Une architecture de contrôle distribuée pour l'autonomie des robots

Arnaud Degroote

Université de Toulouse

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par :
Institut National Polytechnique de Toulouse (INP Toulouse)

Discipline ou spécialité :

Systèmes embarqués

Présentée et soutenue par :
Arnaud Degroote

le : vendredi 5 octobre 2012

Titre :

Une architecture de contrôle distribuée pour l'autonomie des robots

---

Ecole doctorale :
Systèmes (EDSYS)

Unité de recherche :
LAAS - CNRS

Directeur(s) de Thèse :

Simon Lacroix

Rapporteurs :

Herman Bruyninckx
Jacques Malenfant

Membre(s) du jury :
Rachid Alami
Marie-Pierre Gleize
Simon Lacroix
Charles Lesire-Cabaniols
Bruno Patin
Serge Stinckwich

Table des matières

Table des figures

Remerciements

Ce manuscrit est le résultat de quatre années de travail, au LAAS-CNRS, à Toulouse. Par ces quelques mots, je souhaite remercier tous les gens qui ont rendu cette expérience passionnante, autant du point de vue scientifique qu'humain.

Tout d'abord, merci à mon directeur de thèse, Simon Lacroix, pour m'avoir donné ma chance et m'avoir toujours suivi dans mes travaux.

Merci à mes rapporteurs, Jacques Malenfant, et Herman Bruyninckx, pour leurs précieuses relectures, chacun ayant apporté un éclairage différent sur mes travaux. Merci aussi à Bruno Patin pour sa relecture très précise et les idées que nous avons pu échangées par la suite.

Merci à tous ceux qui ont passé du temps avec moi sur le terrain, avec le robot, dans des conditions parfois difficiles. Sûrement la partie la plus difficile, mais aussi la plus riche. Merci donc à Simon, Matthieu Herbb, Matthieu Warnier, Cyril Roussillon, Cyril Robin, Red, Hung et Vivien.

Merci aussi à tous ceux avec qui j'ai eu l'occasion de discuter, à côté d'un café, ou au coin d'un couloir. Merci en particulier à Séverin pour nos discussions souvent flamboyantes. Merci aussi à toute la team BSD du LAAS.

Enfin, merci à ma famille pour m'avoir toujours soutenue. Merci à Sophie pour m'avoir remonter le moral dans les périodes où il était le plus bas, et aussi pour me faire décrocher quand le travail devenait trop envahissant.

Abstract

For simple tasks in a controlled environment, the coordination of the internal processes of a robot is a relatively trivial task, often implemented in an ad-hoc basis. However, with the development of more complex robots that must operate in uncontrolled and dynamic environments, the robot must constantly reconfigure itself to adapt to the external conditions and its own goals. The definition of a control architecture to manage these reconfigurations becomes of paramount importance for the autonomy of such robots.

In this work, we first study the different architectures proposed in the literature, and analyse the major issues that a control architecture must address. This analysis led us to propose a new architecture, decentralized, generic and reusable, integrating an artificial intelligence approach (use of logical reasoning, dynamic propagation of constraints) and a software engineering approach (programming by contract, agents). After a presentation of the concepts underlying this architecture and an in-depth description of its operation, we describe an implementation which is used to control of a ground robot for navigation, exploration and monitoring tasks. Results are presented and analyzed.

In a second part, we focus on the modeling and verifiability of such a control architecture. After analyzing different solutions, we present a comprehensive model of the proposed architecture that uses linear logic. We then discuss the different possible approaches to assess the properties of reachability and safety within this model.

Finally we discuss different ways to enrich this work. In particular, we discuss possible extensions to the control of a multiple cooperating robots, but also the need for stronger links between the control layer and the modeling.

Résumé

Pour des tâches simples ou dans un environnement contrôlé, la coordination des différents processus internes d'un robot est un problème relativement trivial, souvent implémenté de manière ad-hoc. Toutefois, avec le développement de robots plus complexes travaillant dans des environnements non contrôlés et dynamiques, le robot doit en permanence se reconfigurer afin de s'adapter aux conditions extérieures et à ses objectifs. La définition d'une architecture de contrôle efficace permettant de gérer ces reconfigurations devient alors primordiale pour l'autonomie de tels robots.

Dans ces travaux, nous avons d'abord étudié les différentes architectures proposées dans la littérature, dont l'analyse a permis d'identifier les grandes problématiques qu'une architecture de contrôle doit résoudre. Cette analyse nous a mené à proposer une nouvelle architecture de contrôle décentralisée, générique et réutilisable, selon une démarche qui intègre une approche "intelligence artificielle" (utilisation de raisonneur logique, propagation dynamique de contraintes) et une approche "génie logiciel" (programmation par contrats, agents). Après une présentation des concepts qui sous-tendent cette architecture et une description approfondie de son fonctionnement, nous en décrivons une implémentation, qui est exploitée pour assurer le contrôle d'un robot terrestre d'extérieur dans le cadre de tâches de navigation, d'exploration ou de suivi. Des résultats sont présentés et analysés.

Dans une seconde partie, nous nous sommes penchés sur la modélisation et la vérifiabilité d'une telle architecture de contrôle. Après avoir analysé différentes solutions, nous décrivons un modèle complet de l'architecture qui utilise la logique linéaire. Nous discutons ensuite des différentes approches possibles pour montrer des propriétés d'atteignabilité et de sûreté de fonctionnement en exploitant ce modèle.

Enfin nous abordons différentes voies d'enrichissement de ces travaux. En particulier, nous discutons des extensions possibles pour le contrôle d'un ensemble de robots coopérants entre eux, mais aussi de la nécessité d'avoir des liens plus forts entre cette couche de contrôle, et les approches de modélisation des fonctionnalités sous-jacentes.

Chapitre 1

Introduction

Motivations

Ces dernières années ont vu apparaitre sur le marché de nombreux petits robots permettant d'automatiser des tâches simples : robot aspirateurs, robots tondeuses, robots de piscine ... Dans le même temps, dans les laboratoires, on a pu voir des robots réalisant des tâches plus complexes tels que PR2 allant chercher des bières dans un réfrigérateur, ou bien Rosie préparant des crêpes. Ces démonstrations impressionnantes sont en fait réalisées dans un environnement finement contrôlé, réduisant de fait l'autonomie réelle du robot. Actuellement, on peut donc trouver d'un côté des robots relativement autonomes mais réalisant des tâches très simples, et de l'autre des robots réalisant des tâches complexes, mais avec une autonomie assez faible.

Il existe de nombreuses raisons à cette dissymétrie. Un robot aspirateur va utiliser des modèles extrêmement simples, sans planification de tâche ni de trajectoires. Il se contente de réagir efficacement aux stimuli de l'environnement, tout en réalisant sa tâche principale : aspirer la poussière. Au contraire, la préparation d'une crêpe demande beaucoup plus d'informations, des modèles plus complexes, une planification des différentes sous-tâches à effectuer. Mais afin de rendre le problème aisé à résoudre, un certain nombre d'hypothèses fortes sont faites sur les modèles (localisation et types des objets connus à priori, environnement fixe, un seul humain, ...), réduisant par là même l'autonomie réelle du robot.

Ce manque d'autonomie n'est pas uniquement lié à la complexité intrinsèque des tâches à réaliser, mais aussi à l'organisation des différentes briques logiques utilisées. En effet, dans ces démonstrations, les différents éléments sont en général configurés et connectés de manière statique, sans possibilités réelles d'évolutions. Si cela s'avère suffisant pour des tâches ou des environnements simples (ou contrôlés), cela reste insuffisant pour un robot versatile et autonome dans un environnement complexe et dynamique. Dans un tel environnement, le robot doit non seulement posséder des algorithmes efficaces, mais surtout les choisir et les coordonner de la manière la plus appropriée en fonction de l'environnement : son autonomie ne dépend pas uniquement des algorithmes qu'il embarque et de la qualité de ses composants logiciels, mais surtout de la possibilité de se reconfigurer afin de choisir les techniques les plus efficientes dans un environnement donné.

Point de vue de cette thèse

C'est selon cet axe organisationnel que nos travaux sont orientés. Dans nos travaux, nous avons essayé d'analyser les relations entre les différents composants et comment ils peuvent être organisés et (surtout) réorganisés en fonction du contexte courant. De cette analyse, nous essayons de déduire un certain nombre de grands principes qui sont à la base d' une *architecture de contrôle*. Une telle architecture a non seulement pour vocation de lier dynamiquement les différents composants entre eux mais aussi de proposer une sorte de guide conceptuel pour le(s) développeur(s) lors du développement d'un robot autonome.

Souvent, de telles architectures sont centrées sur la considération des actions du robot. Nous proposons une approche différente, en nous concentrant sur la notion d'information et les relations entre ces informations. L'avantage de cette décomposition est double. Tout d'abord, la notion d'information est indépendante d'une implémentation de robot donnée, contrairement à la réalisation d'une action. Se concentrer sur la notion d'information permet donc à priori d'apporter une solution plus générique. En second lieu, les composants sous-jacents sont majoritairement définis par les informations qu'ils prennent en entrée et les informations qu'ils produisent. Construire une architecture de contrôle sur cette notion d'information permet donc de faire un lien plus direct, plus simple à comprendre avec les composants qu'elle doit contrôler.

Lors du développement de cette architecture, nous avons pris soin de respecter différents principes de génie logiciel, souvent négligés par d'autres travaux. En premier lieu, la séparation des problèmes. Notre architecture se concentre uniquement sur la problématique de coordination des différents composants et ne traite pas des problèmes de planification (comment réaliser cette tâche) ou d'analyse de la situation (comment interpréter la situation actuelle). Ces problèmes sont, du point de vue de notre analyse, du ressort de composants sous-jacents. Nous avons aussi essayé de considérer la problématique de la modularité et de la composabilité de l'architecture de contrôle. Les composants que l'architecture de contrôle coordonne sont développés de manière relativement indépendante les uns les autres, et peuvent être réutilisés dans différents contextes. Il en doit être de même pour l'architecture de contrôle. Cela passe en particulier par la décomposition de l'architecture en un ensemble d'agents qui échangent des messages pour produire une politique de contrôle globale. Chaque agent peut alors être développé de manière indépendante des autres agents, aidant ainsi à la modularité de l'architecture.

Enfin, nous nous sommes efforcés de construire une solution sur laquelle il était possible de raisonner globalement. Pour cela, nous avons essayé de limiter le nombre de mécanismes qui composent cette architecture, et de définir de manière claire les différentes règles qui s'appliquent. Cela rend alors la possibilité de raisonner au niveau système plus simple. Dans un second temps, nous nous sommes concentrés sur la possibilité de formaliser l'architecture, afin de mécaniser le raisonnement sur l'architecture complète. Cela permet alors d'aider le développeur dans la construction d'une couche de contrôle pour un robot donné, en détectant différents types d'erreurs au cours du développement (et pas uniquement durant les phases de tests d'intégration).

Contributions

Cette thèse présente trois principales contributions :

– Dans un premier temps, nous avons défini une architecture de contrôle distribuée, générique et versatile. Elle permet de contrôler dynamiquement un ensemble de composants sous-jacents afin de s'adapter au mieux à la situation courante, que ce soit en terme d'environnement, mais aussi en fonction de l'historique de la tâche.
– Un second travail important a consisté à proposer une implémentation efficace de cette architecture, puis de définir avec son aide la couche de contrôle d'un robot terrestre doté de capacités de navigation autonome. La solution résultante a alors été évaluée à la fois en simulation et sur un robot réel, qui est désormais toujours contrôlé par cette implémentation.
– Enfin, une dernière contribution concerne la formalisation de l'architecture proposée. Cette formalisation originale basée sur la logique linéaire permet de raisonner de manière unifiée sur notre architecture : elle pourra notamment servir comme aide au développement, en permettant de vérifier si certaines situations sont atteignables ou non.

Organisation du document

Ce document est divisé en deux grandes parties. La première partie est consacrée à la définition d'une architecture de contrôle. Cette partie est structurée en quatre chapitres :

– Le chapitre 2 analyse le problème du contrôle d'un robot autonome. Il définit de manière plus précise la notion d'architecture de contrôle, puis les exigences que nous lui associons. Notre contexte de travail est ensuite précisé, *i.e.* l'environnement matériel et logiciel avec lequel nous allons travaillé et le type de missions que nous avons considérés (navigation autonome). Ce contexte sera exploité pour illustrer le reste de cette partie.
– Le chapitre 3 recense et analyse les principales architectures de contrôle proposées dans la littérature, en particulier vis-à-vis des exigences précédemment explicitées.
– Le chapitre 4 définit de manière précise l'architecture proposée. Après une vue haut niveau de notre solution, nous descendons au niveau des agents pour décrire finement leur comportement. Ensuite, nous étudierons les interactions entre ces différents agents, en particulier pour gérer les fautes ou les conflits.
– Le chapitre 5 revient sur l'implémentation de l'architecture proposée. Dans un premier temps, il explicite les choix d'implémentation guidés par des critères de performance et de robustesse. Dans un second temps, il s'intéresse au développement d'une couche de contrôle pour un robot terrestre. Enfin, dans une dernière partie, plusieurs expérimentations sont effectuées pour montrer les interactions entre les différents agents, et les performances globales du système.

La seconde partie du manuscrit concerne elle les aspects liés à la formalisation de notre architecture. Elle est composée de trois chapitres :

– Dans un premier temps, le chapitre 6 expose les différents bénéfices que nous offre une formalisation de l'architecture. Puis, il s'intéresse et compare différents modèles présentés

dans la littérature qui pourraient être exploités pour la formalisation de notre architecture.
- Le chapitre 7 définit le modèle basé sur la logique linéaire que nous avons choisi. Il présente ensuite comment cette approche formelle permet de modéliser de manière fine notre architecture de contrôle.
- Enfin, le chapitre 8 propose une prospective sur l'utilisation de cette formalisation pour la validation. En particulier, nous nous intéresserons aux problèmes d'atteignabilité et à la vérification de propriétés de sûreté.

Une conclusion générale termine le manuscrit, et recense les différentes problématiques ouvertes et perspectives concernant l'architecture et la formalisation développées.

Première partie

Une architecture de contrôle

Chapitre 2

Problématique

Dans ce chapitre, nous allons expliciter ce que nous appelons architecture de contrôle et les exigences associées à une telle architecture. Dans un second temps, nous décrivons le contexte d'étude de cette thèse, qui servira de fil d'Ariane tout au long du manuscrit.

2.1 Architecture de contrôle

Nous allons définir ce qu'est une architecture de contrôle. Pour cela, nous nous basons sur l'analyse 4C, proposé par M. Radestock, sur les systèmes distribués et nous faisons le parallèle avec le domaine de la robotique. Dans un second temps, nous explicitons les exigences propres à la robotique d'une architecture de contrôle.

2.1.1 L'analyse 4C et son impact sur les architectures robotiques

Analyse 4C Dans (Radestock and Eisenbach, 1996), M. Radestock revient sur le développement des systèmes distribués. Afin de faciliter le développement de tels systèmes, il propose de diviser le problème en quatre grandes sous-parties :

- la partie Communication, où est définie la manière dont les différents composants peuvent échanger des informations entre eux
- la partie Calcul définit comment est implanté le comportement de chaque composant, et donc détermine ce qui va être échangé entre les composants
- la partie Configuration décrit quels sont les composants dans le système et avec quels composants ils peuvent communiquer : il s'agit de savoir où sont les informations et où elles peuvent être envoyées.
- la partie Coordination définit les schémas d'interactions entre les composants, en particulier dans quelles conditions ils peuvent communiquer.

Architecture fonctionnelle Les deux premiers points ont été largement couverts pour le domaine de la robotique et correspondent à ce qu'on appelle classiquement la couche fonctionnelle d'un robot. De nombreuses solutions ont été proposées et il semble maintenant acquis que les cadres logiciels basés sur des composants sont une des meilleures solutions pour répondre à des besoins d'expérimentations rapides, mais aussi de modularité et de réusabilité (on peut citer GenoM (Fleury et al., 1997), OROCOS (Bruyninckx, 2001), OpenRTM (Ando et al., 2005), ORCA (Brooks et al., 2007) ou ROS (Quigley et al., 2009) – *e.g.* le lecteur pourra trouver une revue plus pointue de ces différents cadres logiciels dans (Shakhimardanov et al., 2010) ou (Brugali and Shakhimardanov, 2010) ainsi que dans l'ouvrage (Brugali, 2007)). Toutefois, souvent, ces deux problématiques restent entremêlées, chaque cadre logiciel utilisant son propre protocole de communication pour échanger des informations. Orocos, et plus récemment Genom3 (Mallet et al., 2010) proposent un plus grand découplage entre ces deux problématiques.

Architecture de contrôle Bien que ces systèmes permettent d'assembler différents composants, ils ne proposent généralement pas de moyens de les contrôler finement, *i.e.* de configurer, d'ordonnancer, de déclencher et suivre l'exécution de ces différents composants. Comme le montre (Coste-Manière and Simmons, 2000; Coste-Manière and Espiau, 1998), cette notion d'assemblage et de contrôle est primordiale pour la bonne réalisation de missions autonomes. De nombreuses solutions ont été proposées pour le domaine de la robotique sous divers noms (elles seront détaillées et analysées dans le chapitre 3) : *architecture décisionnelle*, *architecture de supervision*, *architecture de contrôle* ... Dans la suite du manuscrit, nous utiliserons le terme d'architecture de contrôle. Selon l'analyse de M. Radestock, il s'agit principalement de la tâche de coordination, avec en filigrane une partie configuration, souvent relativement statique.

2.1.2 Exigences pour une architecture de contrôle

Une architecture de contrôle a donc pour vocation de coordonner les différents éléments de la couche fonctionnelle afin d'effectuer différents types de missions, tout en ayant la capacité de réagir efficacement à différents types d'évènements, le tout dans un monde difficilement prédictible. Pour cela, elle doit considérer et traiter les problématiques suivantes :

- **Réaction et délibération.** Un robot autonome évolue dans un environnement complexe et dynamique et doit donc réagir en conséquence. Dans le même temps, pour réussir des missions nécessitant des comportements complexes, il doit être capable de considérer des modèles et des plans à plus long terme. L'architecture doit donc permettre de prendre en compte ces différents niveaux de raisonnement.

- **Gestion de la concurrence.** Un robot autonome est un système hautement concurrent, où de nombreuses tâches parallèles vont potentiellement accéder et/ou modifier un nombre fini de ressources, tant logicielles que matérielles. Le robot, au travers de l'architecture de contrôle doit être capable de raisonner sur son état global courant et sur la mission en cours afin d'assurer un comportement efficace et cohérent, en particulier en évitant les conflits sur ces différentes ressources.

– **Robustesse.** La majorité des fonctionnalités à bord d'un robot peuvent échouer (en particulier l'architecture de contrôle), soit pour des raisons internes (erreurs dans l'implémentation d'un algorithme, algorithme inapproprié pour la situation, échec d'un des capteurs ou actionneurs, ...), soit pour des raisons liées à l'environnement physique du robot. Toutefois, dans tous les cas, l'échec d'un des composants ne doit pas entraîner l'échec de la mission courante : l'architecture doit donc être capable de détecter et de raisonner sur les différentes causes possibles d'échecs, et si possible de trouver une solution alternative pour achever la mission.

– **Vérification.** Un robot autonome est un système complexe, agissant dans un monde complexe. Pour garantir la sécurité des hommes interagissant avec le robot, ou bien l'intégrité du robot, il est nécessaire de pouvoir certifier certaines propriétés du robot telle que "le robot va-t-il toujours s'arrêter avant de heurter un humain ?". Traditionnellement, les développeurs utilisent des tests pour se convaincre que de telles propriétés sont vérifiées par le système, mais les tests peuvent difficilement couvrir tous les cas possibles dans un environnement complexe et dynamique. Ils ne peuvent donc qu'augmenter la confiance dans le système, pas le garantir. Il est donc important qu'une telle architecture soit modélisable afin de lui appliquer des méthodes formelles de vérification.

– **Modularité et composition.** Le développement d'un robot versatile est souvent un processus long et itératif et dépendant de nombreux acteurs. Il est donc primordial d'avoir une architecture aussi modulaire et composable que possible : l'introduction de nouveaux comportements ne doit, dans la mesure du possible, ni casser les assemblages existants, ni demander de réécriture majeure de ceux-ci. De plus, il doit être possible d'ajouter ou de supprimer dynamiquement des fonctionnalités. Ces propriétés existent au niveau de la couche fonctionnelle et doivent être étendues à l'architecture de contrôle.

Au final, le but d'une architecture de contrôle est donc de faciliter le travail des développeurs en leur proposant un cadre de travail assez expressif pour définir des schémas de contrôle et les stratégies de récupérations spécifiques à certaines fautes, tout en étant capable de raisonner sur elle-même, afin de réparer, de manière la plus automatique possible, les erreurs liées à la concurrence ou à une faute du système.

2.2 Contexte d'étude

Nous allons maintenant décrire dans quel cadre nous travaillons. Ce contexte servira de fil rouge au cours de ce manuscrit, ancrant dans un cas concret les problématiques posées précédemment, et permettant d'illustrer les différents mécanismes de notre architecture.

On considère le cas d'un robot terrestre autonome (ou UGV [1]) tel que celui présenté dans

1. pour Unmanned Ground Vehicle

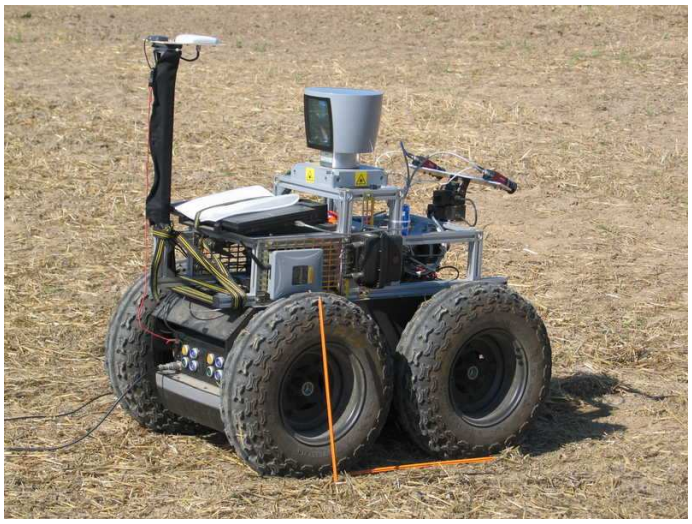

FIGURE 2.1 – Le robot Mana

la figure 2.1. Il a vocation à mener différents types de missions, telles l'exploration d'une zone inconnue, la recherche de victimes ou de cibles, le suivi de cibles, ou bien le transport en terrain inconnu.

2.2.1 Le cadre logiciel GeNoM

Pour cela, le robot possède un certain nombre de composants logiciels (ou modules) encapsulés au travers du cadre logiciel GeNoM (Fleury et al., 1997). Un module GeNoM est composé des éléments suivants :

– **un ensemble de canaux de données**, représentés par un nom unique et le type de donnée qu'il transporte. Ils sont appelés *poster* dans le vocabulaire GeNoM, mais on peut retrouver la même notion dans ROS avec le système de *topic*, ou dans YARP (Fitzpatrick et al., 2008) via le système de *port*. Dans la suite du document, nous utiliserons la notion de *port* pour définir cette notion de flux de données, qui sont identifiés de manière unique, par un identifiant de type chaîne de caractère. Ces ports sont mis à jour de manière continue.

– **un ensemble de requêtes**, définies par leur nom, des paramètres d'entrées / sorties, et les erreurs possibles. Cela correspond à des méthodes de configuration du module, ou à la demande d'exécution d'un service (de manière discrète dans le temps). Lors de l'exécution d'un service, GeNoM renvoie un identifiant d'activité, permettant en particulier de savoir si une requête est terminée (et si oui, avec quel résultat) et d'annuler une requête en cours. On retrouve ce genre de mécanismes dans ROS au travers des systèmes de *services* et d'*actions*.

2.2.2 Couche fonctionnelle d'un robot terrestre

Localisation Une des capacités les plus importantes pour un robot mobile est sa capacité à se localiser. Pour cela, nous nous basons sur différents composants matériels et leurs modules associés, à savoir un *GPS*, un *gyroscope*, une centrale inertielle *IMU*[2] et une mesure de l'odométrie fournie par le module *locomotion*. Ces différentes informations sont fusionnées de manière simple dans le module *PoM*[3]. Une autre approche de localisation plus avancée repose sur l'utilisation de techniques de SLAM (Roussillon et al., 2011) et est encapsulée dans le module *rtslam*. Il utilise, en sus des informations liées aux modules matériels précédents les images fournies par le module *vision*. Ces modules ne sont généralement pas contrôlés de manière active : ils contiennent peu de requêtes, et sont surtout définis par leur flux de données.

Stratégie de navigation 2D Une fois localisé, le robot peut essayer de se déplacer. Pour cela, il existe actuellement plusieurs stratégies de navigation. La première est utilisée plutôt sur des terrains plats ou peu accidentés. Elle est basée sur le comportement sensori-moteur suivant. Dans un premier temps, le robot construit une carte d'obstacles grâce au module *2DMap*, puis cette carte est utilisée pour calculer des vitesses de translations et rotations instantanées pour éviter les obstacles grâce à *avoid*. Enfin ces consignes sont exécutées via le module *locomotion*. La carte peut être construite de différentes façons, selon le matériel présent sur le robot. La méthode la plus simple consiste à exploiter un scanner laser horizontal *sick*. Alternativement, il est possible de dériver la carte à partir d'un nuage de points 3D, qui peut être calculé directement à partir d'un *velodyne* (un lidar 3D tournant), ou bien via la *stereovision*, un module exploitant une paire de caméras placées sur une platine orientable.

Stratégie de navigation 3D Sur des terrains plus difficiles, il est possible de naviguer en exploitant un modèle 3D du terrain et des algorithmes de recherche de trajectoires considérant à la fois ce modèle et les contraintes physiques du robot (tel que décrit dans (Lacroix et al., 2002)). La boucle sensori-motrice associée est donc composée des modules *stereovision* ou *velodyne*, *DTM*[4], *3DPath* et *locomotion*. Le robot va donc, tour à tour, générer un nuage de points 3D, le fusionner dans un modèle de terrain, puis utiliser ce modèle pour choisir une trajectoire élémentaire définie par un couple (v, ω), qui va être exécutée par le module *locomotion*.

Stratégie de navigation haut niveau Ces approches sont purement réactives, et permettent d'éviter efficacement les obstacles mais peuvent tomber dans des minima locaux. Pour pallier ces problèmes, un mode de navigation de plus haut niveau (Boumghar and Lacroix, 2011) est disponible au travers du module *navigation*. Celui-ci est en charge de calculer des chemins à long terme, ainsi que de sélectionner les types de navigation possibles sur le terrain courant. Pour cela, il utilise une carte de probabilité d'obstacles (ou "carte de traversabilité") *TMap*[5],

2. pour Inertial Measurement Unit
3. pour Position Manager
4. pour Digital Terran Model
5. pour Traversability Map

qui est construite par une méthode de filtrage bayésienne sur un nuage de points 3D (généré par *stereovision* ou *velodyne*). Ceci définit une boucle sensori-motrice de plus haut-niveau dont le résultat pourra être utilisé par les méthodes de navigation locales précédentes.

But des missions Enfin, la mission peut être définie en terme de coordonnées géographiques à atteindre (ou de zone à couvrir), ou en terme de cibles à détecter et à suivre. Dans ce cadre, deux modules permettent respectivement de détecter une cible et de la suivre (*detectTarget* et *trackTarget*), en utilisant les images venant de *stereovision*. Dans le cadre des expérimentations, il est aussi possible d'utiliser le *velodyne* pour détecter des cibles spécialement couvertes de surfaces réfléchissantes, qui génèrent des échos aisément détectables.

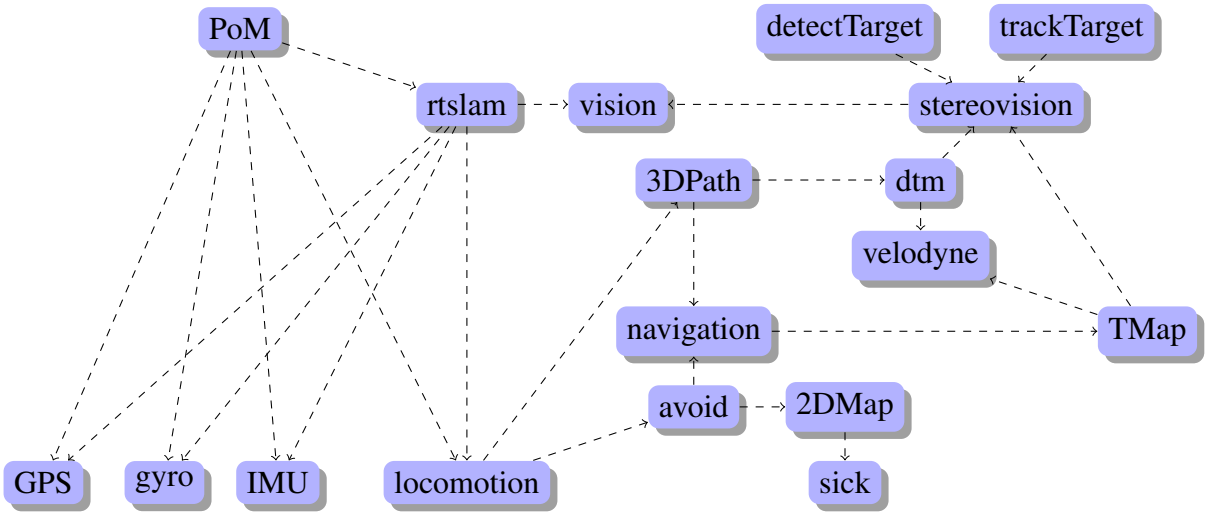

FIGURE 2.2 – Couche fonctionnelle d'un UGV : une solution complète de navigation autonome est proposée au travers d'une population de modules GeNoM. Les flèches représentent les flots de données possibles (à noter que de nombreux modules exploitent la position exportée par le module *PoM* mais ne sont pas dessinés pour des raisons de lisibilité de la figure).

