Master 1 Électronique Électrotechnique Automatique, Ingénierie des Systèmes Temps-Réel

UNIVERSITÉ TOULOUSE III PAUL SABATIER — Université de Toulouse

STAGE
Intégration et commande sous ROS du robot Baxter au sein d'une cellule flexible d'assemblage

Auteur

Bruno DATO

Encadrants

C. Briand, M. Taïx

30 juin 2016

Remerciements

Je tiens à remercier mes encadrant de stage C. Briand et M. Taïx pour m'avoir permis de réaliser ce stage. Je remercie aussi toutes les personnes de l'AIP pour leur accueil au sein de la halle technologique durant toute la durée de mon stage.

Sommaire

Introduction

Dans le cadre de mon année de master ISTR, j'ai effectué un stage au sein de l'AIP/PREMICA afin de prolonger mon projet de TER sur la ligne transitique MONTRAC et de mettre un premier pas dans la robotique en commandant le robot humanoïde collaboratif Baxter. Le but de mon stage était de commander la ligne transitique en interaction avec le robot Baxter de manière synchrone.

Premièrement, je présenterai tous les outils et systèmes avec lesquels j'ai travaillé pendant mon stage ainsi que la solution que j'ai envisagée pour commander la ligne transitique en interaction avec le robot Baxter.

Ensuite, je détaillerai le code mis en place à l'aide du middleware ROS en détaillant toutes les classes utilisées et comment a été établie la communication entre la ligne transitique et le robot Baxter.

Enfin, j'exposerai les commandes que j'ai faites et quels sont les contraintes et objectifs que j'ai fixés.

Chapitre 1

Présentation du stage

1.1 Projet global

Mon stage s'est déroulé au pôle AIP/PRIMECA, à Toulouse, sur le campus de l'université Paul Sabatier à la halle technologique. Ces locaux disposent de divers systèmes tels que des robots mobiles, des robots manipulateurs, une cellule flexible de production, un vidéo projecteur interactif, un réseau de caméras... Dans le cadre de mon stage, j'ai travaillé sur l'interaction entre un des robot manipulateurs : le robot Baxter, et la ligne transitique MONTRAC sur laquelle j'avais déjà effectué mon projet de TER.

Ce stage s'inscrit dans un projet de plus grande envergure visant à faire interagir de nombreux systèmes se trouvant à la halle technologique comme vous pouvez le voir sur la figure 1.1. En effet, depuis 2 ans, il y a eu différents projets et stages concernant le vidéo projecteur interactif, la ligne transitique et le réseau de caméras.

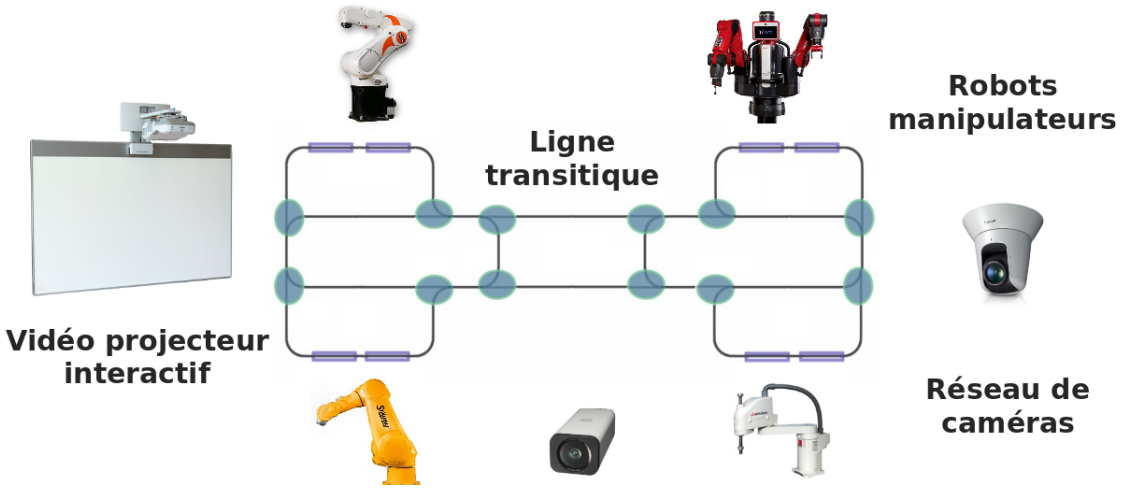

FIGURE 1.1 – Vue globale des systèmes à faire interagir

1.2 Le robot Baxter

Le robot Baxter est un robot humanoïde collaboratif plus communément appelé cobot. Il a été conçu par Rodney Brooks et sa société Rethink Robotics. Un des principaux atouts de ce robot est que sa version dédiée à la recherche peut être commandé à l'aide du middleware ROS qui est très utilisé aujourd'hui en robotique.

FIGURE 1.2 – Le robot collaboratif Baxter

Baxter dispose de deux bras manipulateurs comme vous pouvez le voir sur la figure 1.2, chacun de ces deux bras peut être équipé de pinces ou de ventouses reliés à un système pneumatique. Il dispose de différents capteurs : un accéléromètre, un capteur infrarouge et une caméra sur chaque bras. Il possède aussi une troisième caméra et un sonar au niveau de sa tête, le sonar lui permet de détecter dans l'espace des objets ou des humains lorsqu'ils sont à une certaine distance. Il possède aussi différents boutons sur ces bras et ses épaules qui permettent d'interagir directement avec lui à l'aide de l'écran situé sur sa tête. Enfin il est possible de déplacer ses bras en les saisissant par leur manchette (chaque manchette est munie d'un capteur de pression).

Durant mon stage, j'ai surtout travaillé sur le mouvement des bras manipulateurs équipés de pinces afin de saisir et poser des objets bien qu'ils soient hypothétiques. Chaque bras possède sept degrés de liberté qui sont les angles de rotation *S1*, *E0*, *E1*, *W0*, *W1* et *W2* que vous pouvez voir sur la figure 1.3 ci dessous.

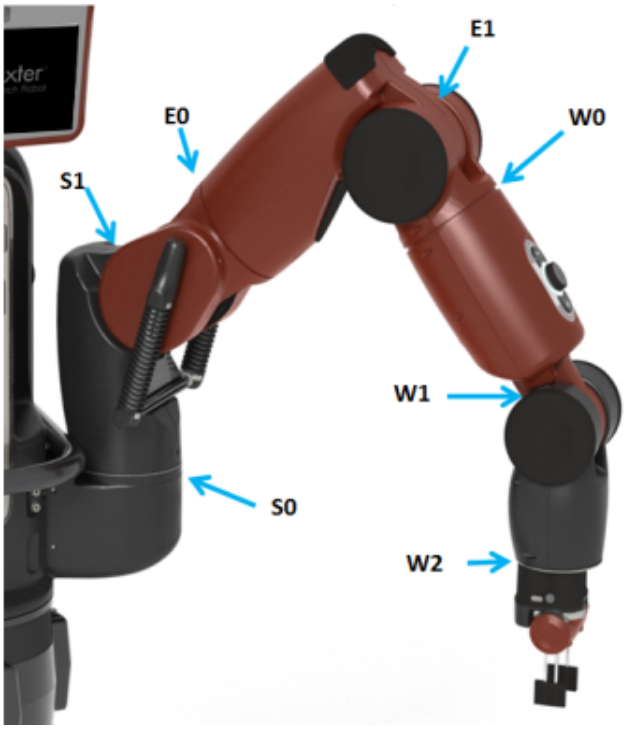

FIGURE 1.3 – Les différents angles des bras du robot Baxter

Pour le mouvement des bras, Baxter dispose de quatre modes différents :

- **Joint Position Control** : Il faut spécifier les valeurs de position que l'on souhaite atteindre pour chacun des sept angles. Le mouvement est ensuite contrôlé automatiquement par Baxter afin d'éviter des collisions entre les deux bras ou avec Baxter lui même mais aussi des positions impossibles. La vitesse est aussi contrôlée. La position est atteinte plus ou moins rapidement suivant la fréquence à laquelle on envoie ces informations au robot.
- **Raw Joint Position Control** : C'est le même fonctionnement que le mode Joint Position Control cependant, le mouvement est moins contrôlé cette fois, il n'y a plus d'évitement des collisions et les bras se déplacent à la vitesse maximale que Baxter peut fournir. Il faut donc faire très attention avec ce mode.

- **Joint Velocity Control** : Il faut spécifier les valeurs de vitesse que l'on souhaite atteindre pour chacun des sept angles. Il y a à nouveau un contrôle pour éviter les collisions cependant la vitesse maximale peut être atteinte.
- **Joint Velocity Control** : Il faut spécifier les valeurs de moment (ou couple) que l'on souhaite atteindre pour chacun des sept angles. Pour ce mode, les contrôles du mouvement sont minimes, il faut donc l'utiliser avec beaucoup de précautions.

Pour les commandes que nous verrons par la suite, j'ai utilisé le premier mode : Joint Position Control, couplé avec un service ROS proposé par le robot. Le service permet, à partir d'une position cartésienne dans l'espace ainsi que d'une orientation souhaitée pour l'extrémité d'un des bras, de fournir les valeurs de position des sept angles qui permettent d'atteindre cette disposition.

1.3 Ligne transitique MONTRAC®

1.3.1 Présentation

La ligne transitique MONTRAC est composée de rails alimentés en énergie électrique sur lesquels circulent des navettes. Les navettes ne peuvent se déplacer que dans un sens sur ces rails.

Pour commander la ligne, on dispose de cinq automates programmables industriels que nous appellerons API. Il y en a 2 de la marque Siemens® et 3 de type Schneider®. Ces automates gèrent les différents actionneurs et capteurs situés sur les rails. Les navettes ne sont pas programmables, une fois allumées elles avancent jusqu'à ce qu'on les arrête. Elles possèdent un capteur de proximité situé à l'avant pour éviter les collisions entre les navettes et les stopper lorsqu'elles rencontrent un obstacle.

Cette ligne comporte 5 "secteurs" dont un central et 4 postes de travail. Chaque "secteur" est contrôlé par un des API qui agit sur la ligne via un système d'air comprimé. Les zones 1 à 3 correspondent aux automates Schneider® et les deux autres zones correspondent aux automates Siemens®. La ligne possède 12 aiguillages comme on peut le voir sur la figure 1.4.

