrétaire de l'AIP-PRIMECA et Monsieur Thierry Canzoneri, membre de l'équipe technique, pour leur accueil et leur disponibilité chaque jour à l'AIP-PRIMECA.

Enfin, nous remercions aussi, Monsieur Bruno Dato, ancien étudiant travaillant sur le réseau de transport robotisé, pour sa gentillesse et son aide lorsque nous étions dans le besoin. Il est venu nous rendre visite afin de répondre à toutes nos interrogations et a su nous éclairer afin que nous puissions avancer dans notre projet.

Table des matières

Table des figures

Liste des tableaux

Introduction

Dans le cadre de notre dernière année d'école d'ingénieur, option Génie Electrique et Automatique, nous avons effectué un projet long d'une durée de 6 semaines au sein de l'AIP-PRIMECA.

La conception et la commande de systèmes de production suscite de plus en plus d'intérêts de nos jours avec la volonté de conserver un tissu industriel fort au niveau national et éviter les délocalisations. Dans ce cadre,un des objectifs de notre formations est de nous former à la conception, à la commande et à l'évaluation de systèmes de production automatisés. Ce projet long s'inscrit donc dans ce contexte puisqu'il s'agit de réaliser la commande d'un réseau de transport industriel

Par ailleurs, l'objectif de ce projet long est également de tester la faisabilité d'un nouveau TER qui pourrait remplacer celui existant.

Ce travail fait suite au projet long effectué en 2016 par d'anciens étudiants ENSEEIHT ainsi qu'au TER puis au stage réalisés par des étudiants de l'Université Paul Sabatier.

Le but de ce projet long était de finaliser la simulation et la commande d'un réseau de transport d'un système de production robotisé. Nous nous sommes particulièrement concentré sur la simulation en vue du temps imparti. En effet, il est important de réaliser une simulation de ce système, au préalable, car cela permet une vérification du plan de production programmé et garantit ainsi une sûreté vis-à-vis du matériel.

Dans ce rapport, nous présentons dans un premier temps l'AIP-PRIMECA, environnement dans lequel nous étions implantés durant 6 semaines. Ensuite, nous présentons les outils et systèmes sur lesquels nous avons travaillé pour faire notre simulation. Puis, nous détaillons l'évolution du projet en expliquant les différentes étapes réalisées et nous dressons un bilan à la fois technique et personnel de cette expérience. Pour finir, nous discutons des évolutions futures envisageables et des améliorations que nous n'avions pas eu le temps de mettre en place.

1 Présentation de l'AIP - PRIMECA

1.1 Réseau national de l'AIP-PRIMECA

Réparti sur tout le territoire et composé de 9 centres régionaux (voir figure 1), le réseau AIP PRIMECA [1] est un réseau académique de chercheurs et d'enseignants chercheurs associé à des moyens technologiques de haut niveau pour la mise en commun de moyens techniques.

FIGURE 1 – Répartition des pôles AIP-PRIMECA en France

Il est le résultat de la fusion entre :

- les A.I.P. (Ateliers Inter-établissements de Productique), créés en 1984 et utilisées comme support expérimental de formations approfondies dans le domaine de la Productique.

- PRIMECA (Pôles de Ressources Informatiques pour la MECAnique), créés en 1991 dans le but de promouvoir l'utilisation des outils informatiques dans la conception des produits mécaniques.

1.2 Le pôle AIP-PRIMECA de Toulouse

FIGURE 2 – Halle technologique de l'Université Jean Jaures, Toulouse

Les établissements toulousains (Université Paul Sabatier, INP Toulouse, INSA Toulouse, LAAS-CNRS) ont développé dès 1983 une politique de site afin de renforcer la formation pratique dans certains domaines nécéssitant des moyens lourds et couteux, en phase avec la réalité industrielle. Cette politique de mutualisation des ressources et des compétences s'est traduite pas la création de trois ateliers inter-universitaires dont l'AIP-PRIMECA, dans les domaines de la Mécanique et de la Productique.

2 Présentation de la cellule flexible

2.1 Le réseau de transport MONTRAC

L'AIP PRIMECA de Toulouse met à disposition de nombreux moyens dont la cellule flexible de production robotisée MONTRAC. Elle est constituée d'un réseau de transport monorail, appelé MONTRAC, sur lequel circulent des navettes. Ces dernières sont alimentées par les rails et sont donc en mouvement dès que la cellule est alimentée. Elles sont aussi autonomes et intelligentes : en effet, le seul moyen de les contrôler est de commander les actionneurs placés sur les rails via des automates.

Afin d'empêcher toute collision, chaque navette possède un capteur de proximité frontal qui permet de la stopper lorsqu'elle est trop proche d'une autre. La figure ci-dessous présente un schéma de la cellule :

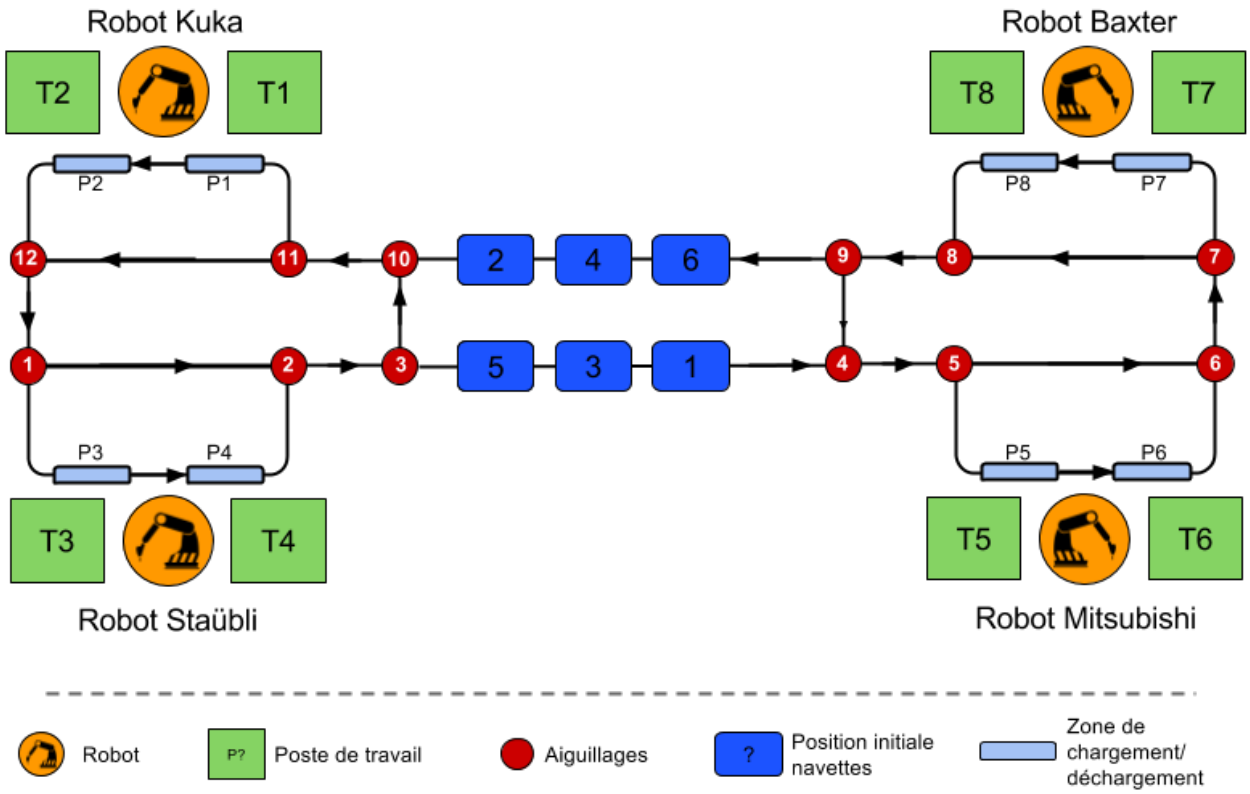

FIGURE 3 – Schéma de la cellule flexible

Comme nous pouvons le voir, la cellule est composée de 4 zones de travail (chacune séparée en 2 zones de chargement/ déchargement représentée en bleu et 2 postes de travail en vert) sur lesquelles 4 robots réalisent des opérations sur les produits transportés

par les navettes. Ces zones sont desservies par des monorails en aluminium sur lesquels les navettes circulent, tout en respectant le sens de circulation indiqué par les flèches noires. Ce sens unique de circulation est imposé par l'alimentation latérale des rails et permet d'éviter les risques de collisions frontales. Le routage est réalisé grâce aux 12 aiguillages présents (A1 à A12 représentés par des cercles rouges sur la figure 4). Le tout est divisé en 5 zones (ici invisibles), contrôlées chacune par un automate programmable.

2.2 Capteurs et Actionneurs

Les capteurs permettent de connaître la position des navettes, des aiguillages et des ergots. Les ergots servent à bloquer les navettes au niveau des zones de chargement/déchargement, ce qui permet aux robots de manipuler les produits positionnés sur les navettes, sans risque de les déplacer accidentellement. Les actionneurs permettent de commander les points d'arrêts des navettes, les aiguillages et la position des ergots. Les listes des capteurs et actionneurs sont données ci-dessous, ainsi qu'un schéma de leur répartition dans la cellule :

RxG	Positionner l'aiguillage x à gauche
RxD	Positionner l'aiguillage x à droite
Dx	Déverrouiller l'aiguillage x
Vx	Vérrouiller l'aiguillage x
STx	STx = 0 arrête la navette au niveau de l'actionneur (=1 : libération de la navette)
PIx	Blocage/déblocage des navettes sur le zone de chargement

Tableau 1 – Tableau des actionneurs

CPx	Capteur de position. Vaut 1 quand une navette est sur le capteur
PSx	Capteur de Stop situé juste en face d'un actionneur STx pouvant arrêter la navette
CPIx	Vaut 1 quand l'ergot PI est sorti
DxD	Vaut 1 quand l'aiguillage est à droite
DxG	Vaut 1 quand l'aiguillage est à gauche

Tableau 2 – Tableau des capteurs

Dans la conception des simulations précédentes (voir Annexe 8), les étudiants avaient considéré les capteurs CPIx des postes de travail comme capteurs de position et capteurs de stop (au niveau des postes). En revanche, sur la maquette réelle ce sont des capteurs de position des ergots (=1 si l'ergot est sorti). Pour palier à ce problème, nous avons rajouté des capteurs PSx sur la simulation (associés aux stops STx), qui ont pour but d'être capteurs de position (que la navette soit à l'arrêt ou en mouvement). Ensuite, nous avons attribué les bonnes fonctionnalités aux capteurs CPIx et donc adapté l'interface utilisateur de la commande locale.