Synthèse La figure 2.2 résume les interactions possibles entre les différents composants de la couche fonctionnelle. Il apparait assez clairement un certain nombre de conflits possibles. Par exemple, les tâches de navigation en milieu complexe et de suivi de cibles utilisent toutes deux une ressource *stereovision*. Toutefois, les contraintes imposées par les deux tâches ne sont en général pas compatibles : la première nécessite de regarder le sol, devant le robot, tandis que l'autre doit suivre la position estimée de la cible. Plusieurs solutions sont possibles, en fonction des ressources matérielles disponibles. En présence d'un velodyne, la tâche de navigation peut utiliser ce capteur laissant la stéréovision pour la tâche de suivi de cible. Si le seul capteur est disponible est un unique banc stéréopixel, une des tâches ne pourra pas être effectuée. Cela met bien en exergue la problématique des accès concurrents dans un système robotique. Cette illustration montre aussi que l'UGV possède en général plusieurs stratégies possibles (qui peuvent toutes échouer) pour une même tâche : l'architecture de contrôle doit pouvoir être capable de

sélectionner une stratégie alternative si la stratégie sélectionnée échoue. De plus, le choix d'une stratégie dépend directement des fonctionnalités disponibles sur un robot donné, choix qui peut être limité tant par la puissance de calcul que par le matériel embarqué. L'architecture de contrôle devra donc s'adapter en fonction des capacités réelles du robot. Enfin, cette couche fonctionnelle va évidemment évoluer (nouvelles stratégies, nouvelles tâches), et l'architecture de contrôle doit donc être assez modulaire et ouverte pour coordonner facilement de nouveaux modules.

2.3 Synthèse

Nous avons précisé ce que nous entendions par architecture de contrôle et définit cinq exigences primordiales pour une telle architecture : **gestion de temps de délibérations distincts**, **gestion aisée de la concurrence**, **robustesse**, **vérifiabilité** et enfin **modularité et composabilité**. Nous avons ensuite introduit notre contexte de travail, la robotique terrestre extérieure et décrit de manière fine la couche fonctionnelle utilisée au LAAS pour ce type de robot.

La suite de cette partie a donc pour vocation à proposer une architecture de contrôle permettant de répondre à ces exigences, puis à définir une couche de supervision (ou couche de contrôle) propre à ce domaine, *i.e.* une instance spécifique de l'architecture pour gérer ce domaine robotique particulier.

Chapitre 3

État de l'art

Ce chapitre passe en revue les principales solutions architecturales proposées dans la littérature pour le contrôle d'un robot autonome. Nous les présentons selon quatre groupes : langages de contrôle, architectures multi-agents, architectures à trois couches et à deux couches, et les analysons selon les caractéristiques présentées dans la section 2.1.

Les premières architectures de contrôle pour des robots autonomes s'articulaient autour d'une décomposition du système en trois sous-parties, appelées séquentiellement, à savoir la perception, la planification, puis l'exécution (paradigme nommé SPA [1]) (Nilsson, 1980). Le sous-système de perception construit un modèle de l'environnement à partir des données perçues, puis la planification calcule un plan sur la base de ce modèle, *i.e.* une séquence d'actions pour atteindre le but, qui est exécuté par le sous-système d'exécution (cf figure 3.1). L'exécution étant à priori un problème simple, la majorité des travaux se concentre sur les phases de modélisation et de planification.

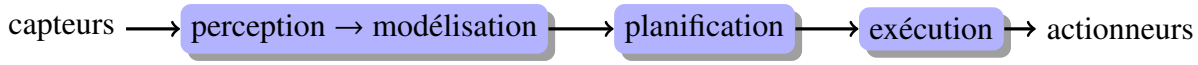

FIGURE 3.1 – L'architecture SPA est définie par trois grands blocs, perception, planification, et exécution, appelés en séquences.

Toutefois, comme l'analyse E. Gat dans (Gat, 1991b), cette approche présente deux problèmes majeurs. Tout d'abord, la planification et la modélisation sont des problèmes très difficiles, surtout considérés au niveau global d'un robot. Ensuite, l'exécution des plans en boucle ouverte n'est pas satisfaisante dans un environnement dynamique. Deux grandes alternatives apparurent pour répondre à ces défis : les architectures réactives et les architectures dites "en couches". Les architectures réactives rejettent le paradigme classique de l'Intelligence Artificielle, à

1. Sense-Plan-Act

savoir l'utilisation de plans et de modèles symboliques comme base de l'autonomie. Elles soutiennent que l'autonomie vient de l'interaction entre des comportements élémentaires – l'une des plus connues étant probablement l'architecture Subsumption (Brooks, 1990). De nombreux travaux suivirent cette voie, en particulier des systèmes multi-agents. Dans ce cadre, la question importante est de savoir comment interagissent les différents comportements ou agents. Les architectures dites "en couches" conservent une notion de plan symbolique de haut niveau mais ajoutent un niveau intermédiaire entre ce planificateur de haut niveau et les comportements réactifs. Cette couche intermédiaire a pour but d'exécuter le plan, mais est capable de réagir à la situation courante. Malheureusement, l'ajout de ce niveau intermédiaire entraîne un certain nombre de difficultés. En particulier, la disparité de représentation des informations entre les différentes couches réduit l'efficacité des échanges, ce qui implique entre autres des réparations de plan peu efficaces. Pour pallier ces problèmes, des architectures "à 2 couches" apparaissent. L'idée principale de ces architectures est de proposer un modèle unifié de l'exécution et de planification, et d'entrelacer en permanence ces deux phases. Enfin, en parallèle de ces solutions architecturales, l'émergence de cadres logiciels basés composants d'une part, et de langages dynamiques puissants d'autre part, a favorisé l'utilisation des seconds pour coordonner les premiers.

Ce chapitre va maintenant analyser plus en détails ces quatre grands types de solutions, en particulier à la lumière des besoins exprimés dans le chapitre précédent.

3.1 Langages de contrôle

Introduction

Les systèmes basés composants offrent le plus souvent la possibilité d'être contrôlés par des langages génériques de haut niveau. On peut citer TCL pour GeNoM, mais aussi Python pour ROS, ou bien Ruby ou Lua pour Orocos. L'avantage principal de ces outils est que n'importe quel développeur peut définir des schémas de contrôle, sans avoir besoin d'apprendre de nouveaux paradigmes ou outils. Le revers de la médaille est qu'ils ne fournissent globalement aucune solution native aux grands problèmes du domaine : il faut donc à chaque fois proposer des solutions *ad hoc* à ces différents problèmes, ce qui est source de nombreuses erreurs. Plusieurs approches ont été proposées pour essayer de répondre à ces problématiques.

Langages dédiés

La première approche a été de proposer de nouveaux langages de contrôle, orientés robotique, mais qui restent proches des paradigmes classiques. Le langage de script Urbiscript (Baillie, 2005) est un langage objet (à prototype) concurrent, *i.e.* il fournit un ensemble de primitives pour la création de tâches parallèles et leurs synchronisations. Toutefois, la gestion des accès concurrents reste manuelle via la création de sections critiques : le problème de la concurrence reste donc entier. De plus, Urbi propose différents mécanismes pour l'utilisation d'évènements discrets, permettant d'implémenter facilement des schémas réactifs mais ne possède aucune gestion

à long terme d'un plan. Une autre approche intéressante est le langage graphique VPL [2] développé par Microsoft, dans la suite Microsoft Robotics Studio. Celui-ci permet graphiquement de créer des interactions entre les différents composants, et il est ainsi facile de développer et déboguer des problèmes de coordinations. Il s'appuie sur une bibliothèque spécialisée CCR [3] qui gère les problèmes d'asynchronisme et de concurrence. Cette approche semble intéressante, mais le CCR reste une boîte noire pour le développeur : il semble difficile de raisonner sur les échecs et de proposer des solutions spécialisées pour un domaine d'application spécifique. De plus, on peut se demander si la programmation graphique est adaptée quand on considère des robots de plus en plus complexes.

Bibliothèques dédiées

Une seconde approche consiste à enrichir un langage générique par un ensemble de bibliothèques dédiées à la robotique. Par exemple, SMACH (Bohren and Cousins, 2010) ajoute à Python le support de machines à états concurrents hiérarchiques et les utilise pour coordonner des tâches robotiques. De manière relativement similaire, RFSM ajoute le support de machine à états hiérarchiques à Lua pour contrôler des robots utilisant Orocos. Les machines à états finis sont un paradigme maintenant classique, bien défini (Harel and Naamad, 1996) et sur lequel il est relativement facile de raisonner. Toutefois, le nombre d'états pour un robot complexe autonome peut facilement devenir très grand, même en utilisant des principes de hiérarchisation, et il devient alors difficile de maintenir et développer ces machines à états finis. De plus, il nous semble que cette approche est difficilement compatible avec un environnement dynamique non prévisible, pouvant donc entraîner des transitions non prévues. Enfin, même si ici aussi, il existe une notion de tâches parallèles, aucune gestion automatique des problèmes de concurrence n'est prévue.

Langages dédiés internes

Une dernière approche consiste au développement de langages dédiés internes (DSEL [4]) qui consiste à définir dans un langage hôte un sous-language spécialisé pour un domaine donné, permettant à la fois d'exprimer facilement ses besoins, mais conservant la richesse d'un langage développé par ailleurs. Frob (Peterson and Hager, 1999) étend le langage fonctionnel Haskell dans deux directions. Tout d'abord, il propose un cadre réactif à travers la notion de FRP (Programmation Fonctionnelle Réactive). Au-dessus de ces constructions, il ajoute la notion de tâches et leurs interactions avec la notion d'événements et d'erreurs. L'utilisation d'une monade [5] pour la gestion des tâches permet de les combiner (séquentiellement ou parallèlement). Encore une fois, le problème d'un accès partagé à une ressource n'est pas considéré, et il faut se rabattre sur les mécanismes offerts par Haskell. La robustesse et la vérifiabilité ne sont pas non plus débattues,

2. Visual Programming Language
3. Coordination and Concurrency Runtime
4. Domain-Specific Embedded Language
5. Une monade est une structure algébrique dérivée de la théorie des catégories. En informatique, elle permet de représenter et encapsuler un certain type de calcul (exception, probabilité, abscence de résultat, ...). Cette construction est particulièrement utilisée dans les langages fonctionelles pour séparer les calculs purs (sans effets de bord) des calculs à effets de bords (accès disque, modification d'un état, ...)

mais il semble possible de pouvoir encoder un certain nombre d'invariants grâce au système de type riche du langage hôte.

Synthèse

Même si ces langages de contrôle permettent de rapidement développer des schémas de contrôle pour un robot, ils sont le plus souvent insuffisants pour créer des contrôleurs généraux pour un robot autonome. Différents aspects nécessaires à la robotique autonome ne sont en effet pas explicitement traités, tels que la gestion des plans à long terme ou la gestion des accès concurrents, et doivent donc être spécifiés par le développeur, au risque de commettre des erreurs. Les questions de robustesse et de vérifiabilité quant à elles sont des problématiques non traitées : si certains langages fortement typés pourraient permettre d'encoder et garantir certaines propriétés, pour la plupart, il n'y a pas d'approches directes permettant de telles garanties.

3.2 Les architectures multi-agents

Architectures réactives

L'architecture Subsumption (Brooks, 1990) propose de décomposer les différents processus d'un robot en un ensemble de comportements élémentaires (possiblement liés), qui sont représentés par des petites machines à états finis. Pour construire des comportements plus complexes, Brooks introduit la notion d'inhibition qui va modifier le contenu d'un lien par la valeur d'un autre. Cette architecture a démontré des résultats très intéressants pour la navigation et l'évitement d'obstacles. Toutefois, d'après R. Hartley (Hartley and Pipitone, 1991), l'architecture Subsumption manque d'outils pour gérer la complexité, et il lui est donc difficile de coordonner des tâches de plus haut niveau.

Systèmes multi-agents

En s'inspirant des grandes lignes des architectures réactives, de nombreux systèmes multi-agents sont apparus. Le problème clef de ces architectures est la sélection et la coordination des bons comportements au moment courant.

Une approche temps-réel L'architecture ARTIS (Soler et al., 2000) définit un ensemble d'agents temps-réels, et les ordonnance selon un ensemble de priorité fixes. Cette approche garantit des propriétés temps réel dur, mais limite la flexibilité de l'architecture, et ses possibilités de reconfiguration. Ces exigences sont nécessaires dans certains domaines, mais sont probablement trop drastiques en général pour un robot accomplissant des tâches plus complexes.

Une approche basée enchère Dans (Busquets et al., 2003), D. Busquets propose une architecture où les agents se coordonnent à l'aide d'un système d'enchères. Une enchère dans ce cadre est un réel dans l'intervalle $[0, 1]$ et indique si une opération est nécessaire plus ou moins rapidement. Ce système d'enchères permet de sélectionner l'action courante à exécuter et donc

d'éviter les problèmes de concurrence au matériel. Toutefois, ce système d'enchères centralise la décision dans un unique agent, l'échec de celui-ci entraîne donc l'arrêt complet du système. De plus, chaque agent utilise une fonction d'enchère qui lui est propre, on peut se demander si ce système permet, quelque soit la fonction enchère de chaque agent, d'avoir un comportement global cohérent.

De la théorie de l'organisation Les travaux de P. Giorgini (Giorgini et al., 2001) analysent différentes stratégies de coordination, certaines classiques (boucle fermée, architecture en couches), et d'autres venant des *théories de l'organisation* (structure-en-5, coentreprise [6]). Dans la structure-en-5 (montrée figure 3.2), le robot est décomposé en cinq sous-systèmes importants qui sont placés dans une configuration initiale, avec au centre un agent qui coordonne les mouvements du robot et gère les différentes fautes. Dans le cas de la coentreprise (présentée figure 3.3), le système est organisé autour d'un agent central manager, jouant d'un côté le rôle de coordinateur et de l'autre d'interface de contrôle pour définir les missions. Ces différents types d'organisations sont analysés sous l'angle de quatre thématiques : coordination, prévisibilité, tolérance aux fautes, et adaptabilité. Au vu de ces critères, les méthodes structure-en-5 et coentreprise sont plus efficaces que les méthodes classiques d'organisation. Toutefois, l'étude reste théorique, et autant que nous le sachons, il n'existe pas d'implémentation effective de ces architectures sur un robot réel. De plus, ces deux propositions ont un point de décision central unique, créant ainsi un point de faiblesse unique (il peut fauter, ou mettre de plus en plus longtemps à prendre une décision).

Une approche décentralisée Au contraire, (Innocenti et al., 2007) propose une architecture complètement décentralisée, en utilisant la logique floue pour coordonner les différents agents. Ces travaux sont illustrés par la fusion de deux contrôleurs pour une tâche de navigation. Toutefois, la fusion correcte de ces contrôleurs grâce à la logique floue dépend de plusieurs paramètres définis empiriquement. Cela pose le problème de la généricité de l'approche : si certains paramètres doivent être redéfinis manuellement à chaque ajout ou suppression d'un contrôleur, la solution proposée reste relativement spécifique à un problème donné. Enfin, on a vu précédemment que des architectures réactives comme Subsumption étaient efficace pour des tâches de navigation mais avaient des difficultés pour contrôler des tâches plus complexes. On peut donc légitimement se demander si cette architecture peut coordonner des tâches plus complexes que la navigation.

Synthèse

Les approches multi-agents sont par essence modulaires. La connaissance des comportements existants n'est en effet pas nécessaire pour l'écriture de nouveaux comportements, et elles sont relativement tolérantes aux fautes car le plus souvent, si un agent échoue, d'autres comportements prennent le relais. Mais l'introduction de nouveaux comportements dans un système préexistant peut avoir un impact négatif sur ce système : le développeur doit dans de nombreux cas manuellement réorganiser ces agents ou modifier leur stratégie d'organisation. Cela réduit

6. joint venture

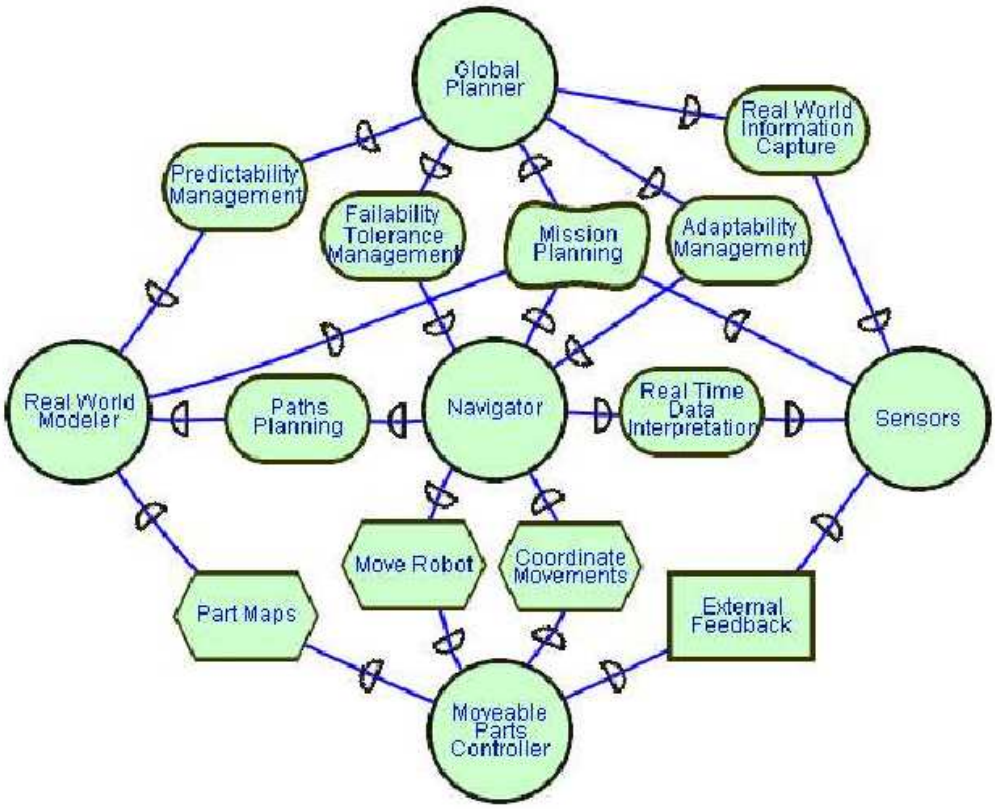

FIGURE 3.2 – Un système robotique multi-agents organisé selon le principe structure-en-5 (extrait de (Giorgini et al., 2001)). Les 5 composants principaux sont représentés par un rond (navigateur, planificateur, capteurs, modèles du monde, contrôleur de déplacements). Les liens entre eux représentent les échanges possibles, le type des échanges étant définis par les boîtes sur ces liens.

donc la réutisabilité et la modularité de l'architecture. De manière plus générale, même si le comportement individuel de chaque agent est relativement facile à prédire, il est souvent difficile de raisonner sur le comportement global d'une architecture multi-agents, et donc d'apporter des garanties concernant le comportement du robot.

3.3 Les architectures "3-tiers"

Du besoin d'une couche intermédiaire

Dans (Gat, 1997b), E. Gat analyse différentes architectures existantes en se basant sur l'état interne des algorithmes qu'elles manipulent. L'architecture SPA utilise des algorithmes qui maintiennent des états du monde à long terme. Les problèmes apparaissent quand les modèles internes ne coïncident plus avec le monde. Pour répondre à ces problèmes de synchronisation, les archi-

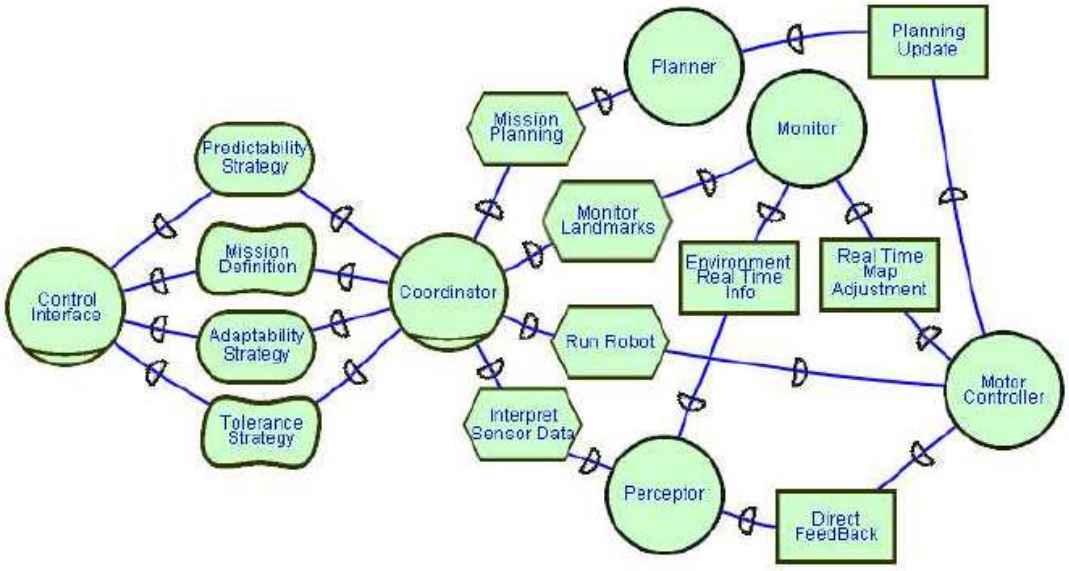

FIGURE 3.3 – Un système robotique multi-agents organisé selon le principe de la coentreprise (image tirée de (Giorgini et al., 2001)). Le coordinateur fait l'interface entre l'interface de contrôle (à gauche), et les sous-fonctions à coordonner (planificateur, moniteur, perception, contrôleur de déplacement).

tectures réactives considèrent des solutions sans états ("le monde est son meilleur modèle"), mais cette approche semble trop limitée pour exécuter des tâches complexes. La définition d'une couche intermédiaire vient alors naturellement, pour faire le lien entre la couche fonctionnelle, possédant peu ou pas d'états internes de la couche décisionnelle, qui considère des modèles et plan à très long terme. De nombreuses solutions utilisant ce modèle ont été proposées, nous allons maintenant en analyser quelques unes.

Quelques instances d'architectures 3-tiers

Architecture 3^T L'architecture 3^T, proposée dans (Bonasso et al., 1995), est décomposée en trois parties, comme illustré dans la figure 3.4 : le *Adversarial Planner*, la bibliothèque RAP (Firby, 1989) qui contient différentes méthodes, dépendant du contexte courant, pour implémenter une tâche, et l'interpréteur RAP. Lorsqu'un robot muni de cette architecture doit effectuer une mission, le *Adversarial Planner* synthétise le but en un ensemble partiellement ordonné de tâches de haut niveau. Dans un second temps, il sélectionne par unification et en fonction du contexte une implémentation dans la bibliothèque RAP. L'interpréteur RAP va alors décomposer cette tâche en une séquence d'actions élémentaires, puis va déclencher les comportements élémentaires associés (appelés *skills*). Dans le même temps, il va déclencher un certain nombre de moniteurs, qui vont l'avertir de changements dans l'environnement, mais aussi détecter les conditions de suc-

cès de la tâche. Dans ce cas, l'information est remontée au planificateur, qui va alors déclencher l'exécution de la prochaine tâche.

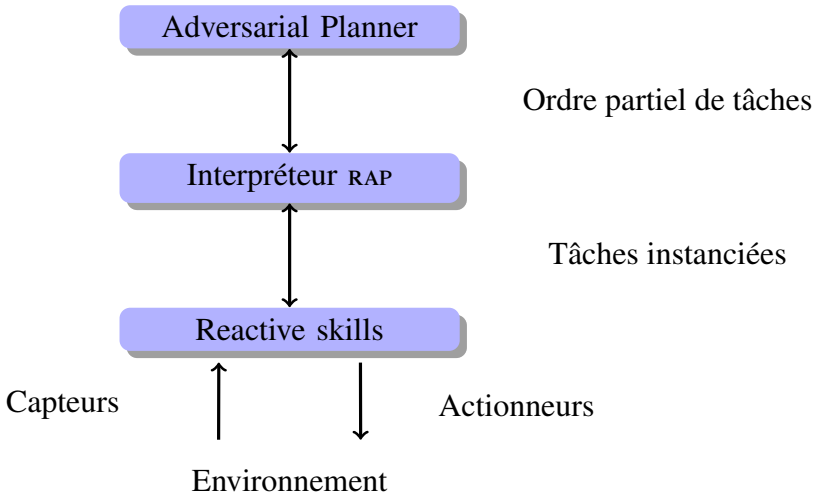

FIGURE 3.4 – L'architecture 3^T décompose le robot en trois couches, le niveau déductif instancié par l'Adversarial Planner, le niveau exécutif instancié par l'interpréteur RAP, et la couche fonctionnelle est composée d'un ensemble de *skills*.

Architecture ATLANTIS L'architecture ATLANTIS (Gat, 1991b) introduit un changement important de paradigme. Contrairement à 3^T, c'est la couche intermédiaire (le séquenceur) qui contrôle le planificateur, *i.e.* qui décide, en fonction de son état interne, quand le planificateur doit être appelé à nouveau. Pour décrire des tâches au niveau du séquenceur, le développeur utilise le langage ALFA (Gat, 1991a). Contrairement à RAP qui est avant tout un langage permettant d'exprimer un plan, ALFA est un langage plus complet, qui permet de définir des comportements et des réactions plus complexes.

Architecture LAAS L'architecture LAAS (Alami et al., 1998), illustrée par la figure 3.5 propose une décomposition légèrement différente. Le niveau décisionnel est composé d'un planificateur symbolique et d'un superviseur. Sémantiquement, ce superviseur est analogue au séquenceur d'ATLANTIS. Les tâches de supervision sont décrites dans le langage PRS (Ingrand et al., 1996). Même si il possède des constructions impératives, PRS propose majoritairement des constructions "orienté vers le but". Le développeur va expliciter les buts à atteindre, et le moteur PRS va chercher dans sa base une recette qui correspond à ce but et l'exécuter. Le niveau exécutif lui n'a aucune capacité de décision, il est uniquement responsable de sélectionner et synchroniser dynamiquement les appels adéquats à la couche fonctionnelle. (Py and Ingrand, 2004) enrichit ce niveau exécutif en lui permettant de formellement vérifier la validité des requêtes envoyées à la couche fonctionnelle, et ce afin d'éviter d'entrer dans un ensemble d'états prédéfinis interdits. Ainsi, le comportement global est plus sûr et cohérent.

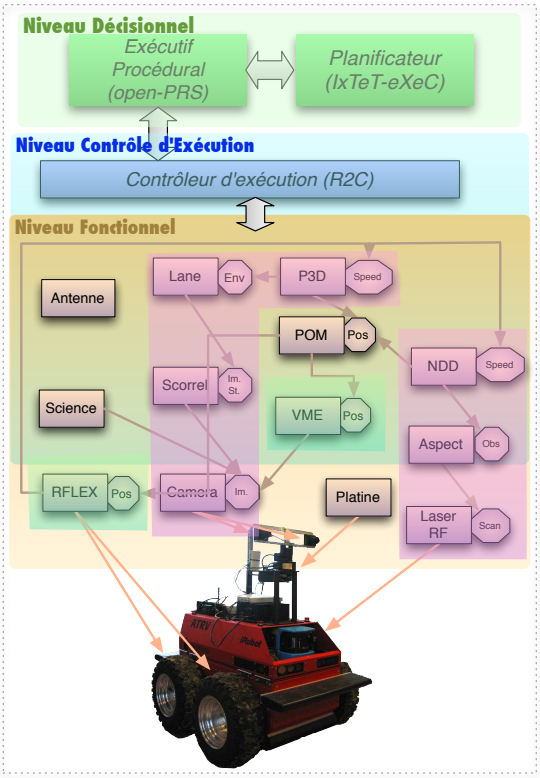

FIGURE 3.5 – Instance de l’architecture LAAS pour un UGV. En haut, la couche décisionnelle, contenant le planificateur et le superviseur PRS, au milieu la couche exécutive, constituée de l’utilitaire R2C qui vérifie l’intégrité des requêtes aux modules fonctionnelles. En bas, la couche fonctionnelle constituée d’un ensemble de modules GeNoM. Les interactions entre couches sont représentées par les grosses flèches.

Remote Agent System L’architecture Remote Agent System (Bernard et al., 1998) propose un certain nombre d’extensions intéressantes. Tout en décomposant les plans de haut niveau en un ensemble de tâches plus petites et en vérifiant leur bonne exécution, la couche exécutive EXEC permet aussi de gérer les ressources matérielles du robot. Pour cela, elle alloue explicitement des ressources à des tâches, et va les suspendre si une ressource est temporairement non disponible. Les procédures de EXEC sont décrites dans le langage ESL (Gat, 1997a). Comme PRS, ESL permet de définir des tâches en termes de but à atteindre, mais aussi de facilement définir des tâches concurrentes. Enfin, en cas d’erreur, EXEC peut compter sur l’outil spécialisé MIR pour effectuer un diagnostic et proposer une solution adéquate. Pour cela, il utilise un moteur de prédiction de l’état du robot en fonction des commandes envoyées. Si celui-ci détecte des divergences entre l’état attendu et l’état courant, il va alerter EXEC puis proposer une solution à l’aide de sa base de

données de conflits.

