Université Paul Sabatier

Master Intelligence Artificielle et
Reconnaissance des Formes
Master Robotique : Décision et Commande

Manuel développeur

Navigation Autonome de Robot Mobile

Auteurs:
Thibaut Aghnatios
Marine Bouchet
Bruno Dato
Tristan Klempka
Thibault Lagoute

Tuteurs:
Frédéric Lerasle
Michaël Lauer
Michel Taix

13 mars 2017

Suivi du document

Nom du document	Version Majeure	Version Mineure	Date de création	Dernière version
Manuel développeur	A	4	13/03/2017	30/03/2017

Auteurs du document

Rédaction	Intégration	Relecture	Validation Interne
Equipe	??	??	??

Validation du document

Validation	Nom	Date	Visa

Liste de diffusion

Le rapport du projet est diffusé à l'ensemble des clients et des intervenants externes aux projets.

Historiques de révision

Version	Modification apportée	Auteur	Date
A.0	Création du document	Bruno Dato	13/03/2017
A.1	Sections 2.2 et 1	Marine Bouchet	??/03/2017
A.2	Section 2.2	Tristan Klempka	??/03/2017
A.3	Section 2.4	Thibaut Aghnatios	30/03/2017
A.4	Sections 2.3 et 2.5	Bruno Dato	30/03/2017

Contents

1 Introduction

Ce manuel donne les informations nécessaires à l'utilisation et à l'évolution du projet "Navigation autonome d'un robot mobile" du Master 2 IARF et RODECO. Celui-ci permet, à ce jour, d'effectuer une navigation autonome d'un Turtlebot dans un environnement connu, contenant des amers dont leur position et leur orientation sont connus. Ces indices visuels, récupérer par la kinect, capteur RGB-D, permettent de supprimer les erreurs systématiques de la localisation des capteurs proprioceptifs embarqués – de l'odométrie.

La navigation s'effectue entre la position initiale, dont la position est connue ou dans la zone de visibilité d'un amer, et une position finale :

Scénario	A	B
Départ	Position absolue	Position inconnue mais dans une zone de visibilité d'un marqueur
Arrivée	Position	Position

DESCR + deco

B Si pas de position en entrée
Le robot effectue une recherche de marqueur dans son champs visuel
Tant que pas de marqueur en vue
Le robot tourne de ?
Il effectue un autre recherche de marqueur dans son champs visuel

A Tant que le but est plus loin que le prochain marqueur dans sa direction
Le robot se déplace jusqu'à la zone de visibilité du marqueur le plus proche dans la direction du but
Le robot effectue une recherche de marqueur dans son champs visuel
Tant que pas de marqueur en vue
Le robot tourne de ?
Il effectue un autre recherche de marqueur dans son champs visuel
Le robot se déplace jusqu'au but

2 Navigation avec amers 2D dans un environnement connu

La tâche de navigation autonome se découpe en cinq briques : ++ de textes résumé texte

Détection qui permet de repérer les amers dans le champs de vision quand ils se présentent (cf. AR

Localisation qui permet de savoir où se trouve le robot, en utilisant soit les données d'odométrie interne, soit les données d'observation (cf. NOUS

Trajectoire qui permet de se déplacer en boucle fermée d'un point A à un point B (cf. NAV

Commande "

Visibilité qui permet de générer une carte contenant les positions des amers et leurs orientations ; et de renvoyer à partir de cette carte le nombre d'amers visibles.

Supervision qui gère le bon déroulement de la tâche, le déplacement du point de départ au point d'arrivée (cf.

2.1 Solution mise en place

L'architecture globale de notre produit est décrite par le schéma suivant :

Figure 1: Architecture ROS

2.2 Détection et localisation

2.2.1 Détection

Utilisation

Nous détaillons ici les étapes nécessaires afin d'utiliser le noeud de détection. Ce noeud est lancé indépendamment des autres noeuds du système. Le fichier ROS de launch permet de lancer le noeud.
roslaunch localisation.launch

Ce fichier configure et lance la détection d'amer de type AR code à l'aide d'une bibliothèque spécialisée couplée à un nœud ROS. Plusieurs paramètres dans ce fichier peuvent être configurés.