Ci-dessous, un schéma de la cellule flexible modifiée avec capteurs :

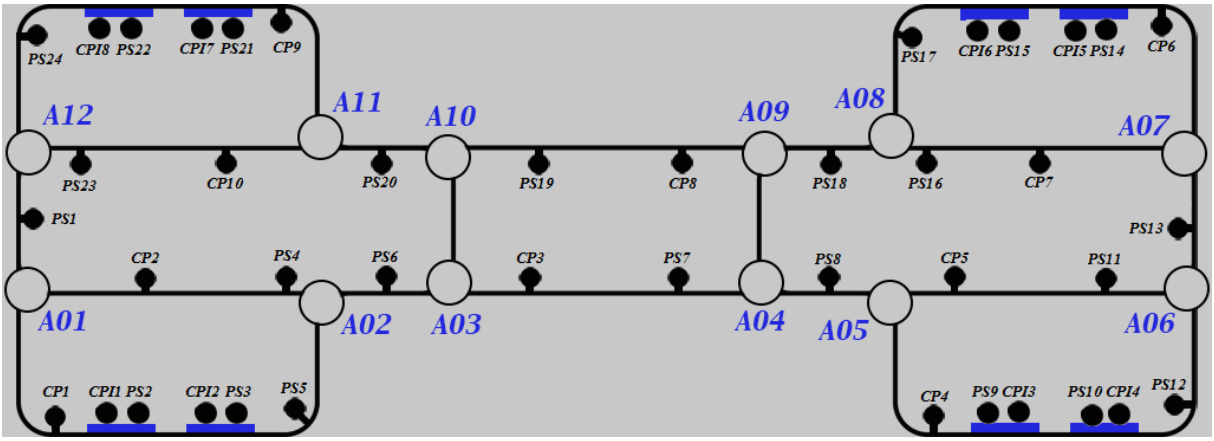

FIGURE 4 – Schéma de la cellule avec capteurs

Les actionneurs STOP/START (STx), non visibles ici, permmettent de bloquer les navettes et sont placés avant les aiguillages et au niveau des postes. Les actionneurs PIx font sortir l'ergot, permettant de stabiliser la navette au niveau des postes de travail. De plus, à chaque aiguillage sont associés deux capteurs DxD et DxG qui permettent de détecter sa position (respectivement droite ou gauche).

3 Objectifs et organisation du travail

3.1 Objectifs

A l'heure actuelle, l'utilité de cette maquette est de pouvoir développer des Travaux Pratiques destinés à des étudiants de cycle supérieur issus de différentes formations de Toulouse, en particulier pour l'ENSEEIHT qui souhaiterait remplacer une ancienne maquette qui ne fonctionne plus. Ces TP devraient permettre de former les étudiants à la commande haut niveau de systèmes de production en respectant un ordonnancement pré-établi et les gammes de production. Il s'agirait de contrôler la destination des navettes, ou encore de charger/ décharger différents produits au niveau des zones de travail robotisées.

Ce projet s'inscrit dans la problématique de l'usine du futur, dans le sens où l'ensemble des différents capteurs présents sur la cellule permettent une communication constante entre les machines et les humains. En effet, il y a maintenant plusieurs années que l'on commence à parler d'usine du futur, ou « usine 4.0 », capable de faire communiquer les machines entre elles. Son principe est "qu'à chaque maillon des chaînes de production et d'approvisionnement, les outils et postes de travail communiquent en permanence grâce à Internet et aux réseaux virtuels. Machines, systèmes et produits échangent de l'information, entre eux ainsi qu'avec l'extérieur" [2].

Dans notre cas, il y a une communication constante entre l'utilisateur et la cellule, mais également une communication constante entre chaque navette, les rails et les robots. Il devient alors beaucoup plus aisé de localiser la source d'une panne et de rediriger la production sur d'autres machines afin d'éviter l'arrêt total de la production durant le temps de manutention de la machine en panne [3]. Nous produisons ainsi un suivi en temps réel de l'avancement des différentes tâches de la production et de l'acheminement, ce qui permet d'avoir une gestion et une planification de la production plus performante.

Notre projet s'organise autour de deux objectifs principaux :

- Le premier objectif est de continuer le développement d'une simulation 3D correspondant au mieux à la maquette réelle. Celle-ci permettra aux étudiants de réaliser des tests préalables, avant de tester leur application d'ordonnancement et de commande sur la maquette réelle. Ils pourront ainsi contrôler leur code et corriger les erreurs sans aucun risque pour le matériel. De plus, dans le cas où les groupes de TP seraient nombreux, nous avons considéré que la maquette peut se diviser en deux parties symétriques, ce qui permettra de faire travailler plus d'élèves en même temps sur la maquette réelle, plutôt que devoir attendre qu'elle soit libre.

- Le second objectif est de créer un espace de travail pour les étudiants : la couche haute. Ils devront coder certaines parties et l'envoyer à la simulation afin de la tester par le biais du middleware ROS (Robot Operating System).

D'un point de vue pédagogique, l'objectif est que les étudiants se forment aux problématiques de gestion de ressources partagées rencontrés sur les chaines de production. La modélisation de ce problème se fera à l'aide de réseaux de Petri. Afin que celui-ci ne soit pas trop compliqué à mettre en place, l'idée est que les étudiants aient accès à des fonctions de haut niveau pour coder la couche haute du programme.

Il est aussi facilement envisageable de faire réaliser une petite partie de la couche basse du programme pour faire découvrir aux étudiants d'autres aspects de la programmation objet, les notions de programmation distribuée ou même des bases sur ROS. Enfin les étudiants pourront faire un lien avec le TER portant sur les problématiques d'ordonnancement utilisant le logiciel ARENA présent à l'AIP-PRIMECA. Ce lien sera établi grâce à la possibilité de jouer des séquences de lancement de produits précises, qu'ils auront préalablement simulé dans ARENA, sur le système réel (ou du moins dans un premier temps sa simulation V-REP).

3.2 Organisation

Nous avons basé l'organisation de notre projet sur la méthode Agile. Cette méthode consiste à livrer régulièrement des fonctionnalités à forte valeur ajoutée ainsi qu'une application fonctionnelle finale.

Chaque semaine, nous établissions une répartition des tâches et les objectifs à atteindre à la fin de la semaine. En effet, à chaque semaine était organisée une réunion durant laquelle nous présentions l'avancement du projet et les futures évolutions à faire. Nos encadrants nous aidaient en commentant notre travail afin de mettre en avant les corrections et/ou améliorations fonctionnelles à réaliser pour la semaine suivante. De plus, ces réunions nous permettaient aussi d'éclaircir certains points, en particulier la trame du futur TER à préparer pour les prochains étudiants.

Par ailleurs, chaque réunion menait à la rédaction d'un compte-rendu afin de pouvoir conserver une trace écrite de ce qui avait été dit et de pouvoir envoyer, un bilan et un ordre du jour à nos encadrant.

Nous avons eu l'avantage de ne pas partir de zéro. Une plateforme de simulation et un environnement commande sur maquette réelle avaient déjà été réalisés. Ainsi, nous avons commencé à lire tous les livrables mis à notre disposition. Mais pour comprendre le travail effectué, nous avons dû passer par une phase de prise en main de ROS et de V-REP. En effet, ces deux outils nous étaient alors inconnus. Nous avons donc suivi de nombreux tutoriels que l'on retrouve dans la bibliographie en [4], [5], [6] et [7] afin de comprendre leurs fonctionnements et donc les codes de nos camarades pour commencer à remplir nos objectifs (voir partie 4.1).

Il est important de noter que par manque de temps, ce projet long se concentre sur la partie gauche de la simulation.

Concernant la répartition, nous nous sommes, dans un premier temps, divisés en 3 groupes de 2 personnes que nous avons adaptée par la suite. L'ensemble des tâches

réalisées et l'intégration globale sont présentées dans la suite du rapport.

Cette organisation nous a permis de nous répartir les tâches à réaliser chaque semaine. Chaque membre du groupe s'occupait d'un package, le codait et réalisait les différentes simulations nécessaires aux tests de validation. C'est vers la fin du projet long que nous avons petit à petit commencé l'intégration des différents packages et effectué les modifications réglant au mieux les problèmes.

4 Prise en main des outils mis à disposition

4.1 Outils ROS et VREP

4.1.1 ROS

Comme dit précédemment, pour la réalisation du projet, nous avons hérité des choix d'outils utilisés par les groupes précédents.

Le premier est le middleware **ROS (Robot Operating System)**, une plateforme de développement logiciel qui fournit des bibliothèques et des outils pour aider les développeurs de logiciels à créer des applications robotiques. Pouvant fonctionner sur un ou plusieurs ordinateurs, il procurent de nombreuses fonctionnalités telles que l'abstraction du matériel, le contrôle des périphériques de bas niveau, la transmission de messages entre les processus et la gestion des *packages* installés.

De manière simple, ROS permet de créer des sous-programmes appelés noeuds ou *nodes* en anglais, qui peuvent communiquer entre eux à l'aide de messages synchrones ou asynchrones. ROS offre une architecture souple de communication inter-processus et inter-machine. Dans le cas de messages asynchrones, les processus ROS *nodes*, peuvent communiquer avec d'autres via des topics.

La connexion entre les *nodes* est gérée par un master et suit le processus suivant :

1- Un premier *node* avertit le master qu'il a une donnée à partager

2- Un deuxième *node* avertit le master qu'il souhaite avoir accès à une donnée

3- Une connexion entre les deux *nodes* est créée

4- Le premier *node* peut envoyer des données au second

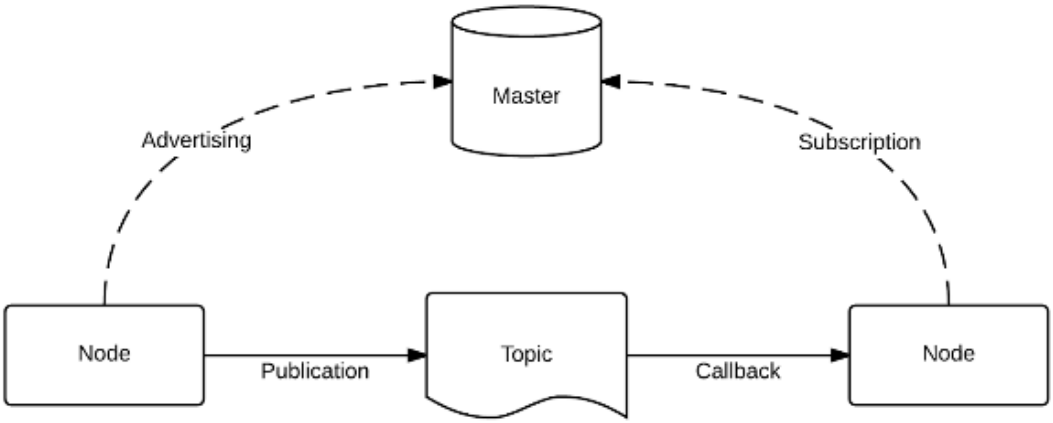

FIGURE 5 – Diagramme de fonctionnement du middleware ROS

Un *node* qui publie des données est appelé un *publisher*, et, un *node* qui souscrit à des données est appelé un *subscriber*. Un *node* peut être à la fois *publisher* et *subscriber*. Les messages envoyés sur les topics sont, pour la plupart, standardisés, ce qui rend le système extrêmement flexible.

Pour les messages synchrones, que l'on appelle *services*, les nœuds ont seulement accès aux services offerts par chacun d'entre eux, par l'intermédiaire de *clients*. Sans entrer dans le détail, les entrées et sorties de chaque *service* sont définies au niveau des nœuds où ils sont instanciés, et elles sont connues des noeuds *clients* correspondants. Il est important de remarquer que ROS permet une communication inter-machine, des *nodes* s'exécutant sur des machines distinctes, mais ayant connaissance du même master peuvent communiquer de manière transparente pour l'utilisateur. Ceci représente une des grandes forces de ROS.

De plus, ROS est sous licence open source, ce qui permet d'avoir accès à une grande communauté et de disposer de nombreux tutoriels. ROS est aujourd'hui, officiellement supporté par plus de 75 robots. La grande souplesse de ROS lui permet d'être déployé sur des robots très différents (robot mobile, bras industriel, multicoptère) et qui évoluent dans des milieux variés (terrestre, aérien, marin et sous-marin).

