UNIVERSITÉ DU QUÉBEC À MONTRÉAL

INTÉGRATION D'UN PLANIFICATEUR DE TÂCHES AVEC CONCURRENCE D'ACTIONS ET INCERTITUDE SUR LE TEMPS AU SEIN D'UNE ARCHITECTURE ROBOTIQUE HYDRIDE

MÉMOIRE

PRÉSENTÉ

COMME EXIGENCE PARTIELLE

DE LA MAÎTRISE EN INFORMATIQUE

PAR

GUILLAUME ONFROY

MARS 2015

REMERCIEMENTS

Ce mémoire n'aurait pu être écrit sans l'aide de nombreuses personnes qui m'ont assisté dans la réalisation des différents travaux autour de ma maîtrise. C'est à ces personnes en particulier que je souhaite adresser mes remerciements.

Je tiens tout particulièrement à remercier Éric Beaudry, professeur au Département d'informatique de l'UQAM et mon directeur de recherche, pour son savoir, son aide et sa patience extraordinaire durant ces deux années.

Je remercie également Mathieu, Sylvain et Nicolas pour leurs conseils avisés et leur présence à mes côtés durant les cours et les heures passées au laboratoire, ainsi que François Ferland du laboratoire IntRoLab de l'Université de Sherbrooke, pour son aide précieuse dans le développement de mon projet.

Enfin, je ne remercierai jamais assez mes parents de m'avoir offert la chance d'accomplir tous les projets que j'ai souhaité entreprendre et de m'avoir encouragé et soutenu jusqu'au bout dans leur réalisation.

TABLE DES MATIÈRES

LISTE DES FIGURES

LISTE DES LISTINGS

LISTE DES ABRÉVIATIONS

AI Artificial Intelligence

AJS Actuplan Java Server

CoMDP Concurrent Markov Decision Process

HBBA Hybrid Behaviour Based Architecture

HRI Human-Robot Interactions

IA Intelligence Artificielle

ICAPS International Conference on Automated Planning and Scheduling

MDP Markov Decision Process

PDDL Planning Domain Definition Language

ROS Robot Operating System

STRIPS STanford Research Institute University Problem Solver

RÉSUMÉ

Le domaine de la robotique mobile est en constante recherche d'autonomie, c'est-à-dire qu'on cherche à concevoir les robots les plus autonomes possible et capables de prendre des décisions par eux-mêmes afin d'accomplir leurs missions avec le minimum d'intervention humaine. Le domaine de la planification en intelligence artificielle apporte des solutions en matière d'autonomie.

Dans ce mémoire, notre intérêt est de doter les robots de capacités de raisonnement sous incertitude temporelle, c'est-à-dire que nous voulons les rendre capables de choisir des actions même si elles présentent des incertitudes en termes de durée. Le projet présenté dans ce mémoire consiste à améliorer le planificateur ActuPlan pour qu'il puisse être intégré à une architecture robotique hybride, c'est-à-dire combinant les paradigmes d'architecture délibérative et à base de comportements. ActuPlan est un planificateur de tâches qui présente une approche novatrice à la résolution de problèmes de planification avec actions concurrentes et incertitude sur le temps. À l'origine, ActuPlan était un planificateur expérimental de type *offline* (hors-ligne), c'est-à-dire que la génération de plan était réalisée à partir d'un état initial, sans permettre son évolution au sein du système. Dans ce mémoire, nous présentons comment nous avons adapté le planificateur ActuPlan pour le rendre *online* (en ligne). Par la suite, nous présentons comment ce dernier a été intégré à une architecture robotique hybride. Nos expérimentations, sur une plateforme robotique réelle et dans un environnement simulé, démontrent le fonctionnement avancé du planificateur.

MOTS-CLÉS : intelligence artificielle, planification, raisonnement sous incertitude, robotique mobile, architectures robotiques.

INTRODUCTION

La création de machines autonomes, douées d'une intelligence proche de celle de l'homme et pouvant interagir avec leur environnement est un problème qui fascine le monde et occupe les chercheurs depuis fort longtemps. Encouragés par les évolutions technologiques et les fictions décrites dans les romans et les films, les balbutiements de l'intelligence artificielle (IA) ont entrouvert les possibilités d'évolution et de création de machines qui pourraient un jour égaler, voir même surpasser, les capacités cognitives, intellectuelles et physiques de l'homme.

Afin de répondre à ce désir de concevoir de telles entités, la recherche en IA, une branche de l'informatique, a grandement permis, depuis plusieurs dizaines d'années, le développement d'approches, d'algorithmes, de techniques et plus largement, de technologies nous rapprochant constamment d'une machine dite intelligente.

L'IA est un vaste domaine de recherche, valide pour de nombreux domaines d'application. Chacun de ces domaines requiert des connaissances en IA pouvant être très différentes et proposer des approches issues de travaux de recherche, qui peuvent également différer grandement les unes des autres. Le domaine d'application visé dans ce mémoire est la robotique mobile. En robotique, on cherche à rendre les robots le plus autonomes possible, afin qu'ils puissent accomplir différentes tâches avec le moins d'intervention humaine possible.

La planification, un sous-domaine de l'IA, apporte des solutions aux besoins en matière d'autonomie. Des planificateurs automatiques permettent de générer des plans qu'un robot peut ensuite exécuter. Cependant, la planification est d'une

grande complexité algorithmique. Même les tâches robotiques les plus simples peuvent rapidement s'avérer difficiles à planifier, tant les informations pour les encoder et les traiter peuvent être complexes et nombreuses. De par cette grande complexité, les planificateurs sont encore aujourd'hui limités à certaines classes de problèmes nécessitant de nombreuses hypothèses simplificatrices. Les travaux de recherche en planification en IA visent à repousser plus loin ces limites.

Le domaine de la robotique est riche en défis en matière de planification. Cela s'explique par le fait que les robots mobiles évoluent dans des environnements réels. Ces derniers étant continus, dynamiques, incertains, etc., la génération de plans est une problématique difficile. Dans ce mémoire, notre objectif est de permettre à des robots de prendre des décisions sous incertitude temporelle. Cela est motivé par le fait que la durée des actions d'un robot peut être incertaine.

Dans le domaine de la planification en IA, plusieurs planificateurs ont été proposés pour générer des plans sous incertitude temporelle. Il y a notamment le planificateur ActuPlan (Beaudry *et al.*, 2010a), qui permet de gérer des actions concurrentes (simultanées) et avec incertitude sur la durée des actions. Cependant, ActuPlan est un planificateur de type *offline* (hors-ligne). Les planificateurs de ce type sont généralement des planificateurs expérimentaux développés à titre de preuve de concept. Ces derniers se limitent à générer des plans pour un état initial connu et ne gèrent pas le volet exécution des plans. Ainsi, un planificateur *offline* ne peut généralement pas être directement exploité dans des systèmes réels.

L'objectif principal du projet réalisé dans le cadre de ce mémoire a été d'améliorer le planificateur ActuPlan afin de le rendre *online* (en ligne).Un planificateur modifié de cette façon est capable de gérer l'aspect dynamique d'un environnement réel, en combinant la génération et l'exécution des plans. Un objectif secondaire de ce projet a été d'intégrer le planificateur ActuPlan dans une architecture robotique,

afin d'évaluer ses performances sur une plateforme robot réelle.

Le présent mémoire est organisé comme suit. Les deux premiers chapitres introduisent les éléments de base nécessaires pour comprendre les travaux présentés aux chapitres suivants. Le premier chapitre introduit des notions de base en IA, en planification et résume le fonctionnement du planificateur ActuPlan. Le chapitre 2 introduit des notions de base en robotique et présente les architectures classiques utilisées dans ce domaine. Une introduction à l'environnement ROS (*Robotics Operating Systems*), un environnement orienté services spécialisé pour la robotique, est également présentée. Le chapitre 3 présente l'intégration du planificateur ActuPlan dans l'architecture HBBA, architecture robotique hybride, développée au laboratoire de robotique IntRoLab de l'Université de Sherbrooke. Le chapitre 4 présente les expérimentations réalisées et discute des résultats obtenus. Une conclusion vient clore le mémoire.

CHAPITRE I

NOTIONS EN PLANIFICATION EN INTELLIGENCE ARTIFICIELLE

L'intelligence artificielle (IA) est une branche de l'informatique qui vise à doter une machine (un système informatique) de capacités normalement associées à l'intelligence humaine. Cela inclut notamment des capacités d'interaction, de raisonnement, de résolution de problèmes et d'apprentissage. Un des objectifs fondamentaux en IA est de créer des entités (informatiques) capables d'être autonomes. L'objectif à long terme est de proposer des systèmes dont les capacités en termes d'intelligence se rapprochent ou même dépassent celles de l'homme, capacités leur permettant d'aborder, comprendre et résoudre des problèmes complexes de façon autonome et sans intervention humaine.

Dans le domaine de l'IA, il existe fondamentalement deux façons d'aborder le problème de création d'entités dites intelligentes : l'IA faible et l'IA forte. L'IA faible (*weak AI*) est une vision très pragmatique, qui vise à simplement créer des systèmes permettant d'apporter des solutions à des problèmes concrets que l'homme cherche à résoudre (ex. : jouer aux échecs, conduire une voiture, entretenir une conversation, etc.). L'IA forte (*strong AI*) a un objectif plus ambitieux, c'est-à-dire concevoir une entité disposant d'une «vraie» intelligence sans que celle-ci ne soit explicitement programmée, comme chez l'humain.

L'IA comprend de nombreuses sous-disciplines, chacune visant des capacités différentes de l'intelligence (ex. : raisonnement, résolution de problèmes, reconnaissance de formes, apprentissage, etc.). La sous-discipline visée dans le présent mémoire est la **planification**. Puisqu'il s'agit d'une approche pragmatique, nécessitant des modèles explicites, la planification appartient davantage au courant de l'IA faible. La planification en IA vise à concevoir des algorithmes de génération de **plans** permettant de résoudre automatiquement des problèmes donnés. Dans ce chapitre, nous introduisons certaines notions en IA et en planification nécessaires à la compréhension des travaux présentés dans ce mémoire.

1.1 Agents intelligents

En intelligence artificielle, on appelle une entité intelligente un **agent intelligent**, ou tout simplement un **agent** (Russell et Norvig, 2010). L'exemple typique d'un agent est un robot, tel que Curiosity présenté à la figure 1.1, c'est-à-dire une entité évoluant de manière autonome (ou partiellement autonome) au sein d'un environnement.La figure 1.2 présente l'architecture typique d'un agent. Un agent peut percevoir son environnement grâce à des **capteurs** et agir sur cet environnement à travers des **effecteurs**. Là où un humain a des yeux et des oreilles qui captent des informations sur son environnement, un robot possède des caméras et des lasers. De la même manière, l'humain peut interagir avec son environnement à l'aide de ses bras, ses mains ou ses jambes, tandis qu'un robot peut bénéficier de moteurs, de roues, de bras articulés et de pinces. Un agent n'est cependant pas nécessairement un robot mécanique, à savoir un objet évoluant dans le monde réel : il s'agit d'un concept définissant une entité évoluant dans un environnement, que ce soit le monde réel ou un univers numérique. À ce titre, un agent intelligent peut aussi bien être un robot qu'un personnage de jeu vidéo, ou encore une entité informatique qui réalise un travail invisible aux yeux de l'utilisateur.

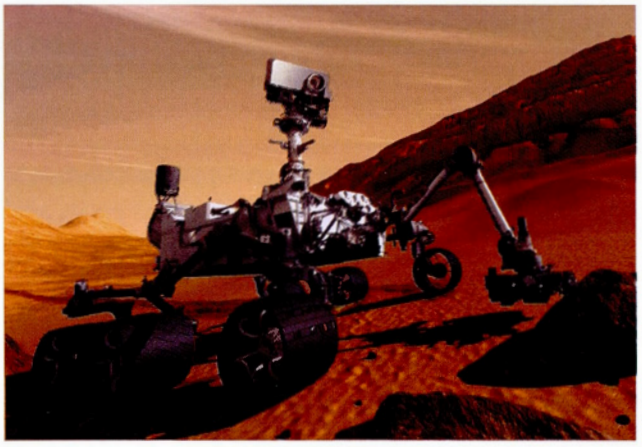

Figure 1.1 Robot Curiosity (Source : NASA)

L'intelligence d'un agent peut être vue, en IA faible, comme une fonction $f : P^* \rightarrow A$, où P est un ensemble de percepts et A un ensemble d'actions. À intervalle régulier, cette fonction retourne l'action à exécuter en fonction d'un historique de percepts (P^*). Le défi fondamental de l'IA consiste à implémenter une telle fonction f. L'algorithme 1 présente un exemple de squelette d'implémentation.

Algorithme 1 Squelette d'un agent

```
1: function AGENT(percept) return action
2:     memoire ← MAJMEMOIRE(memoire, percept)
3:     action ← MEILLEUREACTION(memoire)
4:     memoire ← MAJMEMOIRE(memoire, action)
5:     return action
```

Créer une machine douée d'intelligence est un objectif ambitieux et des approches très différentes peuvent être utilisées selon le domaine d'application pour lequel on souhaite développer un tel système. Par exemple, créer une intelligence artificielle pour rendre autonome un robot explorant la surface d'une lointaine planète est un procédé très différent de celui de création d'un assistant vocal pouvant entretenir une conversation avec un interlocuteur.

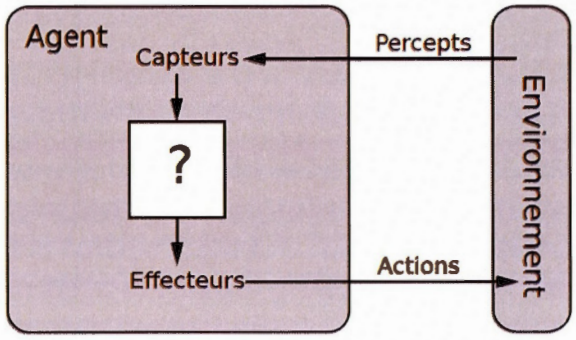

Figure 1.2 Agent interagissant avec son environnement au moyen de capteurs et d'effecteurs (Source : traduit de Russell et Norvig, 2010)

Dans le domaine de la robotique, on cherche la plupart du temps à créer des systèmes améliorant l'autonomie des robots, c'est-à-dire à développer des programmes qui, une fois intégrés aux robots, permettent à ces derniers d'interagir avec leur environnement avec le minimum d'intervention possible de l'homme pour les aiguiller dans leurs tâches. On souhaite, par exemple, qu'un robot puisse trouver seul un chemin pour se rendre à une destination (planification de chemin), actionner ses moteurs pour se déplacer (planification de mouvement) ou encore qu'il puisse choisir et effectuer une séquence d'actions afin d'accomplir un but (planification de tâches).

Ces différentes situations s'apparentent à des problèmes pouvant être résolus de manière efficace par des algorithmes issus d'une branche de l'intelligence artificielle appelée **planification**.

1.2 Planification de tâches

Le domaine de la planification a pour but de concevoir des algorithmes permettant de générer des plans afin de résoudre un problème donné. Les programmes qui incorporent ces algorithmes produisant des plans sont appelés des **planificateurs**.

Afin de parvenir à produire un plan, les algorithmes développés doivent disposer d'un certain nombre d'informations à partir desquelles il sera possible de définir une suite d'actions à entreprendre conduisant au but recherché. Afin de prédire plus efficacement les différentes actions à sélectionner, une approche naturelle consiste à aborder chaque problème pour lequel on souhaite produire un plan avec les spécificités techniques de celui-ci. On parle alors de planificateurs **dépendants du domaine** (Ghallab *et al.*, 2004).

Ce type d'approche de planification peut présenter certains atouts, notamment en termes de performances computationnelles ou de précision et pertinence des actions sélectionnées. Cependant, cette dépendance liée au domaine rend ces planificateurs trop spécifiques et donc trop peu malléables et réutilisables. Pour ces raisons, la recherche dans le domaine de la planification est principalement orientée vers la conception d'algorithmes **indépendants du domaine** (Ghallab *et al.*, 2004).

Planifier consiste à sélectionner et à ordonnancer des actions afin de changer l'état d'un système (Ghallab *et al.*, 2004).La figure 1.3 présente ce modèle de planification qui associe le planificateur à un contrôleur, lui-même évoluant au sein d'un système donné. Les planificateurs indépendants du domaine raisonnent à partir de spécifications abstraites qui constituent le **modèle du monde**.Un problème de planification est défini par trois entrées Σ, s_0 et g, où Σ est une **description du système**, à savoir une description de l'environnement et des différentes actions exécutables, s_0 un **état initial** représentant l'environnement dans l'état actuel du système et g une description du ou des **buts** à atteindre. À partir de ces trois entrées, il est possible, si une solution existe, de générer un plan qui réponde au problème donné. La forme la plus simple d'un plan est une séquence d'actions qui, une fois effectuées, conduisent à un état qui satisfait le but initialement désiré.