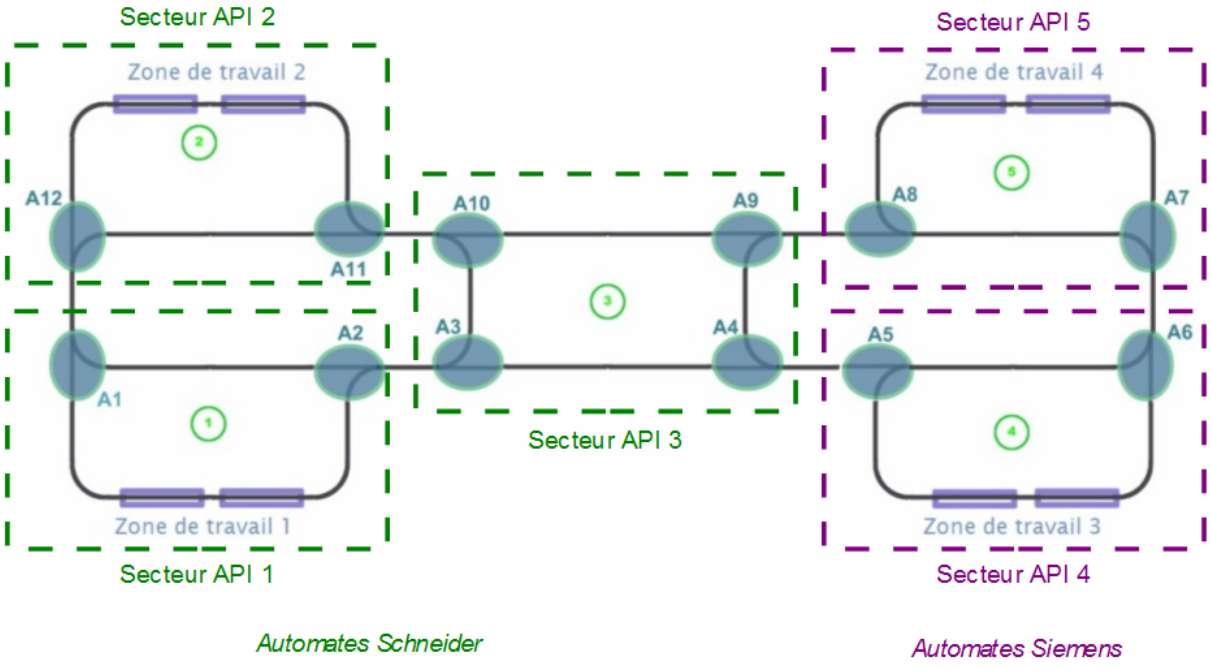

FIGURE 1.4 – Schéma simplifié de la ligne transitique

1.3.2 Capteurs et actionneurs

Les actionneurs permettent de commander les points d'arrêts des navettes, les aiguillages et des ergots pouvant bloquer les navettes à certains endroits lorsqu'elles sont arrêtées. Les capteurs nous permettent de connaître les positions des navettes, des aiguillages et des ergots. Les listes des actionneurs et capteurs sont données ci-dessous.

RxG	Positionne l'aiguillage x à gauche
RxD	Positionne l'aiguillage x à droite
Dx	Dévérrouille l'aiguillage X
Vx	vérrouille l'aiguillage X
STx	Quand STx vaut 0, les navettes s'arrêtent au niveau de l'actionneur
PIx	Blocage des navettes sur le poste de travail

TABLE 1.1 – 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 x est à droite
DxG	Vaut 1 quand l'aiguillage x est à gauche

TABLE 1.2 – Capteurs

Durant ce stage, j'ai commandé de la ligne via les API des zones 1 et 2 (automates Schneider®). Ces automates disposent de 16 entrées et sorties chacun mais toutes ne sont pas utilisées. Les tables 1.3 et 1.4 décrivent respectivement le câblage des API 1 et 2.

On peut voir sur la figure 1.5 une vision globale de la ligne transitique avec tous les capteurs et actionneurs, les zones contrôlées par les API ainsi que les orientations de chaque rail.

OUT	A1		V1		A2		V2		S1		S2		S3		S4		S5		UP1		UP2	
	R1D	R1G	V1	D1	R2D	R2G	V2	D2	ST1		ST2		ST3		ST5		ST4		PI1		PI2	
IN	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15						
	PS1	PS2	PS3	PS5	PS4	D1D	D1G	CP1	CPI1	CPI2	D2D	D2G	CP2	PS6								

TABLE 1.3 – Configuration des entrées/sorties de l'automate Schneider 1

OUT	A1		V1		A2		V2		S1		S2		S3		S4		S5		UP1		UP2	
	R11D	R11G	V11	D11	R12D	R12G	V12	D12	ST20		ST21		ST22		ST24		ST23		PI7		PI8	
IN	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15						
	PS20	PS21	PS22	PS24	PS23	D11D	D11G	CP9	CPI7	CPI8	D12G	D12G	CP10	PS1								

TABLE 1.4 – Configuration des entrées/sorties de l'automate Schneider 2

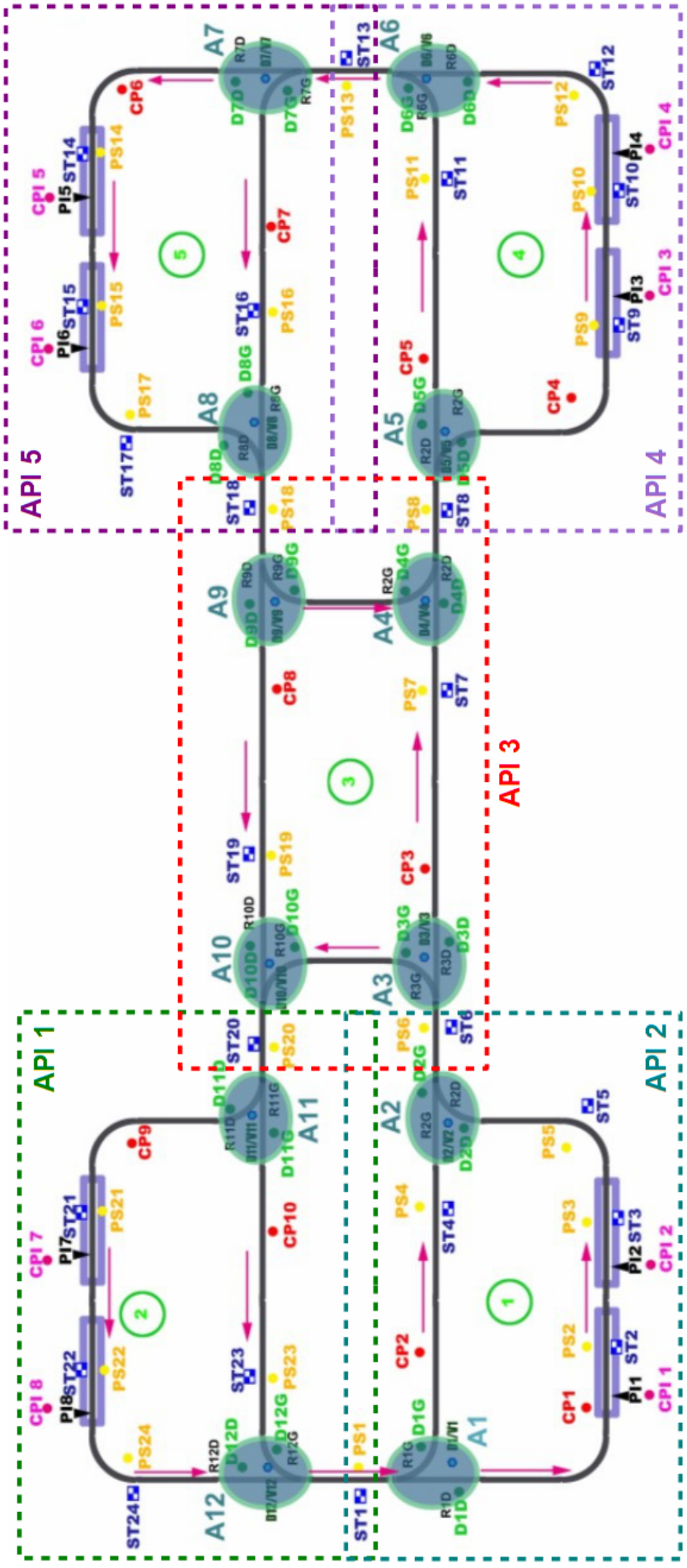

FIGURE 1.5 – Schéma détaillé de la ligne transitique

1.4 Simulation de la ligne transitique

Une simulation de la ligne transitique a été conçue par des étudiant de l'ENSEEIHT à l'aide du logiciel V-REP [1]. Cette simulation se comporte comme la ligne MONTRAC®, bien qu'elle ne possède pas tous les capteurs et actionneurs (les différences sont détaillés table 1.5). Elle permet de valider un grand nombre de commandes avant des les tester directement sur la ligne réelle.