- $marker\_size$: Largeur (cm) des marqueurs AR utilisés
- cam_image_topic : Topic ROS du flux d'images de la caméra.
- cam_info_topic : Topic ROS des infos propres à la caméra. C'est ici que les données de calibrage sont récupérées.
- $output\_frame$: Repère dans lequel sera exprimé le résultat de la détection.

Des paramètres supplémentaires se trouvent dans le fichier *localication_node.cpp*. Le changement de ces paramètre nécessite une compilation du noeud ROS.

- $DEBUG$: Active ou non les sorties de debug. Si cette option est activée, les transformations intermediaires sont publiées.
- $TIMEOUT\_AR\_DETEC$: Temps d'attente maximum pour la reception d'un marqueur. Ce paramètre est utile lors de l'utilisation en réseau. La latence provoque des retards dans la publication des transformées. Il est donc nécessaire de donner une marge de temps au système. Lors de l'utilisation en mode local cette valeur peut être faible.
- NB_MARKER : Nombre de marqueurs total dans la scène.

Une fois le lancement effectué, l'utilisateur peut observé les comportement de ce noeud avec des sorties console.

- $GLOBAL\_SEARCH$: La recherche de marqueur a été demandée.
- $MARKERDETECTED : ID\_MARKER\_DETECTED$: Identifiant du marqueur détecté.
- $LOOKINGFORTF : tf$:Nom de la transformée attendue.

Remarque : Les performances de la détection des marqueurs reposent sur une bonne calibration de la caméra.

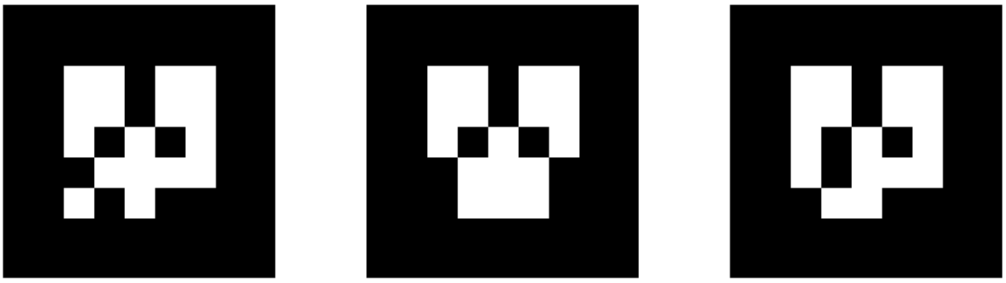

Figure 2: Exemple de marqueurs AR. Les valeurs 3, 4 et 5 sont codés avec ces marqueurs.

Fonctionnement

Afin de détecter les amers dans la scène nous avons recours à la brique ROS : *ar_track_alvar*. Cette brique est une interface qui permet d'intégrer à un projet ROS la librairie Alvar. Alvar est une librairie open source pour la détection de marqueur AR. Cette brique utilise le nuage de points 3D associé à la couleur pour identifier un marqueur. L'information de profondeur permet à l'algorithme de mieux repérer les plans des marqueurs dans la scène. Elle retourne l'identifiant, la position et l'orientation du marqueur dans la position de la */camera_rgb_optical_frame*, dans notre cas. La librairie fournie 55 AR tags et la possibilité d'entendre cette liste facilement. Dans notre cas, nous utilisons uniquement le noeud qui permet de lire plusieurs marqueurs, un par un, dans un même flux vidéo.

Mise en place physique :

Lors de notre projet nous avons utilisé des marqueurs de taille 16×16 cm. On place leur milieu à 31 cm du sol afin que l'axe caméra-marqueur soit le plus parallèle au sol possible.