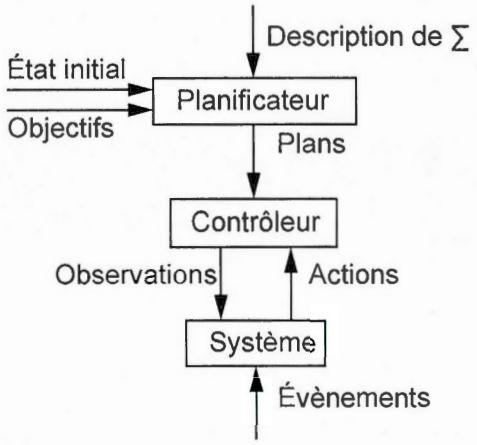

Figure 1.3 Modèle conceptuel d'un système de planification (Source : Ghallab *et al.*, 2004)

Typiquement, un planificateur indépendant du domaine fonctionne à partir de données spécifiées en entrée, à l'aide d'un langage de description représentant le système selon un ensemble de propositions logiques de premier ordre. L'un des premiers planificateurs à avoir été conçu est STRIPS (*Standford Research Institue Problem Solver*) (Fikes et Nilsson, 1972), qui propose son propre langage de définition de problème éponyme. Le listing 1.1 présente un exemple de problème décrit à l'aide de STRIPS.Inspiré de STRIPS, le langage PDDL (*Planing Domain Definition Language*) (McDermott *et al.*, 1998), conçu pour les compétitions aux conférences ICAPS (*International Conference on Automated Planning and Scheduling*), est devenu l'un des principaux langages utilisés pour la description de problèmes de planification.

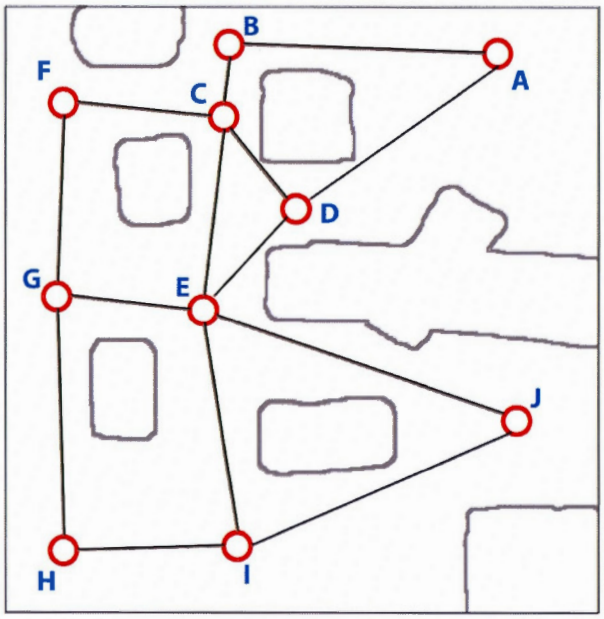

Figure 1.4 Exemple de monde exploité par un problème de planification

```
Actions:
        Goto(Origin, Destination)
        Preconditions:  At(Origin)
        Postconditions: not At(Origin), At(Destination)

        Load(Packet, Location)
        Preconditions:  At(Location), PacketAt(Location)
        Postconditions: not PacketAt(Location), PacketLoaded

        Unload(Packet, Location)
        Preconditions:  At(Location), PacketLoaded
        Postconditions: not PacketLoaded, PacketAt(Location)

Initial state:
        Connected(A,B), Connected(B,C),
        Connected(C,D), Connected(D,A),
        ...
        At(A), PacketAt(G)

Goal:
        PacketAt(J)
```

Listing 1.1 Exemple de problème STRIPS. L'état initial correspond au monde présenté à la figure 1.4.

1.2.1 Hypothèses classiques

En planification, on souhaite créer des planificateurs indépendants du domaine, capables de résoudre des problèmes complexes de la vie réelle. Il suffirait alors de présenter au planificateur une description du système et il serait capable de produire un plan répondant au problème. En pratique, cela n'est pas réalisable, car le monde réel est trop complexe. Il est très difficile de créer un programme qui soit capable de produire à la fois un plan pour un livreur de pizza, pour faire fonctionner Curiosity sur Mars ou encore de résoudre un Rubix-Cube, simplement en lui spécifiant le système du problème. Afin d'être capable de générer des plans dans des délais raisonnables, on doit émettre un certain nombre d'**hypothèses** afin de restreindre le domaine d'application et ainsi **simplifier** le problème. Dans le domaine de l'IA et de la planification, on peut ainsi identifier huit (8) hypothèses qui définissent un **modèle restrictif** de planification (Ghallab *et al.*, 2004).Ces huit hypothèses sont :

- **H0 : Système fini :** il existe un nombre fini d'états et d'actions ;
- **H1 : Observabilité totale :** l'ensemble du système est observable en tout temps ;
- **H2 : Déterministe :** chaque action produit un seul résultat connu à l'avance ;
- **H3 : Statique :** seules les actions de l'agent influent sur le système ;
- **H4 : Buts atteignables :** il existe un plan qui est une solution au problème ;
- **H5 : Plans séquentiels :** un plan est une suite linéaire et ordonnée d'actions ;
- **H6 : Temps implicite :** on ne prend pas en compte de durées dans le temps. Les actions et leurs effets sont instantanés. La durée de chaque action est considérée unitaire ;

— **H7 : Planification *offline*** : pendant la construction du plan, le planificateur ignore les évolutions de l'environnement entre chaque action.

La planification dite **classique** est basée sur un modèle restrictif, tel que défini par les hypothèses ci-dessus. On parle alors de **génération *offline* de plans composés de séquences d'actions au sein d'un système déterministe, statique et fini, avec une connaissance complète de ce dernier, des buts atteignables et une gestion du temps implicite.**

1.3 Approches de planification

Pour résoudre un problème de planification, l'homme peut intuitivement raisonner de la manière suivante : modéliser la situation (état) actuelle, énumérer les options possibles puis évaluer les conséquences de chacune d'entre elles, pour enfin retenir la meilleure option permettant d'atteindre le but recherché. Les planificateurs intègrent une approche similaire pour résoudre les problèmes qui leur sont spécifiés. Une **résolution automatique de problème** revient donc à effectuer une recherche parmi toutes les possibilités offertes par le système.

1.3.1 Recherche dans un espace d'états

Une approche simple pour résoudre un problème de planification est de présenter cet espace de possibilités sous la forme d'un graphe d'états, appelé **espace d'états**. Un espace d'états représente l'ensemble des états possibles du système, où un nœud représente un état et un arc l'action à effectuer pour passer d'un état à un autre. Trouver une solution revient donc à trouver dans le graphe (l'espace d'états) un chemin reliant l'état initial à un état satisfaisant le but. Le plan correspond alors à la succession d'arcs empruntés (ou d'actions effectuées) par ce chemin. La figure 1.5 présente un exemple de recherche dans un espace d'états.

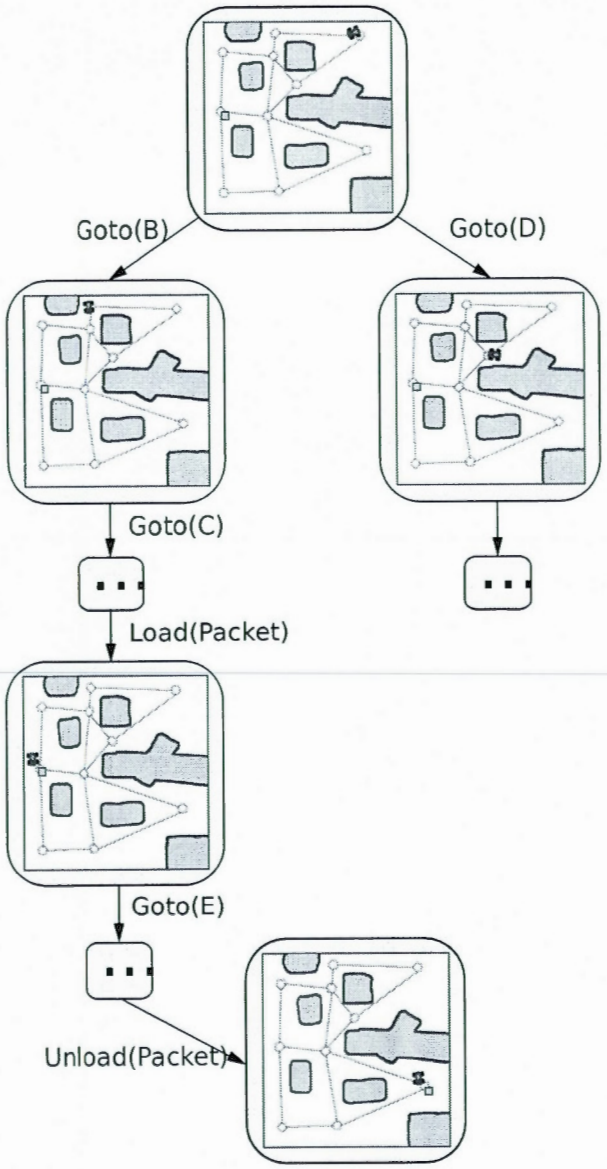

Figure 1.5 Exemple de recherche dans un espace d'états

Il existe plusieurs stratégies pour parcourir un espace d'états et donc trouver un plan. La plus simple d'entre elles est le **chaînage avant** : le point de départ est le nœud modélisant l'état initial du système. On explore ensuite le graphe en parcourant les nœuds jusqu'à arriver à un nœud satisfaisant un état-but. De la même manière, il est possible d'effectuer une recherche selon un chaînage arrière (le point de départ est alors le but) ou encore de combiner les deux approches simultanément.

La contrainte principale avec ces différents algorithmes est de trouver comment améliorer leurs performances en limitant l'espace de recherche. En effet, en considérant le nombre d'actions pouvant être effectuées, à chaque état, appelé **facteur de branchement**, la taille de l'espace d'état est potentiellement exponentielle au fur et à mesure que la recherche progresse. Cependant, la plupart des branches à explorer ne seront en définitive pas utiles pour la recherche de plans, car elles ne permettent pas toujours d'arriver au but recherché. La recherche a permis de développer des techniques de planification plus avancées permettant d'ignorer des parties de l'espace d'états afin de converger plus rapidement vers le but.

Une technique classique permettant d'orienter la recherche dans un graphe d'états est l'utilisation d'**heuristiques**, mécanique introduite avec l'algorithme $A*$ (Hart *et al.*, 1968). L'algorithme A* peut être vu comme une extension de l'algorithme Dijkstra (Dijkstra, 1971). Dans A*, une heuristique est une évaluation du coût restant à partir d'un état vers un état satisfaisant le but. Lorsque l'heuristique est **admissible**, c'est-à-dire qu'elle ne surestime jamais le coût restant, l'algorithme A* trouvera la solution optimale lorsqu'elle existe. Plus une heuristique est informative, c'est-à-dire mieux elle estime le coût restant, plus elle permet de réduire le nombre d'états (nœuds) à examiner au sein du graphe. Un défi important en planification est l'élaboration d'heuristiques indépendantes du domaine. Certaines de ces heuristiques, comme *relaxed graphplan*, ont été proposées (Bonet *et al.*,

1997; Hoffmann, 2001).

Les techniques de planification avancées présentées et utilisées dans le reste de ce mémoire s'appuient sur une recherche selon un chaînage avant dans un espace d'états.

1.3.2 Recherche dans un espace de plans

Une autre approche connue de résolution de problèmes de planification est la recherche dans un espace de plans (Sacerdoti, 1974). Dans ce cadre, l'espace de recherche n'est plus défini par des nœuds représentant des états de Σ. Les nœuds du graphe représentent des **plans partiellement ordonnés**. Un plan partiellement ordonné est un ensemble d'actions présentant des contraintes d'ordonnancement. Dans un espace de plans, les arcs sont des opérations de raffinement de plan. La recherche débute au nœud modélisant un plan vide. On cherche alors à aboutir à un nœud représentant un plan sans défaut, c'est-à-dire qui permet de satisfaire les buts recherchés.

1.4 Au-delà de la planification classique

La recherche actuelle dans le domaine de la planification, même si elle garde comme base théorique la planification classique, ne s'y arrête pas. Les hypothèses utilisées par la planification classique (voir la section 1.2) s'avèrent très rapidement trop contraignantes pour de nombreuses applications pratiques pour lesquelles on souhaite mettre en place un planificateur. Prenons l'exemple d'un robot-taxi évoluant dans une ville parmi d'autres automobilistes et ayant pour mission d'embarquer et de déposer des usagers à divers endroits. L'hypothèse H1 impose une observabilité totale du système et donc de l'environnement, ce qui n'est pas réaliste dans le cas du robot-taxi, qui ne dispose, comme informations

sur l'environnement, que de celles transmises par ses capteurs. Comme l'agent n'a pas une connaissance ni une vue de l'ensemble du réseau routier de la planète, il s'agit donc d'une observabilité partielle.L'hypothèse H2 impose quant à elle un système déterministe. Dans ce cas, on ne connaît cependant pas avec certitude le résultat d'une action entreprise par le robot-taxi. Par exemple, lorsqu'il freine, il n'est pas possible de prévoir la distance de freinage avec exactitude (il peut y avoir du verglas). Il y a donc de l'incertitude. Il en va de même pour chaque hypothèse du modèle de planification classique, pour lequel il est simple de trouver un exemple démontrant qu'il est trop contraignant. Beaucoup de travaux de recherche en planification cherchent à développer des approches qui relaxent une ou plusieurs hypothèses à la fois et qui permettent ainsi de concevoir des planificateurs adaptés à un contexte précis.

Dans le cadre des travaux présentés dans ce mémoire, notre recherche s'axe autour de la **planification sous incertitude et avec actions concurrentes**. Ce sont alors les hypothèses H2 et H5 qui sont relaxées. Cette classe de planification est motivée par la robotique (Wooldridge et Jennings, 1995) et la planification des robots déployés sur la planète Mars (Bresina *et al.*, 2002).

1.5 Planification sous incertitude

Pour relaxer l'hypothèse H2 (système déterministe), une approche classique consiste à modéliser le système comme **processus décisionnels de Markov** (MDP). Un MDP est un modèle stochastique exploité à des fins de prise de décision lorsque l'on se situe dans un état connu, mais où l'effet d'une action est incertain. Un MDP est donc un quadruplet (S, A, P, R) avec S l'ensemble des états possibles du système, A l'ensemble des actions exécutables, $P(s, a, s')$ la probabilité que l'action a exécutée au sein de l'état s conduise dans l'état s', et $R_a(s, s')$, la ré-

compense reçue après transition de l'état s à l'état s'. La figure 1.6 présente un exemple de MDP. Les nombres associés aux flèches représentent les probabilités de transition et les carrés, les récompenses associées aux états.

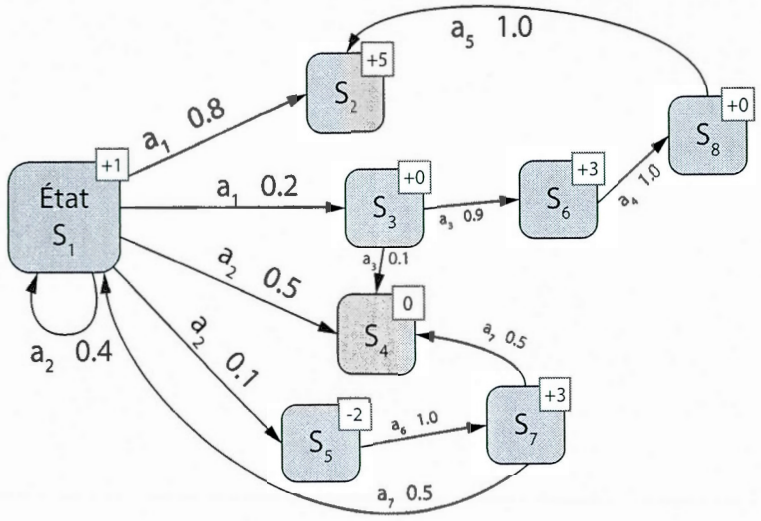

Figure 1.6 Exemple de processus décisionnel de Markov (MDP)

Une solution à un MDP est appelée une **politique** décisionnelle (*policy*). Une politique est une fonction qui associe une action à exécuter dans chaque état. Évidemment, on s'intéresse à trouver la politique optimale, soit celle qui va maximiser le gain espéré d'utilité. Il existe plusieurs algorithmes pour calculer une politique optimale. La méthode de résolution traditionnelle s'appuie sur la technique d'**itération de valeurs** (*value iteration*), basée sur l'équation de Bellman (Puterman, 2009) qui consiste à faire converger la **valeur** de chaque état du système (*quelle est l'utilité espérée de chaque état pour la réalisation de la tâche dans son ensemble ?*). Cette approche de résolution des MDP avec actions concurrentes peut être combinée à des techniques de programmation dynamique comme RTDP (Real Time Dynamic Programming) (Barto *et al.*, 1995) ou L-RTDP (Labeled RTDP) (Bonet et Geffner, 2003). Il existe également l'approche d'**itération de politiques** (*policy iteration*), dont le but est d'associer une action à chaque état

du système.

1.6 Planification sous incertitude et avec concurrence d'actions

1.6.1 MDP Concurrent

Plusieurs travaux de recherche ont été réalisés afin de proposer des solutions à la planification sous incertitude et avec actions concurrentes. Plusieurs approches sont basées sur les MDP, dont les **processus décisionnels de Markov concurrents** (CoMDP) (Mausam et Weld, 2004; Mausam et Weld, 2005; Mausam et Weld, 2006). Conceptuellement, les CoMDP étendent simplement le concept des MDP en considérant non plus un ensemble d'actions A, mais tous les sous-ensembles d'actions possibles (2^A). Les actions contenues dans A doivent alors simplement souscrire au principe d'exclusion mutuelle (Blum et Furst, 1997) : deux actions ne peuvent être exécutées en parallèle s'il existe un conflit entre leurs préconditions, s'il existe un conflit entre leurs effets ou encore si les préconditions de l'une entrent en conflit avec les potentiels effets d'une autre.Enfin, il est nécessaire de mettre à jour le modèle du coût (temps et ressources utilisées) pour un ensemble d'actions. Le temps correspond alors au temps d'exécution de la plus longue des actions et les ressources consommées équivalent à la somme des ressources de chaque action individuelle. Un CoMDP peut être résolu avec les mêmes algorithmes que pour les MDP.