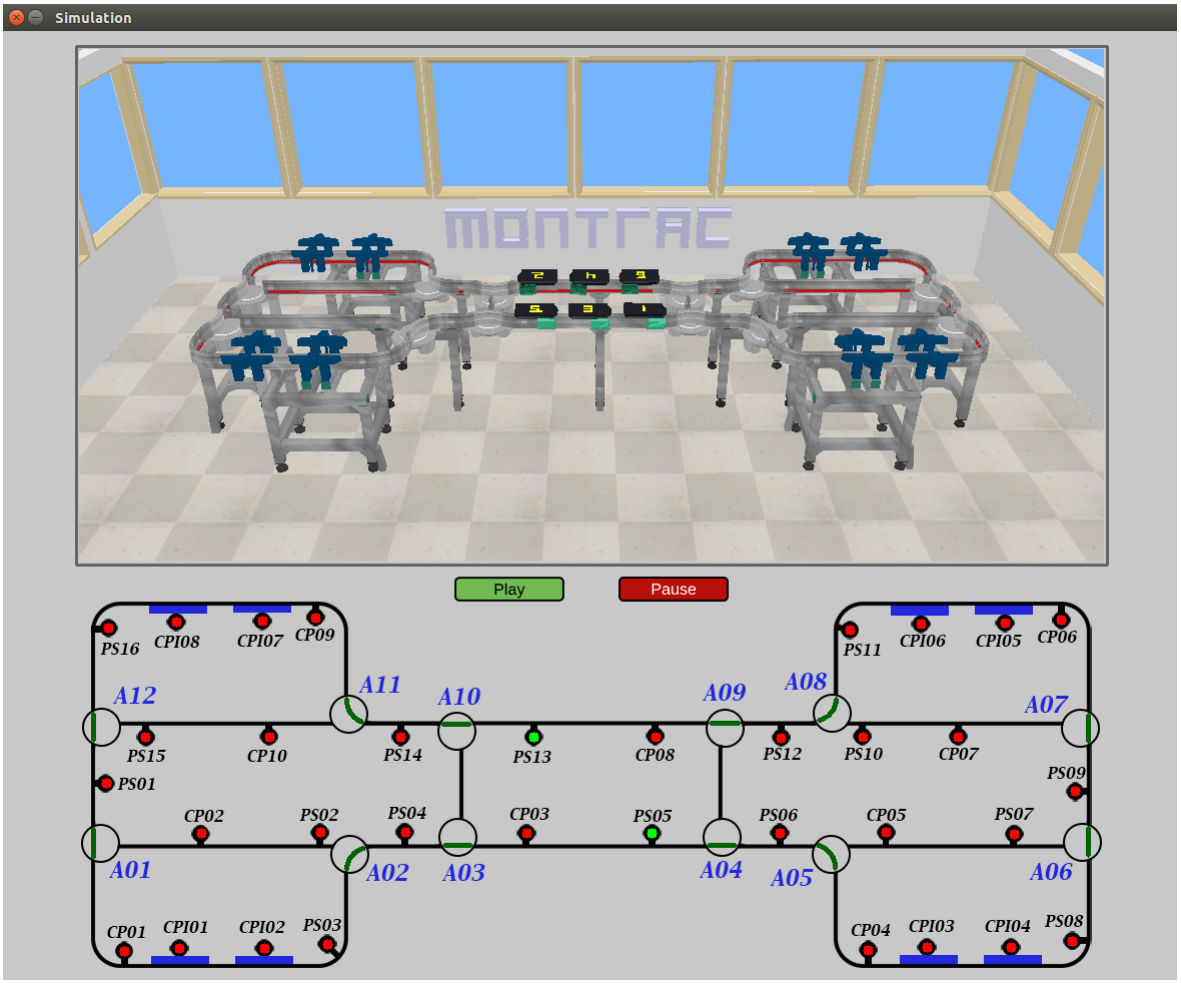

FIGURE 1.6 – Simulation de la ligne transitique

Pour la conception de la simulation, les étudiants on regroupé les capteurs en catégories : CPI, CP, PS, DG et DD qui correspondent à ceux cités précédemment (table 1.2). Il y a cependant quelques différences pour les capteur PS et CPI comme on peut le voir sur la table 1.5.

Ligne transitique	PS1	PS2	PS3	PS4	PS5	PS6	PS7	PS8	PS9	PS10	PS11	PS12	PS13
Simulation	PS1	CPI1	CPI2	PS2	PS3	PS4	PS5	PS6	CPI3	CPI4	PS7	PS8	PS9

Ligne transitique	PS14	PS15	PS16	PS17	PS18	PS19	PS20	PS21	PS22	PS23	PS24
Simulation	CPI5	CPI6	PS10	PS11	PS12	PS13	PS14	CPI7	CPI8	PS15	PS16

TABLE 1.5 – Capteurs de stop de la ligne transitique et de la simulation

1. V-REP est une plateforme de simulation pour tout type de robots **[7]**.

Sur la simulation, les capteur CPI ne sont pas les capteurs de position des ergots mais des capteurs de stop aussi. Pour les actionneurs, il y a 5 registres : RD, RG, LOCK, STOP et GO ; RD et RG fonctionnement de la même manière que ceux de la table 1.1 mais LOCK, STOP et GO ont un fonctionnement différent :

- LOCKx = $\overline{\text{Vx}}$.Dx
- STOPx = $\overline{\text{STx}}$
- GOx = STx

1.5 Problématique et solution mise en place

Comme je l'ai dit précédemment, le but de mon stage était de faire interagir le ligne transitique (ou sa simulation) avec le robot Baxter. Pour cela j'ai utilisé le middleware ROS (Robot Operating System), c'est un ensemble de librairies et d'outils qui permettent de mettre en place toutes sortes d'applications robotiques.

Je me suis servi de ce que mon groupe de TER et moi même avions effectué pour établir la communication entre les automates de la ligne transitique et ROS **[5]**. J'ai alors ajouté la communication de la ligne transitique (ou la simulation) avec le robot Baxter ainsi que sa commande à lui même à l'aide de ROS.

Les commandes de la ligne transitique ont été modélisées par des réseaux de Petri (RdP) afin d'utiliser plusieurs navettes en parallèle. Les commandes du robot Baxter ont été modélisé par des machinés à états finis (MEF), une pour chaque bras du robot.

La commande ainsi réalisé possède l'architecture figure 1.7 que vous pouvez observer ci dessous.

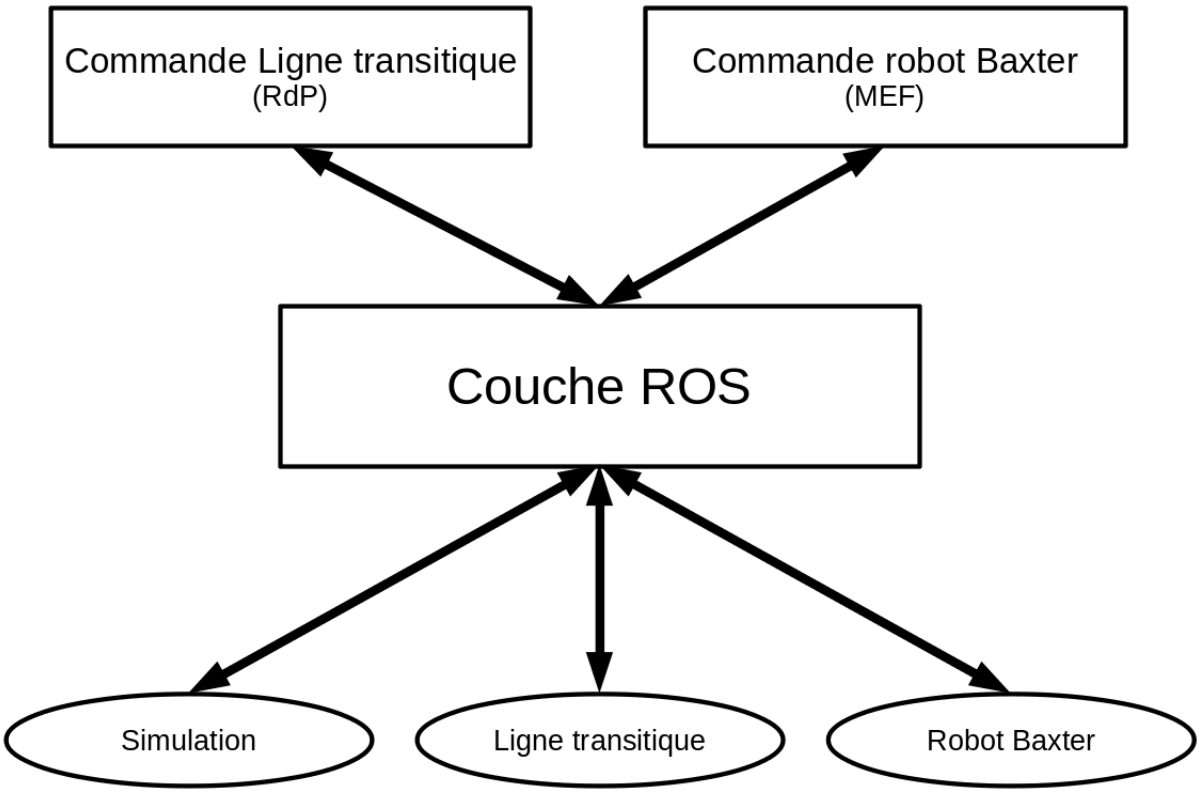

FIGURE 1.7 – Architecture de la solution mise en place

1.6 Présentation du middleware ROS

ROS (Robot Operating System) est une plateforme de développement logiciel qui facilite le développement d'applications robotiques car il permet de créer des pont de communication entre différentes entités de façon très simple. ROS peut être implémenté dans 2 langages : le Python et le C++, j'ai utilisé le C++.

ROS permet l'échange de messages entre nœuds qui sont stockés dans des répertoires nommés packages. Les nœuds sont des exécutables qui utilisent ROS afin de communiquer avec d'autres nœuds. Pour que les nœuds puissent communiquer entre eux, il faut lancer la plateforme en activant un maître. Les nœuds peuvent échanger alors les informations entre eux soit de manière asynchrone via un topic (sujet) ou de manière synchrone via un service. Chaque nœud ROS évolue en parallèle par rapport aux autres.

Le principe des topics est assez simple, il est possible de publier sur un topic ou bien d'y être abonné. Lorsqu'un nœud publie sur un topic, il est le publisher, il envoie un message à des instants choisis par le programmeur (à une certaine fréquence ou sous certaines conditions). Lorsqu'un nœud est abonné à un topic, il est le subscriber, il reçoit tous les messages publiés sur ce topic et pour chaque message reçu, une fonction appelée fonction "Callback" est lancée afin de traiter le message comme on le souhaite. Il est possible de publier sur plusieurs topics et d'être abonné à plusieurs topics à la fois. Pour publier et lire des messages sur un topic, il faut aussi définir les types de messages que l'on souhaite utiliser.

Lorsque qu'un nœud utilise un service, il est le client de ce service, il faut alors utiliser la requête et la réponse de ce service. Ainsi le nœud peut envoyer une requête vers ce service, il recevra ensuite la réponse à sa requête. Le nœud qui fournit le service est le serveur.