Initialisation :

Le nœud *ar_track_alvar* s'initialise dans *localisation.launch* avec les paramètres suivant :

```
<arg name="marker_size" default="16" />
<arg name="max_new_marker_error" default="0.08" />
<arg name="max_track_error" default="0.2" />

<arg name="cam_image_topic"
        default="/camera/depth_registered/points" />
<arg name="cam_info_topic"
        default="/camera/depth_registered/camera_info" />
<arg name="output_frame" default="/camera_rgb_optical_frame" />
```

Lecture du contenu [?] :

On peut écouter les ar_track_alvar_msgs::AlvarMarkers que publie le nœud avec la commande :

```
rostopic echo /ar_marker_pose
```

Performances et évolutions :

L'orientation du markeur trouvé par *ar_track_alvar* doit être : $\vec{z}$ la normale et $\vec{x}$ vers le haut. Il arrive, pour des problèmes inconnus (le réseau ?) que ce repère ne soit pas correctement capturé, avec par exemple, $\vec{y}$ vers le haut. Le robot, à la fin de la localisation se trouve alors couché. Il faudrait alors comprendre d'où vient le problème ou, sinon, ne pas prendre en compte les orientations aberrantes.

DEGRES

2.2.2 Localisation

Utilisation

La node de localisation est lancé grâce à un ficher ROS de launch. Cependant, contrairement au noeud de détection, ce noeud ne se lance pas indépendamment des autres. Il est lancé avec l'ensemble du système de navigation avec la commande suivante.
roslaunch navigation.launch

Ce fichier configure et lance le système de navigation qui comprend la node de localisation. On trouve dans ce fichier des paramètres propre à la localisation pouvant être configurés.

- map_file : Chemin de la carte utilisée pour la navigation. Notre système utilise le noeud ROS *map_server* pour le chargement de la carte.
- $marker\_tf\_publisher\_X$: L'utilisateur renseigne ici, pour chaque marqueur, la transformée qui sera publiée pour le repérer sur la carte. L'orientation est renseignée avec un quaternion.

Avant toute chose, il est important de rappeler que quelque soit la situation :

- $/map \rightarrow /odom \rightarrow /base\_link$ (`cf.http://www.ros.org/reps/rep-0105.html`)
- ATTENTION AU REPERE MAP [?]
- /odom est le repère relatif de l'odométrie, assimilé certain, du robot et est placé au départ sur /map. Il est en quelque sorte le point de départ d'un déplacement. Etant donné que ce repère drift au cours du déplacement du robot, il peut être étonné. La localisation via un amer connu permet de placer le repère /odom au bon endroit, et de réinitialiser l'odométrie, ce qui place /base_link au même endroit. C'est un localisation absolu.
- chaque frame peut avoir plusieurs fils mais qu'un seul parent
- la transformé entre deux frames est décrite par deux attributs :
  - m_origin : Vector3 de translation
  - m_basis : Matrix3x3 de rotation

localisation_node.cpp :

Ce nœud permet d'envoyer la nouvelle localisation du robot si il détecte un marqueur.

La recherche se déclanche uniquement quand la commande haut niveau publie un <std_msgs::Empty> sur le topic $/nav/HLC/askForMarker$.

Dans le cas ou un marqueur est visible :
<u>init</u> /map → /odom → /baselink → /camera_rgb_frame → /camera_rgb_optical_frame
<u>arrivée d'un marker</u> /camera_rgb_optical_frame → /ar_marker_0
<u>traitement</u>
/baselink → /camera_rgb_frame → /camera_rgb_optical_frame → /ar_marker_0
on sait donc /ar_marker_0 → /baselink = /marker_0 → /baselink
sachant /map → /marker_0, on sait /map → /baselink = /map → /odom

Cette transformation est envoyé sur $/new\_odom$ que l'on publie avec le publisher *odom_pub*.
On publie également l'id du marqueur vu sur $/nav/loca/markerSeen$ sous un <std_msgs::Int16>.