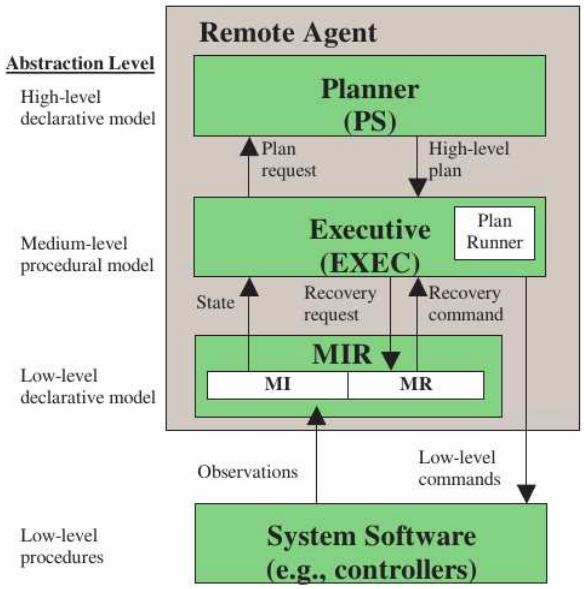

FIGURE 3.6 – Cette vue de l'architecture Remote Agent, extraite de (Muscettola et al., 2002), montre bien les interactions entre les différents éléments qui la composent. L'exécutif EXEC reçoit des plans depuis le planificateur PS, et envoie des commandes à la couche fonctionnelle. Les observations se font à travers le sous-système MIR, qui sert aussi comme solution en cas de fautes.

Analyse

Même si il existe des différences entre ces propositions, l'approche générale est globalement la même : l'ajout d'une couche intermédiaire est nécessaire pour réduire le fossé entre couche fonctionnelle et monde symbolique, permettant ainsi d'exécuter des comportements de haut niveau tout en conservant la capacité de réagir efficacement aux aléas. Mais la création de ce niveau intermédiaire et des outils associés pose différents problèmes. En effet, dans cette approche, chaque outil possède sa propre représentation du monde, du plan, ce qui rend les échanges d'informations difficiles et moins efficaces. Ainsi, le planificateur manque souvent d'informations pour proposer une bonne solution en cas de faute, car il ne comprend pas les causes exactes de l'échec. De même, le niveau exécutif n'a qu'une vue partielle des horizons possibles, ce qui limite la flexibilité et les choix qu'il peut faire. Ces différences de représentation introduisent donc des pertes d'informations, et réduisent significativement la qualité des réactions du robot. Un autre problème des architectures 3-tiers est lié au passage à l'échelle. En effet, chaque niveau raisonne (à une certaine granularité) sur l'ensemble du robot. Par conséquent, le temps de délibération de chaque couche augmente avec la complexité du robot, rendant ainsi le système de moins en moins réactif. Enfin, les solutions logicielles liées à la couche exécutive sont le plus souvent centrées autour d'une base de donnée logique, qui peut être lue ou modifiée par chaque

procédure. D'un côté, cela permet à n'importe quelle méthode de déduire l'état du robot qui l'intéresse. Mais cela entraîne d'importantes difficultés pour raisonner et maintenir le système. En effet, une modification dans la base de données peut avoir des conséquences dans différentes méthodes, ou l'ajout d'un fait dans la base peut entraîner un certain nombre d'effets de bord non prévus. En conséquence, l'ajout ou la suppression de tâche (pour gérer de nouveaux robots ou de nouvelles tâches) est difficile et entraîne souvent une réécriture importante du système exécutif.

3.4 Les architectures 2-tiers

Vers une unification des modèles

Le système CLARAty (Volpe et al., 2001; Estlin et al., 2001) est un des premiers systèmes qui prend en compte les problèmes engendrés par de multiples représentations des données. CLARAty conserve une approche "3-tiers" avec un planificateur symbolique CASPER (Chien et al., 2000) et un système exécutif TDL (Simmons and Apfelbaum, 1998), mais ceux-ci sont couplés dans un système plus global appelée CLEAR . Même si le planificateur et le système exécutif gardent des représentations distinctes, CLEAR permet, comme le montre la figure 3.7, de coupler les bases de plan des deux sous-systèmes, en particulier les changements dans l'un sont automatiquement reflétés dans l'autre. Des heuristiques sont utilisées pour décider qui en cas de faute doit la réparer. L'interaction entre ces deux sous-systèmes reste limitée, CASPER ne pouvant pas tirer parti des constructions plus souples de TDL , et TDL ne pouvant pas utiliser les connaissances de CASPER pour gérer correctement les ressources.

Architecture basée planification

L'architecture IDEA L'architecture IDEA (Muscettola et al., 2002) va proposer un changement profond de paradigme. Elle est entièrement orientée vers la planification, l'exécutif (qui trouve les prochaines actions à exécuter en fonction de l'état courant) n'étant pour les concepteurs qu'une recherche de plan dans un domaine de planification arbitrairement simple. Pour résoudre les problèmes d'efficacité et posés par des temps de réactions distincts, IDEA décompose le système en un ensemble d'agents qui se basent tous sur un même modèle de plan. Chaque agent est composé d'un planificateur et d'un exécutif (comme le montre la figure 3.8), dont les exécutions sont entrelacées. Chaque agent peut posséder des planificateurs ou des exécutifs différents, tant qu'ils respectent le modèle de plan défini par IDEA . Ainsi, un agent bas niveau pourra réagir rapidement avec un planificateur spécialisé, ou un domaine de planification très simple, tandis qu'un agent de haut niveau pourra prendre plus de temps pour délibérer.

L'architecture T-REX T-REX (McGann et al., 2008; McGann et al., 2009; Py et al., 2010) (figure 3.9) est une architecture dérivée de IDEA , mais qui met l'accent sur l'échange et la synchronisation des données entre les différents agents. En proposant une formalisation stricte du temps de réaction maximal d'un agent (un "réacteur" dans la terminologie T-REX) et d'horizon de planification, l'architecture garantit des propriétés fortes sur la cohérence du plan global. Toutefois,

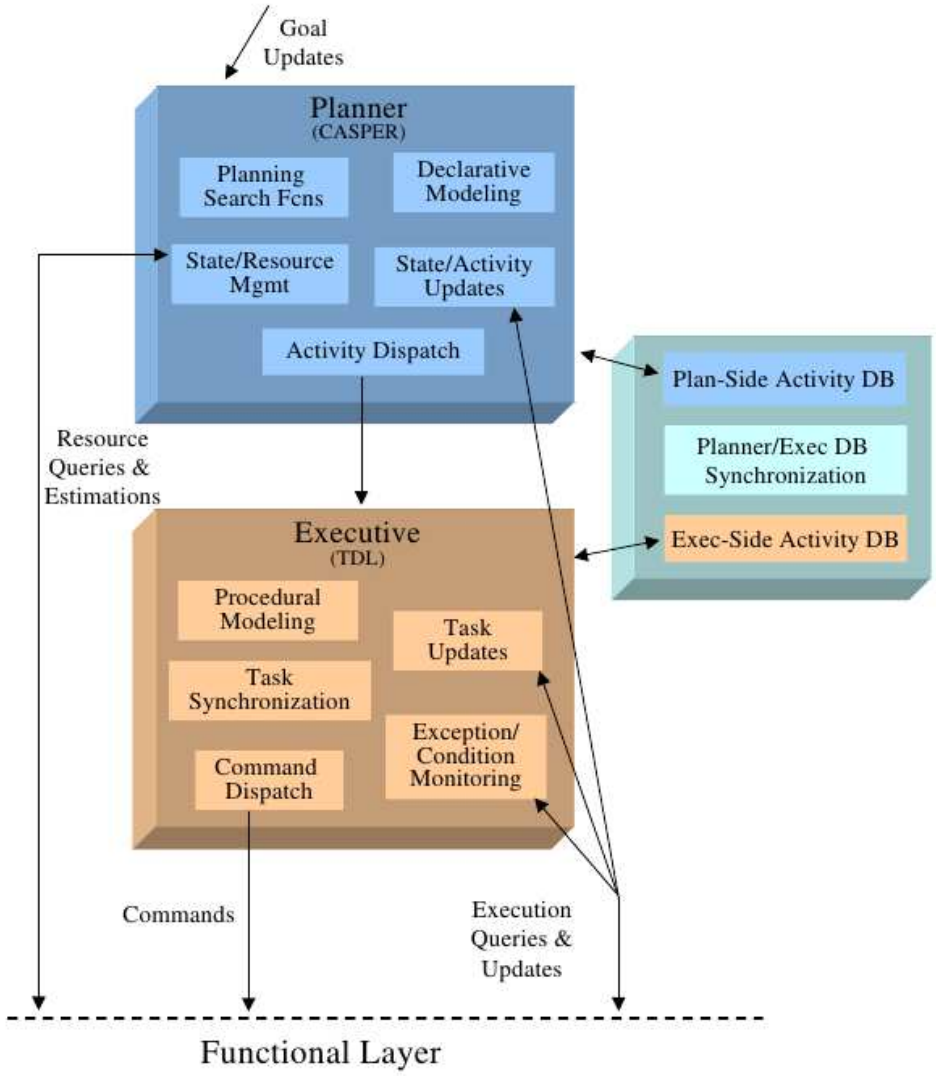

FIGURE 3.7 – Vue d'ensemble de l'architecture CLARAty tirée de (Volpe et al., 2001). On retrouve les deux blocs classiques planificateur et exécutif (respectivement CASPER et TDL), et leurs liens avec la couche fonctionnelle. À droite, le système CLEAR permet de conserver une vue synchronisée entre les modèles de CASPER et TDL .

l'approche implique que les états des réacteurs soient régulièrement synchronisés, nécessitant ainsi un ordre strict entre les réacteurs. Ce pré-requis réduit significativement la modularité et la versatilité de l'approche. En effet, l'ajout de nouvelles variables d'états peut nécessiter des modifications dans la décomposition en agents (afin de respecter la contrainte d'ordre strict entre ces agents), et donc impliquer la modification du domaine des agents en question.

Analyse Ces approches basées planification sont intéressantes à plus d'un titre. En effet, de par leur décomposition en multiples agents, elles permettent de gérer différentes vitesses de ré-

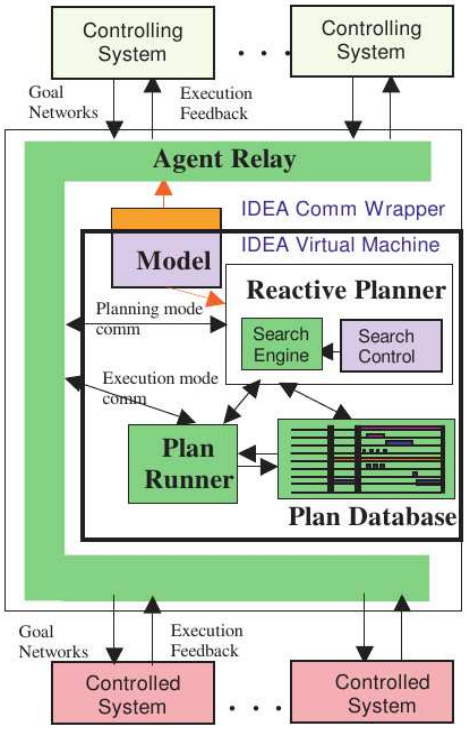

FIGURE 3.8 – Un agent IDEA (image extraite de (Muscettola et al., 2002)). Trois sous-systèmes importants apparaissent, le planificateur réactif, l'exécutif, et la base de plan. Celle-ci maintient l'état courant de l'agent, *i.e.* son état et ses futurs possibles (en fonction des plans calculés).

actions. De plus, l'utilisation de planificateurs permet de gérer les problèmes de concurrence, et garantit la correction des actions effectuées dans les limites des modèles exploités. Toutefois, l'utilisation exclusive d'un certain modèle de plan limite l'expressivité du développeur et les possibilités du robot : on ne peut faire que ce qui est exprimable dans le modèle de plan, ou décidable au niveau du planificateur. Dans T-REX par exemple, on ne peut que faire du raisonnement non conditionnel et non probabiliste, ce qui ne permet pas d'intégrer aisément des paradigmes alternatifs, comme l'apprentissage par exemple.

Architecture basée gestion de plan

Concurrent Reactive Plan Une autre approche consiste non plus à se focaliser sur la planification, mais sur les plans produits. Ainsi, dans l'architecture Concurrent Reactive Plan (Beetz, 2000), l'auteur s'attache à définir un modèle de plan exécutable et générique, puis à développer des transformations sur ces plans. Plus précisément, étant donné un plan courant, et un changement de la situation (par rapport à la situation prévue), l'architecture essaye de modifier le plan afin de le rendre compatible avec la solution courante. Ainsi, en utilisant un modèle de plan "assez riche", il est possible de définir des outils génériques, indépendants du planificateur qui produit les plans pour analyser les plans (entre autre prédire l'état du robot à un temps donné), et

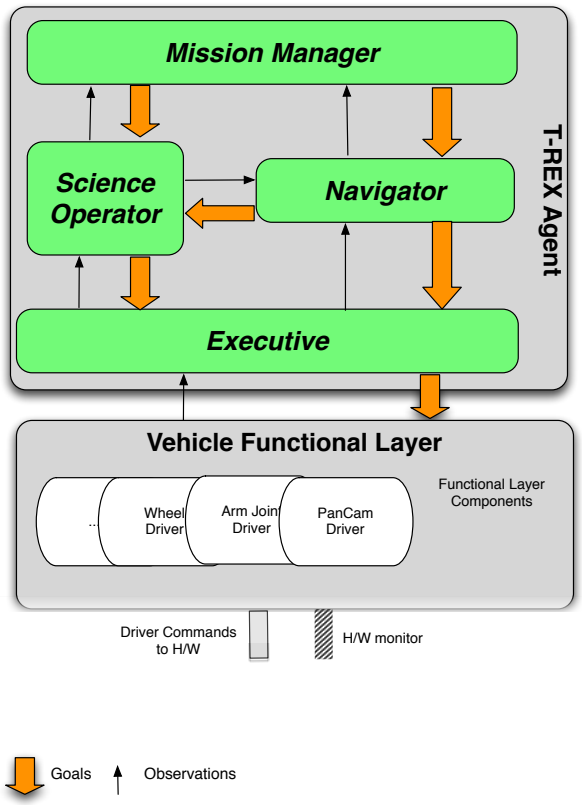

FIGURE 3.9 – Une instance de l'architecture T-REX . La couche décisionnelle est ici composée de quatre réacteurs, le gestionnaire de missions, "l'opérateur science" (la charge utile du robot considéré), la navigation, et l'exécutif. Les flèches oranges représentent les échanges de but, tandis que les flèches fines noires représentent les échanges pour la synchronisation.

les modifier de manière sûre afin de s'adapter à la situation.

Roby L'architecture Roby (Joyeux et al., 2009) étend cette idée selon plusieurs axes. Tout d'abord, le modèle de plan sépare la structure des activités (*i.e.* les relations entre les différentes tâches) de la structure d'exécution, ce qui en simplifie l'analyse. Le moteur d'exécution permet l'exécution non pas seulement d'arbres de tâches mais plus généralement de graphes de tâches. De plus, il propose un système de *transaction* qui permet de manière sûre de modifier un plan en cours d'exécution. Enfin, du point de vue implémentation, l'utilisation d'un langage dédié à l'intérieur de Ruby permet de combiner la souplesse d'un langage dynamique avec les avantages d'un développement basé modèle.

Analyse Bien que l'approche gestion de plan semble plus générique et modulaire que l'approche basée planification (en particulier par l'utilisation de modèles de plans plus génériques), ces architectures semblent répondre moins bien à certain de nos critères. En particulier, dans les approches proposées, il n'existe pas de moyen de gérer explicitement les problèmes de concurrences, deux tâches ne pouvant pas être exécutées en même temps. Plus généralement, les plans ne contiennent pas nécessairement toutes les informations utilisées par le planificateur. Il n'est donc pas toujours possible de modifier de manière sûre un plan quelconque, car l'architecture pourrait violer des invariants utilisés en interne par la planification.

3.5 Synthèse

Les langages de contrôle sont probablement les solutions les plus simples, du point de vue d'un développeur, mais ne satisfont pas la plupart des pré-requis d'une architecture du contrôle. En effet, même si ils proposent une gestion de tâches parallèles, ils ne fournissent généralement pas d'outils pour gérer de manière automatique les problèmes de concurrence. De plus, ils ne fournissent aucun mécanisme pour gérer des plans à long terme, ou de moyens pour assurer la robustesse du comportement du robot.

Les architectures multi-agents sont elles intéressantes car elles encodent naturellement le caractère hautement parallèle d'un robot. De plus, elles sont foncièrement robustes : une faute d'un agent reste généralement confinée à ce seul agent, ne mettant pas en péril l'ensemble du système. Malheureusement, les techniques généralement utilisées rendent le comportement global difficile à prévoir, limitant les propriétés d'extensibilité de l'architecture.

Les architectures "3-tiers", en ajoutant un niveau intermédiaire entre monde symbolique et monde fonctionnel permettent d'améliorer la réactivité (et dans certains cas la robustesse). Toutefois, l'ajout de cette couche crée des différences dans la représentation de ces informations, ce qui rend difficile la coopération entre les différentes couches. De plus, on peut s'interroger sur le passage à l'échelle de telles architectures, tant du point de vue du développement, les solutions logicielles pour la couche exécutive étant souvent peu modulaires, que du point de vue de la réactivité, l'ajout de fonctionnalités augmentant le temps de délibération de chaque couche.

Les architectures "2-tiers" basées planification décomposent le problème du contrôle d'un robot en un ensemble d'agents, chacun mêlant phase de planification et phase d'exécution. Cette

approche améliore ainsi la réactivité et la possibilité du passage à l'échelle du système robotique complet. De plus, une approche basée sur de la planification garantit la correction des actions effectuées. L'approche a toutefois quelques faiblesses, ainsi l'utilisation d'un ordre strict pour T-REX réduit la modularité de l'architecture. Enfin, ces architectures sont limitées par l'expressivité du modèle de plan utilisé et par les capacités d'inférence du planificateur.

Les architectures "2-tiers" basées gestion de plan s'intéressent plus au plan généré et aux outils pour le modifier, de manière continue, afin de l'adapter à la situation courante. Pour cela, elles utilisent des modèles de plan riches, ce qui permet de garder une bonne expressivité. Toutefois, elles ne prennent pour le moment pas en compte les problématiques de concurrence. Enfin, du point de vue de la correction, on peut se demander si les transformations de plans sont toujours correctes, *i.e.* qu'elles ne violent pas un invariant interne des domaines de planification.

Chapitre 4

L'architecture ROAR

Ce chapitre décrit l'architecture ROAR *que nous proposons. Dans un premier temps, nous décrivons les idées générales qui motivent la définition du système proposé. Dans un second temps, la structure d'un agent générique utilisé dans* ROAR *est décrite, ainsi que ses différents mécanismes internes. La troisième partie se concentre sur les interactions entre ces agents, en particulier pour la gestion des fautes. L'ensemble du chapitre est illustré au travers de l'exemple de navigation autonome introduit dans la partie 2.2.*

4.1 Approche générale

4.1.1 De la décomposition en agents

Au vu des exigences que nous avons formulées dans le chapitre 2 et de l'état de l'art, il apparaît nécessaire de décomposer le problème du contrôle d'un robot en un ensemble de sous-systèmes synchronisés, et ce pour trois raisons principales :

- Tout d'abord, comme le montre l'état de l'art et comme l'analyse aussi McGann dans (McGann et al., 2009), une couche exécutive monolithique peut difficilement passer à l'échelle lorsque la complexité du robot augmente, et conserver des propriétés de réactivité. Il est donc nécessaire de décomposer la couche exécutive en sous-systèmes plus simples pour conserver de bonnes propriétés de réactivité, en particulier lorsque la complexité du robot augmente.
- Ensuite, le développement d'un robot, autant de sa couche fonctionnelle que de sa couche de contrôle, est un long travail incrémental. On ne peut donc se contenter d'une base de connaissance globale, sans organisation, où chaque développeur viendrait lire et modifier ce qui l'intéresse, au risque de violer les présupposés d'autres développeurs. Il convient

d'organiser le système global en différents sous-systèmes hiérarchisés avec des interfaces bien spécifiées et interactions claires : cela permet de clarifier le rôle de chacun des développeurs et ce qu'il est en droit de faire.
– Enfin, une même couche de contrôle peut être utilisée sur des robots différents. Il faut être capable d'identifier les sous-systèmes utilisables sur un robot donné en fonction de ses capacités matérielles et refléter cette information au niveau du contrôle. Une bonne décomposition permet de ne déployer que ce qui correspond réellement à l'architecture du robot, et ainsi de s'adapter naturellement à différentes plateformes matériels.

Considérant ces besoins de partitionnement du système en sous-systèmes, tant lors de la phase de développement, que de la phase d'exécution, ainsi que l'aspect intrinsèquement parallèle des tâches d'un robot, il nous semble pertinent de décomposer l'architecture de contrôle en un ensemble d'agents, chacun gérant un sous-système, à des réactivités plus ou moins grandes. Ceci étant posé, il convient maintenant de répondre aux questions suivantes :

– comment décomposer le système en un ensemble d'agents, tout en gardant de bonnes propriétés de modularité, et en favorisant le parallélisme ?
– comment s'organisent ces agents : quels types d'informations échangent ils ? Comment gèrent ils les conflits ou les fautes d'exécutions ?
– enfin, comment être capable de raisonner sur le système, *i.e.* d'avoir des règles assez "simples" pour être capable de prédire son comportement ?

4.1.2 De la décomposition du systèmes en ressources

Des actions...

Un robot autonome a pour but de réaliser des missions de haut niveau, et le rôle de l'architecture est de transformer cette mission de haut niveau en un ensemble cohérent d'actions de plus bas niveau, et d'assurer leur exécution. Une action peut être définie de différentes manières. Le cas le plus simple est le cas d'une action élémentaire, *i.e.* qui correspond directement à une tâche réalisable par un composant de la couche fonctionnelle. Le second cas correspond à une séquence d'actions, potentiellement conditionnelles, qui vont être exécutées séquentiellement ou parallèlement. Le dernier cas est probablement le plus commun : le développeur ne sait pas comment effectuer l'action demandée, mais il connaît un planificateur qui peut résoudre ce problème. Le résultat de ce planificateur peut alors être un plan (consistant à une séquences d'actions) à exécuter ou une construction plus spécialisée qui sera exécutée par un composant spécifique de la couche fonctionnelle (par exemple, une trajectoire, et un contrôleur de trajectoire). Ce dernier cas correspond à l'approche classique SPA. On peut alors voir une action comme un arbre, où les feuilles sont des actions élémentaires, et où les nœuds sont des processus de supervisions de petites boucles de type SPA. Lorsque plusieurs actions sont exécutées en parallèles, on obtient une forêt d'actions, avec possiblement des chevauchements entre les différents arbres : ces chevauchements correspondent à des accès concurrents à un même sous-système.

...aux ressources.

La plupart des architectures de contrôles sont basées sur cette vision “orientée action”, qui est “impérative” dans le sens elle explicite en permanence ce que le robot doit faire (d’abord modéliser, puis planifier, puis exécuter ce plan, puis recommencer...). Nous proposons d’adopter une vue plus déclarative : on ne se concentre pas sur le *comment*, mais sur le *pourquoi*, à savoir les ressources nécessaires et leurs relations. Une ressource est définie ici comme un ressource matérielle (une caméra, un gyroscope) ou comme une information sur le présent (un modèle du monde, l’état du robot) ou sur les futurs possibles (des plans). Par exemple, pour réaliser un certain type d’action, le robot a besoin d’un certain type de plans (dont l’origine importe peu). Si pour calculer ce plan le robot a besoin d’un planificateur, celui-ci nécessitera un modèle qui devra être calculé en amont. On passe donc d’une logique d’actions à une logique de relations entre ressources.

Avantages

Cette décomposition a plusieurs avantages par rapport à une approche basée sur les actions. La première est sa généricité par rapport à l’approche tâche. En effet, la notion de ressource est indépendante d’un robot particulier : on peut, pour un domaine d’application, construire une taxonomie, une ontologie des ressources possibles. Au contraire, la manière de faire une action dépend généralement du robot, et donc expliciter une action comme une séquence d’actions de plus bas niveau est potentiellement moins portable d’un robot à un autre. Par exemple, si un modèle a besoin de nuages de points 3D pour être construit, il n’a pas forcément besoin de savoir si ces données proviennent d’un Kinect, d’un lidar ou du résultat du calcul de stéréovision. Deuxièmement, les problèmes de concurrences sont, de manière inhérente, liées aux ressources, et non aux actions. En considérant en premier lieu ces ressources et leurs relations, il est possible de raisonner sur leurs états, et de détecter, puis d’analyser les problèmes de concurrences liés à ces ressources. Enfin, la majorité des architectures à composants sont fortement axés sur les flux de données échangés entre ces composants (*i.e.* les informations qu’ils produisent ou qu’ils consomment). En se concentrant sur les ressources et non sur les actions, la relation entre niveau fonctionnel et niveau décisionnel se fait plus naturellement, une ressource étant naturellement reliée aux composants qui la produisent.

Retour sur notre contexte d’étude

La figure 4.1 montre comment on peut décomposer la couche de contrôle du robot décrit dans la partie 2.2. On retrouve assez naturellement une décomposition proche de celle de la couche fonctionnelle, mais plusieurs ressources ont été abstraites. Ainsi, on s’intéresse maintenant à la position d’une cible potentielle (la ressource *targetPos*) et non exactement aux méthodes utilisées pour la calculer (les modules *detectTarget* et *trackTarget*). La ressource *pos* est particulièrement importante dans ce contexte : non seulement, elle encapsule la partie localisation, mais elle est aussi celle qui contient les stratégies les plus riches. En effet, une tâche de déplacement ou de suivi de cible va s’exprimer ici comme une contrainte sur la position courante du robot. La ressource *mission* elle, contient l’état courant de la mission, il s’agit donc d’une ressource de

haut niveau, sans équivalent au niveau de la couche fonctionnelle. Cette décomposition, même si elle dérive directement de la couche fonctionnelle de notre UGV reste générique, et les stratégies développées sur cette base peuvent être réutilisées dans d'autres contextes.

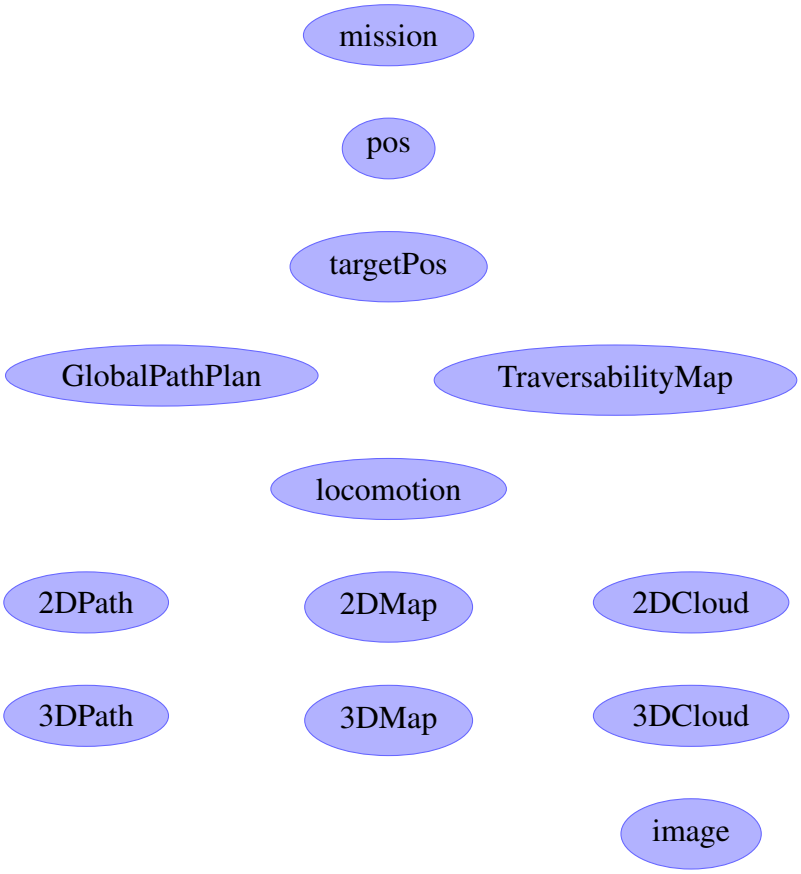

FIGURE 4.1 – Décomposition de la couche de contrôle de l'UGV mana en agents-ressources

4.1.3 Un graphe dynamique d'agents-ressources

Un graphe dynamique

Chaque agent va alors encapsuler une ressource spécifique, et garantit la cohérence de son état. On définit deux types d'échanges principaux entre ces agents. Le premier correspond à la lecture d'une donnée d'un agent par un autre agent, ou autrement dit la synchronisation partielle de la vue d'un agent sur un autre agent. Elle sera notée $A \overset{var}{\dashleftarrow} B$ pour indiquer que l'agent A veut connaître la valeur de *var* dans l'agent B. La seconde est au cœur de notre architecture logicielle. Elle correspond à l'envoi de contrainte d'un agent à un autre agent et est noté $A \xrightarrow{C_i} B$ pour un agent A qui demande à l'agent B d'appliquer la contrainte C_i. Cette contrainte du point de vue de A est un sous-objectif à garantir, et elle devient pour B un but à atteindre. Pour ce faire, l'agent B devra peut-être ajouter d'autres contraintes sur d'autres agents, ou bien appeler des méthodes

de la couche fonctionnelle. Ces mécanismes de propagation seront explicités en détail dans la section 4.2. Ces échanges vont permettent de définir un graphe orienté dynamique, où les nœuds sont les agents, et les messages constituent les arêtes, comme illustré par la figure 4.2. Ce graphe va évoluer en fonction des missions, des agents disponibles, des fautes rencontrées...