Ces différentes techniques et leurs évolutions sont néanmoins souvent confrontées à un problème de croissance exponentielle de l'espace d'états, qui limite rapidement la taille du problème pouvant être résolu. En effet, ces approches utilisent souvent un modèle basé sur une discrétisation du temps et des ressources conduisant ainsi à une explosion de l'espace d'états. Afin de répondre à ce problème, une nouvelle approche a été proposée. Cette dernière se base sur un modèle continu plutôt que

sur un modèle discret. Dans le reste de ce chapitre, nous présentons le planificateur ActuPlan, qui intègre cette approche.

1.6.2 ActuPlan

Afin de gérer l'incertitude sur le temps, les approches présentées jusqu'ici apposent un horodatage (*timestamp*) dans chaque état. Par conséquent, une action dont l'effet est incertain dans le temps sera modélisée selon autant de points dans le temps que les effets peuvent survenir, conduisant ainsi à l'explosion de l'espace d'états.

L'approche utilisée au sein d'ActuPlan (Beaudry *et al.*, 2010a; Beaudry *et al.*, 2012) exploite un modèle de variables aléatoires continues pour représenter l'incertitude sur le temps. Un simple horodatage n'est donc plus utilisé ici pour représenter l'incertitude sur le temps. Chaque état est représenté par un ensemble de **variables d'états** dont chacune présente une caractéristique du système au sein du dit état, à laquelle est associée une valeur. Deux types de variables aléatoires sont utilisées : les **événements temporels** (T), qui décrivent l'occurrence d'un événement (ex. : le début ou la fin d'une action) et les **durées des actions** (D). Chaque action $a \in A$ a une durée $d_a \in D$ représentée par une variable aléatoire. Il existe pour chaque variable aléatoire $t \in T$ une équation qui définit le temps auquel t correspond, dépendamment d'autres variables aléatoires. Par exemple, l'action a commençant au temps t_0 et terminant au temps t_1, t_1 est définie par l'équation $t_1 = t_0 + d_a$. L'ensemble des variables aléatoires définissant les durées des actions est défini par $D = \{d_a | a \in A\}$. La fonction $PDF_{d_a}(u) : \mathbb{R}^+ \rightarrow \mathbb{R}^+$ définit la densité de probabilité que l'action a ait une durée de u unités de temps. Cette approche peut également être généralisée à l'incertitude sur les ressources (Beaudry *et al.*, 2010b).

L'algorithme de planification intégré au sein d'ActuPlan s'appuie sur une recherche à chaînage avant de l'espace d'états, dont les nœuds correspondent de manière traditionnelle aux différents états du système et les arêtes aux actions. En parallèle du graphe de l'espace d'états, l'algorithme génère et maintient un réseau bayésien modélisant les relations de dépendance entre les variables aléatoires définissant les dates des événements et les durées des actions. Le réseau bayésien se présente sous la forme d'un graphe orienté acyclique $B = (N, E)$ où $N = T \cup U$ est un ensemble de variables aléatoires, avec U désignant la fonction retournant la valeur assignée à chaque variable d'état, et E désignant l'ensemble des arêtes modélisant les dépendances entre les variables aléatoires. La figure 1.7 présente un exemple de construction de réseau bayésien permettant de calculer les variables aléatoires correspondant aux variables d'états.

ActuPlan propose également son propre langage de description de domaines et de problèmes, qui partage des propriétés fondamentales avec le langage PDDL (Planning Domain Definition Language), afin de supporter la définition de prédicats et de variables d'états. Le chapitre 4 présente des exemples de définition de problèmes à l'aide de ce langage.

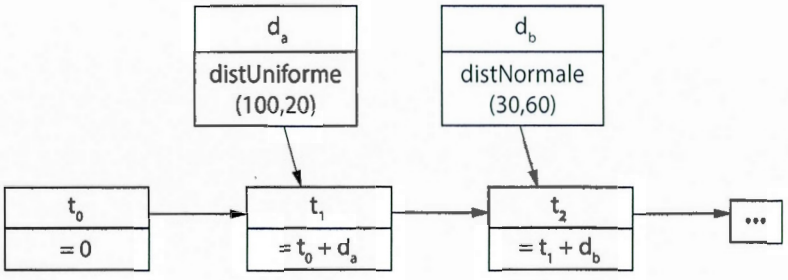

Figure 1.7 Exemple de réseau bayésien construit par ActuPlan

CHAPITRE II

ARCHITECTURES ET ENVIRONNEMENTS EN ROBOTIQUE MOBILE

Le but ultime dans le domaine de la recherche en conception d'architecture robotique et d'interaction homme-robot (HRI) est de concevoir des robots capables de se comporter et d'interagir de manière similaire à celle des hommes. Malgré une intégration de plus en plus importante de robots, notamment au sein de l'industrie, cela reste une tâche relativement complexe de permettre à une machine de réaliser des tâches habituellement réservées à l'homme, telles que conduire une voiture, manipuler des objets, faire les courses, etc. Un certain nombre de paradigmes, d'approches et de technologies ont été proposés et développés afin d'apporter une solution à ce problème. Dans ce chapitre, nous présentons certains de ces grands paradigmes de conception architecturale utiles pour la compréhension de l'architecture HBBA introduite à la section 2.3. Dans un deuxième temps est présenté le framework **ROS**, un écosystème aujourd'hui privilégié pour le développement de systèmes robotiques et sur lequel est basé HBBA.

2.1 Familles d'architectures robotiques

Il existe trois principaux paradigmes de conception architecturale en robotique : l'**architecture délibérative**, l'**architecture réactive** (Arkin, 1998) et enfin

l'**architecture hybride**. Dans les sections suivantes, l'**architecture comportementale**, pouvant être considérée comme une architecture réactive, est aussi présentée.

2.1.1 Architecture délibérative

L'architecture délibérative (figure 2.1) est une architecture de type *top-down* (du haut vers le bas), c'est-à-dire que l'intelligence au sein de l'agent réside dans un ou plusieurs modules de haut niveau, qui prennent des décisions afin de contrôler des modules de plus bas niveau au sein de l'architecture. La manière dont fonctionne une architecture délibérative s'apparente à l'approche de résolution d'un problème de planification présentée dan1s la section 1.2 :

1. L'agent reçoit, à travers ses capteurs, des informations lui permettant de se faire une représentation du monde (son environnement) ;
2. Il possède un but qu'il souhaite accomplir ;
3. Un module de planification de haut niveau produit un plan permettant, s'il existe une solution, de répondre au but recherché ;
4. Des modules de plus bas niveau au sein de l'agent sont activés afin d'exécuter les actions définies dans le plan.

Dans cette logique, un module activé par un module de plus haut niveau peut à son tour mettre en place une logique de type *top-down*, en planifiant puis en activant des modules de plus bas niveau encore. Une architecture délibérative correspond donc à un contrôle hiérarchique.

Ce type d'architecture a pour force de permettre au robot d'être plus omniscient par rapport à son environnement et aux tâches à accomplir pour atteindre le but recherché. Il a cependant pour lacune la nécessité d'encoder des informations relatives au domaine dans lequel il évolue.

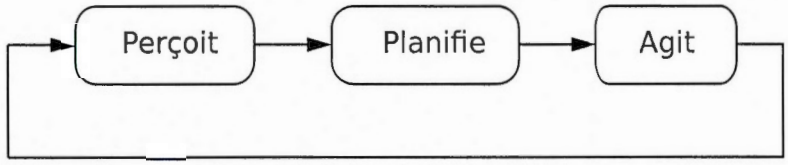

Figure 2.1 Paradigme d'architecture délibérative

2.1.2 Architecture réactive

L'architecture réactive (figure 2.2) est une architecture de type *bottom-up* (du bas vers le haut). Dans ce type d'architecture, on ne cherche pas à développer un module dans lequel réside toute l'intelligence de l'agent. En effet, l'architecture réactive consiste en l'implémentation de plusieurs règles associant percepts et actions du robot. À chaque percept reçu par le robot à travers ses capteurs est associée une action activant les effecteurs du robot. C'est de par la combinaison de ces nombreuses règles d'association que peut émerger naturellement une intelligence de plus haut niveau.

Avec ce type d'architecture, il n'est pas nécessaire d'encoder des informations relatives au système dans lequel le robot évolue. Toutes les informations nécessaires sont extraites depuis les données des capteurs.

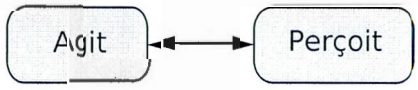

Figure 2.2 Paradigme d'architecture réactive

2.1.3 Architecture comportementale

L'architecture comportementale (Arkin, 1998; Matarić, 1998) est une architecture de type *bottom-up* (du bas vers le haut) et est également considérée comme une

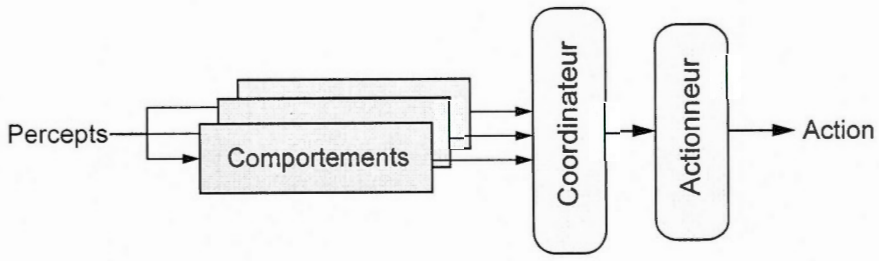

Figure 2.3 Paradigme d'architecture comportementale

architecture réactive. L'idée principale derrière ce type d'architecture est le développement de modules de bas niveau appelés **comportements**. Un comportement est une action ou un ensemble d'actions visant à accomplir un but précis. Au sein d'un robot mobile, on peut par exemple chercher à implémenter les comportements suivants : contourner un obstacle, suivre quelqu'un se déplaçant, identifier les visages autour de soi, etc. Chaque comportement peut prendre comme paramètres les informations reçues par les capteurs ou par d'autres comportements et ensuite actionner les effecteurs du robot, ou encore activer d'autres comportements. Autant que possible, les comportements au sein du robot interagissent entre eux à travers l'environnement et non à travers le système.

L'architecture comportementale a été développée en s'inspirant des observations faites en biologie, qui décrivent chez l'homme, l'animal ou encore l'insecte des comportements en réaction à des stimuli extérieurs.

En raison de sa nature très modulable et évolutive, l'architecture à base de comportement, avec l'architecture hybride, est aujourd'hui souvent privilégiée dans le cadre du développement de systèmes intégrés à un robot mobile.

2.1.4 Architecture hybride

Une architecture hybride (figure 2.4) combine les avantages des différents types d'architecture présentés plus haut, à savoir les performances et la robustesse de l'architecture réactive ou de l'architecture comportementale et l'aspect plus rationnel et omniscient que propose l'architecture délibérative.

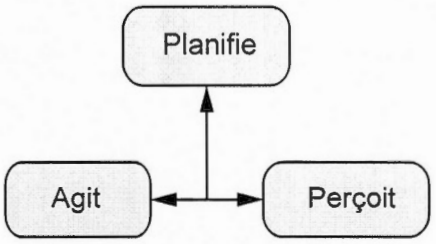

Figure 2.4 Paradigme d'architecture hybride

De manière simplifiée, les architectures réactives sont très efficaces pour répondre à un problème sur le court terme, tandis que les architectures délibératives sont plus efficaces pour résoudre les problèmes sur le long terme. Afin d'être efficace, une architecture hybride est généralement organisée sous la forme de couches (*layers*), avec une couche de bas niveau de type réactif, une couche de haut niveau de type délibératif et une couche intermédiaire permettant la communication entre les deux. On parle alors d'**architecture hybride à trois niveaux** (Peter Bonasso *et al.*, 1995) (figure 2.5).

2.2 *Motivated Behavioral Architecture* (MBA)

La recherche autour de la conception d'architectures hybrides cherche à apporter une solution au problème de la conception de robots nécessitant à la fois une grande réactivité, une souplesse de fonctionnement et la capacité de raisonner et de résoudre des problèmes en se basant sur des connaissances issues d'un niveau

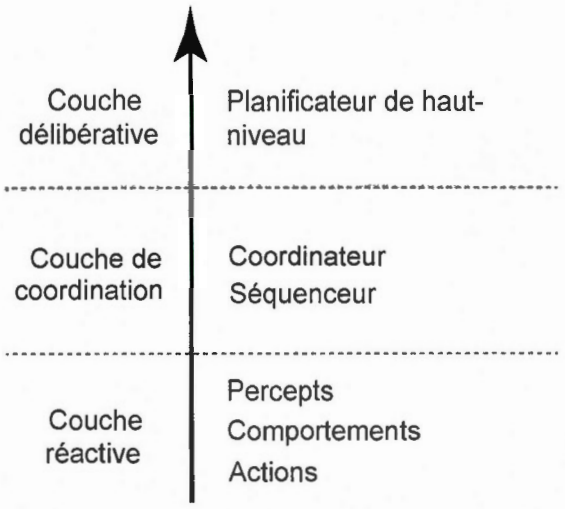

Figure 2.5 Paradigme d'architecture à trois niveaux

d'abstraction plus grand. Tel que présenté dans la section 2.1.4, ces approches sont basées sur une hiérarchisation du processus de planification et de réaction. Au sein de cette hiérarchie, la manière dont est décomposé un problème et est traitée une tâche (via la planification ou autre) est laissée à la liberté du développeur et dépend des capacités du planificateur. En conséquence, il a été proposé une alternative de conception architecturale basée sur le concept de **motivations**. Nous présentons dans cette section l'architecture MBA, dont l'ancêtre est l'architecture EMIB (Michaud, 2002). MBA est elle-même l'ancêtre de HBBA, cette dernière étant l'architecture utilisée dans le cadre des travaux de ce mémoire.

Le principe de MBA (Beaudry *et al.*, 2005) est de conserver une architecture basée sur les comportements, mais d'activer ces derniers dépendamment d'*émotions* émises par différentes sources au sein du système. Ces sources émettrices d'émotions sont multiples et peuvent prendre différentes formes : planificateurs, stimuli extérieurs, scénarios prédéterminés, etc. L'utilisation de sources diverses influençant mutuellement les comportements du robot a un impact sur l'efficacité de

celui-ci à réaliser les tâches qui lui sont confiées, contrairement au fait de ne dépendre que d'un seul module planificateur de plus haut niveau.

Au sein d'une architecture basée sur les comportements traditionnels, l'activation de comportements implémentés dans le robot est étroitement, voire directement liée aux données captées par les différents capteurs. MBA déporte quant à elle l'activation de ces comportements à un niveau d'abstraction supérieur, à l'aide de modules pouvant émettre des émotions. Cependant, certains comportements peuvent toujours être activés directement à la suite de percepts du robot, notamment pour répondre à des besoins plus primitifs, tels qu'un arrêt d'urgence, un dysfonctionnement ou un évitement d'obstacle.

MBA est représentée à la figure 2.6 et s'articule donc de la manière suivante :

- un ensemble de comportements qui activent les effecteurs du robot ;
- un ensemble de sources émettrices d'émotions ;
- un module d'arbitrage des émotions pour la sélection des comportements à activer.

L'arbitrage peut être réalisé par le biais de différentes implémentations. Il peut s'agir d'un système de tri prioritaire, d'un schéma moteur (*Motor schema*) (Arkin, 1989) ou autre.

2.3 *Hybrid Behaviour Based Architecture* (HBBA)

L'architecture HBBA (Ferland, 2011) présentée à la figure 2.7 est une évolution de l'architecture MBA présentée à la section 2.2. Elle conserve le besoin pour les robots d'être capables de prendre des décisions de haut niveau, tout en offrant la réactivité indispensable à leur évolution dans le monde réel. HBBA a été conçue dans le but de permettre à un robot disposant de multiples outils pour interagir et communiquer avec son environnement de devenir une entité indépendante, capable

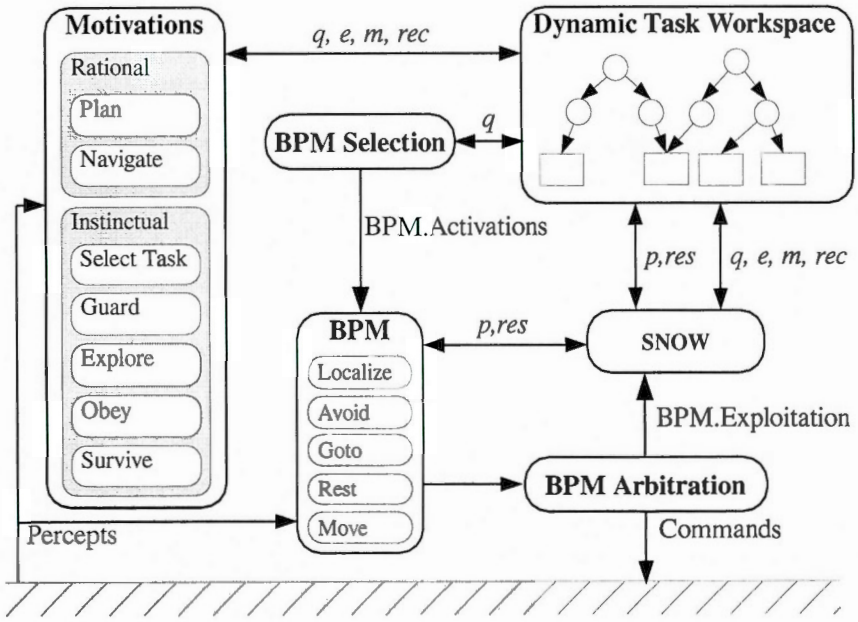

Figure 2.6 Motivated Behavioral Architecture (MBA) (Source : Beaudry *et al.*, 2005)

d'évoluer librement et de manière autonome dans un environnement entouré de plusieurs personnes. Le robot devra être capable, par exemple, de se servir d'une plateforme de télécommunication, d'apporter des objets à des personnes ou dans des bureaux, de surveiller une zone, de partir à la recherche d'informations et d'être capable d'effectuer un rapport sur le résultat de sa recherche. Le but est de proposer une architecture qui permette à une plateforme robotique complexe de gérer ses objectifs en fonction de ses capacités, en prenant en compte les contraintes et l'incertitude qui réside dans l'environnement.