Dans le cas où on ne voit rien on publie : -1

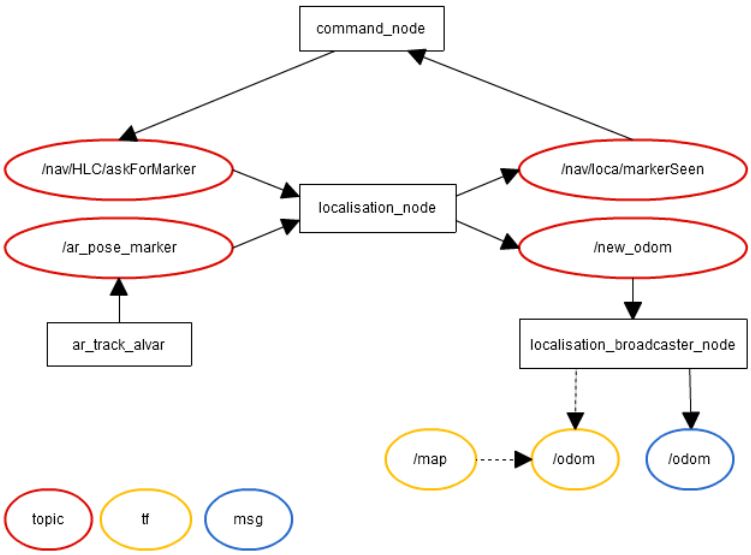

Figure 3: texte de la légende

on_broadcaster_node.cpp :

Ce nœud permet de tout le temps broadcaster la transfom entre /map et /odom. Il souscrit à la <geometry_msgs::Transform> /new_odom, qui contient la "nouvelle" position certaine du robot. Dès que celle-ci est publiée, /odom actualise sa position et on réinitialise le nav_msgs/Odometry : il devient notre nouveau référentiel.

2.3 Commande

Utilisation

Pour la version actuelle du projet, la plupart de la commande est effectué par une brique ROS. La brique *move_base* réalise la commande afin de suivre la trajectoire générée par la brique trajectoire. Cette brique est lancé dans un fichier ROS de launch.
roslaunch turtlebot_proj_nav navigation.launch
La commande peut être configurée à l'aide du fichier XML : *move_base.launch.xml*. Ce fichier XML référence un ensemble de fichiers XML qui configure la commande. Une description exhaustive des paramètres de cette brique peut être trouvée à cette adresse :
http://wiki.ros.org/move_base
Lors de la recherche d'un marqueur, nous reprenons la main et c'est la node *commande_node* qui prend le relais. Celle-ci est également lancée dans *navigation.launch*.

Fonctionnement

Au début du projet une ébauche de commande a été réalisée. Celle-ci est simple, elle permet un déplacement en ligne droite et une rotation sur place. Cette commande est en boucle ouverte. Il peut être intéréssant de continuer ce travail afin de prendre en compte une correction odométrique notamment lors d'un évitement d'obstacle lorsque le système reprend la main sur le *navigation_stack*.
Cette ébauche ce trouve dans le fichier *commande_node.cpp*
Cette node est lancée dans le fichier ROS de launch *navigation.launch*

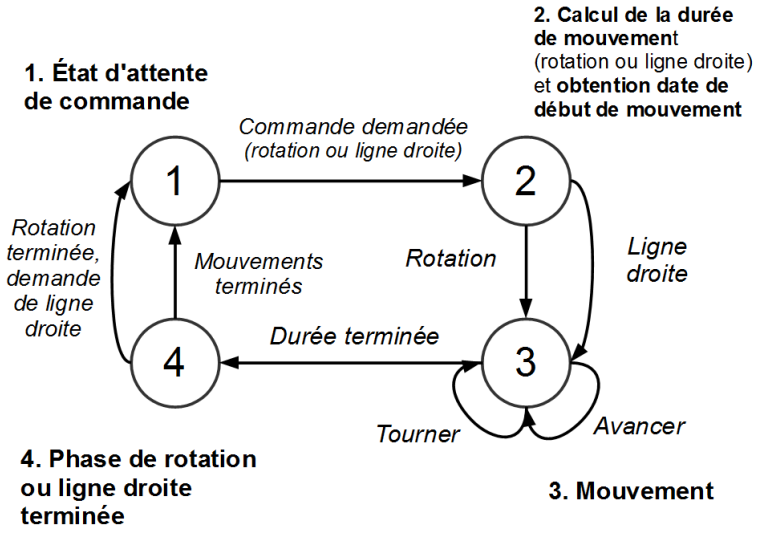

Figure 4: Schéma de la machine à états de la commande utilisée pour la recherche d'une balle et pour la recherche d'un amer