4.1.2 V-REP

Le second outil que nous avons utilisé est V-REP, un simulateur dédié à la simulation de scènes composées de robots et développé par la société Coppelia Robotics. L'intérêt de ce simulateur est qu'il est compatible avec ROS. En d'autres termes, il est vu par nos nœuds ROS comme un nœud spécifique, avec lequel nous pouvons communiquer.

Il permet la simulation de la cellule MONTRAC, se situant à l'AIP. La scène a été initialement créée par le groupe précédent, et nous y avons ajouté de nombreux éléments au cours du développement de notre projet long. En plus des modèles 3D des navettes et des rails de la cellule, on retrouve aussi les modèles de robots KUKA que nous avons utilisé pour simuler notre atelier de fabrication. Nous avons également apporté quelques modifications au niveau de l'emplacement des capteurs (voir paragraphe 2.2). La simulation V-REP finale permet donc de faire évoluer les navettes sur les rails, tout en communiquant l'état des capteurs au reste de notre simulateur, grâce à son noeud ROS.

5 Avancement et Répartition des tâches

L'objectif à la fin de la première semaine était que l'on sache tous utiliser les logiciels ROS et V-REP. Comme notre rôle durant ce projet long a été de mettre en place une simulation fonctionnelle et adéquate qui puissent être utilisée par les étudiants au cours de leur formation., nous avons commencé par comprendre ce qui avait été fait précédemment.

5.1 Simulation de la ligne transitique

La vision globale du projet, cible la réalisation d'une application permettant de basculer facilement entre la simulation et la commande de la cellule MONTRAC ainsi que les robots qui l'entourent. En suivant ces considérations, nous avons identifié les différents acteurs et les cas d'utilisation, c'est-à-dire les lots d'actions que devront réaliser nos acteurs (voir figure 6).

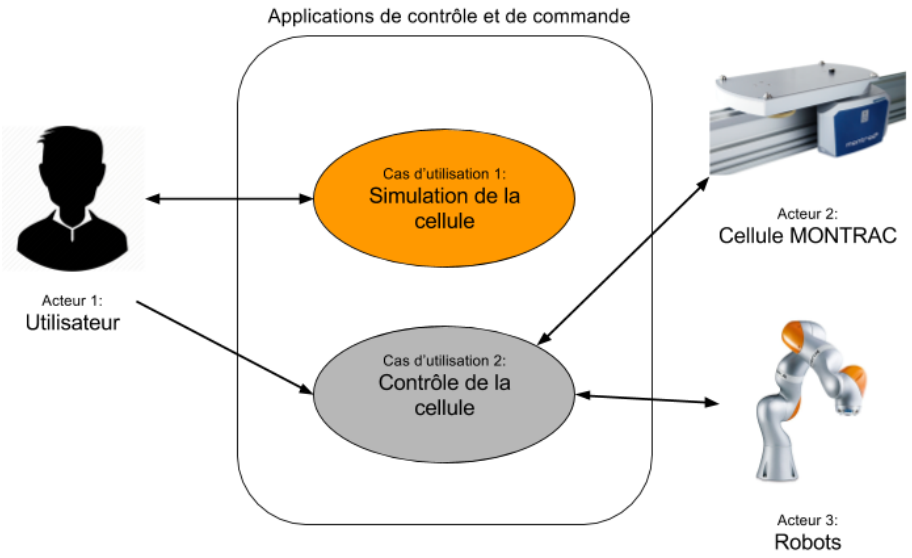

FIGURE 6 – Diagramme des cas d'utilisation

Comme dit lors de l'introduction, lors de ce projet long, nous nous sommes particulièrement concentrés sur la simulation de la cellule. C'est pour cette raison que le cas d'utilisation "Contrôle de la cellule" est grisé sur la figure 6. Cependant, il a été pris en considération pour quelques choix pendant le déroulement du premier cas d'utilisation.

5.2 Installation de la simulation

Pour travailler avec le middleware ROS, il est nécessaire de créer un espace de travail (Workspace en anglais).

Il a été important de suivre un tutoriel approfondi pour savoir initialiser l'espace de travail dans un nouvel ordinateur et pouvoir ainsi utiliser la simulation.

5.2.1 Exécutable

Nous avons pris la décision de développer un code en C++ qui crée un exécutable capable d'installer la simulation sur un nouvel ordinateur de façon automatique.

Ainsi, cet exécutable remplace le tutoriel initial et a pour but d'aider les futurs élèves à démarrer leur travail plus facilement sans refaire tout le tutoriel au préalable. On l'a appelé "setup" et son utilisation est décrite dans le README du livrable final de ce projet.

5.2.2 Modèle des navettes

Afin d'améliorer la simulation et l'adapter à nos besoins, nous avons changé les modèles graphiques des navettes dans l'environnement V-Rep. Il faut préciser que les navettes ne font pas partie de la simulation (fichier .ttt), ce sont des modèles V-Rep (fichiers .ttm) qui sont insérés tout au long de la simulation. Une fois une navette insérée dans la simulation, V-Rep lui donne automatiquement un identifiant unique appelé "handle". Il est important de connaître cet identifiant si l'on veut s'adresser aux navettes et réaliser des fonctions avec elles (comme changer la destination, changer la couleur, détruire la navette, etc.).

Un modèle V-Rep a un code source dans lequel nous pouvons coder plusieurs fonctionnalités. Nous en avons utilisé un de type séquenciel ("non-threaded" en anglais, pour plus d'informations, voir [**?**]).

Ce code est divisé en quatre parties importantes dans lesquelles nous avons mis plusieurs actions des navettes. Les parties sont décrites ci-dessous, ainsi que les fonctions réalisées pour les navettes dans chaque partie :

- **Initialisation :** cette partie s'exécute une seule fois (quand le modèle est inséré dans la simulation). Dans cette partie nous créons un signal de type "entier" pour changer la couleur de la navette depuis un noeud ROS. Ce signal est appelé "[Handle_Navette]_color". Nous avons la possibilité de changer la valeur de ce signal grâce à un service ROS déjà prédéfini, pour agir avec V-Rep.
- **Détection et mise en marche :** cette partie s'exécute à chaque pas de simulation de V-Rep dans laquelle il évalue le signal (de la couleur) de la navette et le met à jour. Nous avons utilisé un code de couleur présenté dans le tableau 3 en Annexe 8 . On associe une couleur à chaque valeur d'entier du signal.
- **Clean-Up :** cette partie s'exécute une fois que la navette est détruite mais nous ne l'utilisons pas.

Comme précisé dans la partie 3.1 le deuxième objectif de ce projet long est de permettre aux étudiants de mettre en pratique leurs connaissances et l'utilisation des réseaux de Petri. Pour cela, il nous a fallu créer la couche basse de l'ensemble du programme afin qu'ils aient à disposition des fonctions haut niveau qui leur permettront de faire évoluer les différents composants de la cellule flexible en simulation. Ainsi, nous avons commencé la conception de nos différents noeuds. Il y a un noeud par "entité" (navette, robot, tâche, aiguillage, etc)

5.3 Noeud Ordonnancement

Afin de permettre aux étudiants de jouer des scénarios de production prédéfinis, nous avons construit un nœud ordonnancement qui se configure par l'intermédiaire du fichier de configuration "ProductConfiguration.config". A l'aide de celui-ci, on définit les produits à fabriquer, leur gamme de fabrication, le temps requit pour chaque étape et le temps d'attente entre les lancements des différents produits. Il permet également de spécifier une limite sur le nombre de navettes en circulation, et nous conseillons de ne pas dépasser 6 navettes afin que la simulation ne soit pas trop lente. Ce fichier est consulté uniquement par les nœuds *Ordonnancement*, *Poste* et *Tâche*, afin de permettre la réalisation de certaines de leurs fonctionnalités.

La création de ce noeud est essentiellement motivée par la volonté d'établir un lien entre ce TER et ce que les étudiants seraient en capacité de simuler à l'aide du logiciel ARENA. Sur ce dernier, ils rechercheraient la séquence optimale de lancement vis-à-vis de critères tels que le taux de production, et, grâce au fichier de configuration, ils pourraient la jouer en simulation et comparer les temps de production avec ceux donner par ARENA. Toutefois, à la vue de la vitesse de la simulation V-REP, ce dernier point ne semble pour l'instant pas réaliste.

D'un point de vue fonctionnel, ce noeud est assez simple, il communique via messages asynchrones avec le nœud *Commande* pour le lancement des navettes. Il se synchronise également avec V-REP à l'aide d'un service ROS disponible dans le package "vrep-common". Cette synchronisation est nécessaire pour permettre le lancement des navettes en accord avec le temps de la simulation, qui évolue plus lentement que le temps OS de manière générale. De plus il souscrit à un topic de la commande locale pour être informé de tout lancement de navette effectué manuellement. Cela permet un retour à 6 navettes maximum si on réenclenche le mode auto.

De plus, ce noeud génère un fichier de log pour analyser les entrées et sorties des produits de la demi-cellule. "Statistic.txt" présente ainsi les dates de lancements des produits de base et celles de sorties des produits finis.

Les capacités offertes sont pour l'instant limitées mais des améliorations sont facilement envisageables dans les évolutions futures de l'application. Celles-ci sont listées dans la partie 7.

5.4 Noeud Navette

Afin de gérer au mieux les navettes, nous avons choisi de créer un nœud *Navette* indépendant. L'idée de ce nœud est venue du fait que la solution que nous avons conçue présente de nombreux nœuds tels qu'*Aiguillage* ou *Poste* qui doivent interagir avec les navettes. La centralisation des informations sur les navettes au niveau d'un nœud unique nous a permis d'en faciliter la gestion.

L'idée est que ce nœud soit informé par les autres nœuds de la simulation des changements opérés sur les navettes. Il va posséder en interne une collection d'objets « Shuttle » présentant de nombreux attributs présentés ci-dessous :

```
class Shuttle
{
public:

// un nom unique est attribué à chaque navette
       std::string name;

// L'attribut destination permet de connaitre la destination de la navette
       int destination;

// L'attribut product permet de connaitre le produit transporter par la navette et
l'étape de la gamme dans lequel il se trouve.
       int product;

// Les attributs handle et handlePlatform sont des identifiants uniques attribuer par
Vrep au navette, ils nous permettent d'identifier les navettes et également de changer
la couleur de la platform lorsque le produit transporter évolue.
       int handle;
       int handlePlatform;

// L'attribut zone pourrait permettre de connaitre en permanence la position de la
navette mais cette fonctionnalité reste à implanter.
       int zone;

       [...]
};
```

Le nœud connaît donc en permanence l'état des navettes dans le circuit. Grâce à cela il va offrir des services permettant la récupération des informations sur les navettes ou encore la recherche des navettes ne transportant pas de produit. Pour l'informer des changements les autres nœuds n'ont qu'à publier des messages sur des topics dédiés. Des topics sont aussi disponibles pour la création et la destruction des navettes.

5.5 Noeud Aiguillage

Les noeuds *Aiguillage* permettent de donner l'ordre aux aiguillages de s'orienter en fonction de la prochaine destination de la navette. Pour cela, nous avons décidé de faire 1 noeud par aiguillage, le tout regroupé dans un même package. En se divisant les tâches, un membre du groupe a fait le code des aiguillages pairs (2 entrées, 1 sortie) et un autre, ceux des aiguillages impairs (1 entrée, 2 sorties).