HBBA est une architecture qui combine deux des paradigmes de conception architecturale présentés dans la section 2.1 : l'architecture hybride et l'architecture comportementale. Telles que présentées précédemment, les architectures hybrides sont généralement organisées sous la forme de couches (architecture à trois ni-

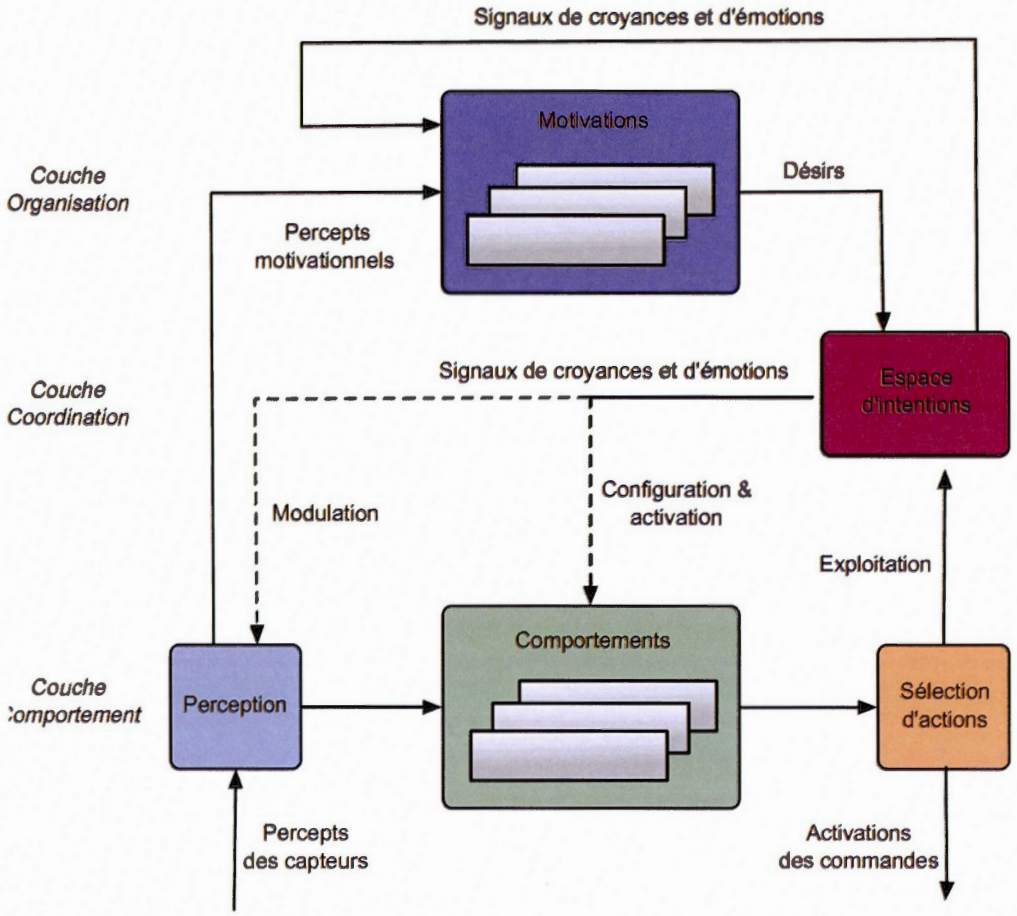

Figure 2.7 Hybrid Behavior-Based Architecture (HBBA) (Source : Laboratoire IntRoLab)

veaux), la couche la plus haute offrant le plus grand niveau d'abstraction. HBBA procède d'une manière similaire et unifie les deux paradigmes en ajoutant des couches par-dessus la couche comportementale. Le principe est simple et réside dans le même mécanisme que celui utilisé avec MBA, à savoir donner la possibilité à différentes sources au sein du système d'émettre des **émotions** au plus haut niveau de l'architecture.Ces motivations se traduisent sous la forme de **désirs** envoyés à l'**espace d'intentions**. Ce dernier collecte les désirs afin d'émettre des **intentions** d'une manière qui limite les conflits et maximise un résultat pertinent.

Par la suite, les comportements au sein de l'architecture sont sélectionnés et activés dépendamment de ces intentions. L'espace d'intentions permet également de détecter des situations dans lesquelles le robot peut se situer à l'aide de modèles d'exploitation de comportements, et ainsi de générer des croyances sur l'état du robot et son environnement.

En définitive, cela fait de HBBA une architecture comportementale très modulable, sans représentation centrale, permettant l'intégration de modules de raisonnement et de planification de haut niveau simplement comme sources de motivations. Le chapitre 3 présente le travail d'intégration du planificateur ActuPlan en tant que module émetteur de désirs au sein de HBBA.

2.4 Environnements pour robots mobiles

Afin de déployer une des architectures présentées dans la section précédente au sein d'un robot, il est utile et souvent nécessaire d'utiliser un environnement logiciel pour coordonner les différents modules que l'on souhaite implémenter. Il existe de nombreux environnements logiciels et frameworks différents, dont le design a évolué avec le temps pour répondre plus efficacement aux besoins des développeurs (Biggs *et al.*, 2013). Ces frameworks se présentent généralement sous la forme d'un *middleware* (intergiciel) dont l'objectif principal est d'offrir un ou plusieurs niveaux d'abstraction supérieurs au matériel (*hardware*) du robot afin de permettre de développer des modules qui exploitent ce dernier, ainsi que de proposer une interface de communication permettant aux différents modules de communiquer entre eux. Les environnements logiciels sont également souvent accompagnés d'un simulateur permettant de simuler un ou plusieurs robots dans un environnement virtuel.

Parmi les frameworks les plus connus et utilisés depuis une quinzaine d'années,

on peut citer :

— Player[1] (Gerkey *et al.*, 2003) (accompagné du simulateur Stage) ;

— Saphira[2] (Konolige et Myers, 1998) ;

— MARIE (Côté *et al.*, 2006) ;

— ROS[3] (Quigley *et al.*, 2009).

Aujourd'hui, l'environnement ROS est privilégié pour la conception et le déploiement de systèmes robotiques, en raison de l'efficacité de son architecture, de sa grande compatibilité avec les machines existantes et de son importante communauté. ROS est l'environnement utilisé dans le cadre du développement de l'architecture HBBA présentée à la section 2.3 et est donc également celui exploité dans le cadre de ce mémoire.

2.5 ROS : *Robotic Operating System*

Robot Operating System (Quigley *et al.*, 2009) (ROS), qu'on peut traduire *système d'exploitation pour robot*), est un ensemble d'outils et de bibliothèques de logiciels permettant de faciliter le développement de programmes visant à piloter ou à rendre autonomes des robots. ROS se présente sous la forme d'un projet collaboratif permettant aux laboratoires ou individus de partager, sous la forme d'un framework commun, des technologies développées dans le but d'offrir un certain nombre de capacités au robot. Développer un ensemble complet de modules permettant de faire fonctionner un robot est une tâche extrêmement complexe ; ROS permet de centraliser l'expertise de tous les développeurs travaillant sur une technologie pouvant être utilisée au sein d'une machine.

---

1. http ://playerstage.sourceforge.net/

2. http ://www.ai.sri.com/ konolige/saphira/

3. http ://www.ros.org/

ROS s'organise sous la forme d'un ensemble de composants de base (*Core Components*) et d'une suite permettant l'intégration de librairies tierces.Les composants de base se présentent de la manière suivante :

- une architecture spécifique pour la communication entre les différents composants (*Communications Infrastructure*) ;
- des composants spécifiques au robot (*Robot-Specific Features*) ;
- un ensemble d'outils pour le développement.

Les librairies tierces pouvant être intégrées à ROS sont quant à elles les suivantes :

- **Gazebo** [4] : un simulateur 3D pour robots multiples avec prise en compte de la dynamique et de la cinématique ;
- **OpenCV** [5] : une librairie pour la vision par ordinateur (*computer vision*) ;
- **MoveIt !** [6] : une librairie pour la planification de mouvement (*motion planning*) (LaValle, 2006).

2.5.1 Architecture logicielle

Tel que précisé précédemment, ROS dispose de sa propre architecture en termes de conception de modules et de communication entre ces derniers. Nous présentons dans cette section plus en détail les mécaniques qui régissent le développement de ces modules, afin de mieux comprendre l'architecture qui sera proposée au chapitre 3.

---

4. http ://gazebosim.org/
5. http ://opencv.org/
6. http ://moveit.ros.org/

Paquet

Le développement sous ROS s'effectue sous forme de modules ou paquets (*packages*). Pour chaque fonctionnalité, ensemble de fonctionnalités ou module requérant une certaine indépendance, un développeur travaillant sous ROS peut (et doit) développer un paquet ROS soumis à ses propres dépendances et contexte de compilation et exécution. Un paquet ROS compile les fichiers sources spécifiquement développés par le développeur ainsi qu'un certain de nombre de dépendances requises par ce code. Chaque paquet peut contenir les ressources suivantes :

- des fichiers sources ;
- des librairies ;
- des exécutables ;
- des scripts ;
- et d'autres artefacts.

Les dépendances peuvent prendre la forme de bibliothèques tierces ou d'autres paquets ROS.

Un paquet ROS peut, une fois compilé et dépendamment de l'implémentation du développeur, proposer plusieurs programmes à exécuter. Ces programmes sont appelés des **noeuds**.

Pile

Une pile ROS (*stack*) est une collection de paquets. Il s'agit généralement d'une agrégation de fonctionnalités telles que la navigation, la localisation, les interactions, la vision, etc. Une pile présente simplement un intérêt organisationnel et permet éventuellement d'automatiser certaines tâches de compilation, mais n'influence pas l'exécution du code développé sous ROS.

Nœuds

Un nœud ROS (*node*) est un exécutable à l'intérieur d'un paquet connecté au réseau ROS. Un nœud est donc simplement un programme développé dans le but de proposer une fonctionnalité et qui profite de l'écosystème ROS afin de communiquer facilement avec d'autres nœuds. Les nœuds utilisent une librairie cliente de ROS afin de communiquer entre eux selon les spécifications et protocoles de communication définis par le système ROS. Par exemple, on peut développer un nœud qui a pour fonction de demander au robot de se déplacer d'un point A à un point B. Ce nœud peut alors faire appel à d'autres nœuds, chargés d'effectuer une planification de chemin ou encore d'activer les moteurs du robot.

Pour chaque instance d'exécution d'un nœud ou d'un ensemble de nœuds, il est nécessaire d'exécuter préalablement le **ROS Master**. ROS Master est un paquet offrant un ensemble de services déployant l'infrastructure de communication de ROS, permettant ainsi aux différents nœuds de se trouver et de communiquer entre eux. Il est également nécessaire d'exécuter **rosout**, équivalant sortie standard (*stdout/stderr*), ainsi que le serveur de paramètres (*parameter server*). Ces trois éléments constituent le **ROSCore**, noyau obligatoire pour toute exécution de programme lié à l'environnement ROS.

Les nœuds peuvent publier ou s'abonner à des **topics**. Ils peuvent également déployer ou utiliser un **service**. La figure 2.8 présente un exemple de configuration de nœuds communiquant entre eux à travers l'utilisation de topics et de services.

Topics

Un **topic** est un canal de communication par lequel les nœuds peuvent s'échanger des messages. Ils répondent à une architecture de type publication/souscription (*publish/subscribe*) (Eugster *et al.*, 2003) permettant ainsi de découpler la source

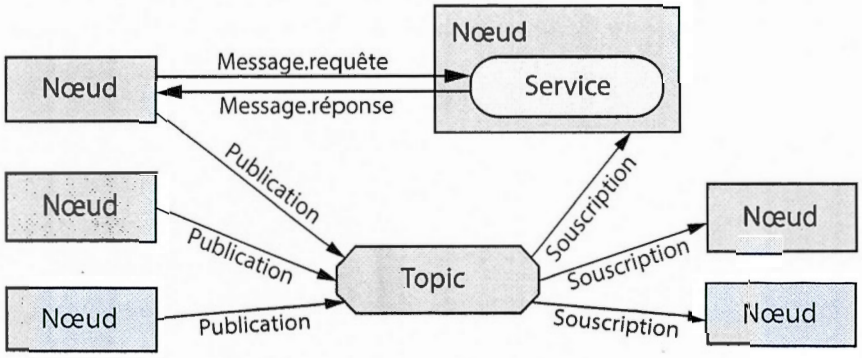

Figure 2.8 Exemple d'architecture déployée avec l'environnement ROS

de production d'informations de celui qui la consomme. Cela permet aux nœuds de ne pas avoir à se préoccuper de avec qui ils communiquent : si un nœud doit partager une information, il lui suffit de déployer un topic et d'y publier des données. Le ou les nœuds intéressés par ces données n'ont ensuite qu'à s'y abonner, sans avoir à connaître précisément le nœud à l'origine de l'information.

Les topics sont utilisés pour une diffusion de données continue à une fréquence choisie. Par exemple, le flux de données relatif à la position courante du robot peut être publié sur un topic spécifique, à une fréquence de 1 hertz, ou encore la force exercée sur ses actionneurs sur un autre topic à une fréquence de 20 hertz.

Si un nœud souhaite interroger une tierce partie pour une information ponctuelle, on doit faire appel alors à un **service**.

Services

Le modèle de communication adopté pour les topics est efficace et très flexible, mais il ne répond pas à tous les besoins en termes d'échange d'informations au sein d'un système distribué tel que ROS. Afin d'ouvrir une voie de communication « déconnectée », de type requête/réponse, ROS propose les **services**.

Un service est défini par une paire de messages : un pour la requête, l'autre pour la réponse. Un service est créé et déployé par un nœud. D'autres nœuds peuvent ensuite appeler le service via l'envoi d'un **message** faisant office de requête et ensuite attendre de recevoir un autre message en réponse. Un appel à un service peut être vu comme un appel à une fonction distante.

Message

Comme cela a été précisé plus haut, l'envoi et la réception de données dans le cas d'utilisation de topics et services se réalisent via l'envoi de **messages**. ROS propose ainsi aux développeurs de définir, via une syntaxe particulière, leurs propres messages. Un message est une structure de données comprenant un certain nombre de champs typés. Les types classiques que l'on retrouve dans la majorité des langages de programmation, à savoir les entiers (*integer*), les flottants (*floating point*), les booléens, etc., sont supportés, ainsi que les tableaux de type primitif. Une fois un message défini, il est également possible de l'utiliser à l'intérieur d'un autre message. La figure 2.1 présente l'exemple de **personne.msg**, un message implémentant les données d'une personne.

```
string   prenom
string   nom_de_famille
uint8    age
uni32    score
```

Listing 2.1 Exemple de message ROS : *personne.msg*

CHAPITRE III

INTÉGRATION DU PLANIFICATEUR ACTUPLAN DANS L'ARCHITECTURE HBBA

L'architecture HBBA (*Hybrid Behaviour Based Architecture*) a été initialement conçue pour l'intégration de capacités d'interaction homme-machine dans des robots, afin de créer des robots agissant de manière naturelle avec les humains. Un tel contexte d'utilisation peut amener à privilégier des architectures robotiques plus réactives, telles que les architectures hybrides ou à base de comportements, parfois aux dépens de fonctionnalités de planification de plus haut niveau. Dans la section 2.3, nous avons présenté HBBA comme une architecture hybride à base de comportements, mais offrant la possibilité d'intégrer facilement des modules de haut niveau grâce au mécanisme d'émission de désirs. Les sections suivantes présentent l'approche utilisée pour développer un module permettant d'exploiter le planificateur ActuPlan (section 1.6.2) au sein de HBBA, afin de permettre au robot de gérer ses tâches en présence d'incertitude sur la durée des actions.

De manière générale, cette tâche consiste à intégrer deux technologies l'une avec l'autre, alors que chacune est développée indépendamment, dans des contextes et avec des langages de programmation différents. Le défi derrière ce mandat est de proposer une approche pertinente pour combiner ces deux technologies, en prenant en compte les contraintes techniques imposées par chacune et en conservant leur

intérêt et leur puissance lorsqu'elles sont utilisées individuellement.