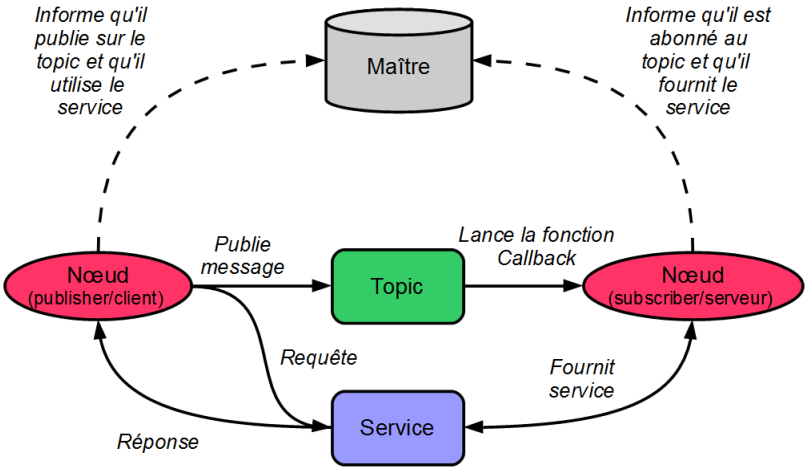

FIGURE 1.8 – Fonctionnement du middleware ROS

Chapitre 2

Développement sous ROS

2.1 Les topics et services de Baxter

Afin de communiquer avec Baxter, le robot possède plusieurs topics ROS sur lesquels il publie régulièrement pour envoyer les valeurs de ses capteurs et d'autres auxquels il est abonné pour pouvoir le commander. Je présenterais ici les topics et un service que j'ai utilisés pour la commande de Baxter ainsi que d'autres topics qui peuvent s'avérer utiles.

2.1.1 Les topics des états

/robot/state

Ce topic permet de savoir si le robot est activé avant de le commander, il utilise des messages de type "baxter_core_msgs/AssemblyState.msg" :

```
bool enabled
bool stopped
bool error
uint8 estop_button
uint8 estop_source
```

/robot/joint_states

Ce topic permet de connaitre l'état de chacun des angles du robot (nom, position, vitesse et effort), il utilise des messages de type "sensor_msgs/JointState.msg" :

```
std_msgs/Header header
string[] name
float64[] position
float64[] velocity
float64[] effort
```

/robot/limb/left/endpoint_state et /robot/limb/right/endpoint_state

Ces topics permettent de connaître la position des extrémités des bras en cordonnées cartésiennes (pose.position), ainsi que leur orientation en quaternions (pose.quaternions) et les efforts qui sont exercés à ces extrémités (twist et wrench). Les messages sont de type "baxter_core_msgs/EndpointState.msg" :

```
geometry_msgs/Pose pose
geometry_msgs/Twist twist
geometry_msgs/Wrench wrench
```

avec "geometry_msgs/Pose.msg" :

```
geometry_msgs/Point position
geometry_msgs/Quaternion orientation
```

/robot/end_effector/left_gripper/state et /robot/end_effector/right_gripper/state

Ces topics permettent de connaître l'état de la pince du bras gauche ou droit, ils utilisent les messages de type "baxter_core_msgs/EndEffectorState" :

time timestamp
uint32 id
uint8 enabled
uint8 calibrated
uint8 ready
uint8 moving
uint8 gripping
uint8 missed
uint8 error
uint8 reverse
float32 position
float32 force
string state
string command
string command_sender
uint32 command_sequence

2.1.2 Les topics de commande

/robot/set_super_enable

Ce topic permet d'activer le robot afin de le commander, il utilise des messages de type "std_msgs/Bool.msg" :

bool data

/robot/limb/left/joint_command et /robot/limb/right/joint_command

Ces topics permettent de commander les bras du robot suivant les modes décrit dans 1.2, ils utilisent les messages de type "baxter_core_msgs/JointCommand.msg" :

int32 mode
float64[] command
string[] names

/robot/end_effector/left_gripper/command et /robot/end_effector/right_gripper/command

Ces topics permettent de commander les pinces gauche ou droite, ils utilisent les messages de type "baxter_core_msgs/EndEffectorCommand" :

uint32 id
string command
string args
string sender
uint32 sequence

2.1.3 Le service Inverse Kinematics

Ce service[1] fournit les angles pour le bras gauche ou droit (en fonction du service) pour une position et une orientation de l'extrémité du bras. La position est en coordonnées cartésiennes et l'orientation est en quaternions. La requête et la réponse sont définies dans le fichier "baxter_core_msgs/SolvePositionIK.srv" :

1. /ExternalTools/<left/right>/PositionKinematicsNode/IKService

Requête :
geometry_msgs/PoseStamped[] pose_stamp
sensor_msgs/JointState[] seed_angles
uint8 seed_mode
Réponse :
sensor_msgs/JointState[] joints
bool[] isValid
uint8[] result_type

Afin d'utiliser les quaternions, je suis passé par les angles d'Euler ψ, Θ et φ qui sont plus intuitifs à utiliser.

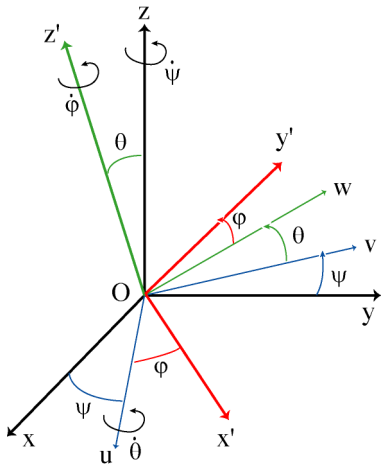

FIGURE 2.1 – Angles d'Euler

Ainsi, pour utiliser les quaternions à partir des angles d'Euler, on utilise les équations de passage suivantes :

$$\begin{aligned} x &= sin(\psi/2).sin(\Theta/2).cos(\varphi/2) - cos(\psi/2).cos(\Theta/2).sin(\varphi/2) \\ y &= sin(\psi/2).cos(\Theta/2).cos(\varphi/2) - cos(\psi/2).sin(\Theta/2).sin(\varphi/2) \\ z &= cos(\psi/2).sin(\Theta/2).cos(\varphi/2) - sin(\psi/2).cos(\Theta/2).sin(\varphi/2) \\ w &= cos(\psi/2).cos(\Theta/2).cos(\varphi/2) - sin(\psi/2).sin(\Theta/2).sin(\varphi/2) \end{aligned} \tag{2.1}$$

2.2 Les topics de communication entre Baxter et la ligne transitique

Afin d'établir la communication entre le robot Baxter et la ligne transitiquen j'ai défini des topics supplémentaires qui utilisent tous des messages de type "std_msgs/Bool.msg" :

- **/pont_BaxterLigneTransitique/<left/right>_arm/attente_prise** : Ce topic permet au robot Baxter d'indiquer à la ligne transitique qu'il est en attente d'une prise pour le bras gauche ou droit.
- **/pont_BaxterLigneTransitique/<left/right>_arm/prise_demandee** : Ce topic permet à la ligne transitique de demander un prise au robot avec le bras gauche ou droit.
- **/pont_BaxterLigneTransitique/<left/right>_arm/prise_effectuee** : Ce topic permet au robot de notifier la ligne qu'une prise avec le bras gauche ou droit a été effectuée.

2.3 Le nœud Commande_Baxter

Le nœud Commande_Baxter permet de commander le robot Baxter à une fréquence de 100 Hz, il utilise les différents topics et services que l'on peut voir sur la figure 2.2.

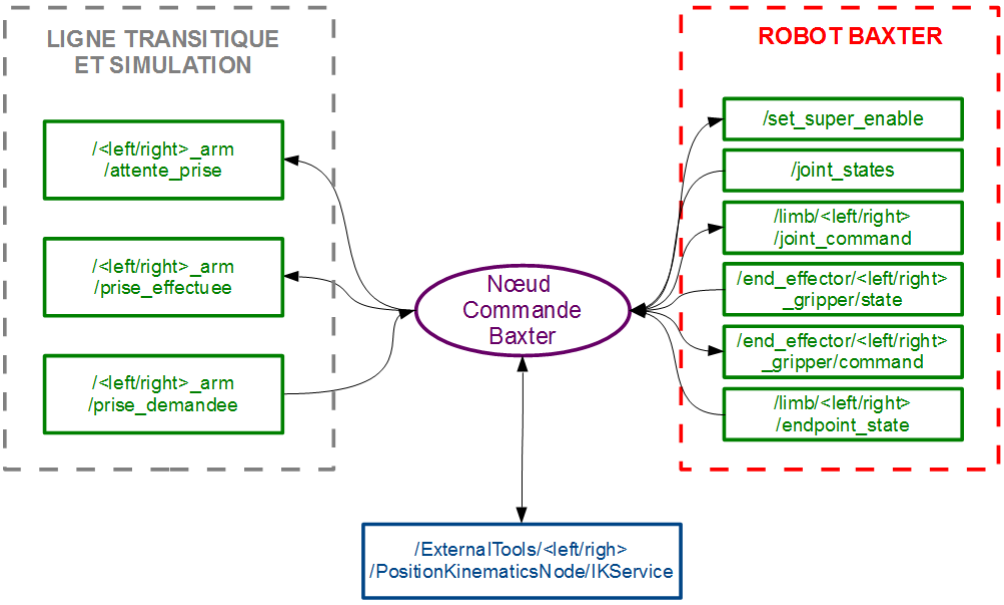

FIGURE 2.2 – Nœud Commande_Baxter avec ses topics et services

Pour ce noeud, j'ai créé les classes Baxter, Baxter_left_arm et Baxter_right_arm afin de commander le robot et plus particulièrement chaque bras indépendamment de l'autre.

2.3.1 La classe Baxter

Je présenterai ici la classe Baxter que j'ai créée afin de pouvoir commander les bras du robot Baxter.

Attributs

- **pub_enable** : de type Publisher afin de publier sur */robot/set_super_enable*.
- **sub_robot_state** : de type Subscriber pour souscrire à */robot/state*.
- **Bras_droit** et **Bras_gauche** : Objets de type Baxter_right_arm et Baxter_left_arm pour commander les bras.
- **enableRobot** : message pour le topic */robot/set_super_enable*.
- **robotSate** : message reçu par le topic */robot/state*.