5.5.1 Composition des noeuds aiguillages

L'aiguillage est géré par une fonction principale exécutée de manière répétitive : c'est cette fonction qui va prendre la décision de faire tourner l'aiguillage et de faire redémarrer une navette. Pour cela, elle utilise un certain nombre de fonctions bas niveau telles que les fonctions :

- **START()** et **STOP()** (de façon similaire, **START_DROIT()** , **START_GAUCHE()**, **STOP_DROIT()**, **STOP_GAUCHE()** pour les aiguillages avec deux entrées).
Elles permettent d'activer et désactiver les stops en entrée de l'aiguillage

- **DROIT()** et **GAUCHE()**.
Elles font tourner l'aiguillage. Mais elles sont bloquantes : elles se terminent lorsque l'aiguillage est en position.

Cette fonction principale a également connaissance des états des capteurs qui la concerne par des attributs mis à jours par des callbacks.

Les aiguillages ayant deux sorties ont également besoin de connaître la destination de la navette pour s'orienter. Pour cela, ils utilisent le handle de la navette pour faire appel au service du noeud *Navette* pour avoir la destination. Par conséquent, les aiguillages doivent s'échanger les handles des navettes afin d'utiliser le service.

L'ensemble de ces fonctionnalités est donc disponible via d'autres fonctions bas niveau telles que **get_Sh_Handle()** pour avoir le handle de la navette qui arrive, et **Send_Sh()** qui permet d'envoyer le handle de la navette qui passe à l'aiguillage suivant (ou au poste si l'aiguillage fait dériver la navette vers un poste). L'utilisation de ces fonctions nécessite dans certains cas un paramètre d'entrée tel que l'origine de la navette (1 si elle vient de droite et -1 si elle vient de gauche) pour les aiguillages à deux entrées, ou la destination (1 ou -1) pour les aiguillages à deux sorties. Le logigramme explicatif de l'aiguillage 11 se trouve en Annexe 8.

5.5.2 Cas des aiguillages A3 et A10

Ces aiguillages sont particuliers parce qu'il n'y a pas de capteur entre les deux. Pour les autres aiguillages, lorsqu'une navette passe, elle s'arrête à l'entrée de l'aiguillage (ou poste) suivant qui s'orientera en fonction de sa destination. Cependant, ce n'est pas le cas pour ces aiguillages, lorsque l'aiguillage A3 s'oriente à gauche, il est important qu'il soit synchronisé avec l'aiguillage A10 afin que la navette puisse faire demi-tour.

Dans un premier temps, nous avons envisagé de ne faire qu'un noeud pour les deux aiguillages, ce qui permet de les synchroniser facilement puisqu'ils sont gérés ensemble. La solution obtenue n'était pas efficace : si deux navettes devaient aller tout droit, une en A10 et une autre en A3, les deux ne pouvaient pas passer en même temps. En effet, la solution consistait à considérer ces deux aiguillages comme un seul, avec deux entrées et deux sorties. Cependant, si on utilisait un algorithme similaire aux cas précédents, une seule navette n'aurait pu passer l'aiguillage commun A3-A10 (géré par un seul noeud) et donc deux navettes n'auraient pas pu aller tout droit en même temps : ce qui rend cette

solution inefficace. Nous avons donc décidé de créer deux noeuds pour ces deux aiguillages, synchronisés par deux topics : l'un permettant à l'aiguillage 3 de signaler au 10 qu'une navette veut faire demi-tour, et l'autre pour que l'aiguillage 10 prévienne l'aiguillage 3 qu'il est en position puis que la navette est sortie de l'aiguillage comme le montre la figure suivante :

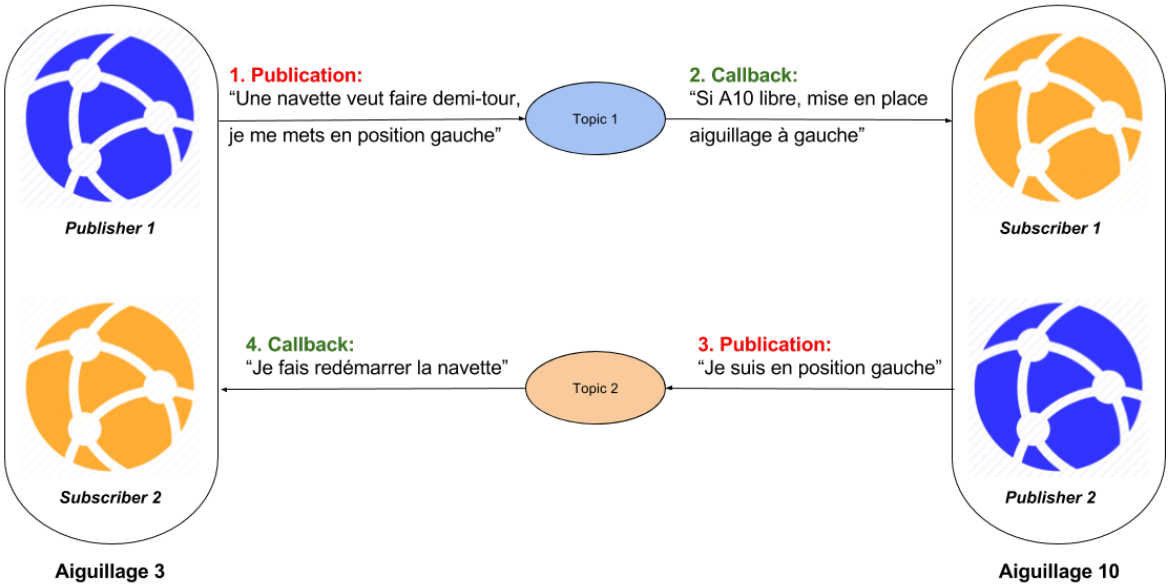

FIGURE 7 – Schéma explicatif des deux topics gérés par les aiguillages 3 et 10

5.6 Noeud Poste

Nous avons séparé les zones d'arrêt devant les robots en deux pour avoir deux fois plus de postes de chargement/déchargement. Il y a donc deux postes devant le robot en haut à gauche (P1 et P2) et deux autres devant le robot en bas à gauche (P3 et P4). Lorsqu'une navette s'arrête à un de ces quatre postes, le robot peut récupérer le produit pour l'amener au poste de travail (tâche) pour le traiter et inversement, le robot peut déposer un produit traité sur la navette arrêtée à un des postes devant lui pour que celle-ci emmène le produit vers une autre tâche.

5.6.1 Réalisation

Dans un premier temps, nous avons fait deux noeuds ROS pour gérer les postes : un pour les postes P1 et P2 et le deuxième pour les postes P3 et P4. L'aiguillage précédant les postes communique au noeud poste le numéro de handle de la navette qui arrive. Grâce au noeud Shuttle, nous pouvons savoir toutes les informations liées à cette navette comme la destination de la navette et le numéro du produit qu'elle transporte (produit = 0 si la navette est vide).

Nous avons rencontré des problèmes au niveau de la synchronisation entre les deux postes d'un même noeud. Par exemple, si la première navette s'arrête au poste P1 et la

deuxième navette veut aller en P2, quand faut-il déclencher le stop du poste P2 pour ne pas bloquer la navette qui part de P1 ?

Pour faciliter la réalisation des postes de chargement/déchargement, nous avons décidé de faire un noeud ROS par poste ce qui nous fait donc 4 noeuds pour la partie gauche de la simulation. Ainsi, si nous prenons comme exemple les postes 1 et 2, l'aiguillage A11 va communiquer au poste P1 le numéro de handle de la navette qui arrive. P1 vérifie la destination de la navette, si elle vaut 1, le stop au niveau du poste P1 est enclenché, sinon, P1 communique à P2 le numéro de handle de la navette. P2 vérifie à son tour la destination de la navette, si c'est 2 alors le stop au niveau du poste P2 est enclenché, sinon, P2 communique à l'aiguillage A12 le numéro de handle de la navette.

5.6.2 Le main

On retrouve le logigramme du poste P1 en Annexe 8 :

La variable *poste_libre* est à 0 quand le poste est occupé. Elle est mise à 1 à l'initialisation et lorsqu'on détecte un front descendant du capteur PS21. La variable *tâche* est à 0 quand il n'y a pas de tâche finie et donc pas de produit à récupérer. La variable *prod_recup* est à 0 quand un produit n'est pas en cours de traitement.

Concernant le fonctionnement des trois autres postes, il suffit d'adapter le logigramme du poste 1 en changeant la destination, le numéro des actionneurs, le numéro des capteurs et le destinataire de l'envoi du numéro de handle de la navette.

5.6.3 Les fonctions callback

Lorsqu'on souscrit à un topic, une fonction callback est appelée, et comme un poste souscrit à 6 topics, il y aura 6 fonctions callback par poste.

Les postes souscrivent à trois topics différents venant de la commande.

- Un topic "redémarrer navette" si on ne veut pas traiter cette navette. Dans la fonction callback associée, on rentre les ergots, on redémarre la navette, on signale que le poste est libre et on publie le numéro de handle de la navette au noeud suivant (*Poste* ou *Aiguillage*).

- Un topic "produit récupéré par le robot". Une fois le message reçu, on met à 1 la variable *prod_recup*, on met le produit et la destination à 0, on envoie les données à la commande locale pour changer la couleur de la navette, on renseigne le nouveau produit et la nouvelle destination au noeud Navette, on rentre les ergots, on redémarre la navette, on signale que le poste est libre et on publie le numéro de handle de la navette au noeud suivant.

- Un topic " produit posé sur navette". Dès la réception du message, on met à 0 les variables tache et prod recup, on envoie les données à la commande locale pour changer la couleur de la navette, on renseigne le nouveau produit au noeud *Navette*, on rentre les ergots, on redémarre la navette, on signale que le poste est libre, on publie le numéro de handle de la navette au noeud suivant.

Ils souscrivent également à un topic venant des tâches contenant le nouveau numéro du produit lorsque l'opération sur celui-ci est terminée. Dans la fonction associée, on récupère le numéro du produit et on met à 1 la variable signalant qu'il y a un produit à récupérer.

Pour récupérer les informations d'une navette, ils souscrivent à un topic venant du noeud *Navette*. Une fois le message reçu, on récupère les différentes informations de la navette.

Afin de connaître l'état des capteurs PSx, les postes sont abonnés au topic venant de VREP. Dans la fonction associée, on attribue à la variable PSx l'état du capteur qui correspond.

5.7 Noeud Robot

La cellule flexible est composée, en plus de la ligne Montrac, de quatre robots industriels (figure 4). Nous avons choisi de les modéliser en simulation uniquement par des modèles KUKA LBR IIWA 14 R820, des robots industriels 6 axes, afin de pouvoir implémenter un code facilement applicable à l'ensemble des robots.

Chaque robot est une ressource partagée entre quatre emplacements qui lui sont associés : deux zones de chargement/déchargement et deux postes de travail, ou tâches, qui permettent de traiter les produits.

Pour commander ces robots, des fonctions haut niveau seront à disposition des élèves afin qu'ils puissent les déplacer d'une position à une autre, de descendre ou monter le bras ou encore d'ouvrir ou fermer la pince. Il est donc nécessaire que ces robots puissent évoluer en parallèle et être commandés indépendamment des autres. Pour cela, nous avons choisi de réaliser un nœud ROS par robot.