3.1 Approche proposée

Dans le cadre de notre projet, il ne s'agit pas simplement de faire cohabiter les deux technologies fonctionnant de manière complètement autonome que sont HBBA et ActuPlan. HBBA se positionne comme l'élément central faisant cohabiter un ensemble de fonctionnalités permettant ultimement à un robot de fonctionner. Dans cette optique, ActuPlan doit être la technologie s'adaptant à HBBA et non l'inverse.

Utiliser un planificateur de tâches de haut niveau tel que ActuPlan afin de résoudre des problèmes de planification pour un robot évoluant dans un environnement non simulé en temps réel implique de conserver en tout temps une représentation de l'état courant, afin de permettre le suivi du plan et éventuellement, de procéder à une replanification si nécessaire. HBBA et ActuPlan étant développés selon deux technologies différentes, conserver une représentation du domaine, du problème et de l'état courant au sein de HBBA impliquerait un travail d'implémentation très conséquent, qui est déjà réalisé avec le planificateur ActuPlan. Partant de ce principe, il a été choisi de conserver le planificateur ActuPlan comme technologie autonome, encapsulée au sein d'un service permettant un échange d'information direct avec HBBA selon un protocole de communication défini. Le service intégrant le planificateur ActuPlan est baptisé **ActuPlan Java Server (AJS)**.

HBBA a été développé à partir des outils proposés par la technologie ROS et s'appuie donc sur des mécanismes permettant le développement et l'intégration de nouveaux modules. L'intégration de l'interface de communication avec le serveur ActuPlan prend donc la forme d'un paquet ROS utilisant les fonctionnalités des autres paquets ROS de HBBA existants. Le paquet ROS intégrant cette interface

de communication avec ActuPlan est appelé `irl_actuplan`.

3.2 Architecture

La figure 3.1 présente l'architecture utilisée pour intégrer le planificateur ActuPlan à HBBA. Dans cette section, nous proposons un descriptif des différents modules et de leurs fonctionnalités respectives. Les sections 3.3 et 3.4 présentent plus en détail la logique implémentée au sein de ces modules.

3.2.1 ActuPlan Java Server

ActuPlan Java Server est le programme serveur encapsulant le planificateur ActuPlan. Fonctionnant de manière complètement autonome, il permet la connexion de multiples clients, la génération et le suivi de plans, ainsi que la simulation du plan par rapport à l'état courant.

3.2.2 HBBA Core

Nous appelons dans le cadre de ce projet **HBBA Core** l'ensemble des modules ROS développés au sein de HBBA permettant l'envoi de désirs, leur traitement, l'émission d'intentions, l'activation des comportements et la sélection des actions. HBBA Core correspond à la plupart des fonctions de l'architecture de HBBA présentées à la figure 2.7.

Il s'agit donc d'un ensemble de paquets ROS, exécutés de manière indépendante au lancement du robot ou de la simulation. Une fois HBBA Core en cours d'exécution, un ensemble de services et de *topics* sont utilisables. Dans le cadre de l'intégration d'ActuPlan à HBBA, le service le plus utilisé est celui permettant l'émission de désirs au reste de l'architecture HBBA.

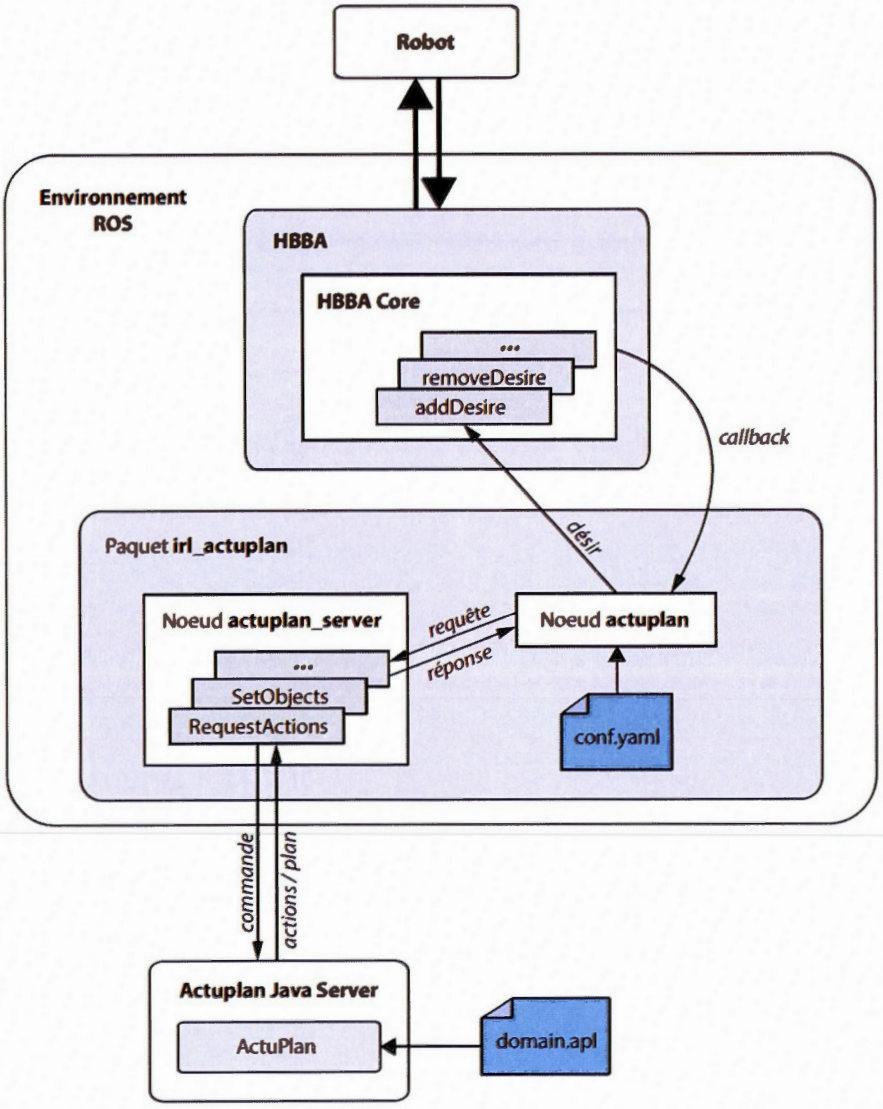

Figure 3.1 Architecture de l'intégration de ActuPlan au sein de HBBA

3.2.3 Paquet `irl_actuplan`

Le paquet ROS `irl_actuplan` est chargé d'implémenter l'interface de communication entre HBBA et le module ActuPlan Java Server, qui intègre le planificateur. Le paquet `irl_actuplan` contient deux nœuds, `actuplan` et `actuplan_server`, tous deux exécutés simultanément après que HBBA Core et ActuPlan Java Server

aient été initialement lancés. Les fonctionnalités de ces deux nœuds sont détaillées à la section 3.4.

3.3 ActuPlan Java Server

ActuPlan Java Server a été conçu dans l'optique d'apporter une solution simple au problème d'intégration du planificateur ActuPlan à HBBA. En proposant un programme encapsulant le planificateur d'origine sous la forme d'un serveur définissant son propre protocole de communication, le planificateur ActuPlan conserve toute son autonomie et les tâches consistant à générer un plan puis à suivre sa validité devient bien moins complexe que de redéfinir les données du problème du côté du client.

Ce procédé permet également d'étendre et d'adapter les capacités du planificateur à une utilisation appliquée au domaine de la robotique. En effet, ActuPlan est un planificateur de type ***offline***, c'est-à-dire qu'il ne prend pas en considération, lors de la planification, les évolutions au sein l'environnement pouvant survenir en cours d'exécution du plan. Cependant, amener un robot à réaliser un certain nombre d'objectifs à travers l'exécution d'un plan est une démarche qui doit être effectué en tenant compte de la dynamique du système. Malgré le fait que le planificateur n'ait pas nécessairement besoin de prendre en compte tous les détails influençant la dynamique du système pour planifier et produire un plan, dans un contexte appliqué à la robotique, il ne peut pas complètement ignorer les évolutions du système. Il est donc nécessaire d'au moins vérifier, lors de l'exécution du plan, que ce dernier reste valide et, si nécessaire, de le modifier ou de replanifier, pour en faire un planificateur de type ***online***.L'encapsulation du planificateur ActuPlan au sein du programme ActuPlan Java Server permet de répondre à ce besoin à travers les deux fonctionnalités suivantes : la vérification du plan par

rapport à l'état courant et la replanification.

3.3.1 Protocole de communication

Lors de son exécution, ActuPlan Java Server (AJS) déploie un serveur de sockets sur un port spécifique, auquel peut se connecter n'importe quel client. Le socket est gardé ouvert durant toute la durée de la session, c'est-à-dire jusqu'à demande explicite d'arrêt de la session par le client, par exemple après que ce dernier ait terminé l'exécution du plan et n'ait plus besoin du planificateur. Une fois la connexion établie, le client peut envoyer une commande à AJS à travers le socket, sous la forme d'une chaîne de caractères. La syntaxe est la suivante :

```
nom_de_la_commande [options]
```

Les commandes offertes par AJS sont les suivantes :

- `RequestPlan`
  retourne le plan courant ;
- `RequestActions`
  retourne la ou les prochaines actions à exécuter ;
- `Objects objets`
  initialise les objets du système ;
- `WorldState monde`
  initialise le monde ;
- `CurrentState etat_courant`
  initialise l'état courant du système (peut être l'état initial) ;
- `Goal but`
  initialise le but ;
- `SetTime temps`
  définit le temps courant d'exécution du plan ;

— `Completed action`
indique la terminaison de l'action en cours ;

— `Status action@temps_fin_estimé`
indique l'état de complétion d'une action.

3.3.2 Suivi et vérification du plan courant

L'exécution de chaque action qui compose un plan produit par le planificateur mène à un nouvel état. L'exécution de l'ensemble des actions du plan modélise donc une séquence d'états correspondant aux différentes étapes du robot au sein de l'environnement.La commande `Completed` permet de signaler que l'action en cours d'exécution par le robot est terminée. Cela fait progresser le plan. Au prochain appel de la commande `RequestActions`, le planificateur renvoie alors les actions suivantes du plan à exécuter.

Cependant, lorsque le robot est en cours d'exécution d'une action requise par le plan, il se situe dans un état intermédiaire, parmi une infinité d'états possibles, qui n'est pas directement monitoré ou même connu par le planificateur. Dans ce contexte, il est impossible pour le planificateur ActuPlan d'évaluer si l'état courant dans lequel se situe le robot par rapport au plan permet toujours de satisfaire le but. Afin de répondre à ce problème, ActuPlan Java Server étend les capacités du planificateur en lui permettant de recevoir à tout moment une information sur l'état de complétion de l'action en cours d'exécution par le robot.

La commande `Status` permet de recevoir l'état d'une action en cours d'exécution par le robot. En option, la commande indique le temps nécessaire estimé (par le robot) auquel l'action sera complétée. À partir de ce temps estimé, ActuPlan Java Server met à jour les informations internes du planificateur, simule l'exécution du reste du plan et réestime sa probabilité de réussite.

Tel que présentée à la section 1.6.2, l'incertitude liée au temps est modélisée au sein du planificateur ActuPlan, à travers un ensemble de variables aléatoires maintenues dans un réseau bayésien. Lorsqu'une action est indiquée comme terminée ou qu'une estimation est faite par le robot, le temps correspondant à la fin de l'exécution de l'action **n'est plus sujet à de l'incertitude, mais correspond alors à une valeur constante connue**. Afin de simuler l'exécution du reste du plan et ainsi tester sa validité, il est donc nécessaire de modifier, au sein du réseau bayésien, la variable aléatoire modélisant l'incertitude sur le temps de fin d'exécution de l'action par une constante définie par le temps retourné par le robot. Il est alors possible de réévaluer l'ensemble des variables aléatoires composant le réseau bayésien afin de prendre en compte cette nouvelle variable. La figure 3.2 illustre ce changement de variable au sein du réseau bayésien.

3.3.3 Replanification

Un approche simple et classique pour intégrer un planificateur dans un système consiste à alterner planification et exécution (Lemai et Ingrand, 2004). ActuPlan étant un planificateur *offline* produisant systématiquement un plan complet, reproduire une telle mécanique de planification *online* consisterait à replanifier systématiquement. Replanifier étant une tâche potentiellement coûteuse en termes de temps et de ressources computationnelles, il est préférable de ne replanifier que quand cela est réellement nécessaire, c'est-à-dire quand le plan n'est plus valide.

À la suite du processus de mise à jour de la variable aléatoire modélisant le temps de fin d'exécution et de réévaluation du réseau bayésien décrit à la section 3.3.2, il est possible de simuler l'exécution du plan et ainsi d'obtenir une **probabilité de succès** de ce dernier, basée sur les nouvelles variables aléatoires générées.

Si la probabilité de réussite est inférieure au seuil choisi (p. ex. une probabilité

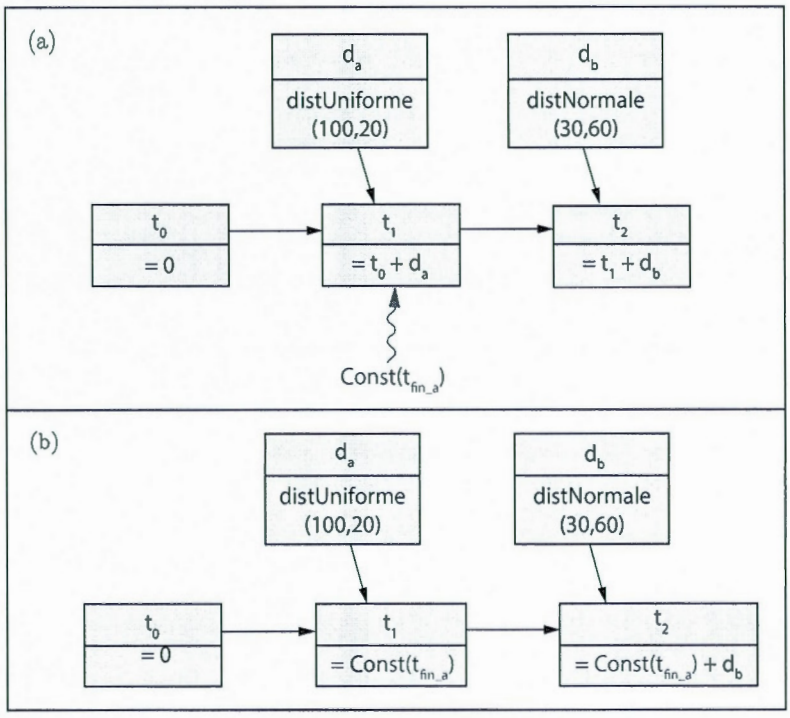

Figure 3.2 Mise à jour des variables aléatoires du réseau bayésien après complétion d'une action. (a) présente le réseau bayésien tel que généré initialement par le planificateur. (b) présente le réseau bayésien après modification des variables aléatoires correspondant au temps de fin de l'action a.

de 0.8), on considère le plan comme invalide. On procède à alors une tentative de replanification, en considérant comme nouvel état initial l'état courant dans lequel se situe le robot. Si, à la suite du processus de replanification, aucun nouveau plan n'est trouvé, cela signifie que le problème ne possède pas de solution au regard de l'état et du but actuel. Pour l'instant, lors de telles situations, le but est abandonné. Comme travaux futurs, nous pourrions utiliser une version d'ActuPlan capable de satisfaire partiellement les objectifs fournis (Labranche et Beaudry, 2014).

3.4 Paquet `irl_actuplan`

3.4.1 Nœud `actuplan_server`

Le nœud **`actuplan_server`** a pour fonction d'ouvrir un socket et de déployer un certain nombre de services ROS permettant une communication simplifiée et standardisée avec ActuPlan Java Server pour les autres nœuds ROS. Un service ROS devant définir publiquement un **message** d'entrée et de réponse [1], tout autre module ROS peut exploiter ce service et ainsi facilement communiquer avec AJS. Chaque service déployé par le nœud **`actuplan_server`** se charge de transformer le message d'entrée en un ensemble de données respectant le protocole de communication avec AJS. La liste des services déployés pour la communication avec AJS est la suivante :

— **`SetObjects`** : prend en paramètre un ensemble d'objets et envoie la chaîne de caractères correctement formatée à AJS afin d'initialiser les objets du problème ;

1. Si l'appel au service est réalisé depuis le code source du module, le langage utilisé est le C++ ou le Python. Le service peut également est appelé depuis une invite de commande une fois le module ROS lancé, le message d'entrée étant alors transmis au service au format YAML.

— `SetState` : prend en paramètre un état (la représentation du monde (*world state*) ou l'état initial du problème (*initial state*)) et envoie la chaîne de caractères correctement formatée à AJS afin d'initialiser l'état correspondant ;

— `SetGoal` : prend en paramètre un ensemble de littéraux correspondant aux buts du problème et envoie la chaîne de caractères correctement formatée à AJS afin d'initialiser les buts du problème pour le planificateur ;

— `UpdateCurrentState` : prend en paramètre un littéral correspondant à une donnée de l'état courant ayant évolué et envoie la chaîne de caractères correctement formatée à AJS afin de mettre à jour l'état courant du planificateur ;

— `RequestActions` : demande à AJS d'envoyer l'action ou l'ensemble des actions suivantes du plan ;

— `RequestPlan` : demande à AJS d'envoyer l'ensemble du plan trouvé par le planificateur.

3.4.2 Nœud `actuplan`

Le nœud `actuplan` est principalement chargé d'émettre des désirs à travers HBBA, dépendamment des actions retournées par le planificateur. Il utilise directement les services déployés par le nœud `actuplan_server` afin de communiquer avec ActuPlan Java Server.