Méthodes

- **Baxter(noeud)** et **~Baxter()** : Ce sont le constructeur et le destructeur de la classe Baxter. Le constructeur permet de définir les topics du publisher **pub_enable** et du subscriber **sub_robot_state**. Il permet aussi d'initialiser les objets **Bras_droit** et **Bras_gauche**.

- **Init()** : Cette méthode permet d'activer le robot et de commander des positions par défaut des bras.

- **Callback_robot_state(msg)** : Cette fonction met à jour l'attribut **robotSate**à chaque fois qu'il a publication sur le topic */robot/state*.

- **Update()** : Cette méthode publie le message **pub_enable** sur le topic */robot/set_super_enable* et met à jour les bras du robot (envoie des commandes pour les bras).

- **Afficher_Etats(EP1,EP2)** : Cette fonction affiche les états de chaque bras du robot.

2.3.2 Les classes Baxter_left_arm et Baxter_right_arm

Je présenterai ici les attributs et méthodes de la classe Baxter_right_arm qui sont quasiment les mêmes que pour la classe Baxter_left_arm.

Attributs

- **pub_joint_cmd**, **pub_gripper_cmd**, **pub_prise_effectuee** et **pub_attente_prise** : types Publisher qui permettent de publier respectivement sur les topics :

  /robot/limb/right/joint_command,

  /robot/end_effector/right_gripper/command,

  /pont_BaxterLigneTransitique/right_arm/prise_effectuee,

  /pont_BaxterLigneTransitique/right_arm/attente_prise.
- **sub_joint_states**, **sub_endpoint_state**, **sub_gripper_state** et **sub_prise_demandee** : types Subscriber qui permettent de souscrire respectivement aux topics :

  /robot/joint_states,

  /robot/limb/right/endpoint_state,

  /robot/end_effector/right_gripper/state,

  /pont_BaxterLigneTransitique/right_arm/prise_demandee.
- **client_inverse_kinematics** : de type ServiceClient, il permet d'utiliser le service */ExternalTools/right/-PositionKinematicsNode/IKService*.
- **msg_JointCommand** : message pour le topic */robot/limb/right/joint_command*.
- **msg_EndEffectorCommand** : message pour le topic */robot/end_effector/right_gripper/command*
- **msg_prise_demandee** : message mis à jour à l'aide du topic */pont_BaxterLigneTransitique/right_arm/prise_demandee*.
- **msg_prise_effectuee** et **msg_attente_prise** : messages pour les topics */pont_BaxterLigneTransitique/right_arm/prise_effectuee* et */pont_BaxterLigneTransitique/right_arm/attente_prise*.
- **jointState** : message mis à jour à l'aide du topic */robot/joint_states*.
- **endpointState** : message mis à jour à l'aide du topic */robot/limb/right/endpoint_state*.
- **endEffectorState** : message mis à jour à l'aide du topic */robot/end_effector/right_gripper/state*.

Méthodes

- **Baxter_right_arm(noeud)** et **~Baxter_right_arm()** : Ce sont le constructeur et le destructeur de la classe. Le constructeur assigne chaque Publisher, Subscriber et ServiceClient à son topic ou service. Il initialise aussi tous les messages

- **Callback_joint_states(msg)**, **Callback_endpoint_state(msg)**, **Callback_gripper_state(msg)** et **Callback_prise_demandee(msg)** : Ces fonctions permettent de mettre à jour les attributs **jointState**, **endpointState**, **endEffectorState** et **msg_prise_demandee**.

- **IK(x,y,z,psi,teta,phi)** : Cette méthode met à jour l'attribut **msg_JointCommand** à partir d'une position en coordonnées cartésienne et d'une orientation en angles d'Euler.

- **Position(right_s0,right_s1,right_e0,right_e1,right_w0,right_w1,right_w2)** : Cette fonction permet de modifier l'attribut **msg_JointCommand** en choisissant directement les angles du bras.

- **Position(angle,num)** : Cette méthode permet de modifier l'attribut **msg_JointCommand** en ne modifiant qu'un seul angle du bras.

- **Prise()** et **Pose()** : Ces fonctions permettent de modifier l'attribut **msg_EndEffectorCommand** qui permet de fermer ou ouvrir la pince du bras.

- **Position_attente()**, **Position_prise()**, **Position_pose()**, **Descente_prise()** et **Descente_pose()** : Ces méthodes modifient l'attribut **msg_JointCommand** pour mettre le bras dans les différentes positions nécessaires à la commande du bras.

- **Prise_effectuee()**, **Attente_prise()** et **Prise_demmandee()** : Ces fonctions permettent de communiquer avec la commande de la ligne transitique.

- **vitesse_nulle()** et **Position(x,y,z)** : Ces méthodes permettent de vérifier ou bien que le bras ne se déplace plus en vérifiant que la vitesse des angles est quasiment nulle ou bien de vérifier que le bras a bien atteint la position qui lui a été commandée.

- **Pince_fermee()**, **Pince_fermee_pos()** et **Pince_ouverte()** : Ces fonctions permettent respectivement de vérifier que la pince tient un objet, que la pince et en position fermée et que la pince est en position ouverte. Pour mes commandes, j'ai utilisé la fonction **Pince_fermee_pos()** et non la fonction **Pince_fermee()** car le robot n'attrapait pas réellement un objet.

- **Position_prise_OK()**, **Position_pose_OK()**, **Descente_pose_OK()** et **Descente_prise_OK()** : Ces méthodes permettent de vérifier que les positions nécessaires à la commande du bras sont bien atteintes.

- **Update()** : Cette fonction permet de publier sur les topics tous les messages qui permettent de commander le bras du robot. Sans elle, les informations de peuvent arriver jusqu'au robot.

2.3.3 Le main

Dans le fichier main_commande.cpp, on déclare et on initialise le nœud ROS "commande_baxter", un objet de type Baxter et on établit la fréquence du nœud à 100 Hz. Dans une boucle infinie, on a 2 MEF codées en mise en œuvre directe. À la fin de chaque boucle, on lance la méthode **Update()** et les fonctions Callback [2] pour mettre à jour les attributs qui on besoin d'être mis à jour.

2. Les fonctions Callback sont lancées à l'aide de ros : :spinOnce()

2.4 Le nœud Commande

Le nœud Commande permet de commander la ligne transitique ou sa simulation à une fréquence de 25 Hz, il utilise les différents topics que l'on peut voir sur la figure 2.3.

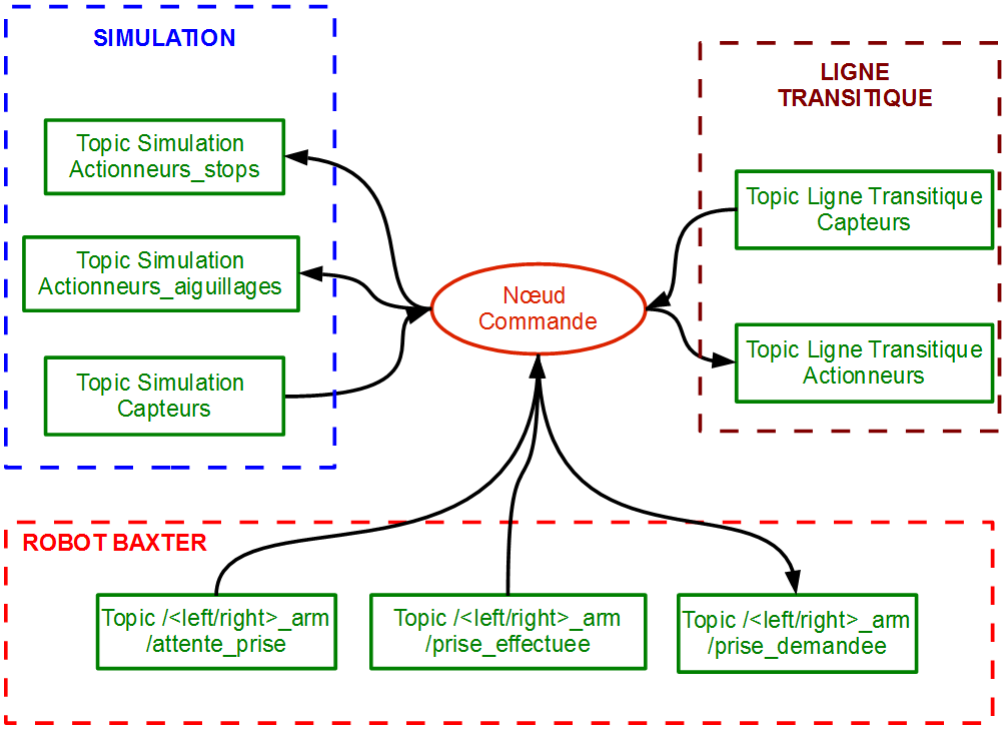

FIGURE 2.3 – Nœud Commande avec ses topics

Pour ce nœud, j'ai ajouté la classe Communication_Baxter afin de pouvoir communiquer avec les classes Baxter_left_arm et Baxter_right_arm qui permettent de commander les bras du robot Baxter.

2.4.1 Les classes Capteurs et Actionneurs

Pour établir la communication entre le nœud *Commande* et la simulation ou la ligne transitique, j'ai utilisé les classes Capteurs et Actionneurs que nous avions créées pendant notre TER.

La classe Capteurs

La classe Capteurs permet d'actualiser l'état des capteurs pour la commande que ce soit ceux de la simulation ou bien de la ligne transitique.

ATTRIBUTS

- **sub_capteurs_ligne** : Permet de s'abonner au topic *Ligne Transitique Capteurs* (Subscriber).
- **sub_capteurs_simu** : Permet de s'abonner au topic *Simulation Capteurs* (Subscriber)
- **PSx**, **DxD**, **DxG**, **CPx**, **CPIx** : Tableaux contenant les valeurs des différents types de capteurs qui seront mis à jour pendant l'exécution du nœud.
- **SIMULATION**, **LIGNE** : Indicateurs permettant de savoir si la commande communique avec la ligne transitique (**LIGNE**=1 et **SIMULATION**=0) ou avec la simulation (**SIMULATION**=1 et **LIGNE**=0).