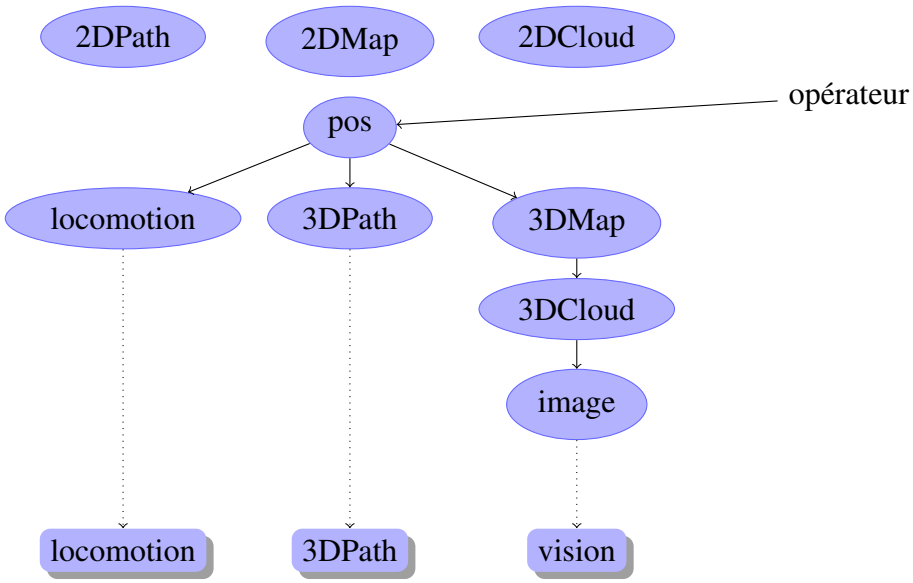

FIGURE 4.2 – Vue partielle du graphe d'agents après l'ajout d'une contrainte sur la position du robot par l'opérateur. Les agents sont représentés par des ellipses, les modules fonctionnels par des rectangles. Les flèches pleines correspondent à la relation de contraintes, et les flèches pointillées à une commande sur un module de la couche fonctionnelle. Dans ce cas précis, le système a choisi la stratégie reposant sur des modèles 3D.

Des contraintes

La définition des contraintes échangées entre les agents doit être assez expressive pour que le développeur puisse facilement exprimer ses besoins, mais suffisamment simple pour que le développeur, aussi bien que l'agent, puisse raisonner dessus – le premier pour prédire le comportement global du système, le second pour gérer les différentes contraintes auxquelles il est assujetti, tout en garantissant la cohérence de la ressource qu'il contrôle. Dans l'architecture ROAR, nous nous plaçons dans le cadre de la logique du premier ordre. Nous proposons d'utiliser des formules atomiques (*i.e.* uniquement composées de prédicats, variables et constantes) pour définir les contraintes. Le développeur peut définir ses propres prédicats et donc exprimer de manière souple des contraintes sur différents domaines. De l'autre côté, l'approche logique va permettre de faire différents raisonnements sur l'état interne de l'agent, et notamment décider quel comportement choisir pour assurer une contrainte et détecter les contraintes concurrentes (section 4.2.3).

Gestion des erreurs

Lors du déroulement d'une mission, de nombreuses causes d'échecs sont possibles, que ce soit une faute liée à la couche fonctionnelle, l'impossibilité temporaire de répondre à une contrainte, la disparition d'un agent... Les agents vont, dans un premier temps, chercher des solutions locales. Dans le cas où il n'y aurait pas de solution locale, plusieurs stratégies sont possibles. Dans le cas d'un problème liée à la concurrence, le système cherchera une solution visant à améliorer le nombre de tâches faites en parallèle, grâce à un nouvel arrangement des contraintes. Ce mécanisme sera expliqué en détail dans la section 4.3.3. Dans le cas d'une erreur plus générique et non traitée localement, l'architecture remonte dans le graphe des contraintes, via une technique de backtracking. L'utilisation d'un historique lors de la remontée dans le graphe permet d'éviter les chemins conduisant à une faute, et accélère le choix d'une bonne stratégie de récupération. Cette approche sera détaillée dans la section 4.3.2.

4.1.4 Synthèse

L'architecture ROAR propose donc de découper l'architecture de contrôle d'un robot autonome en un ensemble d'agents, encapsulant chaque ressource du robot. Les échanges de contraintes sous forme de formules logiques atomiques entre les agents forment un graphe dynamique, et permettent l'évolution du système. Ces formules logiques seront ensuite utilisées pour raisonner sur la cohérence de chaque ressource et déclencher les comportements garantissant ces contraintes. Deux mécanismes génériques sont proposés pour résoudre les problèmes au niveau globale du système. Ces différents mécanismes vont maintenant être détaillés dans les sections suivantes.

4.2 Structure d'un agent ROAR

4.2.1 Vue d'ensemble

La figure 4.3 illustre le modèle général d'un agent ROAR. On y trouve deux grands types d'entités, les tâches et les recettes.

Les tâches définissent des transitions possibles entre deux états logiques. Lors de la réception d'une contrainte par un agent, celui-ci, à l'aide de son moteur logique, va décider si il peut ou non, en fonction de son contexte courant, gérer cette nouvelle contrainte. En cas de succès, il calcule un ensemble de tâches à exécuter. C'est ce niveau logique qui garantit la cohérence de la ressource encapsulée par l'agent. Étant donné que la description d'une tâche dépend uniquement de formules logiques et du contexte de l'agent, celle-ci est générique et peut être utilisée dans différents contextes robotiques.

Les tâches décrivent des modifications d'états logiques, mais n'explicitent pas comment sont effectués ces changements. C'est le rôle de la couche exécutive de sélectionner une recette, en fonction du contexte courant, pour réaliser une tâche. Le développeur peut créer plusieurs recettes pour une même tâche : cela peut permettre d'implémenter plusieurs stratégies en fonction des agents / capacités disponibles, gérer explicitement certaines fautes... À cette granularité, des

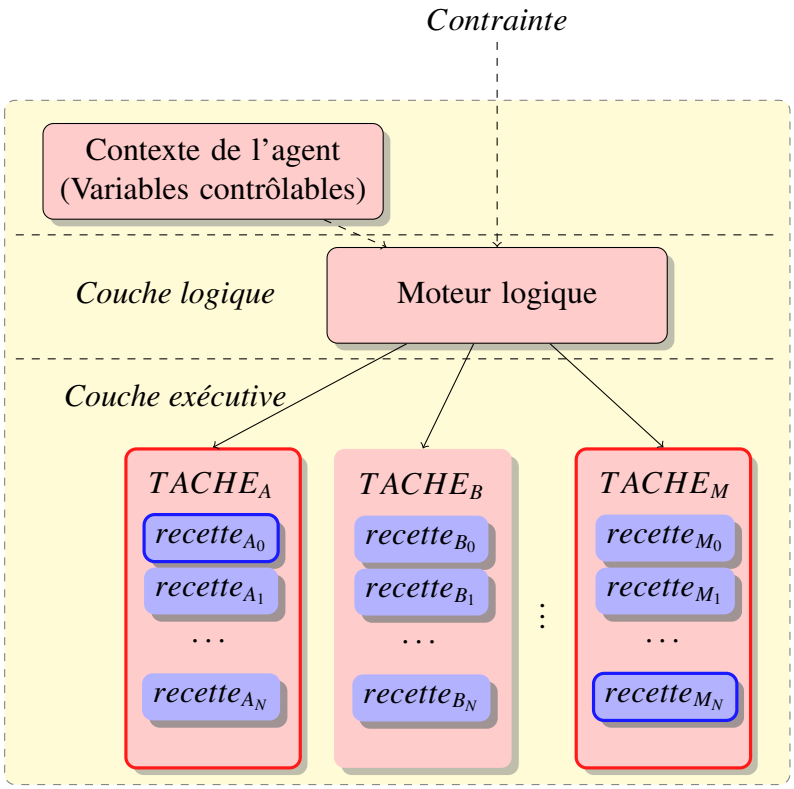

FIGURE 4.3 – Structure générale d'un agent-ressource. Les tâches entourées par des lignes grasses rouges sont les tâches actuellement sélectionnées par la couche logique, les recettes entourées de gras sont les recettes sélectionnées par la couche exécutive – plusieurs tâches peuvent être activés en parallèle, mais seulement une recette peut être activé pour une tâche active.

appels à la couche fonctionnelle peuvent être effectués. Dans ce cas là, la recette n'est utilisable que dans certain environnements spécifiques.

La séparation des concepts de tâches et de recettes a deux grandes vocations :

- réduire la complexité pour sélectionner les actions à effectuer, et donc réduire le temps de calcul pour faire ce choix. En effet, en considérant m tâches auxquelles sont associées n recettes par tâche (en moyenne), on a $n + m$ choix possibles, contre $n * m$ en l'absence de hiérarchisation.
- permettre de proposer différentes stratégies pour la même transition d'un état logique à un autre. En particulier, il est possible d'implémenter différentes stratégies pour une même transition afin de gérer les différentes plateformes logicielles et matérielles disponibles.

Autrement dit, cette hiérarchisation permet à la fois de gagner au niveau réactivité, mais aussi au niveau de la portabilité sur les différentes plateformes robotiques.

Nous allons maintenant détailler les parties "contexte", "couche logique" et "couche exécutive" d'un agent en décrivant la syntaxe utilisée pour les décrire et en explicitant les algorithmes qu'elles contiennent. Chaque partie sera illustré au travers d'un exemple implémentant une stra-

tégie de navigation autonome.

4.2.2 Contexte d'un agent

Le contexte d'un agent contient deux grands types d'informations. D'un côté, il peut définir un ensemble de types spécifiques ainsi que les prédicats qui leur sont associées. De l'autre, il permet de définir le contenu de l'agent, à savoir ses données internes et celles qu'il expose.

Interface

L'architecture ROAR arrive inclut un certain nombre de types classiques préexistants, à savoir les types `bool`, `int`, `double`, et `string` et leur prédicats associés (opérateurs d'égalité et de comparaisons). Toutefois, ces types sont souvent insuffisants pour exprimer des problèmes robotiques : par exemple, on peut vouloir exprimer des primitives géométriques comme des points, des distances, et exprimer des relations sur ces types. L'architecture offre donc la possibilité au développeur de définir des nouveaux types et prédicats sur ces types.

Le mot-clé **struct** permet de définir de nouveaux types produits (*i.e.* au sens produit cartésien) de manière similaire à la construction **struct** du langage C. Les lignes 4 et 7 du listing 4.1 illustrent cette possibilité via la définition des types `position` et `speed`. Les lignes 10-11 déclarent deux fonctions qui travaillent sur ces types. À ce niveau, les fonctions sont uniquement déclarées, et non implémentées. Elles sont implémentées manuellement, au niveau du langage hôte (en l'occurrence C++ pour notre implémentation, voir la section 5.1). Cela permet de conserver de bonnes performances et de faire facilement le lien avec la couche fonctionnelle des robots. Enfin, la ligne 14 définit une règle logique, qui pourra être utilisée dans la couche logique afin de décider si une tâche permet de garantir ou non une contrainte. En l'occurrence, la règle `distance_symmetry` signifie que pour tout A, B point, la distance entre A et B est égale à la distance entre B et A.

Listing 4.1 – Définition de nouveaux types et prédicats : exemple de l'interface localisation

```
1  loc = agent {
2          {
3                  /* in meters */
4                  position = struct { double x, y, z; };
5
6                  /* in m/sec and radians/sec */
7                  speed = struct { double v, w; };
8          }
9          {
10                 double distance(position A, position B);
11                 speed make_speed(double, double);
12         }
13         {
14                 distance_symmetry = distance(A, B) => distance(A,B) == distance(B,A)
15         }
16 }
```

Chaque agent peut ainsi définir ses propres types de données et prédicats. Naturellement, comme le système est ouvert, il est possible que plusieurs agents définissent des prédicats portant le même nom. Pour résoudre ces conflits, chaque type et prédicat est référencé dans la suite par

un nom complet composé du nom de l'agent qui le définit et de son nom réel. Ainsi, le listing précédent définit la fonction `loc::distance` et n'entrera pas en conflit avec la définition de la fonction `time::distance` définie par l'agent *time*. Plus généralement, on définit le domaine d'un type, d'une règle, ou d'une fonction comme l'agent qui le définit.

Contexte

Le contexte de l'agent expose son état interne et externe. Il est défini par trois ensembles de variables, chacun correspondant à des droits d'accès spécifiques pour les autres agents. Les variables contrôlables $\{C_v\}$ correspondent aux variables sur lesquelles les autres agents peuvent mettre des contraintes. Le second ensemble correspond aux variables en lecture seule $\{P_v\}$, que les autres agents peuvent uniquement lire. Elles correspondent généralement à l'information stockée dans l'agent. Enfin, les variables privées $\{P_v\}$ ne sont pas accessibles par les autres agents, et servent à maintenir l'état interne de l'agent.

Une variable est définie par un type et facultativement une méthode d'initialisation, via la syntaxe suivante :

$$< type >< var > [= initializer][, < var2 > [= initializer]...];$$

Le type restreint à la fois les valeurs acceptables pour cette variable, mais aussi les contraintes qu'elle est susceptible d'accepter car le système n'accepte que des contraintes bien formées (correctes au niveau des types utilisés). Deux types de méthodes d'initialisation sont possibles. La première est classique : on assigne une constante à la variable, qui prend cette valeur lors du démarrage de l'agent. La seconde correspond à l'exécution d'une tâche lorsque l'on veut lire la valeur de cette variable. Cela permet d'effectuer des calculs à chaque accès à cette variable, permettant ainsi de refléter un état courant, déduit à partir des données de modules fonctionnels ou d'autres agents. Enfin, si une variable n'a aucune méthode d'initialisation, elle n'a par défaut aucune valeur, et toute expression utilisant cette variable sera évaluée à `undefined`.

Listing 4.2 – Contexte de l'agent *2DMap*

```
1  2DMap = agent {
2      context = {
3          {
4            bool empty = true;
5            position last_update;
6          }
7          { string port = get_port(); }
8          { bool init = false;}
9      }
10 }
```

Le listing 4.2 illustre la syntaxe concrète pour déclarer l'interface de l'agent *2DMap*. Les lignes 3 à 6 définissent les variables contrôlables, à savoir la variable *empty*, initialisée à `true` par défaut (la carte est vide lors de sa création) et la variable *last_update*. Celle-ci ne sera initialisée qu'à partir du moment où l'on rajoutera effectivement des données dans le carte. La ligne 7 définit l'unique variable en lecture : la variable *port* correspond à un identifiant de port et l'agent appellera la tâche *get_port* quand un agent voudra lire la valeur de cette variable. Ici, cela répond à plusieurs besoins : tout d'abord, il faut que le(s) module(s) soi(en)t initialisé(s) avant qu'on

puisse accéder à leur port, et en second lieu, il est possible qu'au cours du temps, l'agent décide de changer de modules fournisseurs. Enfin la ligne 7 définit la variable privée *init*, naturellement initialisée à la valeur `false`.

4.2.3 La couche logique

Programmation par contrat

La programmation par contrat est un paradigme de programmation semi-formel proposé dans (Meyer, 1992). L'idée principale de ce paradigme consiste à expliciter un contrat entre le fournisseur d'une fonction et celui qui l'utilise via un ensemble d'assertions. B. Meyer propose trois types d'assertions :

- les préconditions : des conditions qui doivent être vérifiées avant l'exécution de la fonction, afin de garantir son bon fonctionnement.
- les postconditions : les conditions qui doivent être vérifiées après l'exécution de la fonction.
- les invariants : des conditions qui doivent être vérifiées au cours de l'exécution de la fonction.

Les préconditions peuvent être interprétées de différentes manières. Dans le langage Eiffel (Meyer, 1992), les préconditions non vérifiées déclenchent une erreur fatale afin que le développeur puisse corriger au plus tôt l'appel incorrect. Dans sa version concurrente Scoop (Morandi et al., 2008), dans le cas où des préconditions sont non vérifiées, le fil d'exécution courant est arrêté jusqu'à ce que toutes les préconditions soient vérifiées et que la fonction puisse être exécutée. Dans ROAR, nous donnons un rôle actif à ces préconditions : nous allons chercher à les résoudre de manière active.

Description des tâches

Une tâche dans ROAR est définie par son nom et son contrat. Concrètement, une tâche est définie par trois ensembles d'assertions **pre**, **post** et **maintain** pour respectivement les préconditions, les postconditions et les invariants. De manière similaire aux contraintes, les assertions correspondent à des formules logiques atomiques.

Le listing 4.3 illustre la syntaxe de description des tâches de ROAR. Nous y déclarons quatre tâches pour l'agent *2DMap*. La tâche *init* a un unique effet visible sur l'agent, qui est de faire passer la variable *init* de `false` à `true`. Évidemment, la tâche peut avoir d'autres effets, en particulier ajouter des contraintes sur d'autres agents, ou appeler des fonctions de la couche fonctionnelle. La tâche *fuse* ajoute de nouvelles informations dans la carte. Ses effets observables sont la modification de la variable *empty* (la carte n'est plus vide une fois qu'on a ajouté de l'information) et de la variable *last_update*. Dans ce cas, on n'a pas de renseignement sur la valeur exacte de cette variable, si ce n'est qu'elle satisfera un certain prédicat après l'exécution de la tâche. La tâche *clear* supprime toutes les informations de la carte, par exemple lorsqu'on a détecté des incohérences dans la carte. On voit ici clairement que les tâches *fuse* et *clear* sont incompatibles, car leur postconditions le sont : on ne peut pas avoir *empty* à la fois à `true` et

à `false`. Enfin, la tâche *get_port* va retourner le port courant. Il est à noter qu'elle n'a pas de postconditions et donc sera uniquement appelée pour la mise à jour de la variable *port*. Les différentes tâches de l'agent *2DMap* sont par nature "discrètes", elles n'ont donc pas de clauses **maintain**. Des agents comme *locomotion* avec des sémantiques plus continues tirent eux parti de cette possibilité.

Listing 4.3 – Description des tâches pour l'agent *2DMap*

```
1  init = task {
2    pre = {{ init == false }}
3    maintain = {}
4    post = {{ init == true }}
5  }
6
7  fuse = task {
8    pre = {{ init == true }}
9    maintain = {}
10   post = {
11     { loc::distance(last_update, pos::current) < 0.2 }
12     { empty == false }
13   }
14 }
15
16 clear = task {
17   pre = {{ empty == false }}
18   maintain = {}
19   post = {{ empty == true }}
20 }
21
22 get_port = task {
23   pre = {{ init == true }}
24   maintain = {}
25   post = {}
26 }
```

Critère d'adéquation entre une tâche et une contrainte

Nous avons décrit statiquement la notion de tâches dans ROAR. Nous allons maintenant nous intéresser à comment est fait le lien entre spécifications de tâches et contraintes.

Le domaine d'une expression est l'ensemble des domaines des types et prédicats qui apparaissent dans cette expression. Pour une tâche, la définition est étendue à l'ensemble des domaines des postconditions et des invariants. Ainsi, le domaine pour l'expression

$$loc :: distance(last\_update, pos :: current) < 0.2$$

est l'ensemble {*loc*} car *last_update* et *pos::current* sont de type `loc::position` défini dans l'agent {*loc*} comme montré dans le listing 4.1. Le type `double` apparaît aussi dans cette expression, mais il vient de base dans le système ROAR, et est donc toujours pris en compte. Le domaine de la tâche *fuse* est donc l'ensemble {*loc*} tandis que le domaine de la tâche *clear* est l'ensemble vide.

Pour décider si une tâche T peut gérer une contrainte C, l'agent va essayer de résoudre la clause de Horn suivante :

$$Post_1^T \wedge Post_2^T \wedge \cdots \wedge Post_i^T \wedge Inv_1^T \wedge Inv_2^T \wedge \cdots \wedge Inv_k^T \wedge R_1 \wedge \cdots \wedge R_i \rightarrow C$$

où $Post_i^T$ représente la i^{me} postcondition de la tâche T, Inv_j^T représente le j^{me} invariant de la tâche T et R_j les règles associées au domaine de la tâche. Par exemple, quand l'agent considère le système logique liée à la tâche *fuse*, il utilise les règles associées à l'agent *loc*, en particulier la règle `distance_symmetry`. Autrement dit, une contrainte peut être gérée par une tâche si on peut déduire la formule logique associée à la contrainte depuis les effets de la tâche.

Pour illustrer le type de problèmes générés, considérons que l'agent *2DMap* reçoive la contrainte C suivante :

$$loc :: distance(last\_update, pos :: current) < 0.5$$

L'agent va considérer les problèmes suivants (respectivement pour les tâches *fuse* et *clear*) :

$$\begin{gathered}(loc :: distance(last\_update, pos :: current) < 0.2) \\ \wedge (empty = false) \\ \wedge (\forall A, B\ distance(A, B) = distance(B, A)) \\ \rightarrow (loc :: distance(last\_update, pos :: current) < 0.5)\end{gathered}$$

$$\begin{gathered}(empty = true) \\ \rightarrow (loc :: distance(last\_update, pos :: current) < 0.5)\end{gathered}$$

La résolution d'un tel système mène à différents résultats :

- on peut prouver C directement à partir des prémisses, cela signifie que le contrat de la tâche remplit les besoins énoncés par la contrainte. C'est ce qu'on obtient en particulier lorsqu'on résout le système précédent pour la tâche *fuse*.
- on peut prouver $\neg C$, cela signifie que le contrat de la tâche est en contradiction avec la contrainte, elle n'est donc pas une bonne candidate.
- on ne peut rien déduire, ni montrer C, ni $\neg C$. Cela indique que la tâche exécute des choses orthogonales à celles demandées par la contrainte. C'est le résultat du système précédent pour la tâche *clear*.
- on peut déduire C sous une certaine liste d'hypothèses. La tâche peut correspondre à la contrainte, pourvu que l'on garantisse d'abord que les hypothèses soient vérifiées.

On introduit donc la méthode `proof` qui va résoudre un tel système, et retourner respectivement pour chacun des cas $\top$, $\bot$, [] (la liste vide), et la liste des hypothèses. On utilise la méthode de résolution SLD (Lloyd, 1987) pour résoudre de tels systèmes. Le calcul est stoppé au bout d'un certain temps paramétrable afin d'assurer la réactivité. Toutefois, étant donné la taille les problèmes que l'on considère, il est extrêmement rare d'atteindre une telle limite.

D'une contrainte à un arbre de tâches

Maintenant que nous avons un critère pour décider si une tâche permet de gérer une contrainte, nous allons décrire l'algorithme général pour choisir un ensemble de tâches pour gérer correctement une contrainte, étant donné le contexte courant de l'agent.

Tâches incompatibles Tout d'abord, introduisons le concept de tâches incompatibles entre elles. Deux tâches T_1 et T_2 sont incompatibles si leurs invariants et leurs postconditions sont contradictoires. On note $\{T_1^{\perp}\}$ l'ensemble des tâches incompatibles avec T_1. Ces ensembles peuvent être calculés statiquement.

Contexte dynamique Nous considérons la partition de l'ensemble des tâches en trois sous-ensembles, les tâches disponibles $\{T_d\}$, les tâches en cours d'exécution (plus précisément, les tâches qui ont été sélectionnées pour être exécutées) $\{T_e\}$, et les tâches incompatibles $\{T_i\}$, *i.e.* les tâches incompatibles avec les tâches de $\{T_e\}$. Cette partition est le contexte dynamique d'un agent.

Algorithme général L'algorithme 1 montre comment, étant donné le contexte d'un agent (*i.e.* ici les tâches disponibles), calculer une solution pour aller de l'état courant à un état qui satisfera la contrainte. La fonction `compute_tree` commence par exploiter la fonction `proof` pour déterminer les tâches susceptibles de répondre à la contrainte (entre les lignes 12 et 21). On élimine les cas $\perp$ et [] qui correspondent à des tâches sans rapport avec la contrainte C. On va alors trier les tâches restantes selon les critères suivants :

1. on préfère les tâches qui permettent de gérer directement la contrainte, *i.e.* sans hypothèses supplémentaires.
2. on sépare les hypothèses supplémentaires en hypothèses locales (celles qui vont influer sur l'état de l'agent) et les hypothèses distantes (qui vont impacter un autre agent).
3. on essaye alors d'abord de minimiser le nombre d'hypothèses distantes puis le nombre d'hypothèses locales, l'idée étant ici de privilégier les solutions locales, ayant le moins d'impacts sur le reste du système.

Ensuite, pour chaque tâche, on évalue les préconditions. Les préconditions qui échouent sont considérées comme des nouvelles contraintes à gérer, de même que les hypothèses locales. On va alors chercher récursivement à résoudre ces contraintes. L'algorithme termine lorsqu'il n'y a plus de contraintes à gérer (ligne 4) ou au contraire qu'on échoue à trouver des solutions pour une contrainte (31). L'algorithme termine car à chaque étape, on réduit l'espace de recherche d'au moins une tâche (ligne 25). Celui-ci peut donc devenir vide et entraîner un échec. La diminution des tâches disponibles garantit non seulement la terminaison de l'algorithme, mais assure aussi l'absence de boucles et d'effets de va-et-vient, le premier parce qu'on ne peut réutiliser une tâche directement, et le second parce qu'on interdit les tâches qui ont des effets contradictoires. Une fois que l'agent a calculé un arbre de tâche, il va ajouter cet arbre au contexte dynamique de l'agent, *i.e.* déplacer des éléments de $\{T_d\}$ vers $\{T_e\}$ et $\{T_i\}$.

Illustration Pour illustrer cet algorithme, considérons l'agent *2DMap* dans l'état initial ($init = false$, $empty = true$, *last_update* indéfinie) qui reçoit la contrainte

$$loc :: distance(last\_update, pos :: current) < 0.5$$

Il va d'abord essayer de l'évaluer, mais va échouer car *last_update* n'est pas définie. Il va alors chercher avec l'algorithme précédent à cherche un arbre de tâches permettant de gérer cette contrainte. La seule tâche qui corresponde à cette contrainte est la tâche *fuse* qui a pour précondition *init = true*. L'agent *2DMap* évalue cette assertion à `false` et donc va chercher une solution à cette nouvelle contrainte. Il trouve alors la tâche *init* dont la précondition est évaluée à `true`. L'algorithme termine donc avec comme résultat la séquence de tâches *init* puis *fuse*.

Algorithm 1 L'algorithme compute_tree

```
1:
2:  compute_tree(Tasks, conditions) :
3:  if conditions is empty then
4:     return (⊤, Tasks)
5:  else
6:     if Tasks is empty then
7:        return ⊥
8:     end if
9:     p ← head(conditions)
10:    conditions ← tail(conditions)
11:    possibles ← []
12:    for all T in Tasks do
13:       res ← proof(T, p)
14:       if res = ⊤ or res ≠ [] then
15:          if res = ⊤ then
16:             possibles ← possibles :: T
17:          else
18:             possibles ← possibles :: (T, res)
19:          end if
20:       end if
21:    end for
22:    possibles ← sort(possibles)
23:    for all T in possibles do
24:       conditions2 ← conditions ∪ failed_pre(T)
25:       Tasks2 ← Tasks − {T⊥} − T
26:       next ← compute_tree(Tasks2, conditions2)
27:       if next ≠ ⊥ then
28:          return node :: next
29:       end if
30:    end for
31:    return ⊥
32: end if
```

Gestion de multiples contraintes

Contraintes et tâches actuellement exécutées Lorsqu'une nouvelle contrainte arrive, l'agent va d'abord vérifier si il n'a pas déjà prévu un moyen de la satisfaire. Pour cela, on utilise la fonction `proof` sur les tâches de $\{T_e\}$. Si cela réussit, cela signifie qu'une tâche active, ou qui va être activée permet de gérer cette contrainte, et l'agent va donc juste réutiliser cette tâche.

Détection des problèmes de contraintes concurrentes Lorsque l'algorithme `compute_tree` ne trouve pas de solution, cela peut signifier deux choses : soit l'agent n'a réellement pas de moyens pour effectuer la tâche, soit l'agent n'a pas actuellement de méthodes pour l'effectuer, *i.e.* gérer cette contrainte en même temps que les contraintes courantes n'est pas possible (cas de concurrence). Ces deux cas d'erreurs étant traités différemment (voir section 4.3), il faut que l'agent soit capable de les différencier. Pour décider si il s'agit d'un problème lié à la concurrence, on va simplement appliquer la méthode `proof` sur $\{T_i\}$. Si une de ces tâches permet de gérer la contrainte, alors nous sommes en face d'un problème de concurrence.

Des calculs en parallèles Plusieurs contraintes peuvent arriver dans des temps proches et donc les calculs d'arbres de tâches peuvent se faire en parallèle. Toutefois leur exécution parallèle n'est pas forcément possible, entre autres parce que les deux arbres peuvent contenir des tâches contradictoires. La fonction `compute_task_tree` présentée dans l'algorithme 2 intègre les deux points précédents, à savoir la gestion des contraintes vis-à-vis des tâches déjà en cours d'exécution et la détection des problèmes de concurrence, ainsi qu'un système de transaction atomique. La fonction va lancer les calculs nécessaires, d'abord dans les tâches de $\{T_e\}$, puis la méthode `compute_tree` sur les tâches disponibles. À la fin du calcul, si elle a trouvé une solution, elle va appeler la fonction `commit` qui va modifier, de manière atomique, le contexte dynamique de l'agent, lorsque c'est possible. Soit $\{T_e^l\}$ les tâches localement sélectionnées pour exécution et $\{T_i^l\}$ les tâches incompatibles associées. La fonction `commit` va alors vérifier que

$$\{T_e^l\} \subset \{T_e\} \cup \{T_d\}$$
$$\{T_i^l\} \subset \{T_i\} \cup \{T_d\}$$

Si ces conditions ne sont pas vérifiées, cela signifie que le contexte dynamique a changé de manière incompatible avec la solution trouvée, et il faut donc faire le calcul à nouveau, en considérant le nouveau contexte dynamique. Dans le cas contraire, le contexte dynamique est modifié et on peut alors commencer l'exécution des tâches.