Les **désirs** étant liés aux **intentions** activant elles-mêmes des **comportements** liés aux capacités du robot, HBBA doit définir à l'avance les désirs pouvant être émis au sein de l'architecture. La liste des désirs valides est donc connue à l'avance. Le nœud `actuplan` propose donc un système de couplage dynamique via un fichier de configuration externe, afin de définir les différentes associations entre les actions

retournées par le planificateur et les désirs de HBBA.

À son exécution, le nœud `actuplan` peut également prendre comme paramètre un fichier de configuration (.yaml) afin d'initialiser l'ensemble du problème de planification. Pour cela, le nœud récupère l'ensemble des informations contenues au sein du fichier et les communique à AJS à travers les services du nœud `actuplan_server`.

CHAPITRE IV

EXPÉRIMENTATIONS

Le présent chapitre présente les expérimentations réalisées et les résultats obtenus pour valider le bon fonctionnement des modules ActuPlan Java Server et du noeud ROS `actuplan_server`, qui ont été développés dans le cadre de ce projet. Trois types d'expérimentations ont été effectués :

1. des expérimentations sur un robot réel, soit le robot IRL-1 décrit à la section 4.1;
2. des expérimentations dans le simulateur *Stage* décrit à la section 4.2;
3. expérimentations à l'aide de modules de tests, sans robot ou simulateur.

Les premières expérimentations menées sur le robot IRL-1 ont été réalisées avec une version préliminaire du prototype d'intégration du planificateur ActuPlan. Ces expérimentations avaient deux objectifs : tester le bon fonctionnement général du prototype sur un vrai robot et extraire des données d'expérimentation utiles à l'amélioration du prototype. En effet, tel que présenté à la section 1.6.2, les informations liées à l'incertitude sur la durée d'une action sont encodées sous la forme de variables aléatoires modélisées à travers une distribution. Afin d'intégrer une définition du domaine au sein du planificateur ActuPlan qui soit le plus près possible de la réalité, il est nécessaire de définir un modèle probabiliste modélisant au plus proche l'incertitude sur les capacités du robot IRL-1 à effectuer une action (par exemple se déplacer ou prendre un colis). En raison d'un environnement de

test limité en situation réelle au laboratoire IntRoLab, seules des données sur l'odométrie du robot ont pu être collectées.

4.1 Robot IRL-1

Le robot IRL-1 (figure 4.1), aussi appelé Johnny-0, a été conçu au laboratoire IntRoLab de l'Université de Sherbrooke. Il a été développé dans le but de proposer une plateforme interactive pour l'intégration de divers projets liés au domaine de l'interaction homme-machine. Le robot est composé de deux plateformes principales : AZIMUT-3[1] et un buste muni de différents capteurs, principalement utilisés pour détecter et interagir avec une présence humaine.

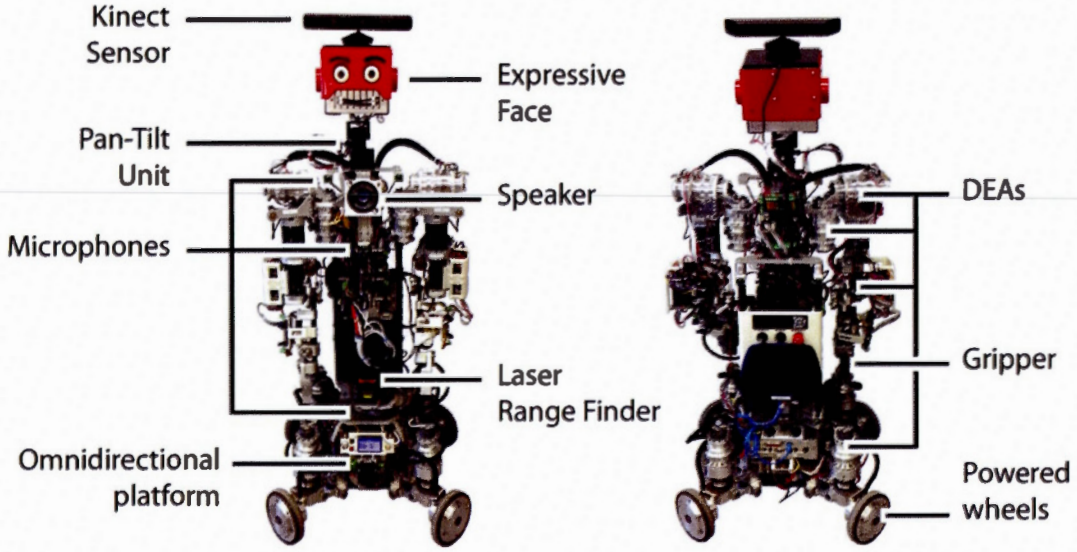

Figure 4.1 Robot IRL-1/Johny-0 (Source : Université de Sherbrooke, laboratoire IntRoLab)

AZIMUT-3 (figure 4.2) est une plateforme mobile qui chercher à proposer une solution pour l'intégration de multiples mécanismes de locomotion sur une même

1. https ://introlab.3it.usherbrooke.ca/mediawiki-introlab/index.php/AZIMUT-3

plateforme robotique. AZIMUT-3 dispose de quatre (4) articulations indépendantes, sur lesquelles peuvent être fixées des roues, des jambes mécaniques, des chenilles ou encore une combinaison de ces mécanismes.AZIMUT-3 possède la fonctionnalité de changer la direction de ses articulations afin de se déplacer sans changer son orientation, ce qui le rend omnidirectionnel.

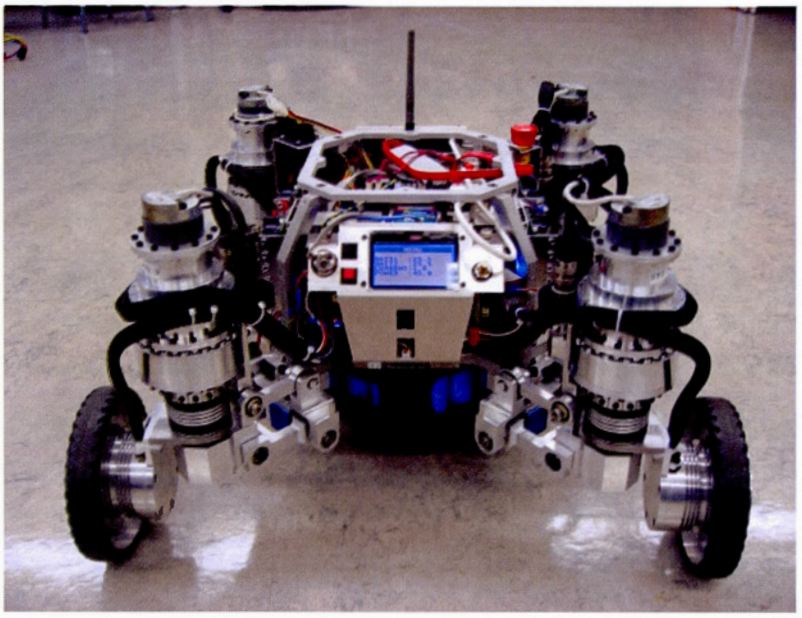

Figure 4.2 Plateforme robotique AZIMUT-3 (Source : Université de Sherbrooke, laboratoire IntRoLab)

La plateforme supérieure de IRL-1 (le buste) est quant à elle composée des capteurs suivants :

- un capteur laser de proximité (Hokuyo UTM-30LX) détectant la présence de jambes sur 180 degrés ;
- une caméra stéréo Microsoft Kinect ;
- une caméra pour la détection de visages ;
- une matrice de huit microphones pour localiser, suivre et séparer les sons.

IRL-1 peut ainsi interagir avec son environnement et avec une présence humaine de différentes façons : voix, expressions faciales, mouvement de la tête, gestes,

déplacements, etc.

4.2 Simulateur *Stage*

Le simulateur *Stage*[2] est un simulateur de robot 2D développé dans le cadre du projet Player/Stage (Gerkey *et al.*, 2003). Il permet de simuler un robot ou un groupe de robots ainsi que leurs capteurs (lasers, sonars, caméra avec détection de couleurs, etc.) et les informations que ceux-ci produisent à travers un environnement à deux dimensions simulé.

Stage a été développé dans l'optique de proposer des contrôleurs simulés correspondant le plus possible au véritable matériel de la machine. Ainsi le robot évoluant dans un environnement simulé *Stage* reflète un comportement relativement proche de la réalité. La figure 4.3 présente une capture d'écran du simulateur *Stage*. Les cônes autour du robot représentent l'environnement perçu par ses capteurs.

Le robot IRL-1 peut être simulé dans *Stage*.

4.3 Modèle de la durée des déplacements

Le planificateur ActuPlan a besoin d'un modèle probabiliste pour prédire la durée des déplacements. La distribution de la durée d'un déplacement ne suit pas forcément une distribution paramétrique comme une loi uniforme ou normale. Pour estimer la distribution, nous avons utilisé un journal des données obtenues après avoir fait rouler le robot pendant quelques heures. À l'aide des données extraites du module de localisation, il est possible d'estimer la durée des déplacements et de construire un modèle.

---

2. http ://playerstage.sourceforge.net/

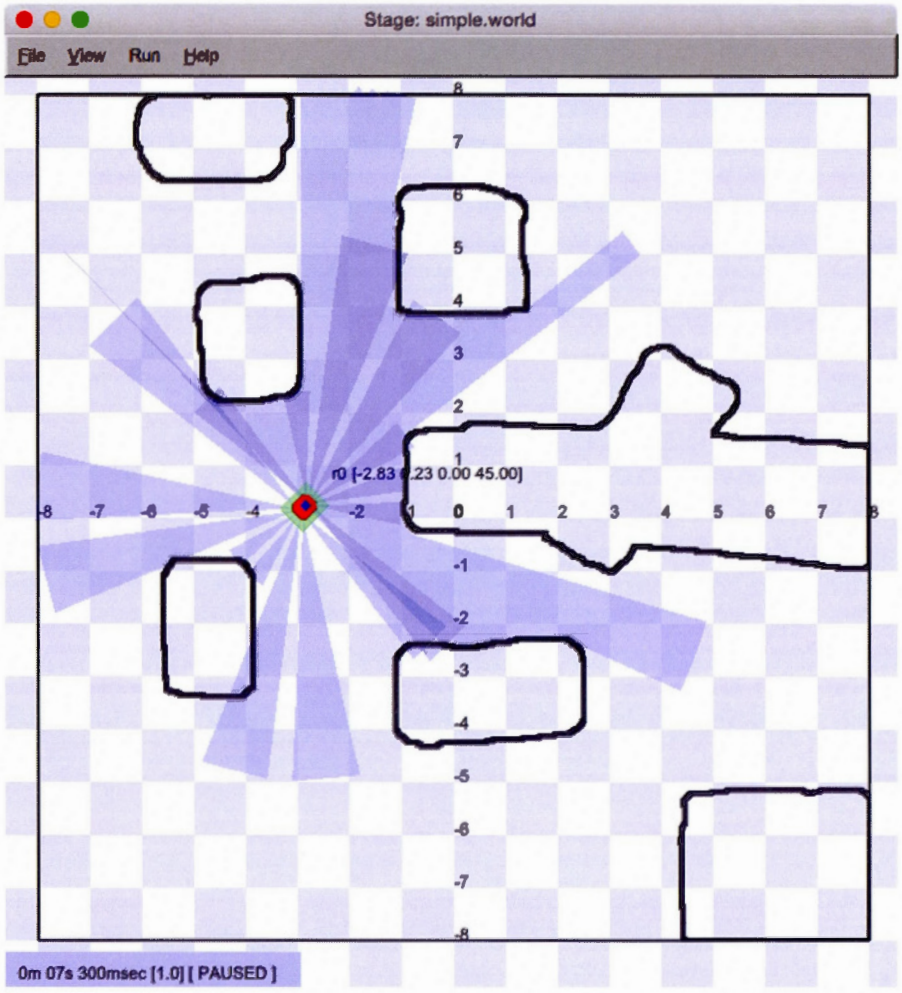

Figure 4.3 Exemple de simulation à travers le simulateur *Stage*

La figure 4.4 présente la distribution projetée, sous forme de fonctions cumulatives de probabilité, sur six (6) distances différentes. Par exemple, cela montre qu'un déplacement de dix mètres dure environ de 20 à 160 secondes. Une fois entrée dans le planificateur ActuPlan, cette distribution est échantillonnée selon la distance des déplacements associés aux actions `Goto`.

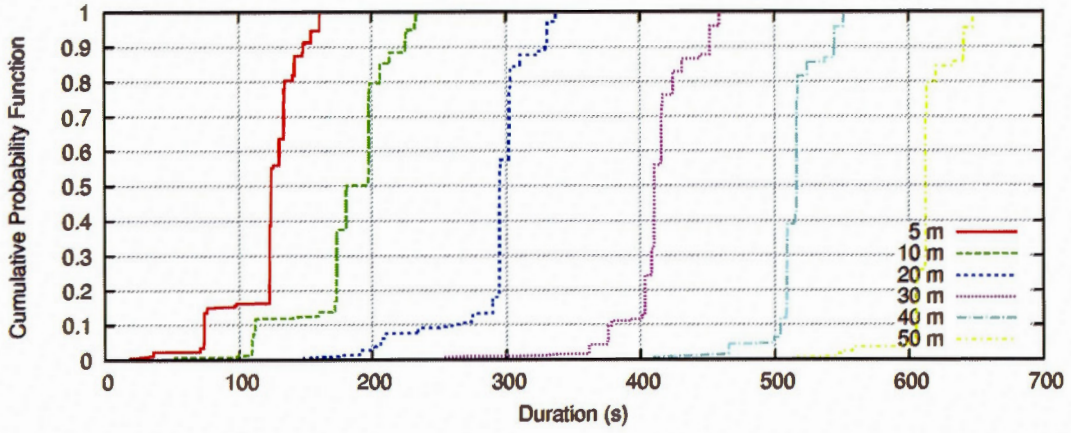

Figure 4.4 Modèle probabiliste des durées de déplacement

4.4 Domaine de planification : livraison de colis avec IRL-1

Le domaine de planification expérimenté est celui du robot livreur de colis. On considère un monde dans lequel il peut exister un ou plusieurs robots, des colis ainsi que des lieux connectés entre eux selon une distance définie. Le robot peut effectuer les actions de se déplacer jusqu'à un lieu, charger et décharger un colis. Le listing 4.1 présente le domaine au complet, tel qu'encodé au sein du planificateur ActuPlan. Au sein de cette représentation du domaine, les lignes 17, 27 et 39 représentent respectivement les distributions aléatoires associées à la durée des actions de déplacement, de chargement et de déchargement d'un paquet. `custom_irl1 orig dest`, à la ligne 17, correspond à la distribution aléatoire basée sur un modèle probabiliste extrait des données d'expérimentation réalisées sur le robot irl_1. Ce modèle probabiliste est présenté à la section 4.3. Les durées des actions de chargement et déchargement de colis sont modélisées sous la forme d'une distribution uniforme sur l'intervalle $[3, 9]$.

```
domain {
    types { Location, Object, Robot }

    world {
       predicate Connected(Location,Location);
       numeric   Distance(Location,Location);
    }

    state {
       function  Location PosRobot(Robot);
       function  Location PosObject(Object);
       function  Object   HasWhichObject(Robot);
    }

    action Goto(Robot r, Location orig, Location dest){
        duration:
           custom_irl1 orig dest;
        conditions:
           @start: PosRobot(r) = orig;
           @start: Connected(orig, dest);
        effects:
           @end: PosRobot(r) = dest;
    }

    action Load(Robot r, Location p, Object o){
       duration:
          uniform 3 9;
       conditions:
          @overall: PosRobot(r) = p;
          @start: PosObject(o) = p;
          @start: HasWhichObject(r) = undefined;
       effects:
          @end: PosObject(o) = undefined;
          @end: HasWhichObject(r) = o;
    }

    action Unload(Robot r, Location p, Object o){
       duration:
          uniform 3 9;
       conditions:
         @overall: PosRobot(r) = p;
         @start: HasWhichObject(r) = o;
         @start: PosObject(o) = undefined;
       effects:
         @end: PosObject(o) = p;
         @end: HasWhichObject(r) = undefined;
    }
}
```

Listing 4.1 Spécification du domaine de planification du livreur de colis défini selon le langage de spécification propre à ActuPlan.

4.5 Expérimentations en situation réelle sur le robot IRL-1

Afin d'exploiter le problème de planification du livreur de colis avec le robot IRL-1, la discrétisation d'une partie de la carte du rez-de-chaussée du laboratoire IntRoLab en un certain nombre de lieux a été réalisée. La figure 4.5 montre la carte obtenue à l'aide d'un logiciel de cartographie intégré à ROS.