2.4 Visibilité

Le but de cette section est de créer une carte de visibilité, c'est à dire une carte contenant nos différents amers et de retourner le nombre d'amers visibles en fonction de la position du robot.
L'intérêt de cette visibilité est d'avoir une carte de visibilité sous RVIZ pour suivre notre navigation d'amers en amers et de savoir si l'on se trouve dans un champ de visibilité pour lancer la détection et la localisation avec notre superviseur.
C'est pourquoi, nous allons expliquer comment nous générons notre carte de visibilité à l'aide de *visib_pgmwriter_node.cpp* et notre node retournant le nombre d'amers visibles : *visib_pgmreader_node.cpp*.

2.4.1 Génération de la carte de visibilité

Tout d'abord pour générer notre carte de visibilité, il faut préalablement avoir une carte de l'environnement (créé virtuellement ou en utilisant la cartographie disponible sur le Turtlebot).

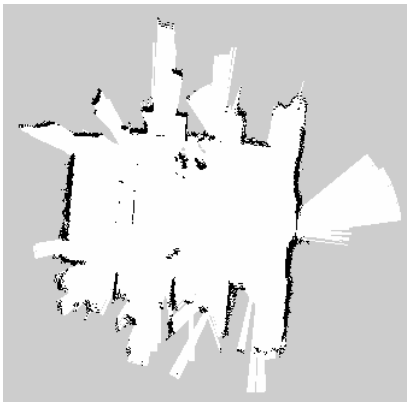

Figure 5: Cartographie de l'environnement avec le Turtlebot

Le node *visib_pgmwriter_node.cpp* ne doit pas être modifié, toutes les configurations se font directement dans le fichier *visib_init.cpp*. En effet, le node lance la fonction *Ecriture_carte_visib*() qui créé un fichier au format PGM l'ensemble des amers définit dans le *graph.xml* se trouvant

dans le dossier */rsc*. Donc pour une taille de carte donnée et pour les configurations effectuées correctement dans *visib_init.cpp*, il suffit de modifier la position et l'orientation de nos amers dans le *graph.xml* pour que la nouvelle carte de visibilité soit généré automatiquement en lançant à nouveau notre node *visib_pgmwriter_node.cpp*.

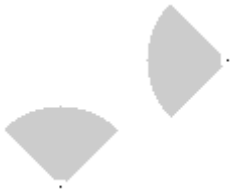

Figure 6: Exemple d'une carte de visibilité

Une fois notre carte générée, il faut pouvoir la visualiser sur Rviz et la superposée avec la carte de notre environnement pour qu'elle soit plus parlante. En lançant le *navigation.launch* notre carte de visibilité est publiée sur le topic *markers_visibility_map*. Il suffit de l'ajouter directement sur Rviz en ajoutant une carte et en se plaçant sur ce topic si la configuration de rviz n'est pas enregistrée. Le résultat obtenu sur rviz est le suivant :