4.2.4 La couche d'exécution

Nous avons décrit comment sont choisies les tâches, nous allons maintenant nous intéresser à leur exécution. Les tâches, telles que nous l'avons décrit, n'ont pas en soi une sémantique exécutable, elles ne font qu'exprimer un contrat. Ce sont les recettes qui vont implémenter ces tâches. Une recette est donc une implémentation spécialisée pour une tâche, avec possiblement

Algorithm 2 compute_task_tree algorithm

```
1:
2:  compute_task_tree(constraint) :
3:  b ← false
4:  while not b do
5:     found ← false
6:     for all T in {T_e} do
7:        if found = false then
8:           res ← proof(T, constraint)
9:           if res = ⊤ then
10:             found ← true
11:          end if
12:       end if
13:    end for
14:    if found = false then
15:       tasks ← {T_e} ∪ {T_d}
16:       res ← compute_tree(tasks, contraint))
17:       if res = ⊥ then
18:          for all T in {T_i} do
19:             res ← proof(T, constraint)
20:             if res = ⊤ then
21:                return ERROR_CONCURRENCY
22:             end if
23:          end for
24:          return NO_SOLUTION
25:       end if
26:    end if
27:    b ← commit(agent, res)
28: end while
29: return OK
30:
```

un contrat plus strict. En particulier, elles vont avoir des préconditions plus spécifiques, qui permettent de décider quelle recette sélectionner pour une situation donnée. On utilisera donc la notion de situation pour désigner l'ensemble des préconditions d'une recette. La recette est alors composée de deux choses : une situation, et une implémentation (ou corps) décrite dans le langage ROAR.

Le langage d'exécution de ROAR

Le langage d'exécution de ROAR (ou langage ROAR) est composé de trois grands types de primitives, les premiers permettant de manipuler les variables de l'agent, les second permettant d'observer le monde, et les derniers permettant de contraindre d'autres agents.

Manipulation de variables La primitive **let** <var> <expr> permet de créer une variable locale et de lui affecter comme valeur le résultat de l'expression *expr*. Le type de la variable est automatiquement déduit depuis le type résultat de *expr*. De même **set** <var> <expr> permet de modifier la valeur de la variable *var* par le résultat de l'expression *expr*. Le type de *var* doit vérifier le type résultat de *expr*. Enfin, la primitive **letname** <var> <expr> assigne à *var* l'expression *expr* : *var* peut alors être vue comme une référence vers cette expression. Il est bien évidemment possible d'appeler des fonctions sans en récupérer le résultat, par exemple, celles qui vont avoir comme effet d'appeler une méthode de la couche fonctionnelle.

Observation La primitive **assert** <expr> va vérifier que l'invariant *expr* est vérifié au cours de l'exécution de la recette. En utilisant cette primitive, il est possible de vérifier que la situation courante est toujours compatible avec les besoin de nos recettes. Si cet invariant est brisé, la recette échoue. Une seconde primitive **wait** <expr> va bloquer tant que la condition *expr* n'est pas vérifiée.

Contrainte Les précédentes primitives étaient passives, elles se contentaient de vérifier des propriétés sur l'environnement. Au contraire, les deux primitives suivantes permettent de contraindre les autres agents. La primitive **make** <expr> envoie la contrainte *expr* à un agent *A* et attend tant qu'elle n'est pas satisfaite. Un échec entraîne l'échec de la recette. L'agent *A* est automatiquement déduit de la contrainte : ROAR considère que la contrainte sur l'agent contrôlant la variable apparaissant en premier dans *expr*. De même, **ensure** <expr> envoie la contrainte *expr* à l'agent *A* qui devra la satisfaire tant que celle-ci n'est pas annulée. Pour cela, la primitive **ensure** renvoie un identifiant de contrainte qui pourra être annulé via la primitive **abort** <id>.

Primitives de contrôle Le langage ROAR de base ne contient pas de primitive de contrôle classique comme **if** ou **while**, le premier parce qu'il peut facilement être remplacé par un prédicat dans la situation, et le second parce qu'il ne nous semble pas utile dans une logique de coordination. Les rajouter n'apporterait cependant pas de difficultés particulières.

Fonctions Enfin, il est possible de déclarer des fonctions locales via la syntaxe **let** <nom_fun> <arg1> <arg2> = **fun** .... Le type des arguments est automatiquement déduit de leur utilisation dans le corps de la fonction. Ces fonctions sont considérés comme des blocs, les contraintes et les invariants sont automatiquement abandonnés en sortie de fonction.

Listing 4.4 – Une recette décrivant le processus de coordination pour la navigation en terrain difficile

```
1  letname mapping_ctr loc::distance(3DMap::last_update, pos::current) <= 0.5
2  letname plan_ctr 3DPlan::goal == 3DPlan::expected_goal
3  letname exec_ctr locomotion::expected_port == locomotion::port
4  letname terminaison_cond loc::distance(pos::current, pos::goal) < 2.0
5  letname monitor_ctr trajectory::expected_port == trajectory::tracked_port
6
7  let follow_goal = fun {
8    let dist loc::distance(pos::current, pos::goal)
9    let max_time dist * locomotion::v_mean * 1.5
10   let now time::current
11
12   assert(time::delay(now) < max_time)
13
14   ensure(plan_ctr where 3DPlan::expected_goal == goal)
15   ensure(exec_ctr where locomotion::expected_port == 3DPlan::port)
16   ensure(monitor_ctr where trajectory::expected_port == 3DPlan::port)
17
18   wait(terminaison_cond)
19 }
20
21 let move_blind_zone speed_ distance = fun {
22   let speed make_speed(speed_, 0.0)
23   let id ensure(locomotion::_speed == locomotion::speed where locomotion::_speed == speed)
24   let cur pos::current
25
26   wait(distance(pos::current, cur) > distance)
27   abort id
28 }
29
30 go_to = recipe {
31   pre = {{3DMap::empty == true}}
32   body = {
33     make(delay(3DMap::last_refresh) < 50.0)
34
35     ensure(mapping_ctr)
36     move_blind_zone(0.5, 3.0)
37
38     follow_goal()
39   }
40 }
```

Exemple Le listing 4.4 exhibe la syntaxe de ROAR au travers d'un recette permettant de naviguer en terrain difficile. On peut y trouver deux fonctions `move_blind_zone` et `follow_goal`. La première a pour but de sortir de la zone aveugle. En effet, les capteurs ont une distance minimum d'acquisition, le robot a donc une zone sans information dans laquelle il ne peut pas planifier. Pour réaliser cela, on utilise une contrainte continue sur l'agent *locomotion* (ligne 23) et on attend que le robot ait parcouru la distance *distance* (ligne 26). La seconde est plus intéressante. D'abord, on estime le temps maximal tolérable pour le déplacement (ligne 8-9) et on ajoute un invariant sur cette durée maximale (ligne 12). On va ensuite contraindre respectivement l'agent

3DPlan à avoir un plan correct, puis l'agent *locomotion* a exécuter ce plan, et enfin l'agent *trajectory* à enregistrer les plans fournis par l'agent *3DPlan*. Enfin l'agent attend jusqu'à ce que le robot soit accès proche du but (ligne 18). La recette *go_to* se contente alors d'émettre une contrainte sur l'agent *3DMap* (un rafraîchissement selon un critère spatial) (ligne 36) puis appelle en séquence les deux fonctions précédentes. On retrouve une boucle SPA classique, avec en plus des moniteurs et des conditions de terminaisons.

Sélection des recettes

De multiples recettes peuvent être associées à une même tâche, afin de s'adapter au mieux à la situation courante. Pour cela, on doit d'abord déterminer quelles sont les tâches acceptables vis-à-vis de la situation courante.

Commençons par définir le domaine d'une recette. On appelle domaine d'une recette l'ensemble des agents auxquelles une recette envoie des contraintes (directement ou via une fonction). Par exemple, le domaine de la recette *go_to* décrite dans le listing 4.4 est :

$$domain(go\_to) = \{3DMap, LocalPathPlan, locomotion, trajectory\}$$

Une recette R est alors un choix acceptable si et seulement si :

- $domain(r) \subset available\_agents$
- $\forall P \in \{Pre_R\}\ eval(P) = true$

Étant donnée une situation donnée, il est possible que l'ensemble des recettes acceptables soit vide. Dans ce cas là, la tâche n'a aucun moyen pour s'exécuter et donc échoue. Au contraire, il est possible qu'il y ait de multiples recettes acceptables. Dans ce cas là, on utilise une heuristique pour choisir la "meilleure" recette. Par exemple, on propose d'utiliser la recette la plus spécifique, *i.e.* celle qui a le plus grand nombre de préconditions. D'autres heuristiques peuvent être définies, comme par exemple minimiser la taille du domaine de la recette (afin de réduire l'impact sur le système global) ou bien exploiter une fonction de coût pour choisir une recette plus "efficace". La définition de ces heuristiques est un point d'entrée qui permet au développeur de définir des stratégies spécifiques de contrôle du robot.

4.3 Interactions entre agents

Nous allons maintenant nous intéresser aux échanges entre les agents, et en quoi ces échanges influencent leur comportement. En particulier, nous allons nous intéresser aux changements d'états lors de la propagation d'une contrainte, puis aux mécanismes de gestion d'erreurs, qu'elles soient liées à une faute ou un problème de concurrence.

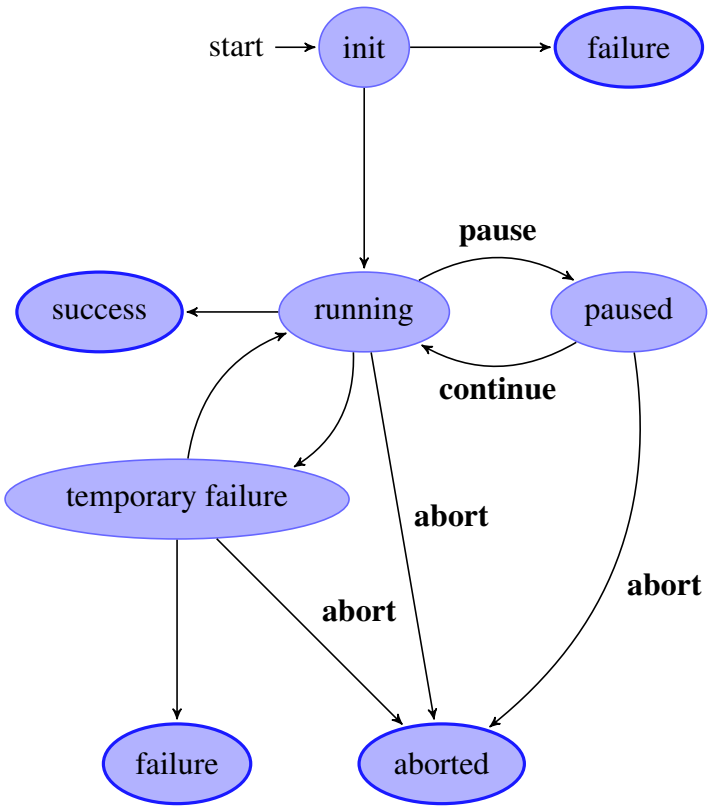

FIGURE 4.4 – Machine à états pour la gestion d'une contrainte. Les transitions en gras correspondent aux transitions déclenchées par un message externe.

4.3.1 Gestion des contraintes

Changement d'états au cours de la vie d'une contrainte

Lorsque un agent reçoit une contrainte, celle-ci passe par plusieurs étapes résumées dans la machine à états figure 4.4. À chaque transition, l'appelant est informé via l'envoi d'un message d'information spécifique. Comme on l'a vu précédemment, lors de la réception d'une contrainte, l'agent va essayer de trouver une solution pour gérer cette contrainte, puis dans un second temps exécuter les tâches qui la satisferont. Si il ne trouve pas de solution, la contrainte passe directement dans l'état *failure*. Dans le cas nominal, elle passe dans l'état *running* au moment où les tâches qui la gèrent commencent réellement à s'exécuter (et non une de ces préconditions). Si nous reprenons l'exemple de la section 4.2.3, la contrainte passe dans l'état *running* lors de l'exécution de la tâche *fuse*. Depuis cet état *running*, plusieurs évolutions sont possibles :

- Le cas nominal est que la tâche se termine avec succès, et donc que la contrainte est effectivement satisfaite : la contrainte passe alors dans l'état *success*.
- Un second cas possible est la rencontre d'un aléa : l'agent va alors chercher une solution localement (ce processus sera décrit plus précisément dans 4.3.2). Dans ce cas, la contrainte passe dans l'état *temporary failure*. Depuis cet état, soit l'agent trouve effectivement une

solution locale, et la contrainte reprend alors l'état *running*, soit l'agent échoue à trouver une solution locale et la contrainte passe dans l'état d'échec définitif *failure*.
– Enfin, l'agent peut réagir à deux stimuli externes. Lors de la réception d'un message `pause`, la contrainte passe dans l'état *paused*, elle pourra revenir dans l'état *running* lors de la réception d'un message `continue`. Et lors de la réception d'un message `abort`, la contrainte est annulée et passe dans l'état *aborted*.

Dépendances entre les contraintes et rôle de l'état *paused*

Dépendances entre contraintes Dans une boucle SPA classique, la phase de planification n'a de sens que si le modèle est valide, et de même l'exécution n'a de sens que si la planification a effectivement construit un plan valide. Il existe donc des liens de dépendances entre ces trois constructions. On retrouve implicitement des dépendances analogues dans ROAR, via l'ordre de déclaration des contraintes (avec les primitives **make** et **ensure**). Si l'on considère à nouveau le listing 4.4, il apparaît assez clairement que la contrainte *plan_ctr* n'a de sens que si la contrainte *mapping_ctr* est bien maintenue, et de même pour pour les contraintes *exec_ctr* et *monitor_ctr*. On considère donc la relation transitive de dépendance entre les contraintes, et celle-ci est directement liée à l'ordre de déclaration de ces contraintes.

Dépendances et changement d'états On ne peut traiter une contrainte que si toutes celles dont elle dépend sont dans cet état, *i.e.* que toutes les contraintes dont elle dépend sont satisfaites. Observons les états dans lesquels la contrainte peut aller à partir de l'état *running*. L'état *success* ne présente pas d'intérêt, car il n'a de sens que dans des requêtes "discrètes" (via la primitive **make**). L'état *temporary failure* est plus intéressant, il indique un échec (au moins) temporaire, mais cela signifie aussi que la contrainte n'est temporairement plus maintenue, et donc qu'on ne peut plus continuer à garantir les contraintes qui dépendent d'elle. On va donc mettre en `pause` toutes les contraintes qui dépendent de la contrainte temporairement en échec, arrêtant ainsi leur exécution. Si la contrainte finit par trouver une solution, l'agent pourra alors envoyer un message `continue` pour réveiller les contraintes qui en dépendent. Ce mécanisme est illustré par la figure 4.5. Il permet d'assurer de manière générique que les dépendances d'exécution sont bien toujours garanties, et que les agents n'essaient pas de calculer ou exécuter des informations à priori non valides.

L'état *paused* L'état *paused* met uniquement en pause l'exécution de la recette actuellement exécutée. En particulier, il ne remet pas en cause le choix des tâches à exécuter, et donc le contexte dynamique. Nous avons envisagé d'interrompre puis de relancer les transitions en cas de solution, mais cette approche avait deux problèmes. Elle est d'une part moins efficace dans le cas nominal car elle implique de recalculer les tâches à exécuter, puis à choisir à nouveau la bonne recette. D'autre part, il est possible que l'agent se fasse "voler" la ressource par un autre agent, qui en avait besoin à ce moment. Étant donné que l'état *paused* est un état transitoire assez court, le contexte dynamique ainsi que le choix de la recette courante restent à priori valides, il est avantageux de les conserver. L'introduction de cet état *paused* répond donc mieux au problème de la gestion des dépendances entre contraintes.

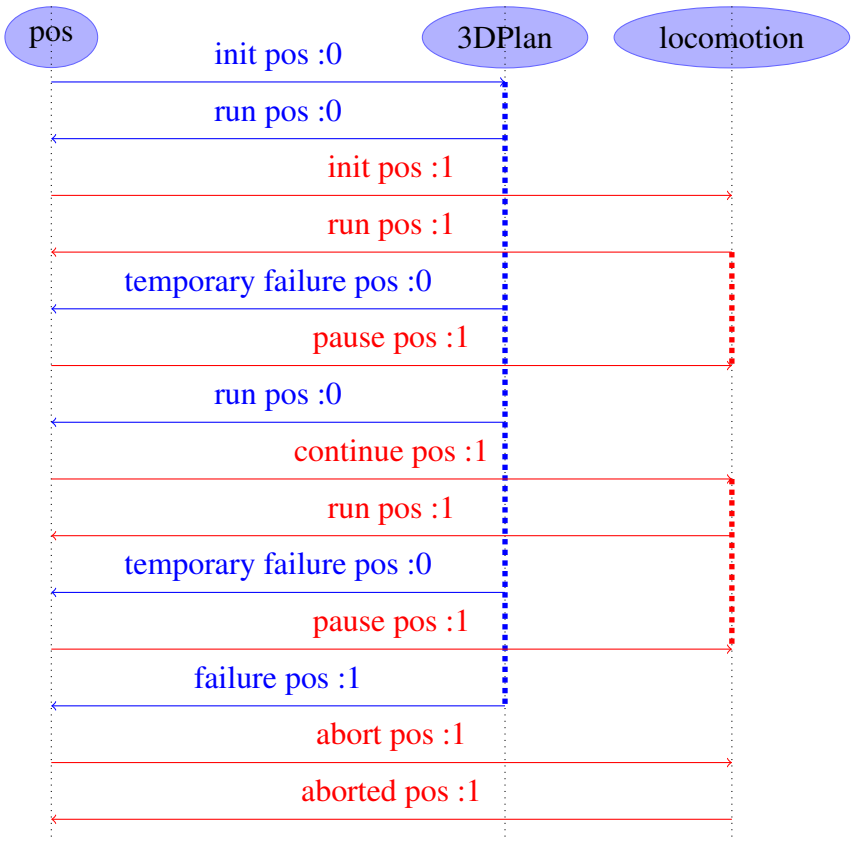

FIGURE 4.5 – Messages échangés entre 3 agents en cas d'échec temporaire. Dans ce cas précis, la satisfaction de la contrainte *pos :1* dépend de la satisfaction de la contrainte *pos :0* : un échec temporaire de cette seconde implique donc la mise en pause de la première.

L'état *paused* et exécution Du point de vue de l'exécution, la pause d'une recette est équivalente aux étapes suivantes :

1. l'interruption de la primitive synchrone courante (**wait**, **abort** ou un appel à la couche fonctionnelle, on suppose ici que les méthodes de la couche fonctionnelle sont incorruptibles)
2. faire suivre des messages *pause* à toutes les contraintes actuellement maintenues
3. attendre le message *continue*
4. faire suivre des messages *continue* à toutes les contraintes actuellement en pause
5. relancer l'exécution de l'opération synchrone interrompue.

Conclusion

Dans cette section, nous nous sommes intéressés aux dépendances entre les contraintes et ce que cela impliquait pour l'exécution des tâches associées. Nous avons présenté les mécanismes

garantissant une exécution correcte en cas d'échec temporaire d'une des contraintes. Nous allons maintenant nous intéresser à la manière dont sont gérées les erreurs lors de l'exécution, d'abord localement au niveau de l'agent, puis plus généralement au niveau du système composé de l'ensemble des agents.

4.3.2 Gestion des fautes

Souvent, au cours d'une mission, des évènements imprévus, voire des erreurs se produisent, et il est essentiel de les gérer correctement afin de poursuivre de manière autonome la mission.

Gestion locales des erreurs

Des solutions spécifiques Au cours des missions, certains problèmes vont apparaître de manière récurrente, que ce soit des limitations des algorithmes du niveau fonctionnel, ou bien des situations particulières. Le développeur peut alors analyser ces problèmes et fournir des solutions spécialisées. Dans (Simmons and Apfelbaum, 1998), les auteurs appellent ce type de solution des "exception handler". Il est possible d'encoder cette solution spécialisée dans ROAR : pour cela, le développeur va implémenter une nouvelle recette ayant comme précondition la présence de l'erreur considérée grâce à la primitive **last_error ?**. Le listing 4.5 étend les définitions du listing 4.4 afin de gérer un échec du planificateur de déplacement. Nous avons observé que les échecs de ce planificateur venaient souvent de l'agrégation erronée de données dans la carte, dues à des petites erreurs de localisation. La solution est alors d'effacer la carte (ligne 11), puis de commencer à en régénérer une (ligne 13). Pour éviter le problème de la zone aveugle, le robot revient sur ses pas via la fonction `backward_traj` puis déclenche le fonctionnement "normal" avec la méthode `follow_goal`.

Listing 4.5 – Gestion d'un échec du planificateur lors de la navigation en terrain difficile

```
1  let backward_traj = fun {
2    let id ensure(exec_ctr where rflex::expected_port == trajectory::port)
3    make(trajectory::backward == true)
4    abort id
5  }
6
7  go_to_planner_error = recipe {
8    pre = {{last_error?(plan_ctr)}}
9    body = {
10     make(3DMap::empty == true)
11
12     ensure(mapping_ctr)
13     backward_traj()
14     follow_goal()
15   };
16 }
```

Une solution générique : choix d'une autre recette Évidemment, il n'est pas possible de fournir une solution spécialisée pour tous les aléas possibles. Pour autant, il ne faut pas échouer dès qu'un événement réellement non prévu surgit. Il faut au contraire essayer de choisir une recette qui peut exécuter la tâche, mais qui n'utilise pas le composant qui vient d'échouer. Pour

cela, nous introduisons deux nouveaux concepts. Tout d'abord, le domaine des contraintes d'une recette est l'ensemble des contraintes qu'émet une recette. Ensuite, le contexte d'erreur est l'ensemble des contraintes que le système a échoué à satisfaire durant le traitement de la contrainte courante. Une recette r est alors acceptable si et seulement si :

- $domain(r) \subset available\_agents$
- $constraint\_domain(r) \cap error\_context = \emptyset$
- $\forall P \in \{Pre_R\}\ eval(P) = true$

Lorsqu'une recette échoue, le contexte d'erreur est rempli avec la contrainte ayant mené à l'échec et le processus de sélection d'une recette reprend. Il est clair qu'à chaque étape, le nombre de recettes acceptables diminue puisque la contrainte fautive apparaît dans au moins une recette (celle qui a été choisi précédemment).

Un autre arbre de tâches Lorsque l'agent ne trouve plus de solution pour effectuer une tâche, cette tâche échoue, et les conditions qu'elle était censée satisfaire doivent être reconsidérées. On va donc appliquer à nouveau l'algorithme 2 en considérant non plus comme ensemble des tâches $\{T_e\} \cup \{T_d\}$ mais cet ensemble privé des tâches fautives. Pour les mêmes raisons que précédemment, ce processus s'achève au pire faute de tâches, auquel cas la contrainte rentre définitivement dans l'état *failure*.

Gestion des erreurs au niveau système

Historique et décision distribuée Lorsque les processus précédents finissent par échouer, l'erreur remonte dans le graphe et est traitée selon le même procédé. Toutefois, comme ROAR est un système distribué sans mémoire, les agents ne gardant pas l'historique des contraintes qui leur ont été envoyées, le système dans sa globalité risque d'essayer de nombreuses fois les mêmes solutions, rendant moins efficace et moins réactive la réparation d'erreur. Pour résoudre ce problème, le contexte d'erreur est échangé lorsque l'erreur remonte dans le graphe. Ainsi, l'agent appelant peut connaître l'ensemble des erreurs survenues lors de la satisfaction des contraintes qu'il a émise, et agir aux mieux en connaissance de cause en rejetant les solutions qu'il sait ne mener à rien.

Illustration La figure 4.6 illustre ce processus. D'abord, une contrainte est assignée par l'opérateur à l'agent *pos* afin de faire déplacer le robot. Dans un premier temps, l'agent *pos* sélectionne la stratégie basée sur les agents *3DMap* (celui-ci utilisant en retour l'agent *3DCloud* puis l'agent *image*), *3DPath* et *locomotion* (figure 4.6(a)). Au bout d'un certain temps, l'agent *3DCloud* échoue à produire des données 3D (par exemple parce que les caméras sont aveuglées ou sous-exposées – figure 4.6(b)). N'ayant pas de solution locale, l'erreur remonte dans le graphe, d'abord à *3DMap*, puis à *pos*. Celui-ci essaye une nouvelle stratégie basée sur les agents *2DMap*, *2DPath* et *locomotion*. Pour construire sa carte, l'agent *2DMap* a plusieurs solutions, soit directement à partir de l'agent *2DCloud*, soit via une transformation des informations de *3DCloud* (figure 4.6(c)). Grâce au contexte d'erreur échangé entre les agents, l'agent *2DMap* va sélectionner l'agent *2DCloud* puisque l'agent *3DMap* est la cause de l'échec dans la branche précédente. Cette dernière configuration est montrée figure 4.6(d).

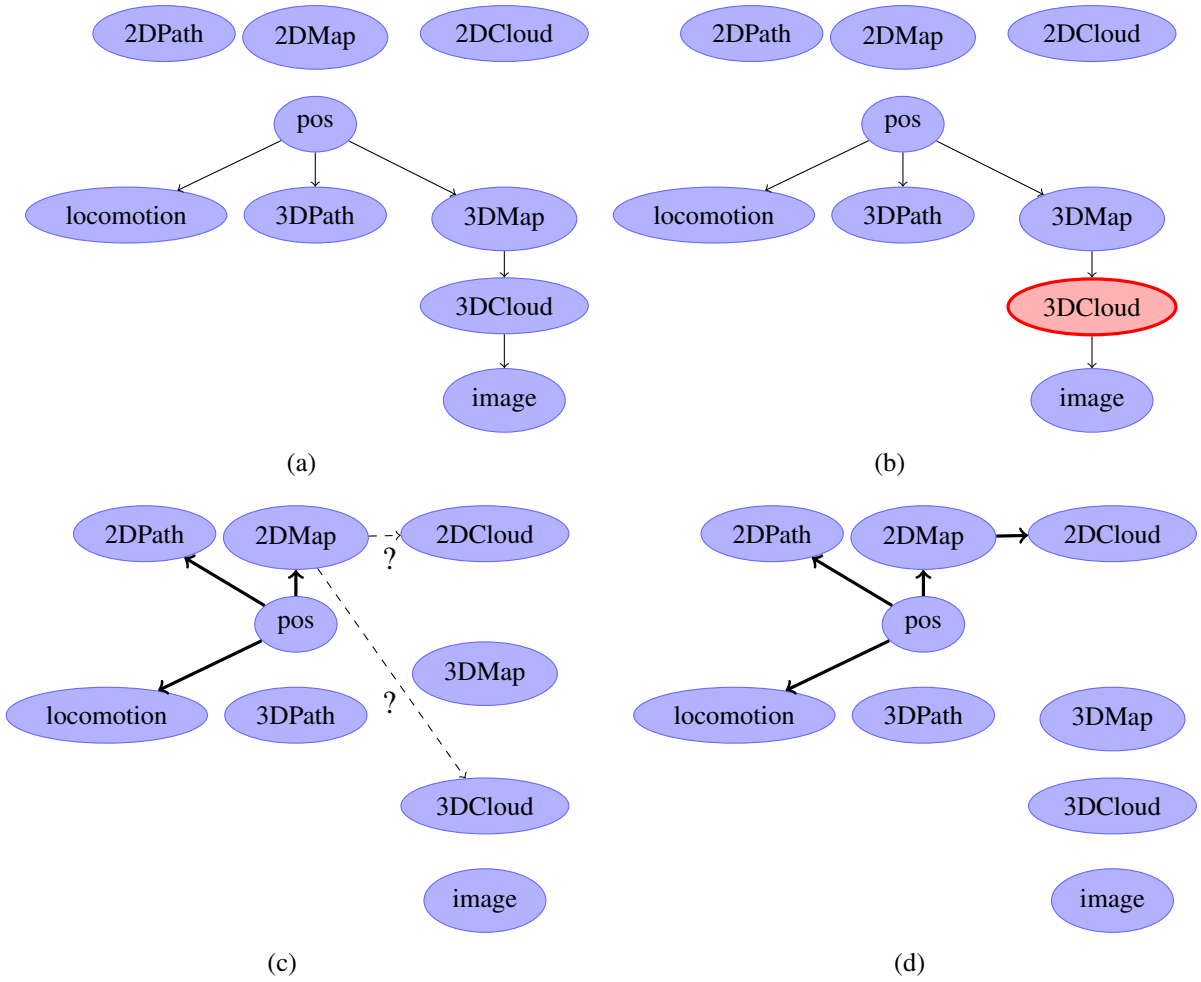

FIGURE 4.6 – Un graphe d'agent dans lequel l'utilisation du contexte d'erreurs améliore la récupération de fautes. Les flèches basiques représentent les contraintes échangées lors de la première stratégie. Les flèches en gras représentent les contraintes échangées lors de la seconde stratégie. Les flèches en pointillés sont les choix possibles de stratégies, dont la sélection sera ici guidée par le contexte d'erreurs.

Limitations Lorsqu'une contrainte met longtemps à être satisfaite (en particulier les contraintes de haut niveau, venant de l'agent *mission* ou directement de l'opérateur), elle peut conserver un historique très long, alors que certains échecs étaient dûs à une situation très temporaire. Le système est alors rapidement sur-contraint, car il s'empêche d'essayer des alternatives qui ont échoué dans un passé possiblement lointain. Plusieurs solutions sont envisageables pour pallier ce problème, même si aucune n'est réellement satisfaisante. La première consiste à "oublier" les échecs au bout d'un certain temps, dépendant de l'agent. La seconde consiste à surveiller les agents qui échouent et à régulièrement vérifier si ils peuvent satisfaire les contraintes qu'ils avaient préalablement rejetées. Cette approche rend les échanges entre les agents beaucoup plus complexes, et est relativement coûteuse en temps de calcul. De plus, elle risque de limiter les

possibilités de parallélisme en contraignant régulièrement l'agent. Enfin, une autre possibilité est de laisser échouer le système jusqu'au plus haut niveau et que celui-ci réessaye une seconde fois après avoir vidé l'historique des échecs, permettant au système de tester à nouveau chaque branche. Même si elle n'est pas optimale car elle ne tire pas rapidement partie d'une "meilleure" ressource, cette solution reste simple à mettre en place et est surtout déterministe, c'est donc celle que nous avons privilégiée.