5.7.1 Réalisation

Chaque composant défini dans le logiciel V-Rep est défini par un entier appelé "handle" qui lui est propre. Nous allons pouvoir contrôler les robots en simulation à l'aide de ces entiers, qui sont obtenus à partir du nom complet des robots. Ce nom sera donc le seul élément qui ne sera pas commun aux différents nœuds ROS associés aux robots.

Une fois le handle associé à un robot récupéré, nous allons pouvoir commander les mouvements de celui-ci. Pour cela, nous avons réalisé plusieurs fonctions internes au nœud ROS permettant le contrôle du robot : une pour atteindre une position prédéfinie, une permettant d'envoyer le robot dans une position définie par ses arguments, une mettant le bras en position basse, une autre pour relever le bras, ou encore une fermant la pince du robot, et une dernière pour ouvrir cette même pince.

5.7.2 Les fonctions internes au nœud ROS

Exceptées celles permettant la commande de la pince, les fonctions que nous venons de citer sont toutes implémentées de la même manière.
Dans un premier temps, on définit un tableau de 7 valeurs. Ce tableau contient la valeur des différents angles du robot, permettant de définir la position de celui-ci. Ensuite, on utilise un service qui communique entre notre nœud ROS et le logiciel V-Rep afin de placer le robot, identifié par son handle, dans la position que nous venons de définir.

Après s'être assuré que ce service avait été correctement appelé, nous en utilisons un deuxième qui permet d'obtenir la position actuelle du robot. Ainsi, nous allons pouvoir tester l'erreur entre cette position réelle et la consigne définie afin de savoir si le mouvement du robot est terminé. On considère que la position souhaitée est atteinte lorsque la différence entre celle-ci et la consigne est inférieure à 0,1%.

Enfin, une fois ce test validé, on envoie des retours en publiant sur différents topics pour signifier à la commande ou aux autres nœuds nécessitant la position du robot que le mouvement est terminé et que la position souhaitée est atteinte. Par exemple, pour envoyer un robot dans une position prédéfinie, on utilise la fonction **EnvoyerRobot(int numposition)**. On peut retrouver le logigramme de fonctionnement du mouvement de robot en Annexe 8.

La commande de la pince est réalisée de manière différente. Pour fermer ou ouvrir la pince, il faut simplement publier un message sur un topic permettant la communication avec l'objet V-Rep, 0 correspondant à l'ordre d'ouverture et 1 à celui de fermeture. De plus, comme pour les fonctions permettant de contrôler la position du robot, des retours sont réalisés pour connaître l'état de la pince à tout moment, mais aussi pour s'assurer que la pince est effectivement fermée ou ouverte. Ce dernier retour ne pouvant être réalisé à l'aide d'un service V-Rep, nous avons implémenté une temporisation basée sur le temps de la simulation. On considère qu'après une seconde V-Rep, la pince a fini son mouvement.

5.7.3 Les fonctions Callback

Ces fonctions internes permettent au robot d'être commandé par le programme principal associé au nœud. Cependant, ce n'est pas ce que nous souhaitons dans ce projet puisque nous voulons que les robots soient contrôlés par l'utilisation de fonctions haut niveau. Aussi, des fonctions callback sont créées. Ces fonctions permettent de réaliser différentes actions après la réception d'un message sur un topic. Le programme principal contient alors uniquement la fonction qui permet d'appeler ces fonctions callback dès la réception d'un message sur le topic associé.

Aussi, chacun des nœuds *Robot* possède 6 fonctions callback qui vont permettre de contrôler le robot à l'appel de fonctions définies dans le nœud *Commande* (cf paragraphe 6.2). Les étudiants peuvent donc envoyer le robot dans une position prédéfinie ou dans une position qu'ils peuvent définir manuellement, monter ou descendre le bras, ou encore ouvrir ou fermer la pince.

Cependant, lors de la réalisation du nœud *Commande*, nous avons pu constater que l'appel de ces fonctions était bloquant et ne correspondait pas à la logique du réseau de Petri. Aussi, une dernière fonction a été implémentée pour pouvoir contrôler entièrement le robot. Lors de la réception du message associé, c'est-à-dire lorsque la fonction haut niveau correspondante est appelée dans le nœud *Commande*, on va envoyer successivement l'ordre de position prédéfinie, celui de commande du bras et celui de commande de la pince. On peut retrouver le logigramme de fonctionnement du contrôle du bras et de la pince du robot en Annexe 8.

5.8 Noeud Tâche

Comme nous l'avons dit précédemment, nous avons choisi de représenter les produits par des couleurs, les objets physiques étant trop compliqués à gérer dans la simulation, notamment au niveau des robots qui sont censés prendre et déplacer ces objets.

Pour simuler le traitement des produits, par exemple l'assemblage de plusieurs composants et donc leur évolution, nous allons augmenter l'intensité de la couleur de base au fur et à mesure de la gamme de fabrication. Chaque couleur est associée à une valeur : par exemple, la valeur 10 est représentative d'un rouge clair, couleur de base du produit A ; 11 étant un rouge plus foncé représentatif du premier stade de traitement du produit. La couleur du produit va donc évoluer au niveau des postes de travail, aussi appelés tâches. La cellule flexible contient 8 postes de travail, qui sont associés deux à deux à l'un des robots industriels présents dans le système.

5.8.1 Réalisation

Lorsqu'une navette contenant un produit arrive au niveau d'une zone de chargement /déchargement, le robot associé à cette zone doit se saisir du produit pour l'emmener à une tâche. Nous avons décidé qu'un produit présent à la zone Px ne pouvait être traité que par la tâche Tx. Le robot ne peut, par exemple, pas emmener un produit présent en P1 en T2 pour y être traité.

Pour pouvoir réaliser des traitements en parallèle les uns des autres, nous avons réalisé un nœud par tâche, c'est-à-dire 8 nœuds au total. Un nœud *Tâche* est associé à une table représentant les postes de travail dans notre simulation. Lors de la dépose d'une pièce sur une de ces tables, la table va prendre la couleur du produit qu'elle doit traiter. Ensuite, après une attente représentative du temps de traitement du produit, la valeur associée à la couleur va être incrémentée et la couleur de la table va changer pour signifier que le traitement est terminé et que le produit traité peut être récupéré pour effectuer la suite de la gamme.

5.8.2 Les fonctions du nœud ROS

Pour débuter le traitement d'un produit, celui-ci doit être posé sur la table. Pour cela, il est nécessaire de connaître la position du robot qui doit amener le produit au poste de travail, ainsi que l'état du bras et de la pince. En effet, il est nécessaire d'attendre que le

robot soit dans la bonne position pour considérer que le produit est posé sur la table et donc qu'il peut être traité. Aussi, le nœud *Tâche* comprend trois fonctions callback qui permettent d'obtenir ces différents retours du robot lié à la tâche. De plus, on y trouve également la gestion du traitement dans une fonction callback. Celle-ci est appelée dès la réception d'un message issu de la zone de chargement/déchargement associée à la tâche. A la réception d'un tel message, on connait donc le produit à traiter. Il nous faut ensuite attendre que le robot soit en position au-dessus de la table, bras descendu et pince ouverte, pour que le produit soit considéré comme posé et prêt à être traité. Pour simuler la pose du produit sur le poste de traitement, on colorise la table à l'aide de la couleur du produit à traiter d'un service V-Rep. Il nous faut ensuite attendre un certain temps représentatif de la durée de traitement du produit et défini dans le fichier de configuration. Une fois cette durée écoulée, le produit est considéré comme étant traité et est incrémenté, donc la couleur de la table fonce.

Enfin, on trouve dans ce nœud une dernière fonction, qui permet de simuler la prise du produit traité. Aussi, une fois que le traitement est terminé et en fonction de la position du robot, c'est-à-dire s'il se trouve au-dessus de la table avec le bras en position haute et la pince fermée, on va considérer que le produit traité a été retiré du poste de travail. Alors, la table reviendra à une couleur de base, ne correspondant à aucun produit.

5.9 Commande locale

Comme dit précédemment, lors de ce projet long, nous nous sommes concentrés sur la partie gauche de la simulation. Pour gérer l'arrivée des navettes sur le circuit, nous les faisons apparaître avant l'aiguillage A10 grâce au noeud *Navette* pour avoir toutes les informations nécessaires et nous les faisons disparaître après l'aiguillage A3. Évidemment, sur la maquette réelle, les navettes ne pourront pas apparaître et disparaître comme dans la simulation, il faudra trouver un moyen pour gérer l'arrivée et la sortie des navettes des parties gauches et droites de la maquette.

Pour voir l'avancement de l'état de fabrication d'un produit, nous avons coloré la plateforme de la navette : plus le produit avance dans sa fabrication, plus la couleur de la plateforme est foncée. Nous avons attribué une couleur par produit, par exemple le rouge pour le produit A, le bleu pour le B ou encore le vert pour le C. Sur la maquette réelle, on ne pourra évidemment pas voir l'avancement de la fabrication d'un produit de la même manière.

Au cours du projet, nous avons mis en place plusieurs manières pour charger les navettes dans la simulation. Tout d'abord, nous avons choisi un modèle de navette (A, B, C, D, E ou F). Ensuite, nous avons mis en place de l'aléatoire sur le choix du modèle. Nous avons, par la suite, choisi d'utiliser le modèle de navette F pour la partie gauche de la simulation. Ainsi, le choix ne se porte plus sur le modèle de la navette à lancer mais sur le produit qui va être lancé et donc sur la couleur. Nous avons créé un bouton (le bouton Shuttle [8]) dans l'interface utilisateur pour pouvoir tester le lancement des produits un à un puis nous avons mis en place de l'aléatoire sur le lancement de produit (de A à F) comme le montre la figure suivante 8. Ainsi, quand on clique sur le bouton, une navette

est lancée avec un produit aléatoire.

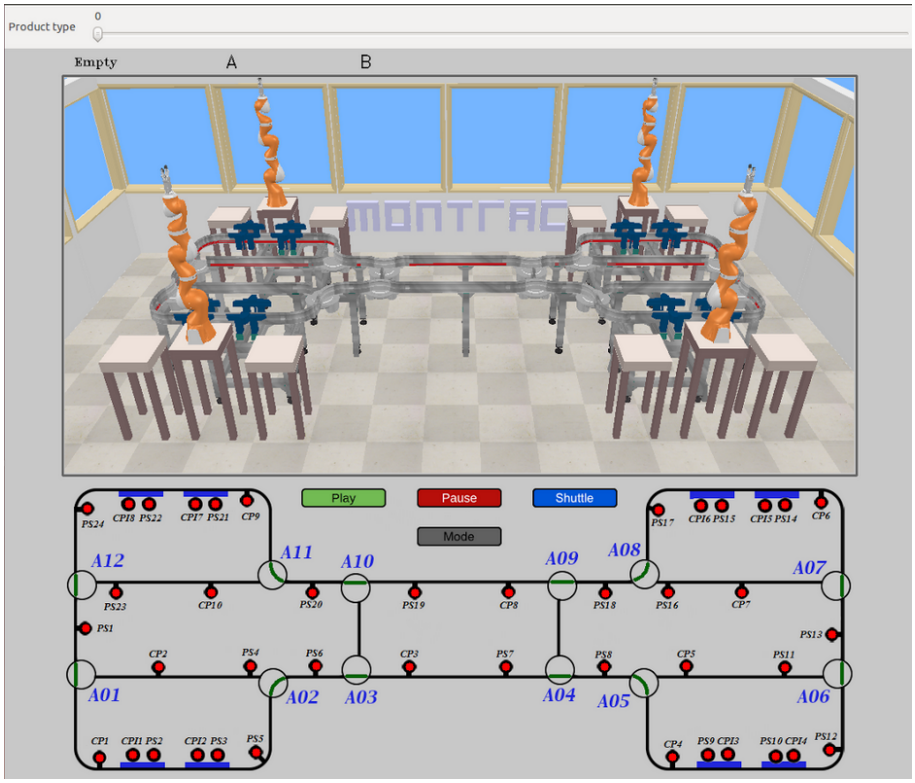

FIGURE 8 – Schéma de la cellule flexible complète

Par ailleurs, nous avons mis en place dans l'interface utilisateur une trackbar pour pouvoir choisir le produit que l'on veut lancer. Ensuite, le noeud ordonnancement a été créé pour pouvoir choisir, entre autre, les séquences de lancement des produits. Finalement, la dernière étape pour gérer le lancement des navettes a été la mise en place d'un bouton dans l'interface utilisateur pour pouvoir choisir le mode de lancement : manuel (avec la trackbar), auto (avec l'ordonnancement) et random (aléatoire). L'utilisateur peut changer de mode que quand la simulation est sur pause.