MÉTHODES

- **Capteurs**(*noeud*)
  C'est le constructeur de la classe, il assigne les attributs **sub_capteurs_ligne** et **sub_capteurs_simu** à leurs topics respectifs. Il initialise aussi tous les tableaux des capteurs ainsi que **SIMULATION** et **LIGNE** à 0.

- **Actualiser**(*PS , DD , DG , CP , CPI*)
  Cette fonction actualise les capteurs déclarés dans le main pour la commande faite par RdP ou MEF.

- **Callback_capteurs_ligne**(*msg*)
  Cette fonction actualise les attributs **PSx**, **DxD**, **DxG**, **CPx** et **CPIx** de la classe Capteurs lorsque la ligne transitique est active.

- **Callback_capteurs_simulation**(*msg*)
  Cette fonction actualise les attributs **PSx**, **DxD**, **DxG**, **CPx** et **CPIx** de la classe Capteurs lorsque la simulation est active.

- **Actualiser_PSx**(*CAPTEURS*), **Actualiser_DxD**(*CAPTEURS*), **Actualiser_DxG**(*CAPTEURS*), **Actualiser_CPx**(*CAPTEURS*), **Actualiser_CPIx**(*CAPTEURS*)
  Ces méthodes permettent d'actualiser les tableaux de capteurs de la classe à partir du message *CAPTEURS* reçu depuis la ligne transitique.

- **MASK**(*registre , numero_bit*)
  Cette fonction n'est pas une méthode de la classe mais elle est très utile pour l'actualisation des capteurs lorsqu'ils sont envoyés par la ligne transitique. Comme nous l'avons vu précédemment, elle permet de récupérer un seul bit d'un mot de données.

La classe Actionneurs

La classe Actionneurs permet d'envoyer les valeurs des actionneurs déterminées par la commande que soit pour la simulation ou bien pour la ligne transitique.

ATTRIBUTS

- **pub_actionneurs_ligne** : Permet de publier sur le topic *Ligne Transitique Actionneurs* (Publisher).
- **pub_actionneurs_simu_aguillages** : Permet de publier sur le topic *Simulation Actionneurs_aiguillages* (Publisher).
- **pub_actionneurs_simu_stops** : Permet de publier sur le topic *Simulation Actionneurs_stops* (Publisher).
- **Actionneurs_ligne** : Mot de données correspondant aux actionneurs à destination de la ligne transitique.
- **actionneurs_simulation_Stop** : Mot de données correspondant aux actionneurs stops à destination de la simulation.
- **actionneurs_simulation_Aguillages** : Mot de données correspondant aux actionneurs aiguillages à destination de la simulation.

MÉTHODES

- **Actionneurs**(*noeud*)
  C'est le constructeurs de la classe, il assigne les attributs **pub_actionneurs_ligne**, **pub_actionneurs_simu_aguillages** et **pub_actionneurs_simu_stops** à leurs topics respectifs. Il initialise aussi **Actionneurs_ligne**, **actionneurs_simulation_Stop** et **actionneurs_simulation_Aguillages** à 0.

- **Envoyer**(*STx , RxD , RxG , Vx , Dx , PIx*)
  Cette méthode permet de mettre à jour les actionneurs de la simulation et de la ligne transitique.

- **publish_actionneurs_ligne**()
  Cette méthode publie le message *Actionneurs* sur le topic *Ligne Transitique Actionneurs*

- **publish_actionneurs_simulation**()
  Cette méthode publie les messages *StopControl* et *SwtichControl* sur les topics *Simulation Actionneurs_stops* et *Simulation Actionneurs_aiguillages*.

- **Ecrire_ligne_STx(***STx***)**, **Ecrire_ligne_RxD(***RxD***)**, **Ecrire_ligne_RxG(***RxG***)**, **Ecrire_ligne_PIx(***PIx***)**, **Ecrire_ligne_Vx(***Vx***)**, **Ecrire_ligne_Dx(***Dx***)**
  Ces méthodes permettent de modifier l'attribut **Actionneurs_ligne** en fonction des tableaux d'actionneurs venant de la commande (main).

- **WRITE(***registre , donnee, numero_bit***)**
  Cette fonction n'appartient pas non plus à la classe Actionneurs mais elle est très utile pour écrire les valeurs des actionneurs un à un pour la ligne transitique. Comme énoncé précédemment, elle permet de modifier un seul bit dans un mot de données.

Fonctions supplémentaires

Afin de simplifier au maximum le main du nœud *Commande*, nous avons créé les fonctions suivantes :

- **Initialisation(***PSx , DxD , DxG , CPx , CPIx , STx , RxD , RxG , Vx , Dx , PIx***)**
  Cette fonction initialise tous les capteurs et actionneurs nécessaires à la commande déclarés dans le main.

- **Deplacer_navettes(***Actionneurs , STx , RxD , RxG , Vx , Dx , PIx , numero_stop***)**
  Cette fonction permet de déplacer les navettes devant n'importe quel stop de la ligne transitique sur la simulation pour choisir la position initiale des navettes. Sur la ligne transitique réelle, on peut disposer les navettes à la main.

- **Mode_ligne(***Actionneurs , STx , RxD , RxG , Vx , Dx , PIx***)**
  Cette fonction permet de mettre la simulation dans la même configuration que la ligne transitique. Durant notre projet, nous n'avons commandé que les automates 1 et 2, nous avons donc bloqué les aiguillages 3 et 10 en position gauche. Cette fonction fait la même chose lorsque l'on commande la simulation et que l'on souhaite que celle-ci soit dans la même configuration que la ligne réelle.

- **Afficher_capteurs(***PSx , DxD , DxG , CPx , CPIx***)**, **Afficher_actionneurs(***STx , RxD , RxG , Vx , Dx , PIx***)**, **Afficher_marquage_RdP(***M , nombre_places***)**
  Ces fonction permettent d'afficher en temps réel les valeurs de tous les capteurs, actionneurs et du marquage du RdP de la commande pour vérifier le bon fonctionnement de la commande.

2.4.2 La classe Communication_Baxter

Je présenterai ici les attributs et méthodes de la classe Communication_Baxter qui permettent à la ligne transitique (ou sa simulation) et au robot Baxter de communiquer ensembles.

Attributs

- **pub_prise_demandee_bras_droit** et **pub_prise_demandee_bras_gauche** : types Publisher qui permettent de publier respectivement sur les topics :

  /pont_BaxterLigneTransitique/right_arm/prise_demandee,

  /pont_BaxterLigneTransitique/left_arm/prise_demandee.
- **sub_prise_effectuee_bras_droit**, **sub_prise_effectuee_bras_gauche**, **sub_attente_prise_bras_droit** et **sub_attente_prise_bras_gauche** : types Subscriber qui permettent de souscrire respectivement aux topics :

  /pont_BaxterLigneTransitique/right_arm/prise_effectuee,

  /pont_BaxterLigneTransitique/left_arm/prise_effectuee,

  /pont_BaxterLigneTransitique/right_arm/attente_prise,

  /pont_BaxterLigneTransitique/left_arm/attente_prise.
- **msg_prise_demandee_bras_droit** et **msg_prise_demandee_bras_gauche** : messages pour les topics */pont_BaxterLigneTransitique/right_arm/prise_demandee* et */pont_BaxterLigneTransitique/gau* emphche_arm/prise_demandee.
- **msg_prise_effectuee_bras_droit** et **msg_prise_effectuee_bras_gauche** : messages mis à jour à l'aide des topics */pont_BaxterLigneTransitique/right_arm/prise_effectuee* et */pont_BaxterLigneTransi* emphtique/left_arm/prise_effectuee.
- **msg_attente_prise_bras_droit** et **msg_attente_prise_bras_gauche** : messages mis à jour à l'aide des topics */pont_BaxterLigneTransitique/right_arm/attente_prise* et */pont_BaxterLigneTransitiqu* emphe/left_arm/attente_prise.

Méthodes

- **Communication_Baxter(noeud)** et **~Communication_Baxter()** : Ce sont le constructeur et le destructeur de la classe. Le constructeur permet d'assigner les Publishers et Subscribers à leur topics respectifs et d'initialiser les attributs.

- **Callback_prise_effectuee_bras_droit(msg)**, **Callback_prise_effectuee_bras_gauche(msg)**, **Callback_attente_prise_bras_gauche(msg)** et **Callback_attente_prise_bras_droit(msg)** : Les fonction Callback permettent d'actualiser respectivement les attributs **msg_prise_effectuee_bras_droit**, **msg_prise_effectuee_bras_gauche**, **msg_attente_prise_bras_droit** et **msg_attente_prise_bras_gauche**.

- **Prise_effectuee_bras_droit()**, **Prise_effectuee_bras_gauche()**, **Attente_prise_bras_droit()**, **Attente_prise_bras_gauche()**, **Demander_prise_bras_droit()** et **Demander_prise_bras_gauche()** : Ces méthodes permettent de communiquer avec le robot Baxter.

- **Afficher_Communication_Baxter()** : Cette fonction permet d'afficher les états de la communication.

- **Update()** : Cette méthode permet de publier les messages de tous les Publishers.

2.4.3 Le main

Dans le fichier main_commande.cpp, on déclare et on initialise le nœud ROS "commande", les objets de type Capteurs, Actionneurs et Communication_Baxter. On établit la fréquence du nœud à 25 Hz : une fréquence inférieure aux fréquences des nœuds permettant de communiquer avec les automates de la ligne transitique. On initialise aussi tous les capteurs et actionneurs de la ligne transitique ainsi que le RdP de l'on utilise pour la commande.

Dans une boucle infinie, on teste tout d'abord si l'on commande la simulation ou la ligne transitique afin de commander la configuration initiale. Ensuite, on actualise les capteurs, le RdP codé en mise en œuvre directe donne les états suivants et active les actionneurs correspondants. Enfin, on envoie ces information vers Baxter et la ligne transitique (ou sa simulation).