4.3.3 Gestion des problèmes de concurrence

Un cas spécial d'échec est lié aux compétitions sur les ressources. Nous avons montré dans la section 4.2.3 comment différencier le cas "classique" d'erreurs avec le cas spécifique des erreurs de concurrence. Nous allons maintenant voir comment sont gérés ce type d'erreurs.

Contexte d'appel Nous appelons contexte d'appel d'une contrainte C la pile des contraintes qui ont mené à cette contrainte. Autrement dit, si $A \xrightarrow{C_1} B$ et qu'en réponse, on a $B \xrightarrow{C_2} D$ puis $D \xrightarrow{C_3} E$, le contexte d'appel pour C_3 est la pile $[C_2, C_1]$. Cette notion est équivalente à la notion de pile d'exécution dans un langage de programmation classique, mais ici dans un contexte distribué : il faut donc explicitement propager cette information le long du graphe des contraintes. Si nous nous intéressons de nouveau à l'exemple précédent, le contexte d'appel pour la contrainte sur l'agent *image* est :

1. $loc :: distance(3DCloud :: last\_refresh, pos :: current) < 0.2$
2. $3DMap :: refresh = true$
3. $loc :: distance(pos :: current, pos :: goal) < 0.5$

Mécanisme d'interactions inter-agents Considérons la situation $A \xrightarrow{C_1} B$ et $D \xrightarrow{C_2} B$. Si C_2 entre en conflit avec C_1, l'agent D va alors récupérer le contexte d'appel lié à C_1, et demander successivement aux agents qui traitent les contraintes du contexte d'appel si ils peuvent trouver une solution alternative à leur problème, sans utiliser explicitement la contrainte C_1 qui mène au conflit. Pour cela, ils vont d'abord chercher une recette pour les tâches actuellement sélectionnées dont le domaine de contrainte n'inclut aucune des contraintes apparaissant dans le contexte d'appel. Si il n'existe pas de solution, les agents pourront essayer de chercher des solutions alternatives au niveau tâche, de manière analogue à l'approche décrite dans 4.3.2 pour la gestion des fautes. L'exécution "normale" continue durant ce processus : la configuration n'est remise en cause que si une solution est réellement trouvée. Si une solution alternative est trouvée, l'agent se reconfigure et informe l'agent D qu'il peut maintenant utiliser la ressource de B. Si aucune solution n'est trouvée, l'erreur de concurrence est transformée en échec classique (*i.e.* il n'existe pas de méthodes pour satisfaire la contrainte dans le contexte courant), et on utilise le processus décrit précédemment pour le gérer. Ainsi, ce mécanisme permet une reconfiguration locale des agents afin d'exécuter plus de tâches en parallèle.

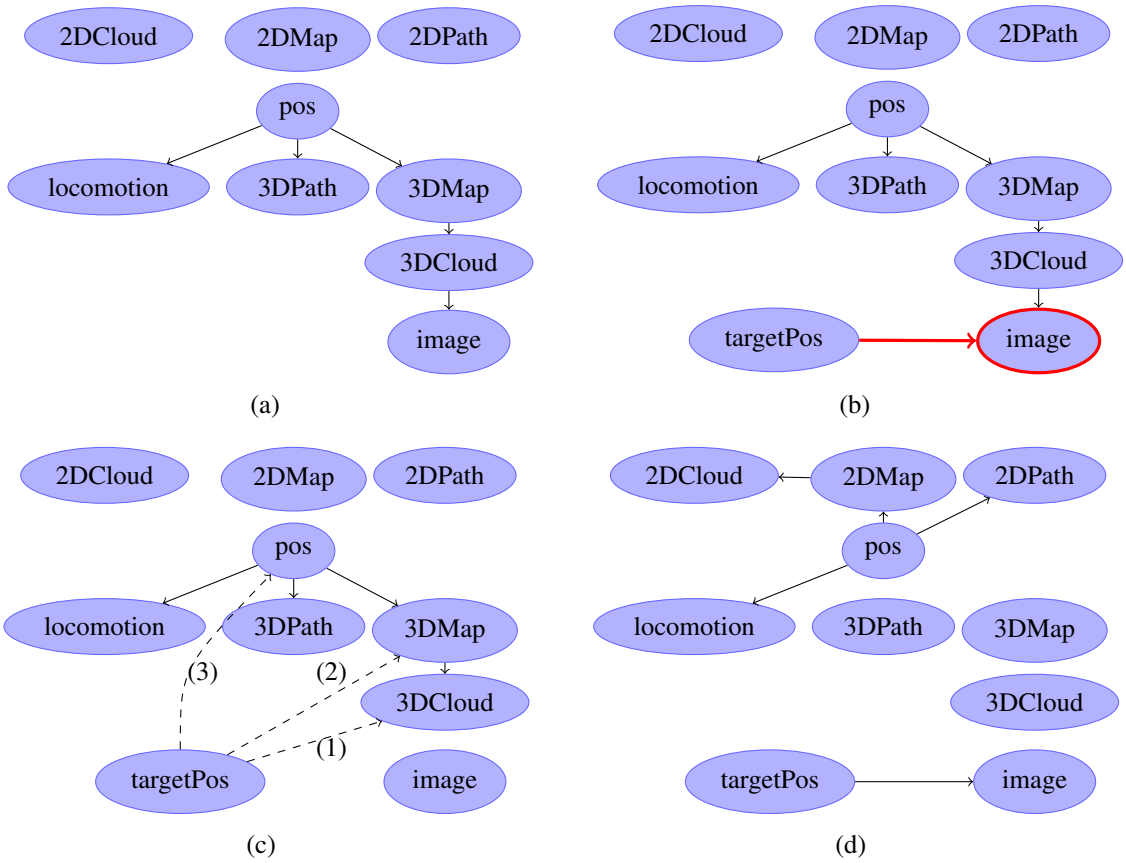

FIGURE 4.7 – Évolution du système après l'apparition d'un conflit sur l'agent *image* (voir le texte associé pour une explication détaillée)

Illustration La figure 4.7 illustre ce mécanisme. Initialement, une tâche de navigation est demandée au robot, générant une contrainte sur l'agent *pos*. Celui-ci choisit une stratégie de navigation basée sur les agents *3DMap*, *3DPath* et *locomotion* (figure 4.7(a)). Au bout d'un certain temps, une nouvelle contrainte est ajoutée sur l'agent *targetPos*. Cet agent utilise une stratégie de tracking visuel de la cible, ce qui implique l'ajout d'une contrainte sur l'agent *image* (figure 4.7(b)), qui doit alors satisfaire deux contraintes non compatibles car les orientations des caméras liées à ces deux tâches sont contradictoires. Il va donc rejeter cette nouvelle contrainte car il y a un conflit. L'agent *targetPos* va alors utiliser le mécanisme précédemment décrit pour résoudre le système. Il va d'abord interroger *3DCloud* afin de savoir si il peut satisfaire sa contrainte sans utiliser la contrainte courante sur *image*. Celui-ci n'ayant pas de telle solution [1], il en informe *targetPos*. Celui-ci continue avec l'agent *3DMap* qui répond aussi par la négative. Il interroge alors l'agent *pos* (figure 4.7(c)). Dans le contexte courant, celui-ci peut exploiter une stratégie de déplacement en 2D, basée sur les agents *2DMap*, *2DPath* et *locomotion*. Il change alors sa stratégie, laissant libre la ressource *image* pour *targetPos*. La configuration finale, montrée figure

1. on considère ici pour l'illustration que le robot n'est pas équipé de lidar

4.7(d) permet ainsi à la fois de naviguer de manière sûre et de suivre la cible.

4.4 Synthèse

Modularité et réactivité Notre architecture de contrôle est donc un système distribué basé sur des agents encapsulant des ressources. En plus d'encoder naturellement le parallélisme dans les tâches d'un robot, cette décomposition en agent répond bien à deux autres grandes problématiques des architectures de contrôle. D'un côté, elle permet une bonne organisation logique, gage de modularité. Cela permet de plus de ne déployer que les agents correspondant aux ressources d'une plateforme robotique donnée, et donc de contrôler différentes plateformes à partir de la même couche de contrôle. De l'autre, elle garantit des temps de délibération dépendant de la ressource considérée : ainsi la partie bas niveau peut rester très réactive tandis que des agents de plus haut niveau peuvent avoir des temps de délibération plus longs.

Gestion de la concurrence La définition explicite du contrat des tâches et l'échange de contraintes sous forme de formules logiques permet aux agents de raisonner sur l'impact de ces contraintes sur leur fonctionnement, et en particulier de détecter les conflits liés à des contraintes contradictoires. Un mécanisme spécifique est proposé pour résoudre ce type d'aléas, et ce afin d'optimiser la configuration du système et maximiser le nombre de tâches parallèles dans le robot.

Expressivité Nous avons introduit le langage de contrôle de ROAR. Il permet d'exprimer simplement des solutions de coordination, dans des contextes parallèles. De nombreuses informations sont automatiquement déduites depuis ces spécifications afin de choisir la meilleure stratégie dans les conditions courantes.

Robustesse La question de la robustesse est considérée selon plusieurs angles. Tout d'abord la décomposition en agents assure qu'il n'y a pas de point de défaillances uniques. Si un des agents disparaît, le reste du système peut continuer à fonctionner. Un mécanisme de dépendances implicites entre les contraintes permet d'assurer qu'aucune contrainte n'est "exécutée" si ses antécédents ne sont pas valides. Pour répondre à des aléas liés à la couche fonctionnelle ou à des situations imprévues, deux mécanismes ont été mis en place. D'une part, il est possible de spécifier explicitement des stratégies de réparation. D'autre part, une solution générique est proposée pour choisir des stratégies alternatives, en évitant les branches ayant précédemment mené à des échecs.

L'architecture ROAR répond donc bien aux problématiques exprimées dans la section 2.1.2. Seule la question de la validation n'est pas traitée explicitement, elle sera abordée dans la seconde partie du manuscrit. Il faut toutefois noter que les règles spécifiées dans ROAR sont relativement simples, et permettent d'évaluer de manière aisée le comportement conjoint d'un ensemble d'agents.

Chapitre 5

Mise en œuvre

Après avoir présenté les principes et algorithmes qui définissent ROAR*, nous nous intéressons dans ce chapitre à son déploiement sur un robot réel. Dans un premier temps, l'implémentation que nous avons faite de* ROAR *est présentée. Dans un second temps, nous décrivons le développement d'une couche de supervision pour un UGV à l'aide de cette implémentation, en la comparant avec une précédente approche. Enfin, nous décrivons et analysons quelques expérimentations que nous avons réalisées avec cette architecture.*

5.1 Implémentation de l'architecture ROAR

Le projet HYPER [1] est une implémentation libre de l'architecture ROAR. Il est composé d'un ensemble de bibliothèques dynamiques et des tests unitaires associés permettant l'analyse du langage présenté, la génération de code associée, le moteur logique, ainsi que la gestion de la communication entre agents. En plus de ces bibliothèques, plusieurs exécutables sont disponibles, en particulier le compilateur, un agent maître servant de serveur de noms ainsi qu'un agent pour centraliser les traces d'exécutions des différents agents. Nous allons maintenant décrire plus précisément ces différentes parties.

5.1.1 Compilation

Vue d'ensemble du processus de compilation

Vue d'ensemble Dans HYPER, les agents sont des processus distincts construits à partir d'un ensemble de fichiers de spécifications. La définition de chaque agent contient un fichier décrivant

1. dont le code est disponible sur `http://trac.laas.fr/git/hyper`

le contexte de l'agent, un fichier décrivant ses tâches ainsi qu'un ou plusieurs fichiers contenant des recettes pour ces différentes tâches. Ces spécifications sont analysées syntaxiquement puis sémantiquement. Dans une seconde phase, le compilateur `hyperc` génère le code c++ correspondant à cette spécification. Si un agent définit des fonctions propres, un squelette c++ est automatiquement généré par `hyperc` qui doit alors être complété par le développeur. Enfin, un compilateur c++ est appelé pour générer l'exécutable de l'agent ainsi qu'une bibliothèque partagée contenant l'implémentation c++ des types et fonctions qu'il exporte. La figure 5.1 illustre ces différentes étapes.

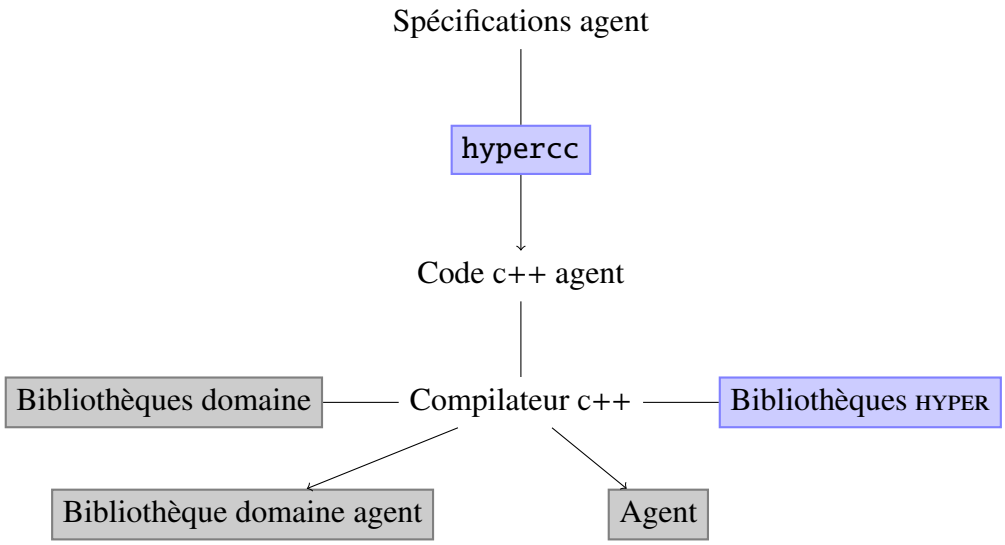

FIGURE 5.1 – Vue d'ensemble du processus de génération d'un agent. Les boîtes bleues correspondent à ce qui est fourni par HYPER. Les boîtes grises sont les résultats de la génération (de l'agent courant, ou de génération précédente).

Du choix de c++ Plusieurs raisons nous ont poussé à choisir de générer du code c++ de haut niveau. Tout d'abord, nous voulions être capable de comprendre le code généré, pour faciliter la phase de débogage. Ensuite cela rend plus facile l'édition de liens avec des bibliothèques externes, en particulier pour contrôler et lire des informations de la couche fonctionnelle d'un robot. De plus, les compilateurs c++ font de nombreuses vérifications et optimisations sur le code, ce qui permet à la fois de détecter un certain nombre d'erreurs dans la génération du code et aussi d'avoir des agents "rapides". Enfin, de manière générale, le c++ offre un bon rapport entre abstractions et performances pour le développement des bibliothèques au cœur des agents HYPER, présentées dans les sections 5.1.2, 5.1.3 et 5.1.4. Sur la base de ces différentes bibliothèques, il est possible de construire manuellement des agents spécifiques (voir en particulier 5.1.5).

Analyse

Un des buts de la phase de compilation est de vérifier un certain nombre de propriétés statiquement. À partir de l'ensemble des contextes des agents, le compilateur `hyperc` peut vérifier différentes propriétés : tout d'abord, il vérifie que l'ensemble des variables utilisées existent et sont du type correct vis à vis du contexte. Ensuite, il vérifie que les variables sont utilisées dans des contextes acceptables, *i.e.* qu'un agent n'essaye pas d'accéder aux variables privées d'un autre agent ou qu'il n'essaye pas de contrôler une variable seulement disponible en lecture. Même si `hyperc` ne peut vérifier que les contrats des tâches sont consistants par rapport aux diverses implémentations, il vérifie que les recettes ne modifient (via la primitive **set**) que des variables d'états apparaissant explicitement dans ce contrat.

Extensions

L'intégration avec différentes abstractions logicielles est toujours un point délicat, nécessitant de générer des parties spécifiques pour chaque type d'architecture fonctionnelle. Nous ne voulions pas que la cœur du compilateur dépende de ces connaissances spécifiques. `hyperc` utilise donc un système de greffons et d'annotations pour gérer les différentes architectures de contrôle. Lorsque le développeur déclare une fonction, il peut ajouter une annotation en tête de cette fonction du type `@tag`. Dans ce cas, le compilateur utilisera le générateur de code du greffon *tag* pour générer le code associé à cette fonction. Le listing 5.1 illustre l'utilisation de ces annotations. On définit ligne 4 la fonction `go_to` annotée par `genom` et ligne 5 la fonction `go_to_ros` annotée par le `ros_action`. Lors de l'utilisation de ces deux fonctions, `hyperc` appellera la génération de code respectivement du greffon *genom* et du greffon *ros_action*. Ce mécanisme permet de garder un cœur générique, et d'externaliser les parties spécifiques à des couches logicielles particulières. Actuellement, seul le greffon pour GeNoM est implémenté.

Listing 5.1 – Exemple d'utilisation d'annotation dans la déclaration de fonctions

```
1 X = agent {
2         {}
3         {
4           @genom void go_to(double, double);
5           @ros_action void go_to_ros(double double);
6         }
7         {}
8 }
```

5.1.2 Logique

La bibliothèque `hyper_logic` traite principalement de la résolution des systèmes de clauses de Horn, utilisée dans 4.2.3. Comme mentionné précédemment, il s'agit principalement d'une implémentation de la méthode SLD (Lloyd, 1987). Toutefois, dans notre contexte, il n'est pas possible de réutiliser un raisonneur prolog classique comme gprolog [2] ou Swi-prolog [3] car les interpréteurs prolog font l'hypothèse d'un monde clos (*i.e.* on connaît tout sur le monde). Ils

2. `http://www.gprolog.org/`
3. `http://www.swi-prolog.org/`

implémentent alors une variante de l'algorithme précédent SLDNF où l'on déduit $\neg p$ de l'échec de l'implication de p. Au contraire, dans ROAR, nous considérons un monde ouvert (un agent ne connaissant par définition pas tout sur le monde). Les échecs dans la déduction d'une proposition p indiquent juste qu'on ne peut rien déduire de particulier sur p à partir de ce système. Les hypothèses utilisées en bout de raisonnement peuvent alors être interprétées comme des nouvelles contraintes sur le système.

Enfin, nous utilisons des techniques de mémorisation lors de la résolution du système. En effet, à chaque nouvelle contrainte C, l'agent essaye de résoudre le système $Task_{spec} \rightarrow C$ où $Task_{spec}$ correspond à la spécification d'une tâche particulière. L'agent va alors conserver, pour chaque tâche, les dérivations qu'il a déjà calculées et les contraintes qu'il a réussi à prouver. Ainsi, lorsqu'un agent reçoit plusieurs fois la même contrainte ou des contraintes proches, il peut résoudre de manière quasi instantanée le problème, et donc améliorer en général la réactivité du système.

5.1.3 Communications entre agents

Messages échangés La bibliothèque `hyper_network` traite les problèmes de communication entre les différents agents. Dans l'implémentation actuelle, les agents communiquent au travers de TCP/IP via un protocole spécifique. Les messages échangées sont constitués d'un entête de taille fixe, contenant le type de message (défini par un `uint32_t`) et la taille des données sérialisées (définie aussi par un `uint32_t`) et des données sérialisées. Le type encodé dans l'en-tête est utilisé côte réception pour décoder correctement les données.

Format d'échanges des données Les données sont encodés en binaire pour des raisons d'efficacité et de compacité via la bibliothèque `boost::serialization`[4]. À l'heure actuelle, tous les agents utilisés dans le système communiquent via `hyper_network`, le format d'échange de données n'est pas en soi un problème. Dans un cadre plus ouvert, il faudrait se tourner vers des formats d'échanges plus standards (mais moins efficaces) comme json, xml, ... Le choix de la bibliothèque de sérialisation a été fait en considérant ce besoin, celle-ci permettant en effet de facilement changer de schémas d'encodage.

Schémas de communications Les communications entre agents sont unidirectionnelles, les canaux de requêtes et de réponses sont donc séparés. Cette séparation rend l'implémentation et le raisonnement sur chaque agent plus simple, car chaque agent n'a de fait qu'un seul point d'entrée par lequel arrivent tous les messages. Lors d'un schéma requête / réponse, chaque requête est identifiée par un identifiant unique composé du nom de l'agent et d'un entier généré par l'appelant. Enfin, les communications sont permanentes, *i.e.* elles ne sont pas détruites et recréées à chaque échange entre deux agents mais au contraire conservées pour une future utilisation. Là encore, ce choix vise à limiter la charge système associée à HYPER.

4. `http://www.boost.org`

5.1.4 Modèles

La bibliothèque `hyper_model` utilise les bibliothèques précédentes pour implémenter un modèle générique d'agent. C'est en particulier dans cette bibliothèque que les différents concepts proposés dans le chapitre 4 sont implémentés.

Patron de conception "proactor" De nombreuses sous-tâches peuvent s'exécuter en même temps dans un même agent, que ce soit l'exécution de plusieurs recettes, la mise à jour de variables, ou le traitement de nouvelles contraintes. Une solution classique pour exécuter ces différentes tâches en parallèle est d'utiliser un processus léger pour chacune de ces tâches mais cela peut être coûteux du point de vue système (en particulier une fois multiplié par le nombre d'agents), et complexe à mettre en place (problème de synchronisation entre les processus, interblocages...). De plus, la majorité des traitements exécutés en parallèles ne sont pas du calcul intensif (à part la partie logique), mais majoritairement des attentes de réponses de la part d'autres agents ou de la couche fonctionnelle. En conséquence, nous avons décidé d'utiliser le patron de conception "proactor" (Pyarali et al., 1997; Schmidt et al., 2000) pour implémenter nos agents, celui-ci consistant principalement en l'appel d'une fonction de continuation à la fin de l'exécution d'une fonction asynchrone. Cette technique est globalement plus efficace qu'une approche basée fil de processus pour notre utilisation, mais elle est aussi plus composable et supprime les problèmes de synchronisation.

Exécution En plus des différents algorithmes présentés dans le chapitre 4, cette bibliothèque contient aussi le code générique pour chaque primitive du langage d'exécution ROAR défini section 4.2.4. Chaque primitive est représentée par un objet distinct ayant une méthode d'exécution `compute()`, une méthode d'interruption `abort()`, une méthode de mise en pause `pause()` et enfin une méthode de redémarrage `continue()`. Un objet de plus haut niveau peut contenir une séquence de ces objets, et ainsi crée un comportement de plus haut niveau. Cet objet a la même sémantique (exécution, interruption, pause, redémarrage) et peut donc être composé dans un objet plus grand (typiquement une recette). La figure 5.2 explicite ces relations entre les différentes classes de la couche exécutive. Ajouter des primitives à ROAR est donc relativement facile : il suffit d'ajouter, du côté compilateur, l'analyse et la génération de code associé à cette primitive, et côté modèle d'ajouter un objet représentant le calcul lié à cette primitive.

5.1.5 Des agents spécifiques

Quelques agents ont été développés directement en c++, pour résoudre des problématiques spécifiques. Nous décrivons ici leur rôle.

L'agent *hyperruntime*

Serveur de noms L'agent *hyperruntime* a une position centrale dans HYPER. Il sert tout d'abord de serveur de nom pour les autres agents. Lorsqu'un agent A veut communiquer avec un agent B, il commence d'abord par demander à *hyperruntime* comment s'adresser à B. Pour le reste

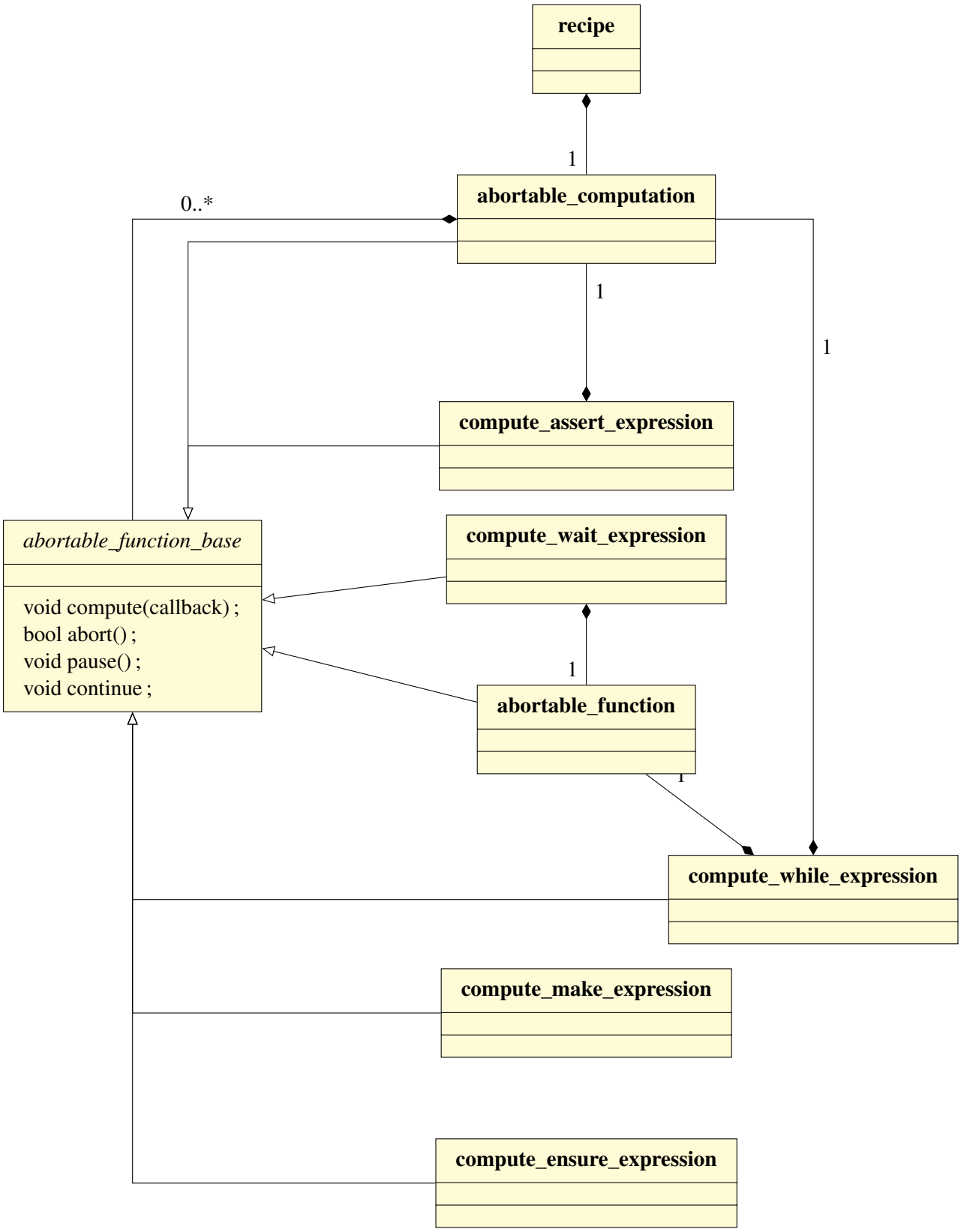

FIGURE 5.2 – Diagramme de classes pour la couche exécutive. Les flèches triangulaires blanches représentent la relation d'héritage. Les flèches en losange noir représentent la relation de composition.

des communications, l'agent *hyperruntime* n'intervient évidemment plus. Au démarrage, chaque nouvel agent va s'inscrire au niveau de *hyperruntime* afin que les autres agents puissent connaître son existence et la manière de communiquer avec lui (type de protocole, port de communication, ...). En l'absence de cet agent, les autres agents ne peuvent démarrer : il a donc un rôle crucial dans l'ensemble des agents. Il n'a toutefois que des capacités extrêmement simples, et il est donc peu probable qu'il échoue.

Un protocole de ping Pour vérifier que les agents sont bien actifs, on utilise un système de ping. Chaque agent va régulièrement envoyer un message à *hyperruntime* pour indiquer qu'il est toujours actif. Si ce dernier ne reçoit pas de tels messages d'un agent *A* pendant deux périodes (100 ms par défaut), l'agent *A* est déclaré mort à tous les autres agents. Ceux-ci peuvent alors interrompre leur attente sur des réponses de *A* (celui-ci n'étant plus supposé répondre) et essayer alors d'autres stratégies.

L'agent *hyperlog*

Il est toujours important d'avoir des traces d'exécution afin de pouvoir comprendre le déroulement d'une mission, débogguer le système ou faire des statistiques. Dans un contexte distribué, cela est d'autant plus complexe que nous avons *n* sources d'informations qu'il faut ensuite agréger. L'agent *hyperlog* répond à ce besoin. Chaque agent envoie des messages d'information sur son exécution à *hyperlog*, celui-ci étant en charge de les réordonner (l'ordre de réception n'étant pas forcément l'ordre d'exécution). Ces traces peuvent alors être affichées en direct (`hyper_viewer`) ou bien rejouées pas à pas ultérieurement (`hyper_replay`). Elles sont aussi disponibles sous forme textuelle pour des analyses plus poussées ou pour faire des statistiques.

L'agent *hyper_ctrl*

L'agent *hyper_ctrl* permet lui de faire le lien avec l'utilisateur. En entrée, il peut prendre des contraintes qu'il fera suivre directement à l'agent concerné ou des ordres de plus haut niveau (`go_to`, `explore`, `track`, ...). Dans ce second cas, les ordres sont traduits sous forme de contraintes via une traduction ad-hoc puis envoyés aux agents concernés. Le protocole d'entrée de cet agent peut varier en fonction du contexte, du simple socket Unix à l'utilisation de protocoles de plus haut niveau comme YARP (Fitzpatrick et al., 2008).