6 Résultats

6.1 Intégration

Une fois que nous avons développé tous les noeuds ROS nécessaires pour la simulation de la cellule, nous avons réalisé une intégration à différents niveaux pour arriver à la version finale du projet. Les différentes intégrations sont décrites dans ce chapitre, ainsi que les tests réalisés pour valider le fonctionnement de la simulation et ses différents outils.

La première étape de l'intégration a été de faire fonctionner l'ensemble des noeuds *Aiguillage* avec le noeud *Navette*, afin de s'assurer que les navettes se rendent bien aux différentes destinations prédéfinies. Pour cela le test était assez simple, nous lancions une navette avec une destination choisie au préalable, puis nous vérifions qu'elle s'y rendait bien sur la simulation.

La deuxième étape concernait l'intégration des noeuds *Poste, Robot ,Tâche* et *Ordonnancement*. Pour l'ordonnancement on s'est assuré que les navettes lancées comportaient bien les bonnes propriétés. Cette vérification fut rendue facile par l'affichage d'un certain nombre d'informations de débogage dans la console et par le système de coloration des navettes mis en place. Pour l'ensemble des noeuds *Tâche, Robot* et *Poste*, on s'est assuré que les navettes s'arrêtaient bien au niveau des postes correspondants à leur destination en fonction du produit qu'elles transportaient. Puis aussi, que ces mêmes produits sont traités, conformément à nos attentes.

La dernière étape de l'intégration est celle qui concerne la couche haute. Il s'agit du noeud *Commande* expliqué ci-dessous, qui joue le réseau de Petri (une partie est définie en Annexe 8 pour les postes 1 et 2). Il permet une gestion simple des destinations des navettes et du bon suivi de la gamme de fabrication.

6.2 Noeud Commande

Le noeud *Commande* est à la fois utilisé comme une interface entre les noeuds *Aiguillage, Poste, Robot* et *Tâche* et le noeud *Commande_locale* lui même servant d'interface entre la commande et la simulation. Mais le noeud *Commande* sert également de superviseur. C'est dans ce noeud que nous avons codé le réseau de Petri répondant au cahier des charges.

6.2.1 Interfaces entre les noeuds aiguillages et postes et la simulation

Lorsqu'un noeud *Aiguillage* ou *poste* souhaite modifier l'état d'un des actionneurs comme les aiguillages, les stops ou les ergots, il publie dans le topic correspondant à l'action le numéro de l'actionneur. Le noeud *Commande* met alors à jour son tableau associé puis le publie dans un topic à destination de *Commande_locale* qui est retransmis

à V-Rep. Par exemple, supposons que le noeud chargé de l'aiguillage 11 souhaite le faire tourner à gauche, il publiera tout d'abord 11 dans un premier topic pour le déverrouillage à destination de *Commande* qui mémorisera que l'aiguillage a été déverrouillé. Il écrira ensuite dans un second topic pour signaler que l'aiguillage doit tourner à gauche. Le noeud *Commande* réagit alors en transmettant un message à *Commande_ locale* qui sera retransmis à *Vrep* pour que l'aiguillage tourne effectivement. L'aiguillage sera ensuite verrouillé de la même façon qu'il a été déverrouillé.
En revanche, les noeuds *Aiguillage* et *Poste* n'utilisent pas le noeud *Commande* pour avoir l'état des capteurs dans V-Rep, ils souscrivent directement aux topics de *vrep*.

On retrouve un schéma précisant les liaisons entre tous les noeuds de la simulation à la fin de notre intégration en Annexe 8.

6.2.2 Superviseur

Le noeud *Commande* va également servir de superviseur du système. C'est dans ce noeud que le réseau de Petri répondant au cahier des charges sera implémenté. Il prendra donc la décision de déplacer les robots, définir les destinations des navettes ...

6.2.3 Cahier des charges

Le cahier des charges général qui doit être respecté par le système est le suivant :

Des navettes arrivent entre les aiguillages 9 et 10 en transportant un certain produit. Le produit transporté doit suivre une certaine gamme : il doit passer par différents postes de travail dans un ordre prédéfini. Par exemple, le produit A devra passer par le poste 1, puis les postes 2, 3 et enfin le poste 4. Une fois la gamme terminée, la navette transportant le produit fini devra aller dans la section entre les aiguillages 3 et 4 où elle sera supprimée de la simulation.

Lorsque le produit transporté par la navette doit aller au poste de travail i (tâche i), la navette doit aller au poste i. Une fois arrivé au poste, le robot doit prendre le produit, et le déposer au niveau du poste de travail. Une fois le produit pris par le robot, la navette est considérée comme vide et repart avec la destination 0 (sur les rails du milieu). Une fois que le poste de travail i a terminé son traitement sur le produit, une navette vide doit être ramenée au poste de travail pour récupérer le produit traité et doit l'emmener au poste de travail suivant.

Pour réaliser le réseau de Petri, nous avons considéré quelques hypothèses supplémentaires :

- Les navettes qui arrivent ne sont jamais vides.
- Une navette vide ne doit doit jamais sortir.
- Bien que physiquement possible, on suppose qu'on ne peut pas prendre un produit au niveau du poste de déchargement 1 pour le déposer au niveau du poste de travail 2 et inversement. Idem pour les postes 3 et 4.

6.2.4 Fonctions utilisées par le réseau de Petri

Pour commander les robots, gérer le redémarrage des navettes au niveau des postes et la destination des navettes, on doit implémenter dans le programme main_commande

le réseau de Petri.Cette fonction contient des instances des classes Commande (vu précédemment) et Robots. La classe Commande va permettre au niveau de la supervision de créer un certain nombre de fonctions qui utiliseront des services et contiendra des attributs qui seront mis à jour par des fonctions callbacks et qui permettront de connaître l'état du système, notamment la présence de navettes vides ou occupées au niveau des postes (et leurs handles).

La classe "Robots" a le même objectif mais uniquement en ce qui concerne les robots et les tâches. Elle possède par exemple une fonction **ControlerRobot()** qui prend en paramètre le numéro du robot concerné, la position souhaitée (dans le plan), en position haute ou basse et pince ouverte ou fermée. Elle contient également des retours sur les mouvements des robots : la fonction **RobotEnPosition()** par exemple qui renvoie 1 si le robot a atteint la position demandée, la fonction **BrasEnPosition()** et **PinceEnPosition()**.

6.2.5 Réseau de Petri

Le réseau de Petri que nous avons réalisé se compose de quatre parties principales associées à chaque poste du système. Il s'agit d'un réseau de Petri coloré dont le marquage de certaines places correspond à un handle de navette. Il contient un certain nombre de places communes aux différentes parties : la place 0 correspond aux navettes dont la destination est d0 (tourne au milieu du circuit). La place 50 (resp. 250) correspond à la disponibilité du robot 1 (resp. robot 2) (partage de ressource). Ces places contiennent un jeton au marquage initial. Toutes les autres places correspondent à une phase du traitement du produit.

Pour implémenter le réseau de Petri nous avons une série de conditions, chacune correspondant à une transition du réseau de Petri. La condition est validée si la transition est franchissable. Dans ce cas, on met à jour le marquage et on effectue les actions associées à la transition (déplacer le robot ...).

Nous avons cependant eu un problème au niveau d'une transition : on exécute une fonction qui doit modifier, en passant par un topic, un attribut du noeud navette. Or on récupère juste après ce paramètre en passant par un service. Il est cependant possible que la variable n'a pas eu le temps d'être mis à jour, ce qui conduit à un bloquage. Pour corriger le problème, nous avons ajouté une attente après l'exécution de la fonction pour être sûr que la variable ait été mise à jour. Cette solution n'est cependant pas en accord avec la modélisation sous forme de réseau de Petri et devra donc être corrigée en ajoutant un retour par exemple et une place.

6.3 Lancement de la simulation

Finalement pour le lancement de tous les noeuds ROS réalisés, nous avons utilisé une fonctionnalité ROS appelée roslaunch. Cette fonctionnalité nous permet de, depuis un fichier écrit en langage interprété XML, pouvoir lancer tous les noeuds, dans le bon ordre et avec tous les arguments nécessaires.

7 Bilan et améliorations possibles

7.1 Bilan

Lors de nos délibérations pour trouver une solution répondant à nos objectifs, il est apparu aux vues des outils à notre disposition que la réalisation de fonctions de haut niveau allait demander un découpage important des fonctionnalités et la création de nombreux noeuds. Cela est dû au fait que sans scheduler, les notions de signaux et de tâches dormantes ne sont pas disponible. Pour recréer un comportement d'OS pseudo temps-réel nous avons utilisés les noeuds ROS. En effet, d'un point de vue fonctionnel, chaque noeud peut être vu comme un programme indépendant ainsi on peut recréer un comportement multithread permettant la parallélisation des prises de décisions. Sans cela, impossible de contrôler l'ensemble du système via un programme C++ mono-threadé, tout du moins sans réaliser un trait grand réseau de Petri avec une programmation proche du Standard Text List, comportant une matrice d'état importante et un nombre de transitions à tester à chaque boucle tout aussi conséquent. La réalisation de programme C++ multi-threads n'est pas impossible mais l'échange de messages synchrones ou asynchrones à l'aide du middleware ROS rend la tâche plus simple à réaliser dans notre cas.

7.2 Améliorations possibles

7.2.1 Niveau Poste

Au point où le projet en est, un produit arrivant au poste P1 sera traité par la tâche T1 et un produit arrivant au poste P2 sera traité par la tâche T2. Il n'y a pas de communication entre P1 et T2, P2 et T1 et T1 et T2. On pourrait imaginer que par la suite, le robot puisse amener un produit arrivé en P2 à la tâche T1. De la même manière, on pourrait supposer qu'un produit devant réaliser la tâche 1 puis la tâche 2 ne soit pas obligé de repasser par une navette, le robot pourrait amener directement le produit de la tâche 1 à la tâche 2.

7.2.2 Niveau VREP

Par ailleurs, dans V-REP, on pourrait rajouter des objets pour représenter les produits. On pourrait garder le système des couleurs mais au lieu de colorer les plateformes des navettes et les tables, on colorerait les objets.