Chapitre 3

Synthèse de commande

Pour commander la ligne transitique en interaction avec le robot Baxter, j'ai tout d'abord commandé la prise d'un objet virtuel avec un des bras du robot, puis avec les deux bras. Ensuite, j'ai créé une commande pour les bras et une autre pour la ligne transitique et je les ai enfin synchronisées.

3.1 Commande du robot seul

Pour commander chaque bras du robot Baxter, j'ai utilisé une MEF pour chaque bras. Dans l'état initial, un bras est en attente d'une prise. Lorsque le robot n'est pas en interaction avec la ligne transitique, une prise est demandé 5 secondes après que la MEF soie dans l'état initial.

Ensuite, pour effectuer la prise, le bras passe par plusieurs positions et informe qu'une prise a été effectuée lorsque le bras remonte après la prise. Pour que le bras passe bien par toutes les positions, on vérifie qu'elles sont bien atteintes et parfois, lorsque ce test ne suffit pas, on attends que la vitesse du bras soit presque nulle avant de commander une autre position.

Lorsque l'on utilise les deux bras en parallèle pour effectuer les prises, les positions des prises pour les deux bras ne sont pas les mêmes. Il n'est donc pas possible de prendre un objet au même endroit avec les deux bras à l'aide de cette commande. Le but des commandes que je vais présenter est principalement de les synchroniser ensembles.

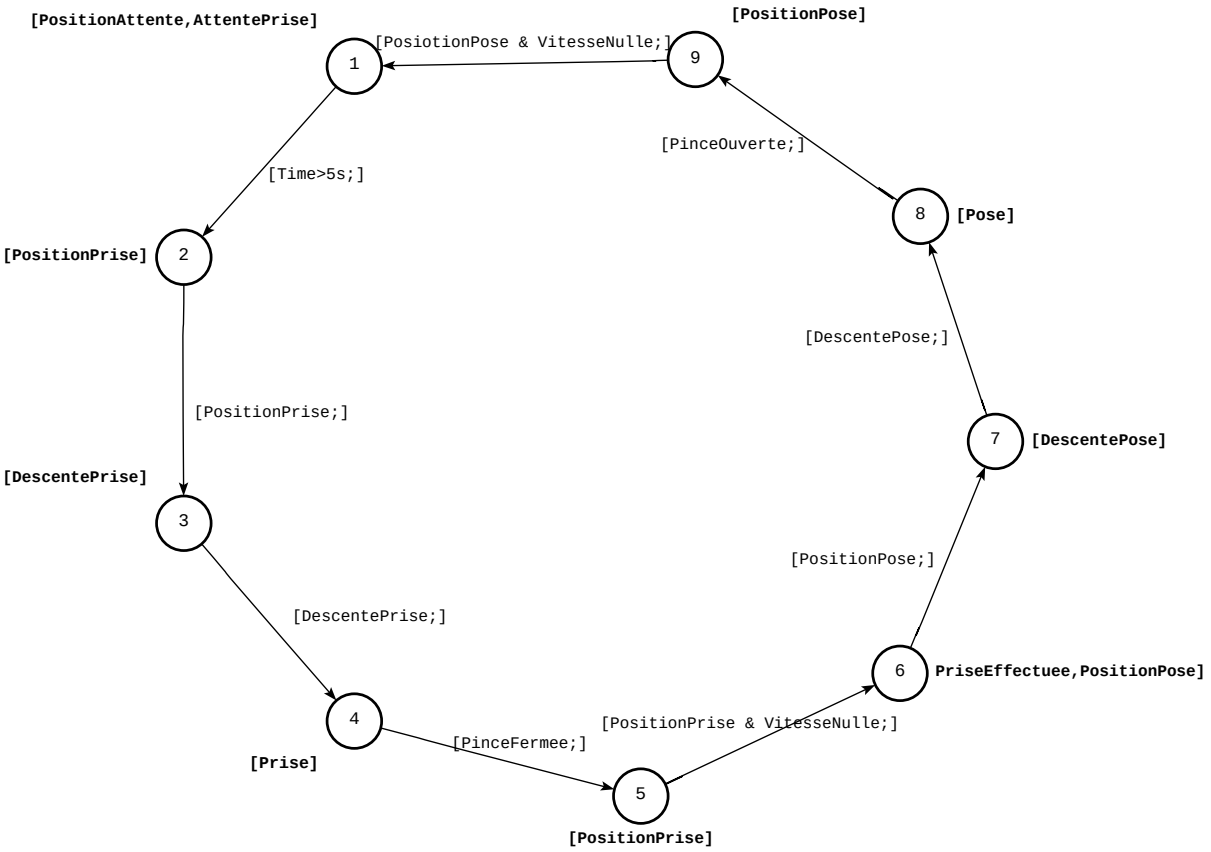

FIGURE 3.1 – Machine à états finis de la commande d'un bras du robot Baxter

3.2 Commande de la ligne transitique MONTRAC en intéraction avec le robot Baxter

Lorsqu'un bras est en interaction avec la ligne transitique (ou sa simulation), on attends qu'une prise soient demandée lorsque la MEF est dans l'état initial. Une prise ne peut être demandée que lorsque le MEF est dans l'état 1, c'est à dire que la prise précédente est terminée ou qu'aucune prise n'a encore eut lieu.

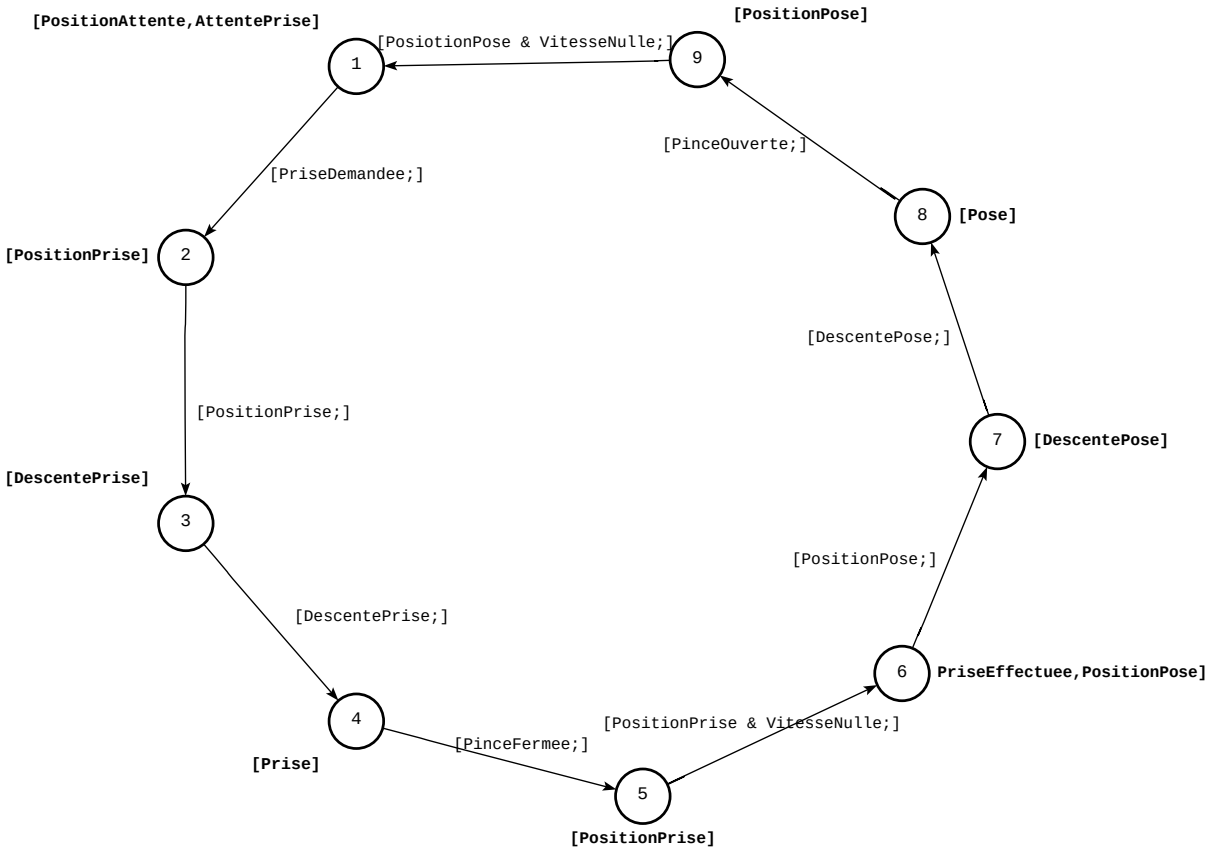

FIGURE 3.2 – Machine à états finis de la commande de chaqu'un des deux bras en interaction avec la ligne transitique

3.2.1 Commande de la ligne transitique en interaction avec un des bras manipulateurs

Pour la commande de la ligne transitique, dans un premier temps, j'ai effectué une commande simple en interaction avec un seul des bras du robot pour laquelle les navettes avancent toujours sur le même circuit (on ne modifie pas la position des aiguillages pendant la commande).

Les navettes démarrent toutes devant le stop numéro 5, elles avancent librement sur les rail tant qu'il n'y a pas d'autre devant. Lorsque l'une d'elles arrive au niveau du stop numéro 22, une prise est demandée au robot Baxter comme s'il y avait un objet à récupérer sur la navette. Une fois la prise effectuée, la navette peut repartir. Lorsqu'une prise est déjà en cours, il faut attendre qu'elle soit terminée pour que la nouvelle demande prenne effet au sein de la commande du robot.

Le jeton dans la place $p10$ permet de ne faire passer qu'un seul jeton représentant les navettes à la fois lorsqu'il y en plusieurs car la dynamique du nœud est trop rapide devant la dynamique de la ligne transitique. Sans cette place les jetons franchissent tous la transition $t1$ alors qu'il n'y a qu'une navette qui se déplace sur la ligne.

3.2.2 Commande de la ligne transitique en interaction avec les deux bras manipulateurs

Pour cette commande, j'ai utilisé les deux bras du robot Baxter. Le principe est le même que pour la commande précédente mais lorsqu'une prise est demandée, c'est le bras qui est disponible qui fait la prise. Ainsi, les navettes attentent moins de temps lorsqu'elles arrivent au stop numéro 22. Le places $p11$ et $p12$ symbolisent la disponibilité de chaque bras.