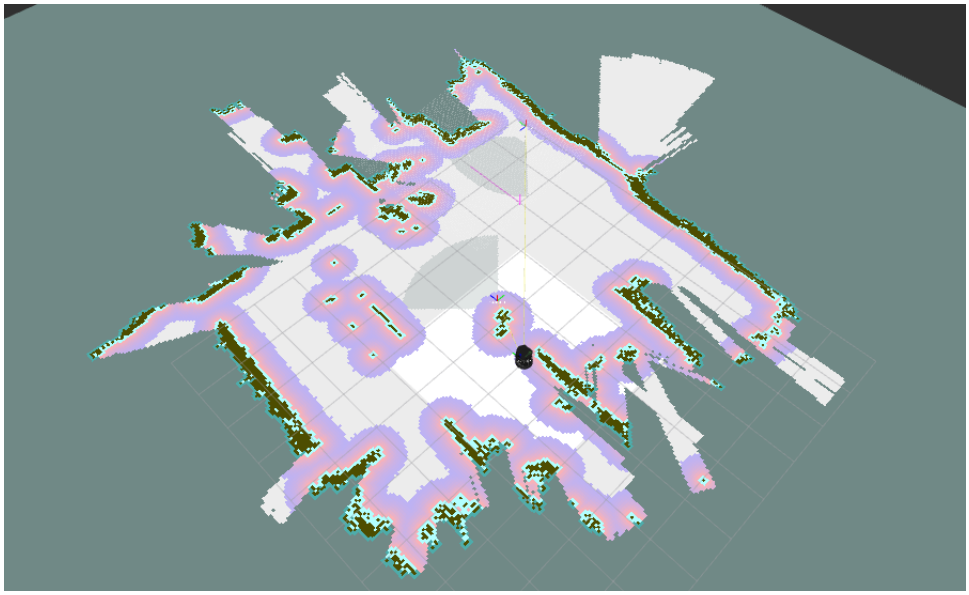

Figure 7: Superposition de nos cartes sur rviz

Au niveau de la configuration de *visib_init.cpp* dans la fonction *Ecriture_carte_visib*(), il faut modifier $intlargeur = 512; inthauteur = 640;$ pour que la carte est toujours les mêmes dimensions que la carte de notre environnement. $intnbr_amers = 2;$ pour prendre en compte le nombre d'amers dans *graph.xml* (mettre 2 si 2 amers dans le graphe). Ensuite dans la boucle for faisant la conversion de la position en m en pixels :

$for(a = 0; a < nbr\_amers; a++)$
{
$x[a] = (x1[a] + 12.2)/0.05;$
$y[a] = -(y1[a] + 13.8 - 32)/0.05;$
$alphamax[a] = pi/4;$
}

Avant tout, il faut savoir qu'on a 3 repères différents.
- On a le repère du robot que l'on nomme R et possède la position en x et y du robot donc R (X,Y).
- Le repère de la carte de l'environnement note R' qui a pour origine (-12.2 ; -13.8) (dans notre cas pour une carte 512x640) donc R' (X+12.2,Y+13.8).
- Ensuite on a le repère de notre carte de visibilité noté Rp qui est positionné en haut a gauche de la carte (vecteur u dans le même sens que x et vecteur v opposé au vecteur y) c'est pourquoi on peut écrire Rp(X+12.2,-(Y+13.8-32)).

Explicitons maintenant les différentes valeurs utilisées : 0.05 correspond à la résolution qui se trouve dans le fichier .yaml dans */map*. Ici 12.2 et 13.8 correspond à l'origine de la carte pour que la superposition des cartes soit correctes. En effet, si on laisse 0;0 la carte se générera a la base du robot sur Rviz donc on aura un décalage lors de la superposition. C'est pourquoi dans le fichier *visib.yaml*, il faut aussi le configurer de cette maniere :
$resolution$: 0.050000
$origin$: $[-12.200000, -13.800000, 0.000000]$
Enfin, la valeur 32 correspond à la conversion pixel/mètres de la hauteur de notre fenêtre soit : $640 * 0.05 = 32$.

2.4.2 Node retournant le nombre d'amers visibles

Maintenant que nous possédons une carte de visibilité, on va l'utiliser pour renvoyer le nombre d'amers visibles à l'aide de la node *visib_pgmreader_node.cpp*. Cette node va aller lire le fichier PGM créé et renvoyer la valeur du pixel pour une position x, y sur cette carte. Si le pixel vaut 15 on est dans une zone blanche donc pas de marqueurs visibles. Pour un pixel de valeur 12, cela veut dire qu'on est dans le champ de visibilité d'un amer. Pour 9, 2 amers visibles et ainsi de suite. Dans notre cas on suppose qu'au maximum 4 amers sont visibles en même temps. Si l'on souhaite plus, il suffit d'augmenter la valeur maximal lors de l'écriture de notre carte de visibilité. En effet, lors de la création de l'entête PGM, on définit la valeur maximale utilisée (dans notre cas 15). Au niveau de cette node, elle n'est plus fonctionnelle lors de notre intégration avec le système complet. Lors des tests avant intégration, la node suivante renvoyée bien le nombre d'amers visibles, cependant en l'intégrant notre fonction $pgm\_imread(char * argv)$ ne stocke plus correctement les valeurs des données. Donc cette node est disponible, cependant il faut la débugger pour qu'elle soit utilisée.