7.2.3 Niveau Simulation

Dans la simulation, nous avions prévu de séparer la partie gauche en 5 zones distinctes afin de savoir plus facilement où se situe chaque navette. Cela aurait été utile pour savoir quelle est la navette vide la plus proche du poste qui en demande une, pour être plus efficace. Cependant, nous n'avons pas eu le temps de mettre en place les zones même si nous avons attribué aux navettes un argument 'zone'.

De plus, il reste toute la partie droite de la simulation à coder, c'est-à-dire, les adapter en fonctions des capteurs et actionneurs adéquats. Pour cela, les différents noeuds créés pourront être réutilisés avec des adaptations à faire. Il reste également à adapter la partie gauche de la simulation sur la maquette réelle.

7.2.4 Niveau Aiguillage

Les noeuds *Aiguillage* ayant deux entrées peuvent être améliorés afin d'optimiser leur fonctionnement. Dans la solution actuelle, si deux navettes sont présentes aux deux entrées de l'aiguillage, ce sera la navette venant de gauche qui passera la première. Cette "priorité" est simplement liée au fait que l'on vérifie si une navette est présente à gauche en premier. Arbitrairement, si une navette est présente, on la fait passer. Deux améliorations différentes sont envisageables pour définir un autre type de priorité. La première consiste à faire passer en priorité la navette présente du côté où est tourné l'aiguillage. Ainsi, on limite le nombre de déplacement de l'aiguillage. Cette solution est assez simple à mettre en place puisqu'il suffit de modifier, dans la fonction principale de la classe, la condition pour traiter la navette. La seconde amélioration consiste à définir des priorités sur les produits transportés par la navette. Si deux navettes sont présentes à l'aiguillage, on fait passer celle qui transporte le produit avec la plus grande priorité. Cette amélioration est plus compliquée à implanter, parce qu'il faut, en plus d'adapter les noeuds *Aiguillage*, mettre en place les priorités sur les navettes dans les noeuds concernés comme le noeud *Navette*.

De plus, nous avons commencé à coder une manière plus facile d'intervenir manuellement en cas de bug. C'est-à-dire, si jamais un aiguillage ne tourne pas (car parfois il se peut que la simulation rencontre des problèmes de limitation), nous avons pensé mettre en place une interface qui s'ouvrirait dès que l'on appuierait sur une touche précise du clavier pendant la simulation, et qui nous permettrait de passer du mode "automatique" au mode "manuel" (pour un seul aiguillage, les autres restant en automatique) pour faire tourner l'aiguillage correspondant, puis de repasser en mode automatique ensuite.

8 Conclusion

Pour conclure, nous avons eu l'opportunité de pouvoir poursuivre le développement d'un ancien projet au sein de l'AIP-PRIMECA à Toulouse.

L'AIP-PRIMECA met à disposition une cellule flexible de production robotisée pour des étudiants de cycle supérieur et avait besoin d'un simulateur qui la modéliserait afin de pouvoir être utilisée au cours de TP.

Notre projet consistait à rendre cette simulation fonctionnelle ainsi qu'un espace de travail dans lequel les futurs étudiants travailleront.

Aujourd'hui la simulation propose un design plus que similaire à la cellule réelle et offre des premières possibilités de travaux pratiques que ce soit pour de l'ordonnancement ou une étude plus approfondie du middleware ROS.

Les livrables rendus en fin de projet vont également permettre aux futurs développeurs de poursuivre notre travail en y apportant de nombreuses améliorations listées dans la partie précédente.

Annexes

Annexe 1

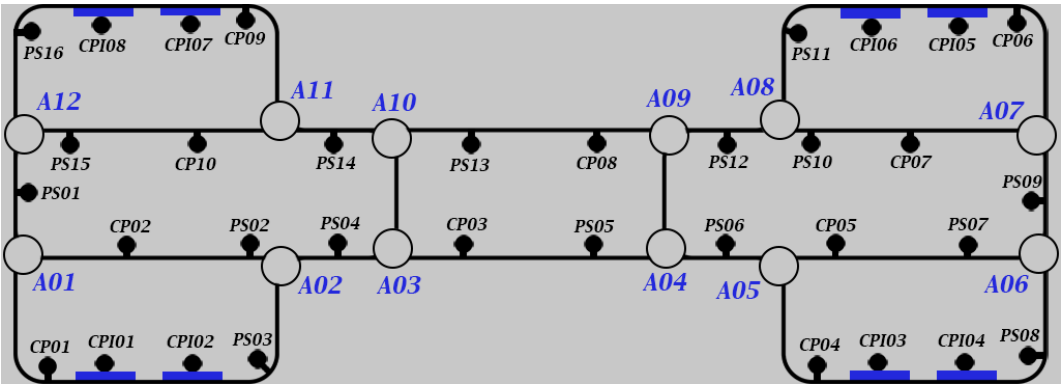

FIGURE 9 – Schéma de la cellule des anciens étudiants

Annexe 2

Code	Color (R, G,B)	Code	Color (R, G,B)	Code	Color (R, G,B)
0	(41, 41, 54)	30	(153,255,153)	50	(255,153,204)
11	(255,102,102)	32	(51,255,51)	52	(255,51,153)
12	(255,51,51)	33	(0,204,0)	53	(255,0,127)
13	(255,0,0)	34	(0,150,0)	54	(204,0,102)
14	(230,0,0)	40	(255,217,172)	60	(204,153,255)
20	(102,178,255)	41	(255,178,102)	61	(178,102,255)
21	(51,153,255)	42	(255,153,51)	62	(153,51,255)
22	(0,128,255)	43	(255,128,0)	63	(127,0,255)
23	(0,102,255)	44	(255,100,0)	64	(76,0,153)
24	(0,0,255)				