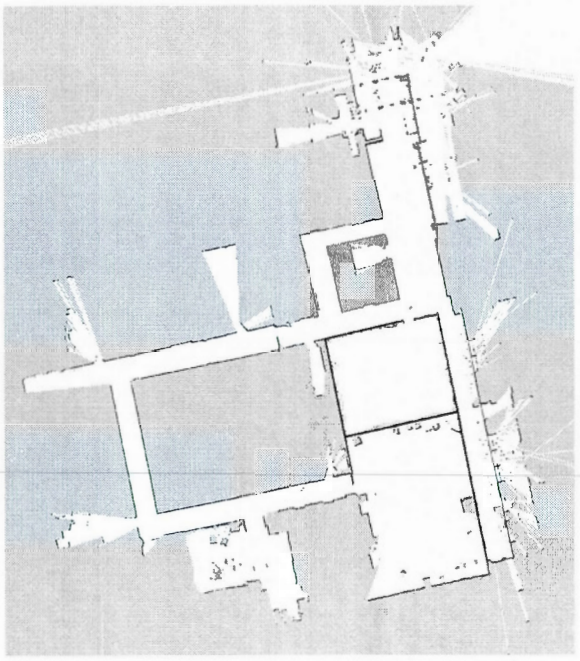

Figure 4.5 Carte construite du rez-de-chaussée du laboratoire IntRoLab

Sur cette carte, dix (10) emplacements (*locations*) ont été ajoutés. Les distances entre les emplacements visibles entre eux ont été estimées à l'aide d'un planificateur de chemin qui considère les obstacles statiques. Ces distances permettent au planificateur ActuPlan d'estimer les durées de déplacement entre les emplacements. La figure 4.6 présente la carte du laboratoire exploitée par le planificateur ActuPlan.

Le problème consiste ensuite en deux (2) paquets positionnés à deux (2)lieux

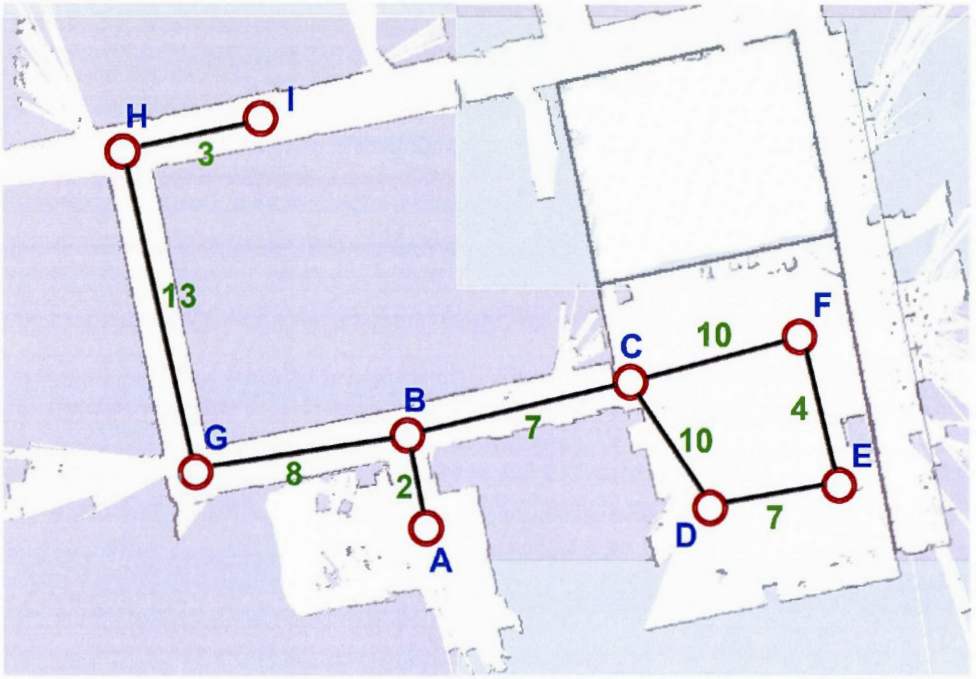

Figure 4.6 Carte du rez-de-chaussée du laboratoire IntRoLab. (a) Carte construite à partir des capteurs du robot IRL-1; (b) Carte partielle discrétisée selon 9 points. Les distances affichées en vert sont en mètres.

différents de la carte que le robot doit respectivement récupérer et déposer à deux autres points différents. La définition complète du problème de planification est présentée à la figure 4.2. Le plan résolvant ce problème produit par le planificateur ActuPlan est présenté au listing 4.3.

```
1  Objects{
2          Object packet1; Object packet2;
3          Robot robot;
4          Location locA; Location locB; Location locC;
5          Location locD; Location locE; Location locF;
6          Location locG; Location locH; Location locI;
7  }
8
9  WorldState{
10         Connected(locA, locB); Connected(locB, locA);
11         Connected(locB, locC); Connected(locC, locB);
12         Connected(locC, locD); Connected(locD, locC);
13         Connected(locD, locE); Connected(locE, locD);
14         Connected(locE, locF); Connected(locF, locE);
15         Connected(locF, locC); Connected(locC, locF);
16         Connected(locB, locG); Connected(locG, locB);
17         Connected(locG, locH); Connected(locH, locG);
18         Connected(locH, locI); Connected(locI, locH);
19         Distance(locA, locB) = 2.0; Distance(locB, locA) = 2.0;
20         Distance(locB, locC) = 7.0; Distance(locC, locB) = 7.0;
21         Distance(locC, locD) = 10.0; Distance(locD, locC) = 10.0;
22         Distance(locD, locE) = 7.0; Distance(locE, locD) = 7.0;
23         Distance(locE, locF) = 4.0; Distance(locF, locE) = 4.0;
24         Distance(locF, locC) = 10.0; Distance(locC, locF) = 10.0;
25         Distance(locB, locG) = 8.0; Distance(locG, locB) = 8.0;
26         Distance(locG, locH) = 13.0; Distance(locH, locG) = 13.0;
27         Distance(locH, locI) = 3.0; Distance(locI, locH) = 3.0;
28 }
29
30 InitialState{
31         PosRobot(robot) = locA;
32         PosObject(packet1) = locG;
33         PosObject(packet2) = locE;
34         HasWhichObject(robot) = undefined;
35 }
36
37 Goal{
38         PosObject(packet1) == locI;
39         PosObject(packet2) == locC;
40 }
```

Listing 4.2 Définition des objets, du monde, de l'état initial et du but pour le problème du livreur de colis, selon le langage de spécification propre à ActuPlan.

```
1  a0 : Goto(robot,locA,locB);
2  a1 : Goto(robot,locB,locC);
3  a2 : Goto(robot,locC,locF);
4  a3 : Goto(robot,locF,locE);
5  a4 : Load(robot,locE,packet2);
6  a5 : Goto(robot,locE,locF);
7  a6 : Goto(robot,locF,locC);
8  a7 : Unload(robot,locC,packet2);
9  a8 : Goto(robot,locC,locB);
10 a9 : Goto(robot,locB,locG);
11 a10 : Load(robot,locG,packet1);
12 a11 : Goto(robot,locG,locH);
13 a12 : Goto(robot,locH,locI);
14 a13 : Unload(robot,locI,packet1);
15 a0 < a1;
16 a1 < a2;
17 a2 < a3;
18 a3 < a4;
19 a3 < a5;
20 a4 < a5;
21 a4 < a7;
22 a5 < a6;
23 a6 < a7;
24 a6 < a8;
25 a7 < a8;
26 a7 < a10;
27 a8 < a9;
28 a9 < a10;
29 a9 < a11;
30 a10 < a11;
31 a10 < a13;
32 a11 < a12;
33 a12 < a13;
```

Listing 4.3 Plan produit par ActuPlan pour le problème du livreur de colis au sein du monde discrétisé du laboratoire IntRoLab.

La figure 4.7 présente un exemple d'enregistrement des données issues d'une session d'expérimentation sur le robot IRL-1. Elle permet d'illustrer le fonctionnement de HBBA au sein du robot. On peut observer après environ 9 secondes d'expérimentation la présence d'un événement de type 5, signifiant que l'exploitation d'un désir a commencé, associée à un désir de type '*Goto*'.Ce premier événement correspond au démarrage du **nœud actuplan**, qui initialise le problème et démarre la planification. Suite à cette exécution, une première action (de type `Goto`) est reçue et le désir correspondant est émis presque instantanément. On observe par la suite, à un temps d'environ 15 secondes, un événement signifiant l'accom-

plissement du but associé au désir.Cet événement déclenche le processus de mise à jour de l'état courant du planificateur, et donc l'envoi de nouvelles actions et l'émission de nouveaux désirs.

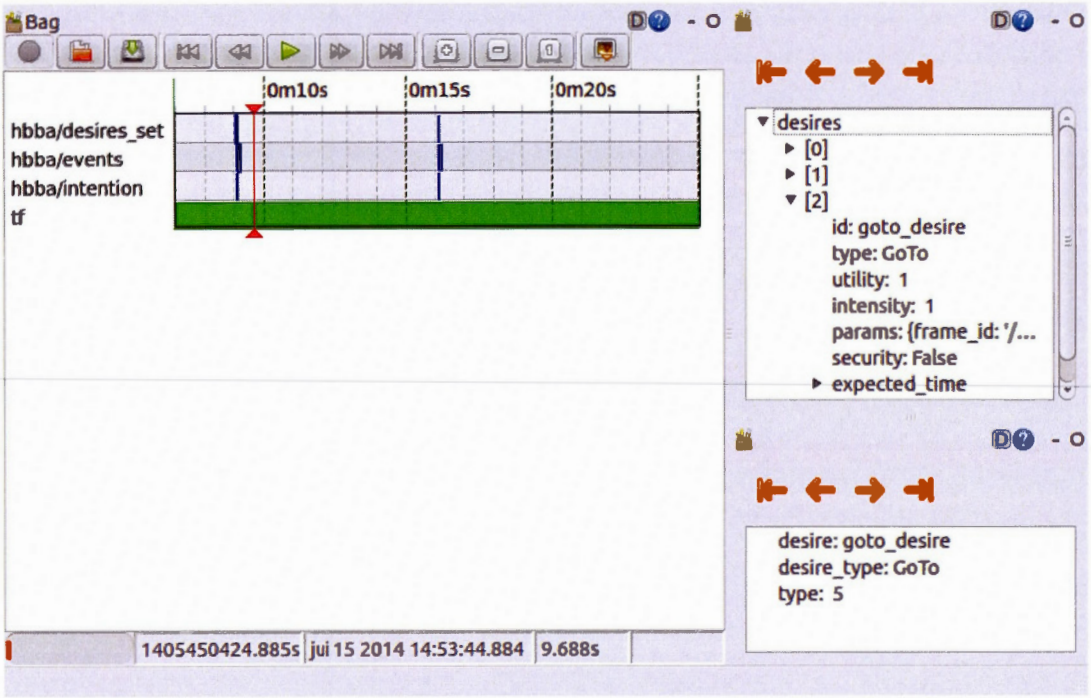

Figure 4.7 Capture de l'enregistrement des données issues de l'expérimentation du prototype sur le robot IRL-1. La partie de gauche présente une frise chronologique des données publiées sur certains *topics* sélectionnés. *hbba/desires_set* et *hbba/intention* représentent respectivement l'ensemble des désirs et des intentions du système, *hbba/events* retranscrit les différents événements relatifs à HBBA (par exemple, "un désir a été ajouté", "un désir a été satisfait", etc.) et *tf* représente les données odométriques du robot. La partie de droite présente le contenu de l'ensemble des désirs du système. On observe à l'instant sélectionné la présence d'un désir de type *Goto* (*déplacement*). En bas à droite se situent les informations relatives aux événements de HBBA.

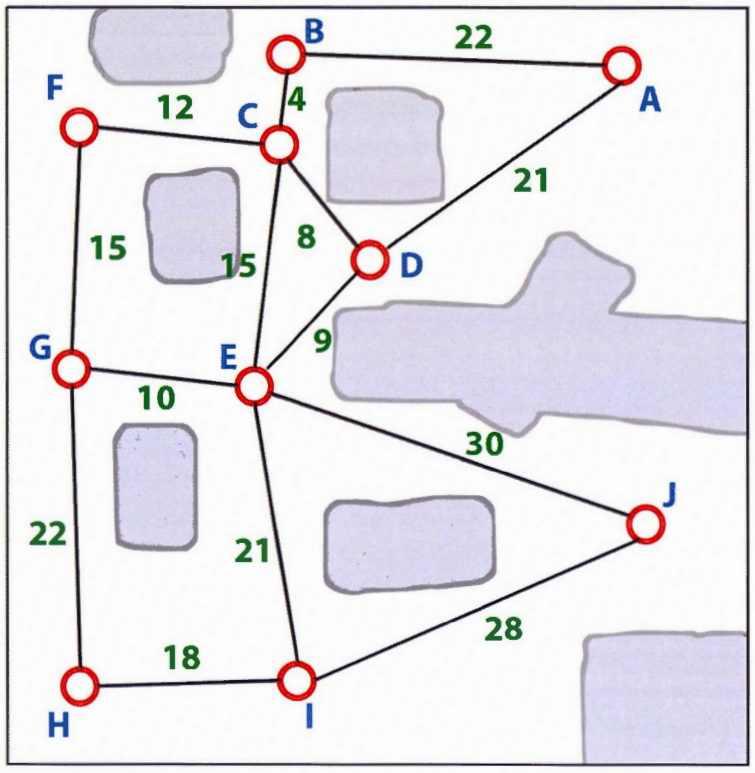

Figure 4.8 Carte exploitée avec le simulateur *Stage*, construite à partir d'une image Bitmap. Les distances affichées en vert sont en mètres.

4.6 Expérimentations en simulation avec *Stage*

Les expérimentations avec le simulateur ont été réalisées avec une carte différente de celle du laboratoire IntRoLab, afin de bénéficier d'une carte offrant une meilleure répartition des nœuds et un plus grand nombre d'arêtes par nœud, et cela afin de proposer un scénario qui reflète mieux les capacités du planificateur. La figure 4.8 présente la carte exploitée avec le simulateur *Stage*. Le listing 4.4 présente un extrait de la spécification du problème du livreur adapté à la nouvelle carte.

```
1  Objects{
2          Location A, B, C, D, E, F, I, J;
3          Object packet1, packet2;
4          Robot robot;
5  }
6  WorldState{
7          Connected(D, A);          Connected(A, D);
8          Connected(D, B);          Connected(B, D);
9          Connected(D, E);          Connected(E, D);
10         ...
11         Distance(D, A) = 36;      Distance(A, D) = 36;
12         Distance(D, B) = 29;      Distance(B, D) = 29;
13         Distance(D, E) = 37;      Distance(E, D) = 37;
14         ...
15 }
16 CurrentState{
17         PosRobot(robot) = I;
18         HasWhichObject(robot) = undefined;
19         PosObject(packet1) = C;
20         PosObject(packet2) = B;
21 }
22 Goal{
23         PosObject(packet1)==F;
24         PosObject(packet2)==A[124];
25 }
```

Listing 4.4 Définition du problème du livreur de colis dans le cadre des expérimentation avec le simulateur *Stage*.

4.6.1 Scénario sans replanification

Au départ de la simulation, HBBA est lancé et le paquet ROS *irl_actuplan* ouvre une connexion à travers un socket avec le programme AJS. Le domaine est chargé et les données du problème sont transmises à AJS à l'aide des commandes `SetObjects`, `SetState` et `SetGoal` présentées à la section 3.4.1.

Le listing 4.4 présente le problème du livreur exploité dans le cadre de la simulation avec le simulateur *Stage*. Le nombre indiqué entre crochets à la ligne 24 représente le temps limite à l'intérieur duquel doit être accompli l'objectif. Tel que présenté à la section 1.6.2, le planificateur ActuPlan prend en compte l'incertitude concernant la durée de réalisation des actions ainsi que leur temps limite, afin de calculer la probabilité de réussite du plan.

Après réception du but, le planificateur produit un premier plan présenté au listing 4.5. La probabilité de réussite du plan calculée est indiquée à la ligne 1. Le robot (HBBA et le paquet ROS *irl_actuplan*) envoie alors sa première commande `RequestActions` à AJS, qui renvoie en réponse la première action à effectuer (`Goto(robot,I,E)`). Lorsque le robot simulé termine l'action au sein du simulateur, il envoie la commande `SetTime` suivie du temps courant, ainsi que la commande `Completed Goto(robot,I,E)` afin de spécifier à AJS la terminaison de l'action. Le planificateur prend alors en considération la fin de l'action, évalue si le plan peut toujours être satisfait à partir du temps spécifié et, enfin, envoie la prochaine action à effectuer (`Load(robot,E,C)`) après réception d'une nouvelle commande `RequestActions` par le robot. Le listing 4.6 présente la suite de commandes envoyées par le robot lors de la simulation.

```
1  Plan found with cost=119.2013123463149; prob=0.822998046875
2  ----------
3  Plan {
4          a0 : Goto(robot,I,E);
5          a1 : Goto(robot,E,C);
6          a2 : Load(robot,C,packet1);
7          a3 : Goto(robot,C,F);
8          a4 : Unload(robot,F,packet1);
9          a5 : Goto(robot,F,C);
10         a6 : Goto(robot,C,B);
11         a7 : Load(robot,B,packet2);
12         a8 : Goto(robot,B,A);
13         a9 : Unload(robot,A,packet2);
14         a0 < a1;
15         a1 < a2;
16         a1 < a3;
17         a2 < a3;
18         a2 < a4;
19         a3 < a4;
20         a3 < a5;
21         a4 < a5;
22         a4 < a7;
23         a5 < a6;
24         a6 < a7;
25         a6 < a8;
26         a7 < a8;
27         a7 < a9;
28         a8 < a9;
29 }
```

Listing 4.5 Trace d'exécution d'Actuplan Java Server lors de la production du plan pour le problème du livreur de colis dans le cadre de la simulation avec le simulateur *Stage*.

```
1  SetTime 0
2  RequestActions
3  SetTime 28
4  Completed Goto(robot,I,E)
5  RequestActions
6  SetTime 48
7  Completed Goto(robot,E,C)
8  RequestActions
9  SetTime 49
10 Completed Load(robot,C,packet1)
11 RequestActions
12 SetTime 59
13 Completed Goto(robot,C,F)
14 RequestActions
15 SetTime 60
16 Completed Unload(robot,F,packet1)
17 RequestActions
18 ...
```

Listing 4.6 Exemple d'ensemble de commandes envoyées par le robot à AJS.