2.5 Superviseur

2.5.1 Comportement de la navigation

Le but du superviseur est de guider le robot pour qu'il atteigne un but de coordonnées (x, y) dans la carte. Afin de gérer le comportement de la navigation à l'aide des amers pour atteindre ce but précis, nous avons mis en place un nœud ROS de supervision : *hightLevelCommand_node*. Ce nœud permet d'utiliser des commande de haut niveau telles que chercher un amer ou bien se déplacer vers un but.

On suppose alors qu'initialement, le robot est dans une position connue a priori et dans une des zones de visibilité. Si sa position est assez proche du but final suivant un seuil que l'on définira par la suite, on commande alors au robot de se déplacer directement vers le but. Si la position initiale est trop éloignée, on se déplace d'amer en amer pour atteindre l'amer qui est le plus proche du but final et enfin se déplacer vers le but final. Tous les amers et les distances entre ces derniers sont représentés par un graphe, ainsi on utilise l'algorithme de plus court chemin de Dijkstra pour déterminer vers quel amer se déplacer lorsque l'on en a détecté un.

Le comportement détaillé du superviseur est décrit par la machine à états figure 8.

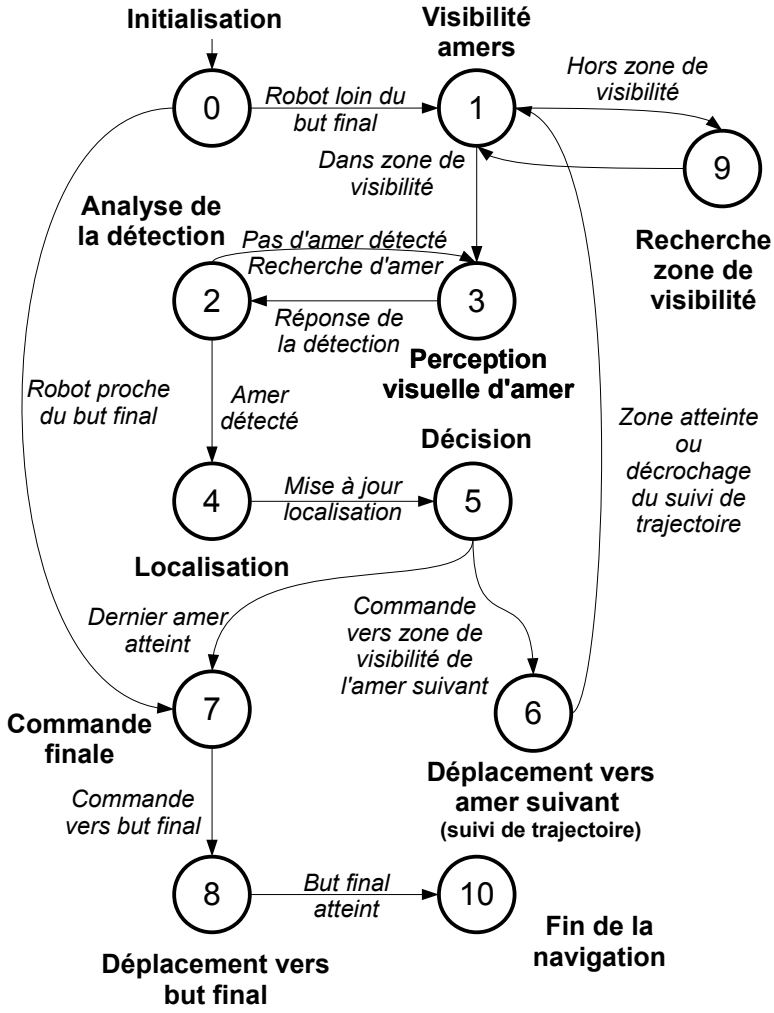

Figure 8: Schéma de la machine à états du superviseur de la navigation entre amers

2.5.2 Commandes de haut niveau

(détail des commandes principales)

2.5.3 Graphe des amers

(construction du graphe et explications)

2.5.4 Lancement du superviseur

(explication des différents paramètres)

3 Conclusion

List of Figures

ANNEXE