TABLEAU 3 – Tableau des codes couleurs utilisés

Annexe 3

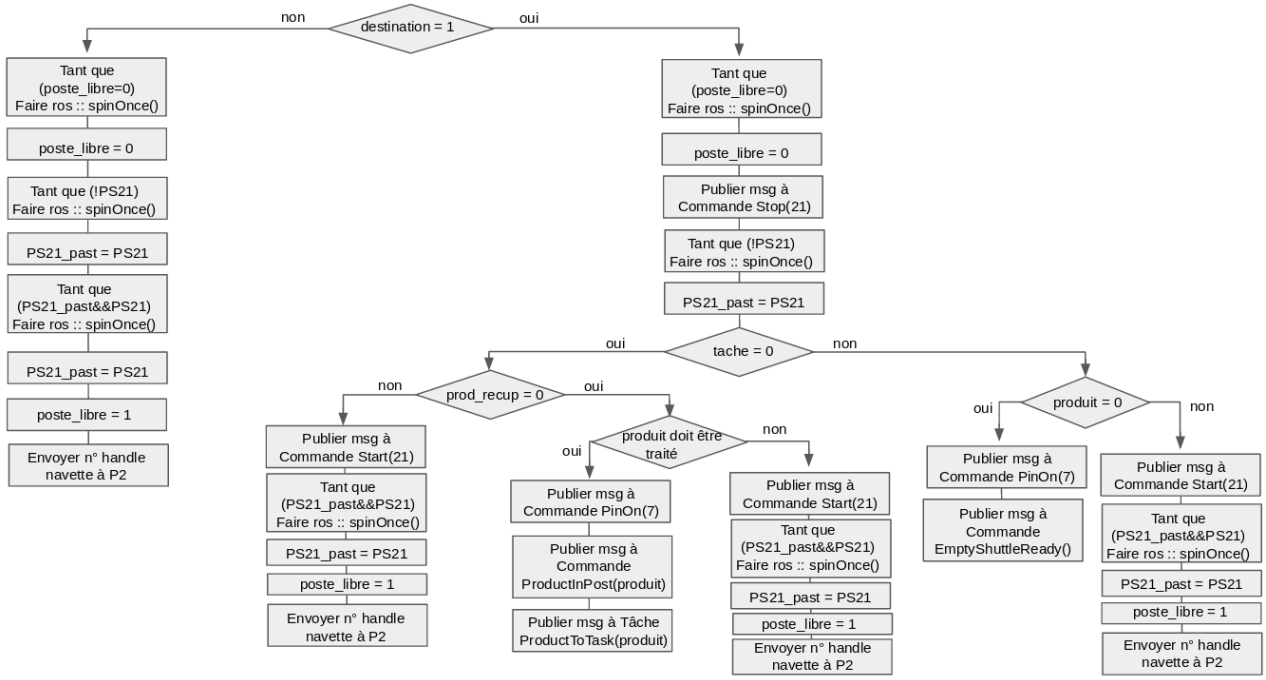

FIGURE 10 – Logigramme de fonctionnement du poste P1

Annexe 4

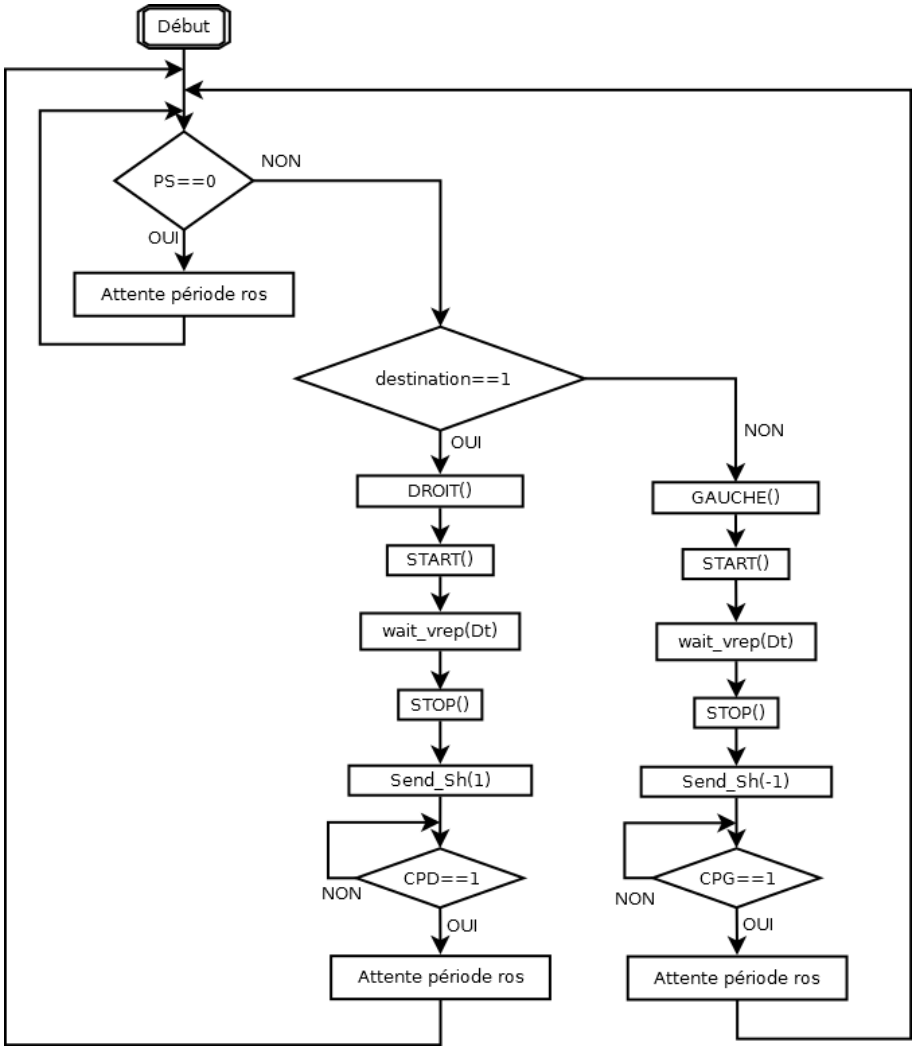

FIGURE 11 – Logigramme de fonctionnement de l'aiguillage 11

Annexe 5

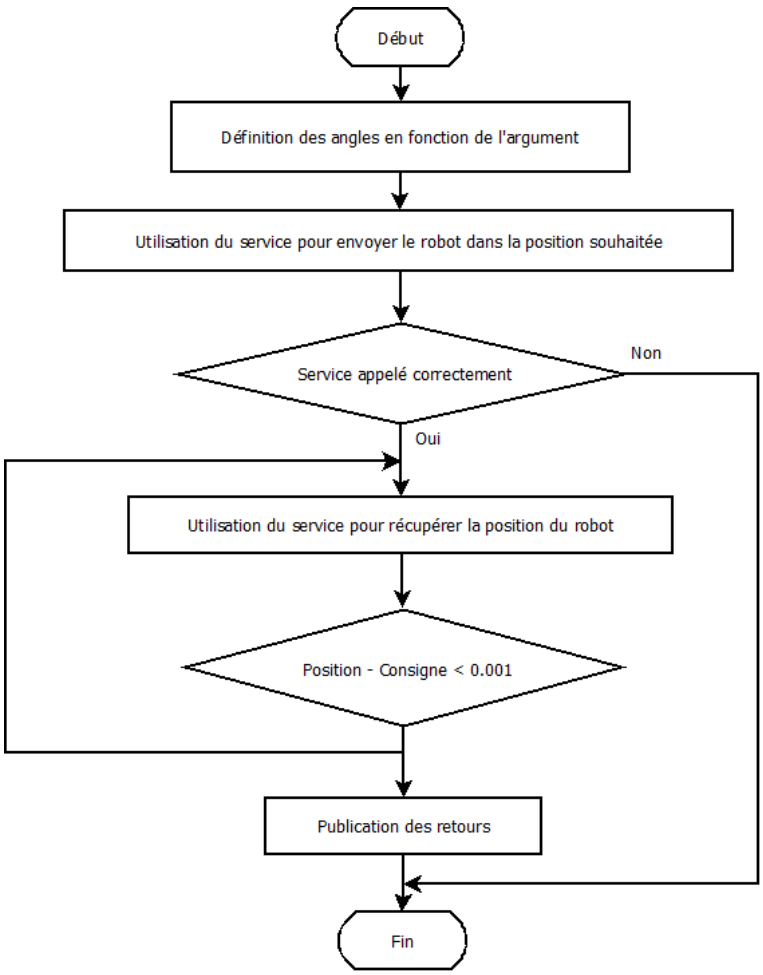

FIGURE 12 – Logigramme de fonctionnement de mouvement du robot

Annexe 6

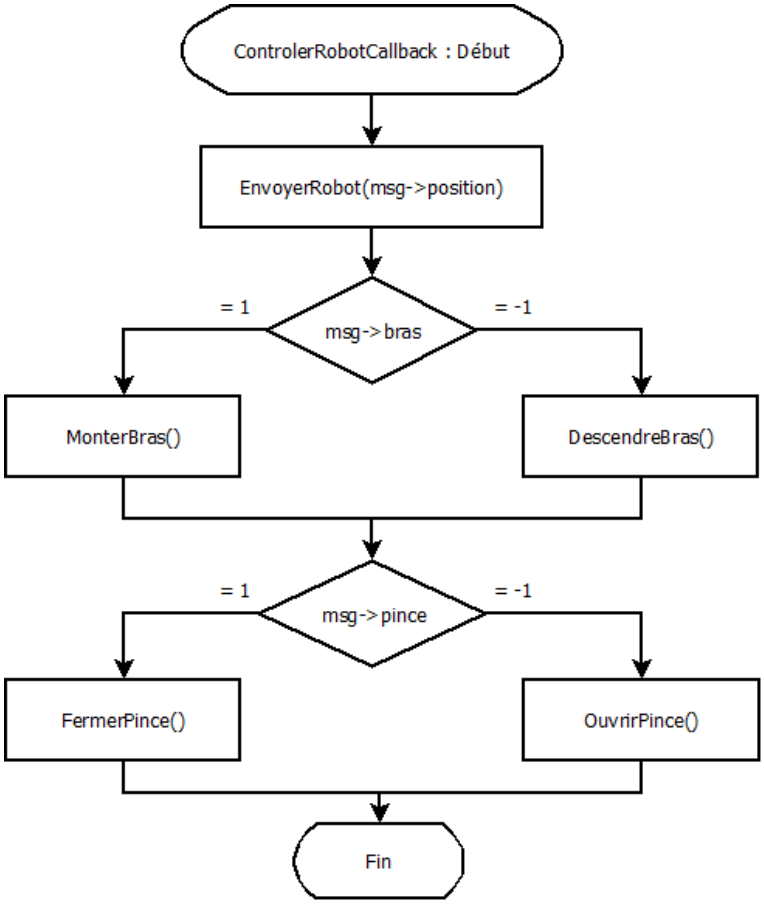

FIGURE 13 – Logigramme de fonctionnement de contrôle du robot

Annexe 7

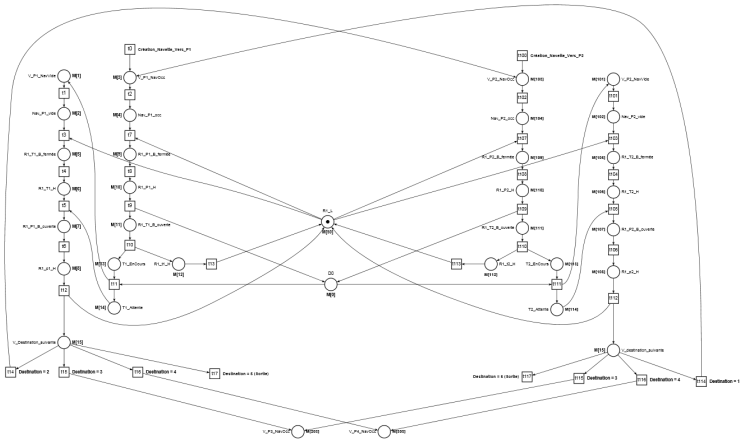

FIGURE 14 – Réseau de Petri des postes 1 et 2

Annexe 8

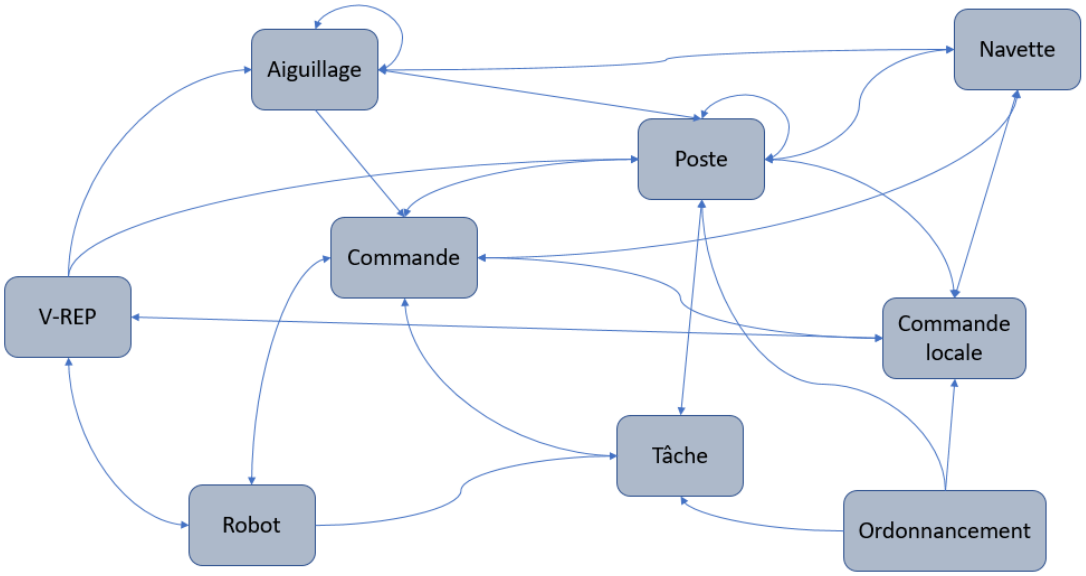

FIGURE 15 – Liaisons reliant tous les noeuds nécessaires à la simulation

Bibliographie

[1] AIP-PRIMECA [En ligne]
http://www.aip-primeca.net/

• Usine du futur

[2] Vinci Energies [En ligne] *L'usine du future sera autonome*
http://www.vinci-energies.com/cest-deja-demain/
pour-une-industrie-intelligente/lusine-du-futur-sera-autonome/

[3] M. Grunow, H.-O. Günther, M. Lehmann *Strategies for dispatching AGVs at automated seaport container terminals.* Pages 587-610, Oct 2006

• Tutoriels

[4] Tutoriel ROS *Liste des topics et services*
http://www.coppeliarobotics.com/helpFiles/en/rosServices.htm#
simRosAddStatusbarMessage

[5] Tutoriel ROS *Apprentissage*
http://www.coppeliarobotics.com/helpFiles/en/childScripts.htm

[6] Tutoriel V-REP *Apprentissage*
http://www.coppeliarobotics.com/helpFiles/

[7] Tutoriel V-REP *Manuel Lua*
http://www.lua.org/manual/5.2/

• Librairie de fonctions

[8] Open CV
http://docs.opencv.org/3.1.0/d4/dd5/highgui_8hpp.html

[9] C++ référence
http://www.cplusplus.com/reference/

• Anciens rapports

[10] BIVEN Cédric, CARDONE Grégoire, GAO Shengheng *Commande d'une ligne transitique MONTRAC* Rapport de TER, Master 1 Électronique Électrotechnique Automatique, Ingénierie des Sytèmes Temps-Réel. Toulouse : Université Paul Sabatier, 2015.

[11] ANTONIUTTI Emilie, BERTIN Thibault, DEMMER Simon, LE BIHAN Clément *Commande et Simulation d'un réseau de transport d'un système de production* Rapport de Projet Long, GEA CDISC. Toulouse : ENSEEIHT, 2016.
https://github.com/ClementLeBihan/CelluleFlexible/blob/master/
Livrables/Rapport_Projet_Long

[12] ANTONIUTTI Emilie, BERTIN Thibault, DEMMER Simon, LE BIHAN Clément *Commande et Simulation d'un réseau de transport d'un système de production* Code source (GIT). Toulouse : ENSEEIHT, 2016.
`https://github.com/ClementLeBihan/CelluleFlexible`

[13] DATO Bruno, ELGOURAIN Ab dellah, SHULGA Evgeny *Commande d'une ligne transitique MONTRAC* Rapport de TER, Master 1 Électronique Électrotechnique Automatique, Ingénierie des Sytèmes Temps-Réel. Toulouse : Université Paul Sabatier, 2016.

[14] DATO Bruno *Intégration et commande sous ROS du robot Baxter au sein d'une cellule flexible d'assemblage* Rapport de stage, Master 1 Électronique Électrotechnique Automatique, Ingénierie des Sytèmes Temps-Réel. Toulouse : Université Paul Sabatier, 2016.