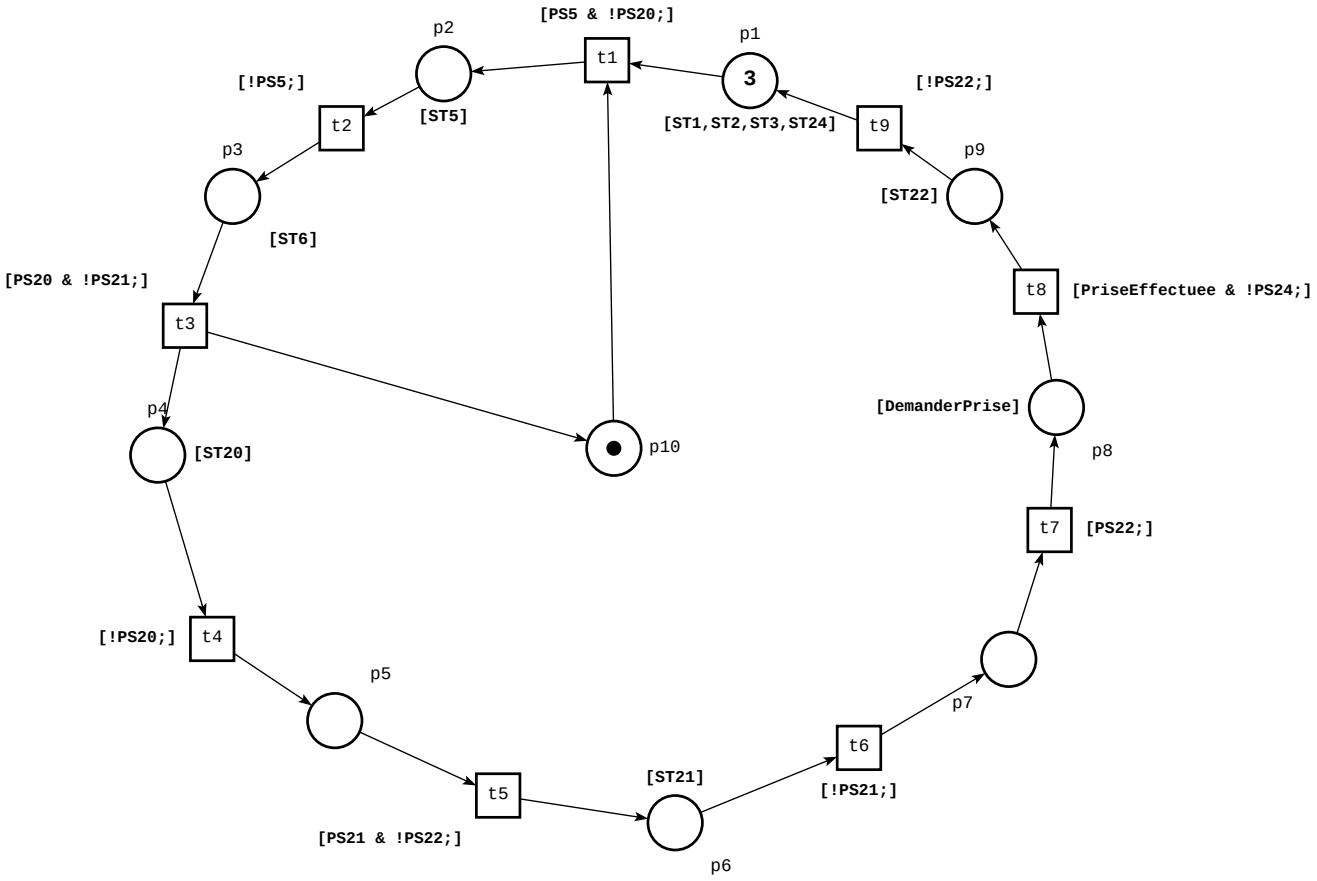

FIGURE 3.3 – Réseau de Petri de la commande de la ligne transitique en interaction avec un bras du robot Baxter

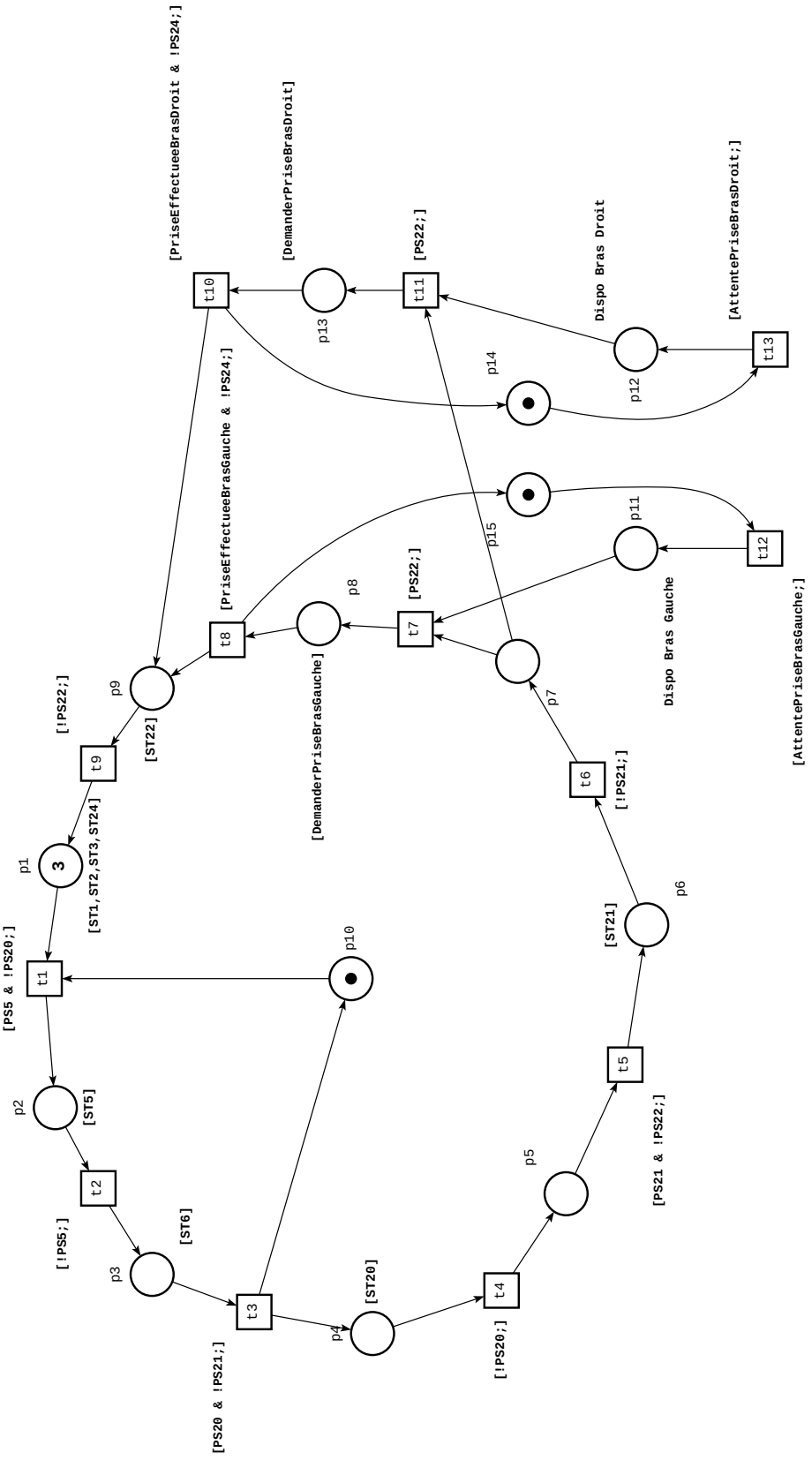

FIGURE 3.4 – Réseau de Petri de la commande de la ligne transitique en interaction avec les deux bras du robot Baxter

Conclusion

Durant ce stage j'ai pu réaliser une commande de la ligne transitique en interaction avec le robot Baxter de manière synchrone tout en apprenant à utiliser le robot à l'aide du middleware ROS. J'ai alors pu renforcer mes compétences concernant ROS.

J'ai donc pu mettre un premier pas dans la robotique en découvrant le robot Baxter et en commandant plusieurs trajectoires à ses bras manipulateurs.

Enfin, ce stage m'aura permis de consolider mes connaissances en informatique notamment en apprenant à utiliser le logiciel GIT qui est très utile pour partager du code et collaborer sur différents projets.

Bibliographie

[1] ABIVEN 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.

[2] 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.
Disponible sur <`https://github.com/ClementLeBihan/CelluleFlexible/blob/master/Livrables/Rapport_Projet_Long`> (Consulté le 26.05.2016)

[3] ANTONIUTTI Emilie, BERTIN Thibault, DEMMER Simon, LE BIHAN Clément. *Simulation de la ligne transitique MONTRAC.* Code source (GIT). Toulouse : ENSEEIHT, 2016.
Disponible sur <`https://github.com/ClementLeBihan/CelluleFlexible`> (Consulté le 26.05.2016)

[4] GORRY POLLET Alexandra. *Commande d'une cellule flexible de production robotisée.* Rapport de stage, IUT Génie Électrique et Informatique Industrielle. Toulouse : Université Paul Sabatier, 2015.

[5] DATO Bruno, ELGOURAIN Abdellah, 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.

[6] AIP-PRIMECA. *Pôle AIP-PRIMECA Toulouse.*
Disponible sur <`http://aip-primeca.ups-tlse.fr/`> (Consulté le 26.05.2016)

[7] Robot Operating System. *ROS.*
Disponible sur <`http://www.ros.org/`> (Consulté le 26.05.2016)

[8] V-REP. *v-rep virtual robot experimentation platform.*
Disponible sur <`http://www.coppeliarobotics.com/`> (Consulté le 26.05.2016)

[9] RethinkRobotics.
Disponible sur <`http://www.rethinkrobotics.com/`> (Consulté le 31.07.2016)

[10] Baxter Research Robot Wiki.
Disponible sur <`http://sdk.rethinkrobotics.com/wiki/Main_Page`> (Consulté le 31.07.2016)