Vue d'ensemble

La figure 5.3 résume les différentes interactions entre les agents standards (ici l'agent *pos*) et les agents spécifiques que nous venons de présenter.

5.1.6 Conclusion

Dans cette section, nous avons brièvement décrit notre implémentation du cadre logiciel proposé. Nous avons décomposé cette implémentation en plusieurs bibliothèques, chacune traitant

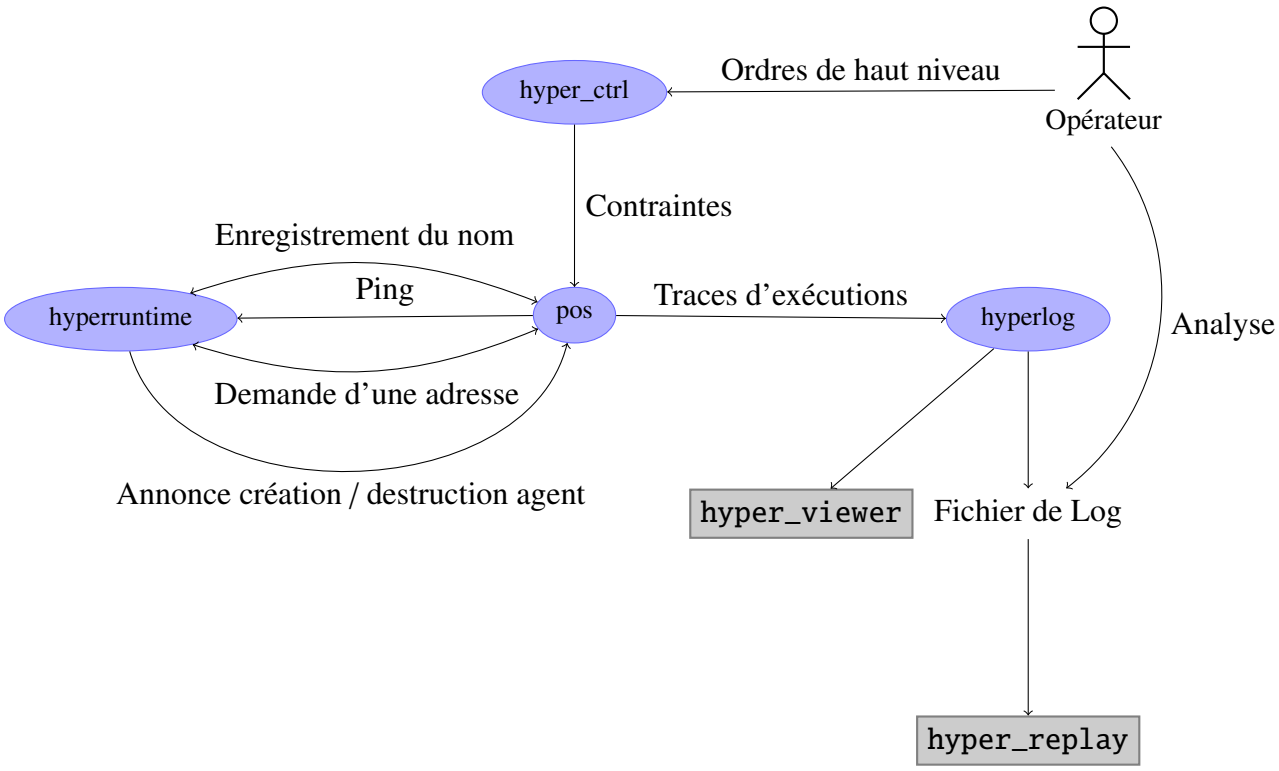

FIGURE 5.3 – Interaction entre les agents “classiques” et les agents “spécifiques”

d'une problématique séparée. Celles-ci sont utilisées pour notre modèle d'agent générique, mais aussi pour développer des agents répondant à des besoins spécifiques. Le développement d'une bibliothèque pour le compilateur permet de réutiliser celle-ci dans différents contextes, en particulier pour écrire des générateurs de code pour des cibles différentes, ou pour être intégré dans un environnement de développement. Enfin, un soin particulier a été apporté pour traiter les questions de robustesse (décomposition en processus, détection des agents qui ne répondent pas, gestion interne des erreurs...) et de performance (mémorisation, patron de conception "proactor", langage compilé...).

5.2 Développement d'une couche de supervision pour un UGV

Nous allons maintenant nous intéresser aux étapes de développement d'une couche de supervision en utilisant HYPER, en les comparant avec NAV_CTRL, un autre outil de développement de couche de supervision.

5.2.1 Une description succincte de NAV_CTRL

NAV_CTRL est une première tentative de conception d'architecture de contrôle que nous avons réalisée au début du doctorat. Si nous devions la classer selon la taxonomie présentée dans le chapitre 3, elle renterait dans la catégorie des langages de contrôle avec bibliothèques dédiées. Dans NAV_CTRL, les tâches et les ressources sont encapsulées dans des objets c++ classiques. Les différentes tâches s'exécutent en parallèle via l'utilisation de processus légers, la synchronisation se faisant classiquement grâce à des primitives d'exclusion mutuelle ou de moniteurs. Enfin, les informations de contrôles sont échangées grâce à des signaux tandis que les informations de données sont lues directement, les processus légers occupant un espace mémoire partagé. NAV_CTRL permet alors d'implémenter facilement des schémas réactifs, basés sur des événements.

5.2.2 Développement comparé de deux couches de supervision

Interfaces de base Nous commençons d'abord par définir des types de base pour représenter l'espace et le temps, ainsi que les méthodes associées à ces types. Pour représenter l'espace, nous utilisons des primitives du type point (x, y) ou (x, y, z), ou point avec direction (x, y, θ). Pour le temps, nous utilisons un couple $(sec, \mu sec)$ analogue à la structure C `timeval`. Dans un second temps, nous définissons des interfaces de contrôle vers les modules de la couche fonctionnelle (GeNoM). Dans le cas de HYPER, chaque module est contrôlé par un agent spécifique, afin de limiter les dépendances de chaque agent, et d'augmenter la granularité possible de déploiement. Cette étape a été légèrement plus longue côté HYPER, car nous devons expliciter les interfaces au niveau de chaque agent, puis écrire manuellement l'appel à la couche fonctionnelle. Toutefois, cette étape peut être (et sera) automatisée grâce à l'outil GeNoM3 (Mallet et al., 2010). Par contre, la sémantique "interrupted_by" de GeNoM, qui spécifie que certaines requêtes du composant peuvent interrompre l'exécution d'autres requêtes, peut être trivialement explicitée dans HYPER, alors qu'elle doit être encodée manuellement dans le cas de NAV_CTRL. Cet encodage

manuel a souvent été la source d'erreurs d'exécution à cause d'interruptions non prévues dans les requêtes couramment exécutées.

Une première méthode de navigation Nous définissons maintenant les contextes des grands types d'agents exposées figure 4.1, puis nous nous intéressons à définir une première stratégie de navigation, qui utilise les modules *locomotion*, *3DPath*, *dtm* et *stereovision*. L'implémentation du cas nominal et la gestion des erreurs spécifiques ne posent pas de problème particulier dans aucune des deux solutions. Toutefois, l'utilisation d'un langage de contrôle exhibe déjà des limitations : l'interruption d'une tâche dans le modèle proposée est difficile, laissant souvent le processus léger dans un état inconsistant et amenant alors à des interblocages. Ces problèmes de concurrences sont difficiles à analyser et à débogguer, et dans ce contexte HYPER montre une certaine supériorité.

Un second mode pour générer des nuages de points 3D Nous nous intéressons maintenant à l'utilisation possible du module *velodyne* pour remplacer le module *stereovision*. Dans HYPER, il s'agit principalement d'ajouter des recettes à l'agent *3DCloud*, le reste du système n'étant pas impacté. Au contraire, une part significative de NAV_CTRL doit être réécrite afin d'expliciter la possibilité d'utiliser soit *stereovision*, soit *velodyne* pour construire des cartes. De plus, ces deux méthodes n'ont pas les mêmes modalités d'utilisation, l'un produisant un nuage dense à courte vision, tandis que le second voit plus loin, mais est beaucoup moins dense. Évidemment, ce besoin de réécriture aurait pu être mitigé par une meilleure analyse du problème : cette analyse est inhérente dans ROAR, le développeur étant invité à expliciter dès le début les différentes ressources et leurs relations. De plus, dans le langage de contrôle HYPER, il est nécessaire d'encoder manuellement la possibilité de passer d'un mode à l'autre pour pouvoir générer une carte *dtm*.

Un second mode de navigation L'ajout d'un second mode de navigation utilisant les modules *2DPath*, *avoid* et *locomotion* mène à des difficultés similaires. Dans HYPER, il s'agit principalement d'ajouter des recettes à l'agent *pos*. Dans NAV_CTRL, au contraire, il est nécessaire d'encoder manuellement que deux implémentations ont un résultat "similaire" (ici le déplacement d'un robot), gérer finement la concurrence, ...

Une navigation de haut niveau L'ajout d'un mode de navigation de haut niveau utilisant *TMap* et *navigation* pour guider la navigation réactive a par contre été assez simple dans les deux systèmes. Dans HYPER, il a fallu réorganiser un peu les tâches de l'agent *pos*, mais sans toucher aux recettes, les modifications ont donc été assez légères. Dans NAV_CTRL, nous avons défini une nouvelle tâche qui envoyait des signaux à la tâche de navigation réactive. L'enchaînement de tâche est donc relativement aisée dans les deux cas. Par contre, HYPER offre une gestion systématique des erreurs, alors que dans NAV_CTRL, elle doit être manuellement encodée. Quand une erreur remonte jusqu'à l'opérateur, celui-ci a plus d'informations dans le système contrôlé par HYPER que dans celui contrôlé par NAV_CTRL.

Taille des systèmes obtenus Le tableau 5.1 représente la taille (en ligne de code) des implémentations des deux couches que l'on vient de présenter et d'une couche de supervision développée précédemment en PRS (Ingrand et al., 1996). Cette dernière implémentation ne contient pas toutes les fonctionnalités des solutions HYPER et NAV_CTRL, en particulier, elle ne contient ni la navigation à haut niveau, ni le suivi de cible. Ce tableau montre en particulier que la solution basée sur HYPER est la plus courte, d'un facteur 2.5 par rapport à NAV_CTRL. Même si la taille du code n'est pas une mesure absolue de la qualité de la solution, un plus petit code implique souvent moins de bugs, et surtout plus de facilité pour le maintenir.

TABLE 5.1 – Nombre de lignes de code dans chaque couche de supervision

PRS	NAV_CTRL	HYPER
4855	9217	3557

5.2.3 Conclusion

Nous avons ici décrit et comparé le développement de deux couches de supervision pour un UGV. Au vu de cette analyse, le système basé sur ROAR répond en pratique à nos exigences. En particulier, il permet de résoudre facilement et de manière homogène les problèmes de concurrence. Contrairement à NAV_CTRL, il est facilement possible d'expliciter plusieurs méthodes pour répondre au même problème. Enfin, il est beaucoup plus robuste, l'échec d'un agent n'entraînant pas l'échec complet de la mission (ce qui est le cas dans NAV_CTRL si un fil d'exécution rentre dans un état inconsistant). Enfin, la solution basée sur HYPER est plus courte et globalement plus facile à maintenir.

5.3 Expérimentations, résultats

Après avoir développé une couche de supervision pour le robot, nous pouvons maintenant l'utiliser pour contrôler la réalisation de missions autonomes par un robot. Nous allons nous intéresser ici plus particulièrement à des missions de navigation. Nous avons mené des évaluations dans deux cadres différents, à l'aide du simulateur MORSE (voir annexe B) et sur le robot réel Mana (décrit dans la section 2.2). Le simulateur, en plus d'offrir des facilités pour le débogage, permet aussi de tester aisément différentes configurations de robots, et différents environnements.

5.3.1 Rappels sur les agents présents sur le robot terrestre

Avant d'entrer dans le résumé des expérimentations, précisons à nouveau les différents agents qui vont entrer en jeu dans ce cadre précis.

La position du robot est maintenue par l'agent *pos*. Il utilise pour cela les informations venant de l'agent *pom*, qui contrôle directement le module de la couche fonctionnelle du même nom. Celui-ci agrège les différentes sources de localisation du robot pour donner la meilleure estimée possible de localisation.

Nous disposons de deux stratégies de navigation possibles, l'une se basant sur des modèles 3D pour des environnements complexes, et l'autre se basant sur des modèles 2D pour des environnements plus structurés. L'agent *mapping3D* encapsule cette notion de carte 3D. Pour la construire, il peut se baser sur les agent *stereopixel* et *dtm* qui contrôlent les modules éponymes qui permettent respectivement de construire un nuage de point 3D à partir d'une paire d'image et de construire une carte 3D à partir de ces nuages de points. Dans cette instance du problème, la carte 2D est encapsulée dans l'agent *obstacle_map*.

La stratégie de navigation 3D est encapsulée dans l'agent *p3d* qui contrôle le module du même nom. La stratégie 2D contrôle elle deux agents *avoid_obs* et *controller* qui contrôlent chacun un module de la couche fonctionnelle. Ces modules vont respectivement trouver une trajectoire dans l'espace des solutions puis calculer les consignes nécessaires pour les réaliser.

L'agent *locomotion* encapsule les capacités du mouvement du robot. Pour cela, dans ce cadre, il contraint l'agent *rflex*, qui contrôle directement le module fonctionnel correspondant.

Enfin, l'agent *trajectory* permet d'observer et de stocker la trajectoire courante, pour pouvoir éventuellement la rejouer.

5.3.2 Quelques expérimentations

Pour présenter quelques uns des tests que nous avons menés, nous utilisons des captures d'écrans venant de `hyper_replay` présentant la configuration des agents à un temps t (*i.e.* les contraintes entre les différents agents) et des chronogrammes représentants les échanges de contraintes au cours du temps entre les différents agents. Ces chronogrammes sont obtenus de manière automatique depuis les logs, les seules modifications manuelles étant le placement des agents afin de rendre le chronogramme plus lisible. Ils s'interprètent ainsi :

- les traits rouges correspondent aux périodes où les agents sont actifs,
- les flèches correspondent au début et à la fin d'une contrainte,
- les traits bleus pointillés correspondent à la séparation des différentes phases décrites dans le texte.

Les échanges de données entre les agents sont omis pour des questions de lisibilités.

Stratégie de déplacement 2D Le robot virtuel utilise ici la stratégie 2D, à savoir la combinaison des modules *obstacle_map*, *avoid_obs*, *controller* et *locomotion*. La figure 5.4 illustre cette configuration. En particulier, on peut voir qu'il n'existe pas de producteur de données télémétriques planes (laser 2D), et donc la carte d'obstacle va être construire à partir d'une carte 3D. En terme de configuration, l'agent *pos* a bien une contrainte sur l'agent *3DMap* afin de rafraîchir régulièrement la carte, celle-ci sera alors utilisée par l'agent *obstacle_map*. Le chronogramme 5.5 explicite les différentes étapes et contraintes échangées par les différents échanges durant la mission. On peut voir apparaître trois grandes phases :

1. dans un premier temps, le robot ne bouge pas, mais construit un modèle du monde qui l'entoure. Cela se traduit par des contraintes successives sur les agents *platine*, *stereopixel* et *dtm*.

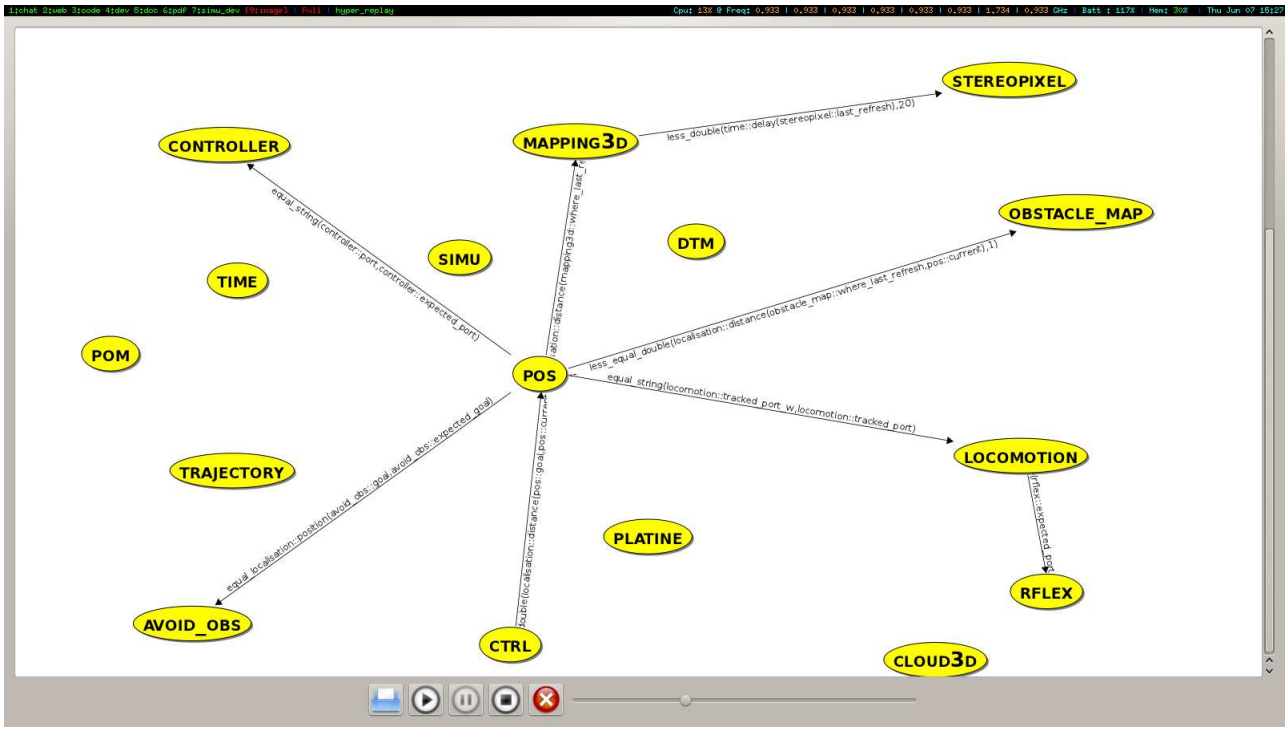

FIGURE 5.4 – Configuration pour la stratégie de navigation 2D

2. dans un second temps, le robot va essayer de sortie de sa zone aveugle. Il avance doucement via des contraintes sur *locomotion* qui en réponse contraint *rflex*. Pendant ce temps, l'agent *3DMap* continue de construire activement son modèle, mais ne modifie plus la direction des caméras.
3. enfin, nous rentrons dans la phase "nominale". L'agent *pos* contraint l'agent *avoid_obs* pour éviter activement les obstacles, tandis qu'il demande à *locomotion* de suivre les consignes calculées par *avoid_obs*. De plus, il continue à construire activement sa carte. On retrouve la configuration montrée par la figure 5.5.

Stratégie 3D La stratégie de navigation 3D repose sur la combinaison des modules *p3d*, *dtm* et *locomotion*. Un module *trajectory* permet d'observer la trajectoire, afin de la rejouer en marche arrière en cas d'échec de progression du robot. La figure 5.6 illustre la configuration nominale pour cette stratégie, qui a été décrite dans la recette listing 4.4. Cette configuration est globalement plus simple que la précédente, car celle-ci utilise directement comme modèle la carte 3D. On retrouve donc une contrainte sur *pos* venant de *ctrl*, qui en réponse maintient des contraintes actives sur *3DMap*, *locomotion* et *p3d*. Le chronogramme 5.7 précise les interactions entre les agents au cours du temps. On retrouve deux grandes étapes :

1. Tout d'abord la création du modèle initial (ligne 33 du listing 4.4), puis la sortie de la zone aveugle (lignes 35-36 du même listing).

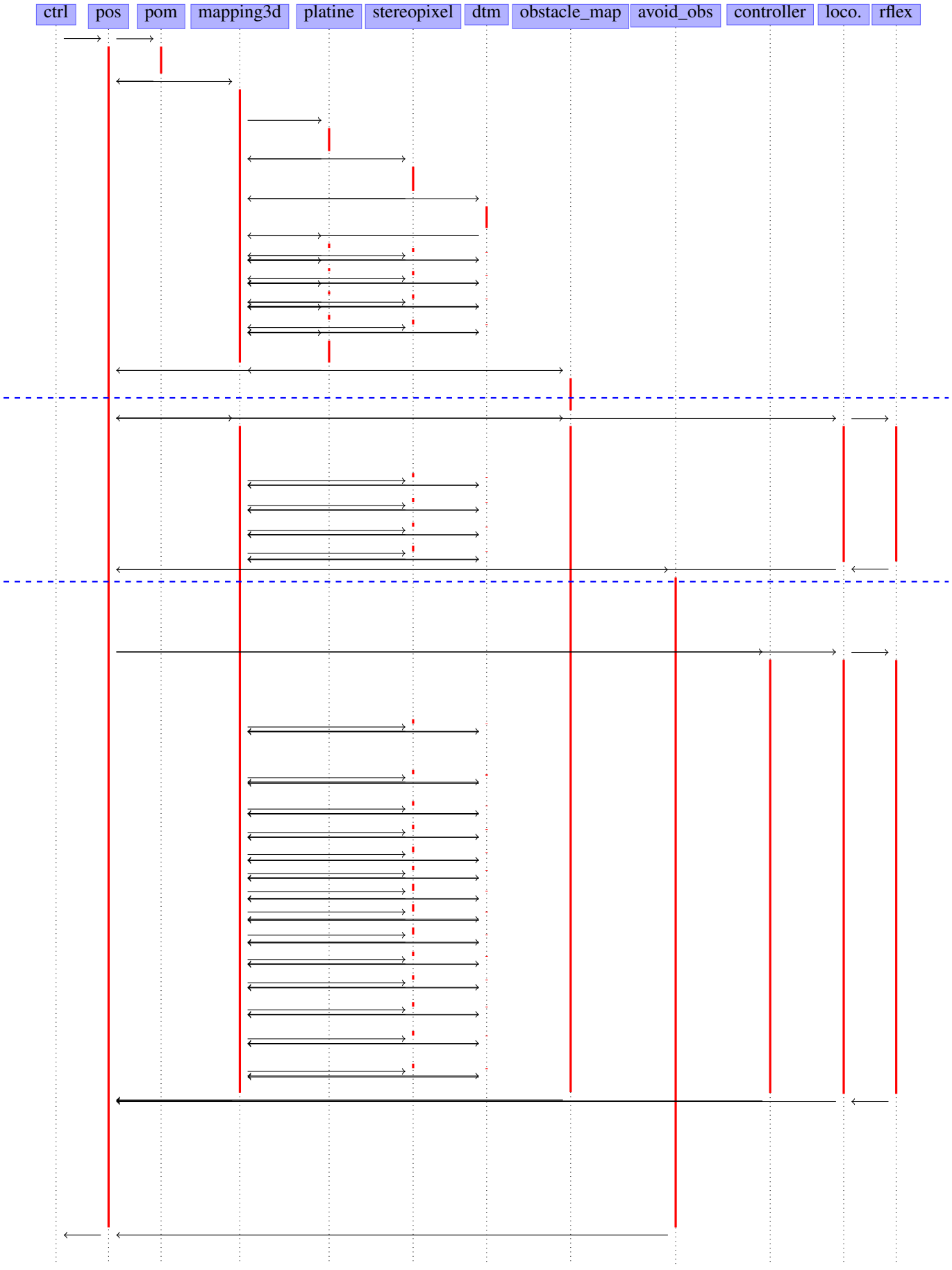

FIGURE 5.5 – Chronogramme d'un cas nominal d'exécution pour une mission de navigation utilisant la stratégie 2D. Correspond à environ 30 secondes de mission.

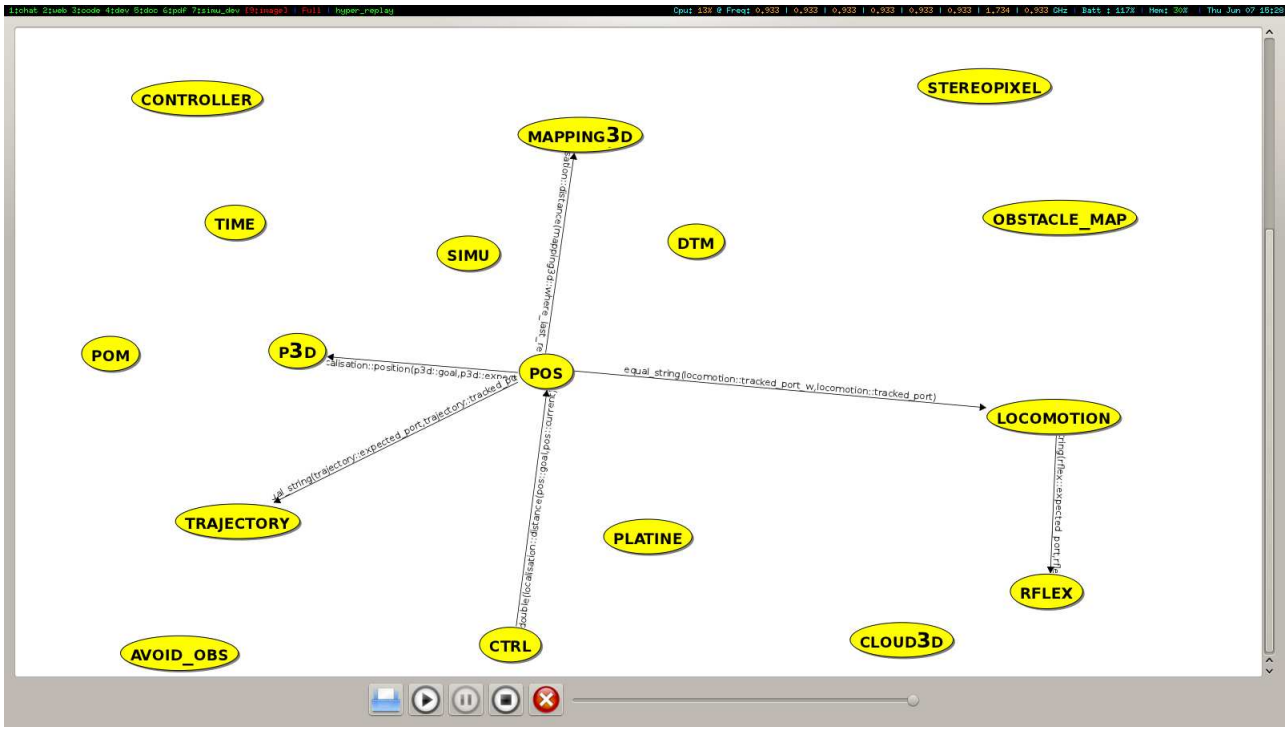

FIGURE 5.6 – Configuration pour la stratégie de navigation 3D

2. dans un second temps, on rentre dans la phase nominale, correspondant à la fonction `follow_goal`, et à la configuration présentée ci-dessus. Deux faits sont intéressants à noter. Le premier est le temps important entre le temps de démarrage de la contrainte sur l'agent *p3d* et celle sur l'agent *locomotion*. Cela correspond en fait à l'exécution de la tâche d'initialisation de l'agent *p3d*, qui consiste à démarrer le module *p3d*, puis à le configurer. En terme d'état de la contrainte, il s'agit de la période où la contrainte est dans l'état *init*, la contrainte suivante n'étant exécutée qu'au moment où celle-ci passe dans l'état *running*. Pour gérer sa contrainte, l'agent *mapping3D* utilise une stratégie basée sur des critères géométriques (une distance parcourue depuis la mise à jour). Cela peut être observé par l'absence de contrainte sur les agents *stereopixel* et *dtm* quand le robot est à l'arrêt (*i.e.* quand il n'a pas de contraintes sur l'agent *locomotion*).