4.6.2 Scénario avec replanification

Malgré le fait qu'une limite de temps soit imposée à l'objectif `PosObject(packet2)==A[124]` et non à l'objectif `PosObject(packet1)==F`, on observe dans l'exemple précédent que le planificateur produit un plan qui satisfait les buts dans l'ordre suivant : livraison de `packet1` puis de `packet2`. Ceci est attendu, car les buts sont satisfaits en respectant la contrainte de temps et en choisissant le chemin le plus court, tout en respectant une probabilité de réussite supérieure au seuil choisi (ici 0.8).

Le listing 4.7 présente un ensemble de commandes envoyées par le robot à AJS résultant d'une expérimentation différente. On observe que le robot parvient au point `E` puis au point `C` dans un temps supérieur au cas précédent, par exemple en raison d'un obstacle rencontré ou d'un problème de planification de chemin. À la réception de la commande `Completed Goto(robot,E,C)`, le planificateur évalue si le plan est toujours valide. Dans ce cas-ci, le plan est invalidé, car la limite de temps sur le but `PosObject(packet2)==A[124]` ne respecte pas une probabilité de réussite satisfaisante (ici 0.498291015625). Le planificateur tente donc de replanifier ; un plan est alors trouvé satisfaisant d'abord l'objectif avec une contrainte de temps. La figure 4.8 présente la trace d'exécution de AJS lors de la validation du plan courant et de la replanification. L'exécution de la commande à la ligne 7 du listing 4.7 provoque le processus de replanification au sein de AJS.

```
1 SetTime 0
2 RequestActions
3 SetTime 32
4 Completed Goto(robot,I,E)
5 RequestActions
6 SetTime 62
7 Completed Goto(robot,E,C)
8 ...
```

Listing 4.7 Exemple d'ensemble de commandes envoyées par le robot à AJS entrainant une replanification.

```
1  ValidateCurrentPlan : probability 0.498291015625
2  Replanning ...
3    ----------
4  Plan found with cost=142.88008862543643; prob=1.0
5    ----------
6  Plan {
7   a0 : Goto(robot,E,C);
8   a1 : Goto(robot,C,B);
9   a2 : Load(robot,B,packet2);
10  a3 : Goto(robot,B,A);
11  a4 : Unload(robot,A,packet2);
12  a5 : Goto(robot,A,B);
13  a6 : Goto(robot,B,C);
14  a7 : Load(robot,C,packet1);
15  a8 : Goto(robot,C,F);
16  a9 : Unload(robot,F,packet1);
17  a0 < a1;
18  a1 < a2;
19  a1 < a3;
20  a2 < a3;
21  a2 < a4;
22  a3 < a4;
23  a3 < a5;
24  a4 < a5;
25  a4 < a7;
26  a5 < a6;
27  a6 < a7;
28  a6 < a8;
29  a7 < a8;
30  a7 < a9;
31  a8 < a9;
32 }
```

Listing 4.8 Validation du plan courant et replanification par Actuplan Java Server dans le cadre de la simulation avec le simulateur *Stage*.

Si à la suite de cela, le robot termine une nouvelle action à un temps ne permettant plus de satisfaire l'objectif `PosObject(packet2)==A[124]`, le planificateur échouera lors de la replanification, car plus aucun plan ne pourra satisfaire le but. Dans ce cas de figure, la mission du robot est tout simplement abandonnée.

4.7 Expérimentations à l'aide d'un module de test externe

Chaque expérimentation sur le robot réel peut prendre de 20 à 40 minutes. Cela s'explique par différents facteurs, comme la nécessité de lancer des scripts pour démarrer, réinitialiser et arrêter les modules du robot, de redéplacer le robot vers

son lieu initial, etc. De plus, il faut prévoir environ 2 à 3 heures pour recharger les batteries après 1 à 3 heures de tests. À travers le simulateur *Stage*, on peut réduire le temps de chaque expérimentation à environ 15 à 30 minutes. Réaliser un grand nombre d'expérimentations sur un robot réel dans *Stage* demanderait donc beaucoup de temps.

Afin d'obtenir un nombre suffisant de résultats issus d'expérimentations différentes, nous avons implémenté un module de test externe, indépendant de tout environnement de simulation robotique et se connectant à ActuPlan Java Server afin de simuler l'envoi de commandes et donc l'accomplissement des actions du plan. Le programme de test génère aléatoirement les temps d'exécution des différentes actions du robot, en se basant également sur le modèle probabiliste calculé depuis les données extraites des expérimentations sur le robot IRL-1.

4.7.1 Protocole d'expérimentation

Le domaine de planification utilisé est le même que celui présenté précédemment au listing 4.4. Ce problème consiste à déplacer un colis du point du point `C` au point `F` et un deuxième colis du point `B` au point `A`, avec une contrainte de temps sur le deuxième colis. Quel que soit le but accompli en premier, le plan fait se déplacer le robot au point `C`. Arrivé au point `C`, le reste du plan change dépendamment du colis livré en premier. Nous avons donc identifié les trois (3) chemins correspondant aux segments du plan propres à la livraison de chacun des colis. On note ainsi le segment `CF`, le segment `CB` et le segment `BA`. Nous observons par la suite le nombre de fois que chacun de ces segments a été emprunté par le robot. Si le robot livre en premier le paquet 1, le plan est défini par un coût optimal de 86 mètres. Dans le cas où le robot livre en premier le paquet 2, le coût est de 100 mètres.

D'autre part, nous réalisons l'expérimentation avec trois versions différentes du

planificateur ActuPlan :

1. ActuPlan déterministe et pessimiste, qui surestime la durée des actions ;
2. ActuPlan déterministe et optimiste, qui assume une durée moyenne des actions ;
3. ActuPlan normal, qui utilise des variables aléatoires pour représenter les temps.

Dans ces tests, nous faisons varier le temps limite d'arrivée du paquet 2 à A à 105 et à 124 secondes. Nous avons exécuté ces tests 10 000 fois et calculé les taux de succès du plan et le coût des exécutions de plan réussis.

4.7.2 Résultats

Le tableau 4.1 présente les résultats obtenus avec un temps limite de 124 secondes. On observe un très faible taux de succès avec la version optimiste du planificateur. Ce dernier génère des plans conduisant le robot à livrer le paquet 1 (sans contrainte de temps) en premier, car il s'agit du plan offrant la durée totale la plus courte. Cela se reflète dans la durée moyenne des plans. Cependant, lorsque le robot utilise la version optimiste, il manque souvent de temps pour livrer le paquet 2 à temps. Quant aux deux autres versions du planificateur, elles arrivent à produire des plans adéquats. Dans le cas de ActuPlan normal, le planificateur génère des plans qui commencent à livrer le paquet 2, car il s'agit de l'unique façon de garantir une probabilité de succès supérieure à 0.8.

Le tableau 4.2 présente les résultats obtenus avec un temps limite de 105 secondes. Dans ce cas-ci, c'est le planificateur pessimiste qui éprouve des difficultés, avec un taux de succès relativement faible (79 %). En effet, à mesure que l'exécution du plan progresse, le planificateur pessimiste estime qu'il manquera de temps pour terminer son plan, ce qui le mène à conclure qu'il n'existe pas de plan pour

Tableau 4.1 Résultats avec un temps limite de 124 secondes sur la livraison du paquet 2

	CF	CB	BA	Taux de succès	Durée moyenne
ActuPlan déterministe optimiste	16030	10561	10344	61.7%	130.4
ActuPlan déterministe pessimiste	9942	19911	19909	99.42%	143.4
ActuPlan	9938	19917	19915	99.4%	143.3

satisfaire ses objectifs.

Tableau 4.2 Résultats avec un temps limite de 105 secondes sur la livraison du paquet 2

	CF	CB	BA	Taux de succès	Durée moyenne
ActuPlan déterministe optimiste	9245	19175	19147	92.5%	141.7
ActuPlan déterministe pessimiste	7892	16016	15997	78.9%	139.53
ActuPlan	8208	16835	16748	82.1%	140.3

Enfin, les résultats présentés montrent que la version normale du planificateur ActuPlan se comporte généralement mieux que les deux autres. Cela démontre la pertinence d'utiliser un planificateur, comme ActuPlan capable de considérer l'incertitude pendant le processus de génération des plans.

CONCLUSION

Les évolutions technologiques dans le domaine de la robotique encouragent le développement de systèmes rendant les machines toujours plus autonomes. Il s'agit d'un problème auquel le domaine de la planification en intelligence artificielle cherche à apporter une réponse. Cependant, la complexité du monde dans lequel évolue un robot oblige les chercheurs à faire des compromis et à développer des planificateurs contraints par les hypothèses aujourd'hui définies par la planification classique, et ainsi proposer des solutions à des problèmes spécifiques. Dans ce contexte, le planificateur ActuPlan vise à proposer une solution pertinente au problème de planification d'actions concurrentes avec incertitude sur le temps.

Un tel planificateur présente un intérêt particulier lors de son utilisation combinée avec un robot.Or ce travail d'intégration n'avait pas encore été réalisé. L'objectif de ce mémoire était d'améliorer le planificateur ActuPlan, un planificateur de type *offline*, afin de le rendre de type *online* et ainsi démontrer son utilité lors d'une utilisation combinée avec un robot mobile.

Dans ce mémoire nous avons donc présenté le travail d'intégration du planificateur ActuPlan comme planificateur de tâches de haut niveau au sein d'une architecture robotique hybride. Il a été démontré qu'une telle intégration pouvait être réalisée en conservant l'efficacité du planificateur et sans transformer l'architecture exploitée par le robot. À l'aide d'expérimentations réalisées à la fois en simulation et sur une machine réelle, a également été démontré l'apport d'un planificateur de haut niveau perméable à l'incertitude pour un robot mobile ayant un certain nombre d'objectifs à accomplir.

À l'avenir, plusieurs pistes pourraient être explorées afin d'améliorer l'intérêt de l'utilisation d'ActuPlan avec un robot. Dans un premier temps pourrait être exploitée la capacité d'ActuPlan à produire des plans de type non conditionnels à la place de plans conditionnels optimaux. Dans un deuxième temps, il pourrait être intéressant d'intégrer une mécanique de stabilité de plan (Fox *et al.*, 2006) lors du processus de replanification, afin de conserver une cohérence dans les actions déjà réalisées par le robot et celles introduites par un nouveau plan après replanification.

RÉFÉRENCES

Arkin, R. C. (1989). Motor schema—based mobile robot navigation. *The International journal of robotics research*, *8*(4), 92–112.

Arkin, R. C. (1998). *Behavior-based robotics*. MIT press.

Barto, A. G., Bradtke, S. J. et Singh, S. P. (1995). Learning to act using real-time dynamic programming. *Artificial Intelligence*, *72*(1), 81–138.

Beaudry, É., Brosseau, Y., Côté, C., Raïevsky, C., Létourneau, D., Kabanza, F. et Michaud, F. (2005). Reactive planning in a motivated behavioral architecture. Dans *Proceedings of the National Conference on Artificial Intelligence*, volume 20, p. 1242.

Beaudry, É., Kabanza, F. et Michaud, F. (2010a). Planning for concurrent action executions under action duration uncertainty using dynamically generated bayesian networks. Dans *Proceedings of the International Conference on Automated Planning and Scheduling*, 10–20. AAAI Press.

Beaudry, É., Kabanza, F. et Michaud, F. (2010b). Planning with concurrency under resources and time uncertainty. Dans *Proceedings of the European Conference on Artificial Intelligence*, 217–222. IOS Press.

Beaudry, É., Kabanza, F. et Michaud, F. (2012). Using a classical forward search to solve temporal planning problems under uncertainty. Dans *Proceedings of the Association for the Advancement of Artificial Intelligence Workshops*, 2–8.

Biggs, G., Rusu, R., Collett, T., Gerkey, B. et Vaughan, R. (2013). All the robots merely players : History of player and stage software. *Robotics Automation Magazine, IEEE*, *20*(3), 82–90.

Blum, A. L. et Furst, M. L. (1997). Fast planning through planning graph analysis. *Artificial intelligence*, *90*(1), 281–300.

Bonet, B. et Geffner, H. (2003). Labeled RTDP : Improving the convergence of real-time dynamic programming. Dans *Proceedings of the International Conference on Automated Planning and Scheduling*, 12–21.

Bonet, B., Loerincs, G. et Geffner, H. (1997). A robust and fast action selection mechanism for planning. Dans *AAAI/IAAI*, 714–719.

Bresina, J., Dearden, R., Meuleau, N., Ramakrishnan, S., Smith, D. *et al.* (2002). Planning under continuous time and resource uncertainty : A challenge for ai. Dans *Proceedings of the Conference on Uncertainty in Artificial Intelligence*, 77–84. Morgan Kaufmann Publishers Inc.

Côté, C., Brosseau, Y., Letourneau, D., Raïevsky, C. et Michaud, F. (2006). Robotic software integration using marie. *International Journal of Advanced Robotic Systems*, *3*(1), 55–60.

Dijkstra, E. W. (1971). *A short introduction to the art of programming*, volume 4. Technische Hogeschool Eindhoven.

Eugster, P. T., Felber, P. A., Guerraoui, R. et Kermarrec, A.-M. (2003). The many faces of publish/subscribe. *ACM Computing Surveys (CSUR)*, *35*(2), 114–131.

Ferland, F. (2011). HBBA - Hybrid Behavior-Based Architecture. Récupéré de `https://introlab.3it.usherbrooke.ca/mediawiki-introlab/index.php/HBBA`

Fikes, R. E. et Nilsson, N. J. (1972). Strips : A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, *2*(3), 189–208.

Fox, M., Gerevini, A., Long, D. et Serina, I. (2006). Plan stability : Replanning versus plan repair. Dans *Proceedings of the International Conference on Automated Planning and Scheduling*, 212–221. AAAI Press.

Gerkey, B., Vaughan, R. T. et Howard, A. (2003). The player/stage project : Tools for multi-robot and distributed sensor systems. Dans *Proceedings of the International Conference on Advanced Robotics*, 317–323.

Ghallab, M., Nau, D. et Traverso, P. (2004). *Automated planning : theory & practice*. Morgan Kaufmann.

Hart, P. E., Nilsson, N. J. et Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *Systems Science and Cybernetics, IEEE Transactions on*, *4*(2), 100–107.

Hoffmann, J. (2001). Ff : The fast-forward planning system. *AI magazine*, *22*(3), 57.

Konolige, K. et Myers, K. (1998). The Saphira architecture for autonomous mobile robots. *Artificial Intelligence and Mobile Robots : case studies of successful robot systems*, *9*, 211–242.

Labranche, S. et Beaudry, É. (2014). Partial satisfaction planning under time uncertainty with control on when objectives can be aborted. Dans *Proceedings of the Canadian Conference on Artificial Intelligence*, 167–178.

LaValle, S. M. (2006). *Planning algorithms*. Cambridge University Press.

Lemai, S. et Ingrand, F. (2004). Interleaving temporal planning and execution in robotics domains. Dans *Proceedings of the National conference on Artifical Intelligence*, 617–622. AAAI Press.

Matarić, M. J. (1998). Behavior-based robotics as a tool for synthesis of artificial behavior and analysis of natural behavior. *Trends in cognitive sciences*, *2*(3), 82–86.

Mausam et Weld, D. S. (2004). Solving concurrent Markov decision processes. Dans *Proceedings of the National Conference on Artifical Intelligence*, 716–722. AAAI Press.

Mausam et Weld, D. S. (2005). Concurrent probabilistic temporal planning. Dans *Proceedings of the International Conference on Automated Planning and Scheduling*, 120–129. AAAI Press.

Mausam et Weld, D. S. (2006). Probabilistic temporal planning with uncertain durations. Dans *Proceedings of the National Conference on Artificial Intelligence*, 880–887. AAAI Press.

McDermott, D., Ghallab, M., Howe, A., Knoblock, C., Ram, A., Veloso, M., Weld, D. et Wilkins, D. (1998). Pddl-the planning domain definition language.

Michaud, F. (2002). EMIB—computational architecture based on emotion and motivation for in-tentional selection and configuration of behaviour-producing modules.

Peter Bonasso, R., James Firby, R., Gat, E., Kortenkamp, D., Miller, D. P. et Slack, M. G. (1995). Experiences with an architecture for intelligent, reactive agents. *Journal of Experimental & Theoretical Artificial Intelligence*, *9*(2-3), 237–256.

Puterman, M. L. (2009). *Markov decision processes : discrete stochastic dynamic programming*, volume 414. John Wiley & Sons.

Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R. et Ng, A. Y. (2009). Ros : an open-source robot operating system. Dans *International Conference on Robotics and Automation workshop on open source software*, volume 3, p. 5.

Russell, S. J. et Norvig, P. (2010). *Artificial Intelligence : A Modern Approach* (troisième éd.). Prentice Hall.

Sacerdoti, E. (1974). Planning in a hierarchy of abstraction spaces. *Artificial Intelligence, 173*(5), 115–135.

Wooldridge, M. et Jennings, N. R. (1995). Intelligent agents : Theory and practice. *The knowledge engineering review, 10*(02), 115–152.