Gestion locale d'une erreur Dans le listing 4.5, nous avions défini une solution spécifique lorsque la stratégie précédente échouait. Le robot efface sa carte, puis revient sur ses pas de quelques mètres en reconstruisant sa carte. Ensuite, il reprend le processus normal (modélisation, planification, exécution). La figure 5.8 représente cette configuration spécifique. L'agent *pos* va contraindre *trajectory* afin qu'il rejoue la trajectoire mémorisée et contraint l'agent *locomotion* à suivre cette trajectoire. Dans le même temps, la carte est reconstruite via une contrainte sur l'agent *3DMap*. Le chronogramme 5.9 présente un scénario où une erreur est spécifiquement injectée pour tester ce comportement. On retrouve trois grandes phases :

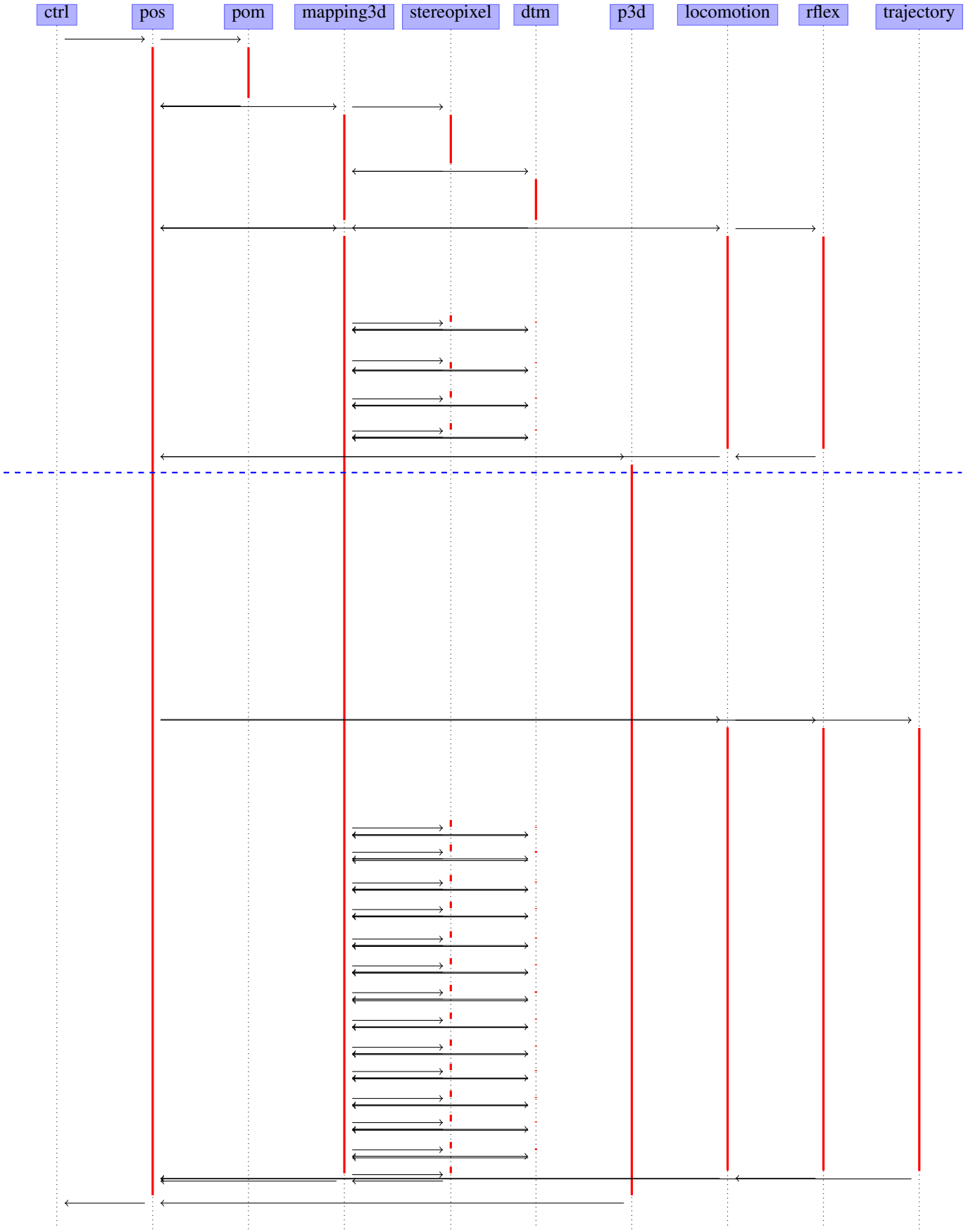

FIGURE 5.7 – Chronogramme d'un cas nominal d'exécution pour une mission de navigation utilisant la stratégie 3D. Correspond à environ 20 secondes de mission.

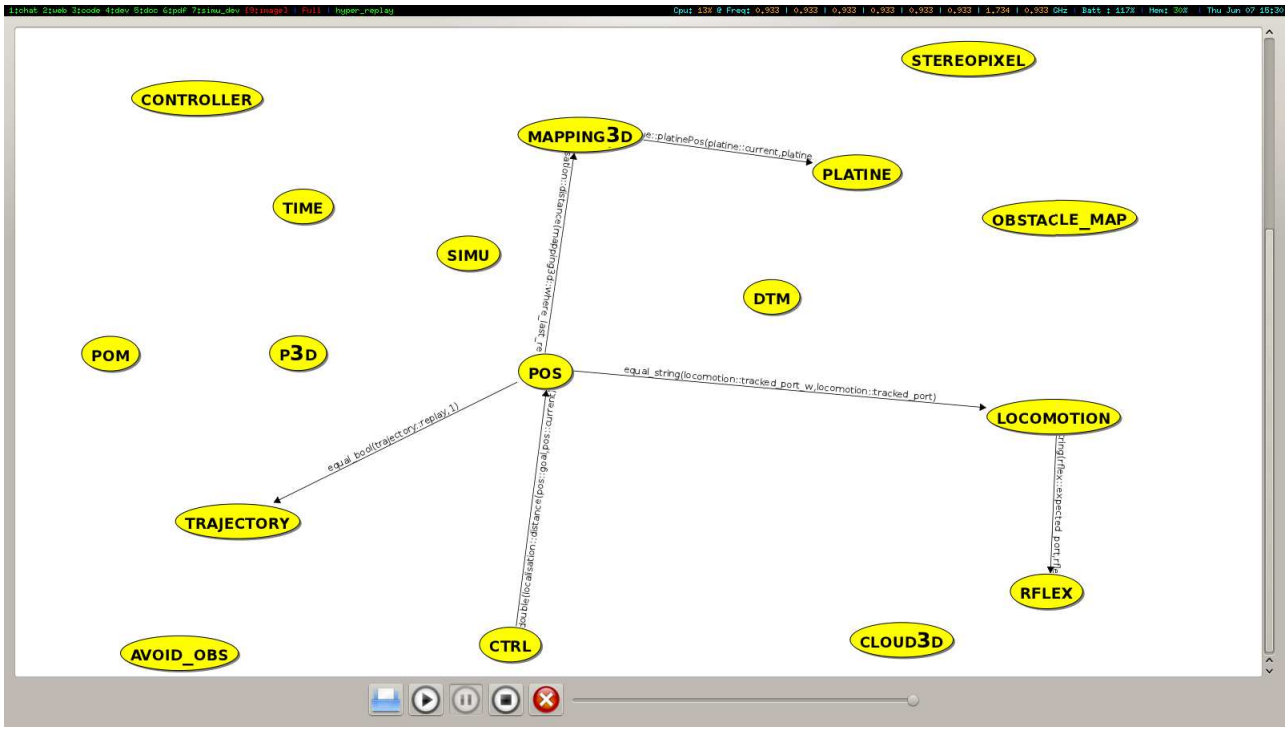

FIGURE 5.8 – Configuration temporaire lors de la réparation d'un échec de navigation réactive

1. phase nominale d'exécution de la stratégie, correspondant au chronogramme 5.7
2. une seconde phase, après la détection de l'erreur afin de réparer l'erreur via cette stratégie spécifique. Le chronogramme 5.10 représente de manière plus fine ce qu'il se passe dans cette période. Après l'échec de l'agent *p3d*, les autres contraintes vont être annulées et on peut observer qu'elles se terminent elles aussi. L'agent enchaîne alors très rapidement sur la stratégie de réparation, en remettant tout d'abord une contrainte sur *3DMap* puis sur *trajectory* et enfin *locomotion*.
3. retour à une phase nominale , *i.e.* l'activation coordonnée des agents *p3d*, *locomotion* et *trajectory* (on peut l'observer de manière plus fine à la fin du chronogramme 5.10).

5.3.3 Performances

Après avoir vu le comportement global du système implémenté en termes d'échanges de contraintes, nous nous intéressons maintenant à ses performances. Il est en effet raisonnable de se demander quel impact les différentes caractéristiques de l'architecture (moteur logique, processus séparé, communication par TCP/IP, ...) peuvent avoir sur le temps de réaction du système.

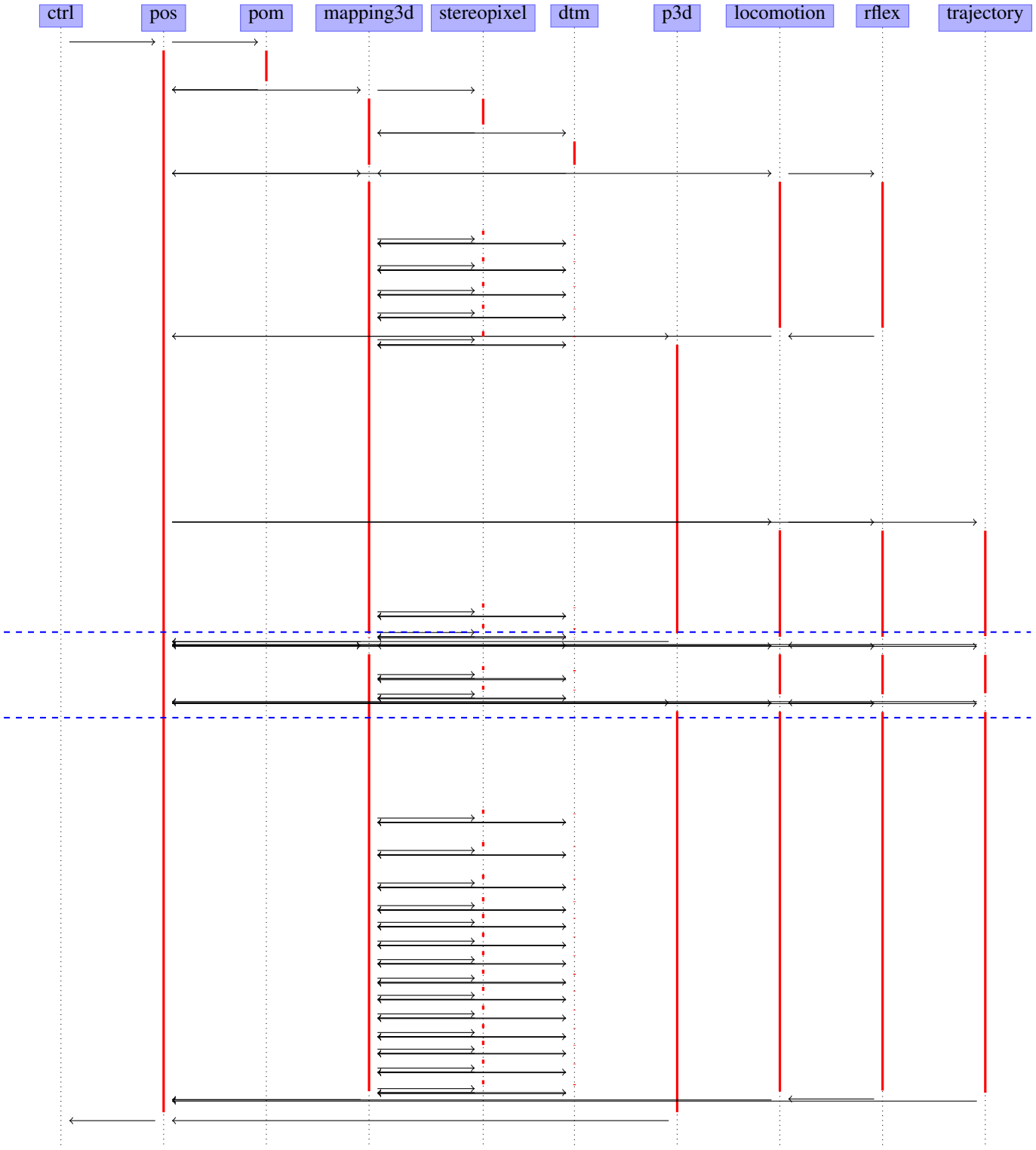

FIGURE 5.9 – Chronogramme d'un cas d'échec d'exécution pour une mission de navigation utilisant la stratégie 3D, puis d'utilisation d'une solution locale. Correspond à environ 30 secondes de mission.

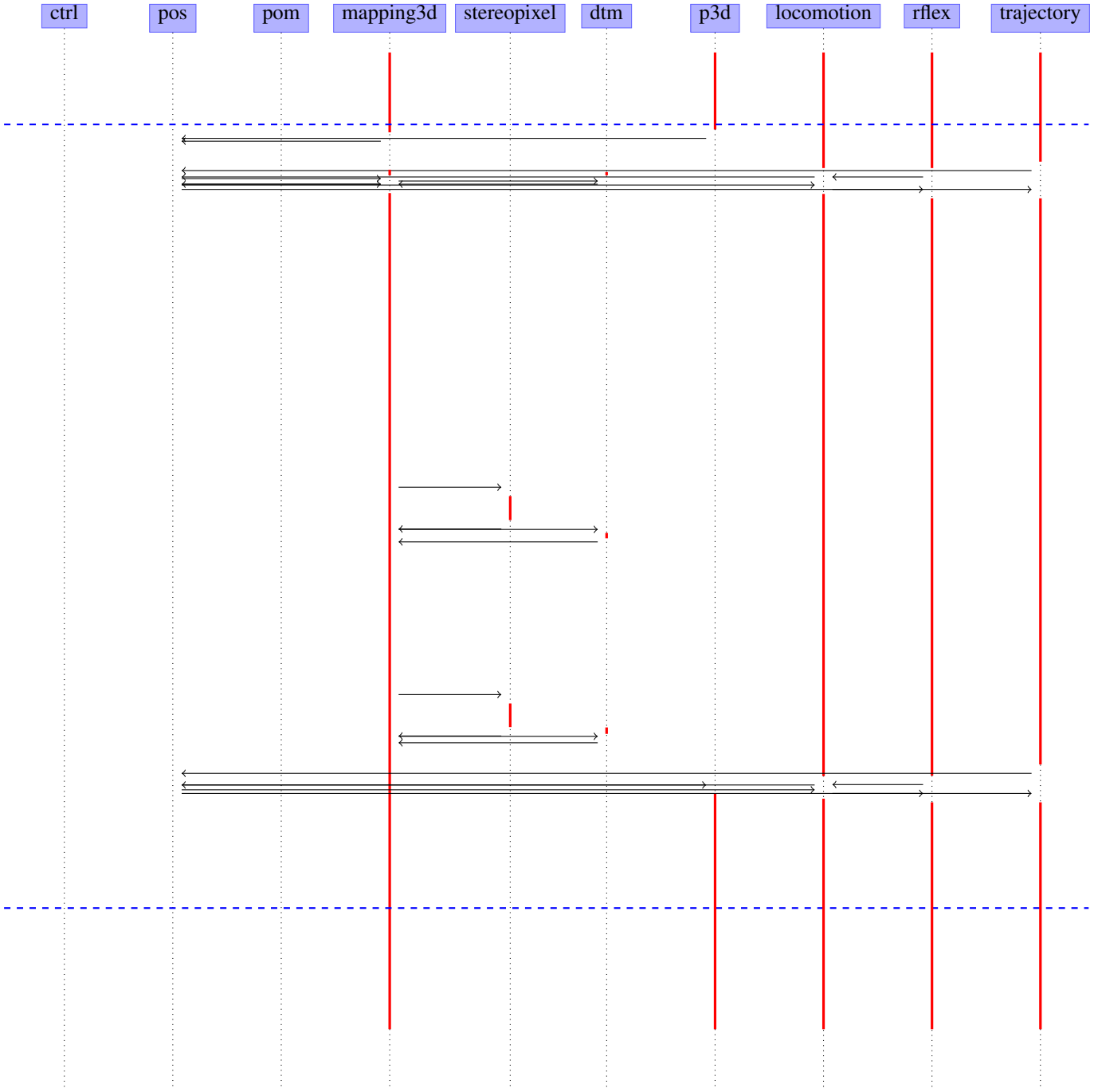

FIGURE 5.10 – Gros plan sur les événements se passant lors de l'injection d'une faute dans la navigation réactive 3D. Correspond à environ 3 secondes.

À l'échelle microscopique

Nous commençons par nous intéresser au cas microscopique. Plus précisément, nous nous intéressons au surcoût entre la gestion complète d'une contrainte, du moteur logique jusqu'à son exécution par rapport à un appel direct à la couche fonctionnelle. Dans un deuxième temps, nous mesurerons le coût d'accès aux variables.

Surcoût d'une contrainte Dans les chronogrammes précédents, nous avons vu que l'agent *3DMap* contraignait régulièrement les agents *dtm* et *stereopixel* qui contrôlent directement des modules fonctionnels. Nous allons donc nous intéresser aux deux contraintes suivantes :

$$time :: delay(dtm :: last\_refresh) < 20$$

$$time :: delay(stereopixel :: last\_refresh) < 20$$

TABLE 5.2 – Temps de réponses de diverses contraintes

	appels	moyenne (ms)	min (ms)	max (ms)	moyenne fonctionnel (ms)
(1)	287	16.01	12.152	1056.58	11.32
(2)	288	98.869	81.272	1130.11	96.88

Le tableau 5.2 présente les mesures que nous avons effectuées. La première chose qu'on peut noter, c'est que la valeur maximale est très éloignée de la moyenne, cela correspond en fait à leur première exécution qui inclut en plus de la tâche "normale" l'initialisation du module, processus qui est relativement long (de l'ordre de la seconde). En moyenne, le surcoût de ces deux opérations reste modique, de l'ordre de quelques millisecondes. Si l'on regarde plus finement, on obtient respectivement pour la première et la seconde contrainte :

- décomposition en arbre de tâches : 1.46 et 1.38 ms
- sélection d'une recette : 1.62 et 0.36 ms
- exécution de la recette : 12.93 et 97.12 ms

Les différentes étapes du calcul ont bien un coût mais celui-ci reste très faible pour des agents simples. Le système reste donc très réactif. Pour des agents plus complexes, le temps de délibération augmentent mais restent raisonnable : par exemple, pour réaliser la tâche *go_to* de l'agent *pos*, celui-ci met en moyenne 4.28 ms pour choisir laquelle exécuter.

Coût d'accès aux variables distantes Dans notre contexte d'études, plusieurs cas sont intéressants à étudier. L'agent *3DMap* accède à la variable *dtm::where_last_refresh* à chaque exécution de sa recette *fuse*, afin de calculer le dernier endroit où il a récupéré de l'information. Cette variable n'a pas de tâche associée et donc le temps pour l'accès à cette variable correspond directement au surcoût d'accès à une variable distante de l'architecture. La variable *pom::current_pos* possède, elle, une tâche associée, qui correspond à la lecture de l'information dans un port de la couche fonctionnelle, nous nous intéressons donc au coût supplémentaire liée à cette opération.

Enfin, nous nous intéressons à la variable *pos::current* qui correspond à la position réelle du robot exposée, elle utilise donc une tâche spécifique qui va lire *pom::current_pos* pour calculer la position du robot.

TABLE 5.3 – Temps d'accès à différentes variables

	appels	moyenne (ms)	min (ms)	max (ms)	fonctionnel (ms)
dtm::where_last_refresh	287	0.18	0.11	0.32	N/A
pom::current_pos	3238	0.84	0.51	2.21	0.002
pos::current	3238	1.06	0.54	3.57	N/A

Le tableau 5.3 résume les temps mesurés pour les accès à ces différentes variables. Là encore, on peut noter qu'il existe un lié coût à notre architecture, mais que celui-ci reste relativement faible. La différence entre l'accès *pom::current_pos* et *pos::current* est en moyenne seulement de 216 μs, incluant la sélection de la bonne recette : la communication en TCP/IP sur une même machine n'entraîne donc pas de réel surcoût.

Au niveau macroscopique

Pour mesure le coût de notre architecture au niveau macroscopique, nous allons étudier le cas de la navigation. Dans le simulateur MORSE, nous allons demandé au robot de réaliser une boucle en spécifiant un certain nombre de points de passage. L'expérience sera répétée 20 fois pour chaque architecture de contrôle (NAV_CTRL et HYPER). Nous utilisons MORSE ici afin de garantir au maximum la répétabilité de l'expérience, et évacuer les problèmes liés aux changements possibles de l'environnement et à l'indéterminisme des acquisitions de données par les capteurs. Le tableau 5.4 présente les résultats obtenus avec la distance entre chaque point de passage, et le temps pris en moyenne par le robot pour l'atteindre.

TABLE 5.4 – Temps (secondes) pris en moyenne pour accéder au prochain point de passage

distance (m)	NAV_CTRL (S)	HYPER (S)
10	9.18	9.32
20	18.84	18.33
20	17.02	16.94
30	28.13	23.94
67	62.95	61.64
50	45.63	40.53
42	41.76	37.74
50	50.99	45.66
20	19.74	16.60
70	75.63	66.33
28	32.76	33.81

Les résultats obtenus montrent que non seulement la solution proposée n'induit pas de surcoût au niveau de la mission, mais qu'en plus, elle est généralement plus rapide que la solution

précédemment développée. Plusieurs explications sont envisageables : implémentation moins efficace, moins bonne gestion du parallélisme (en particulier utilisation de nombreux verrous de synchronisation), code pour chaque tâche moins spécialisé...

5.4 Synthèse

Dans ce chapitre, nous nous sommes intéressés à l'implémentation et à l'exploitation de l'architecture ROAR. Nous avons d'abord présenté les points importants de notre implémentation, en nous focalisant sur les aspects liés aux questions de robustesse et de performance. Ensuite, nous avons expliqué comment nous avons développé une couche de supervision pour un robot terrestre en utilisant HYPER. Dans le même temps, nous avons comparé ce développement avec le développement d'une autre solution de supervision, basée sur une bibliothèque spécialisée en C++. Cela nous a permis de mettre en exergue les points forts de ROAR en ce concerne les problèmes de concurrence, mais aussi les facilités de développement qu'il offre. Enfin, nous avons montré quelques cas d'utilisation de cette couche de supervision. Tout d'abord, nous avons explicité la dynamique d'interactions des agents à l'aide de chronogrammes. Ensuite, nous nous sommes intéressés aux performances du système. Même si HYPER présente un très léger surcoût par rapport à des appels directs à la couche fonctionnelle, il reste très réactif. Au niveau macroscopique, l'architecture proposée semble légèrement plus efficace que la solution plus directe développée précédemment.

Ainsi, nous avons pu montrer que le système proposé est efficace et qu'il répond en pratique aux besoins exprimés dans la section 2.1.2. En effet, la gestion de la **concurrence** en est grandement facilitée. La décomposition en agents "physiquement distincts" améliore la **robustesse** du système de contrôle. De plus, l'approche proposée est globalement plus **modulaire** et **composable** que notre approche précédente. Même si deux agents développés séparément mettent des contraintes inconciliables sur le même agent, l'erreur sera détectée au niveau de l'appelé, gérée correctement, et une erreur explicitant le conflit est remontée à l'appelant. Enfin, par des mesures de performances, nous avons pu montrer que le système reste grandement **réactif**, même si certains agents peuvent évidemment avoir des temps de décision plus longs.

Deuxième partie

Formalisation

Chapitre 6

Formalisation d'une architecture de contrôle : besoins et approches

Dans ce chapitre, nous commençons par nous intéresser aux différentes motivations qui justifient le besoin de formaliser les architectures de contrôle de robots. Ensuite, nous présentons rapidement les principales approches proposées dans la littérature pour développer des modèles formels des architectures de contrôle.

6.1 Du besoin d'une approche formelle

6.1.1 Analyse, extension, et transformation

Dans la première partie du manuscrit, nous avons présenté l'architecture ROAR comme étant définie par un assemblage d'algorithmes, de sémantiques plus ou moins informelles et de machines à états. Pour chaque problème que devait résoudre l'architecture de contrôle, nous avons essayé d'expliciter la solution proposée de la manière la plus idiomatique possible, en particulier en utilisant à chaque fois l'outil le plus approprié pour présenter cette solution. Cette approche pragmatique atteint toutefois des limites quand on veut étudier l'interaction entre ces différentes idées, en particulier dans le cadre d'optimisation ou d'extension. Dans ce contexte, il semble nécessaire de modéliser l'ensemble de l'architecture grâce à une méthode formelle unique, nous permettant ainsi de raisonner globalement sur le système.

Implémentations et transformations L'implémentation telle que décrite dans la section 5.1 est naïve, dans le sens où elle ne fait aucune optimisation. Pour pouvoir faire des optimisations correctes, il faut être capable de montrer que la sémantique du code optimisé est identique à celle du code naïf, ou autrement dit qu'ils ont les mêmes effets observables. Il est donc important de

définir la sémantique formelle d'une architecture décisionnelle si l'on veut pouvoir proposer des optimisations correctes du système.

Extensions Une architecture logicielle ne doit pas être figée, elle doit pouvoir intégrer les évolutions des fonctionnalités robotiques, de la définition des missions, des besoins en autonomie... Pour intégrer ces évolutions, on va chercher à satisfaire de nouvelles propriétés, ou les exprimer de manière plus fine, ou bien encore s'intéresser à de nouveaux problèmes. Lorsqu'on ajoute une fonctionnalité ou une propriété au sein d'une architecture, il est essentiel de s'interroger sur le bien fondé de cet ajout, à savoir si il est nécessaire (*i.e.* il permet d'exprimer et/ou de répondre à des besoins nouveaux) ou redondant. Là encore, il est nécessaire d'avoir une sémantique formelle pour décider si deux expressions sont équivalentes. De plus, on peut s'interroger si cet ajout n'a pas des conséquences non envisagées, brisant certaines propriétés, ou empêchant d'atteindre certains états. Cela nous amène au problème de la vérification, discuté dans la section 6.1.3.

6.1.2 Comparaison, intégration

Comparaisons avec d'autres architectures Comme décrit dans le chapitre 3, il existe de nombreuses architectures de contrôle, chacune avec leurs spécificités. Une approche possible de comparaison est de développer, sur une même couche fonctionnelle, différentes couches de supervision (chacune utilisant une architecture particulière). Il serait alors possible d'utiliser des métriques pour comparer "l'efficacité" de ces différentes solutions, mais cela soulève toutefois plusieurs problèmes. Le premier est évidement la difficulté de mettre en place une telle solution, et les erreurs possibles lors des instanciations des différentes architectures. La seconde est la question de l'interaction entre couche fonctionnelle et couche décisionnelle : en fixant une couche fonctionnelle, on peut rendre plus ou moins ardue la tâche du développement d'une solution de contrôle. Enfin, cette approche n'explicite pas réellement les différences de sémantique (ce qui est exprimable dans une architecture mais pas dans l'autre). Pour répondre à de telles questions, il est nécessaire d'avoir une sémantique formelle pour ces différentes solutions. On peut alors chercher à montrer formellement que certaines combinaisons de constructions dans deux architectures sont équivalentes, ou bien qu'au contraire certaines constructions d'une architecture donnée ne sont pas exprimables dans une autre.

Intégration avec des planificateurs symboliques ROAR a jusqu'à présent surtout été utilisé pour contrôler de manière fine la couche fonctionnelle d'un robot. Il est intéressant de s'interroger sur les interactions possibles entre l'architecture ROAR et un planificateur symbolique classique. La question au centre de ce problème est de savoir ce que représente un plan. Là encore, il s'agit d'un problème de sémantique entre deux systèmes à priori distincts. La définition précise de leurs sémantiques dans un modèle formel commun permet à l'architecture de comprendre le plan, ou de le transformer, de manière correcte, en un ensemble de constructions qu'elle pourra exécuter. Comme pour la comparaison avec d'autres architectures, il est possible que l'on montre

que la sémantique d'un certain type de plan n'est pas exprimable dans ROAR : dans ce cas, il faudra étendre l'architecture ou décider que ces plans ne sont pas exécutables dans l'architecture.

6.1.3 Vérification, aide à la programmation

Les robots sont des agents physiques, interagissant avec le monde physique. Dans ces conditions, il est possible que le robot s'endommage, blesse une personne, ou bien détériore l'environnement. Ce sont évidemment des situations qu'il est indispensable d'éviter, la propriété de sûreté étant fondamentale. D'un autre côté, le développeur aimerait pouvoir vérifier, sans explicitement exécuter le code, que certaines situations sont atteignables : en partant d'un état initial, est-il possible d'arriver dans la situation but ? (propriétés d'atteignabilité).

Pour se faire, les développeurs de robots se basent généralement sur des jeux de tests, plus ou moins complets, qui évaluent le comportement global du robot. Malheureusement, étant donnés la grande complexité d'un robot et l'ensemble des évènements qui peuvent survenir dans l'environnement, il est illusoire de tester toutes les configurations possibles, et donc de garantir certaines propriétés de sûreté sur les robots. On peut alors se tourner vers des méthodes plus formelles, afin de prouver, via différentes techniques ("model checking", preuves automatisées, ...) que certaines propriétés sont vraies. Toutefois, ces méthodes demandent une formalisation initiale avant de pouvoir être appliquées.

6.1.4 Les défis

Ainsi, de nombreuses raisons plaident en faveur d'une formalisation de la couche décisionnelle, en particulier pour des besoins de correction – que ce soit de transformations de code, ou bien par rapport à des critères de sûreté. La difficulté principale de la formalisation d'une architecture décisionnelle vient de sa nature même, à savoir l'imbrication des phases de planification et d'exécution concurrente. Ces deux phases reposent le plus souvent sur des modèles différents, rendant difficile le raisonnement sur l'intégralité de la couche décisionnelle. La planification peut être assimilée à une déduction logique classique et n'interagit pas directement avec l'environnement. Au contraire, la phase exécutive induit des changements concurrents sur l'état du robot, *i.e.* des changements de l'état du robot ont lieu en parallèle, et ont possiblement de l'influence les uns sur les autres. De plus, ces changements sont parfois non réversibles. Ces deux propriétés, concurrence et non réversibilité sont difficiles voir impossibles à modéliser en logique classique.

De nombreux modèles ont été proposés pour représenter les systèmes concurrents, nous allons maintenant en étudier quelques uns, et voir comment ils peuvent se combiner élégamment avec une approche logique.

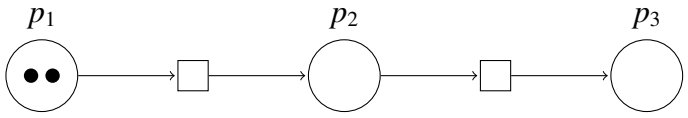

FIGURE 6.1 – Un réseau de Pétri simple, représentant un calcul séquentiel. 3 places et deux transitions sont définies. Le marquage initial correspond aux deux jetons présents dans p_1.

6.2 Modèles formels pour la concurrence

6.2.1 Réseaux de Pétri

Définition

Le modèle des réseaux de Pétri (Wolfang Reisig, 2010) est probablement le modèle le plus connu et le plus utilisé pour modéliser et raisonner sur des systèmes concurrents. De manière informelle, un réseau de Pétri est un ensemble de places, de transitions, et de flèches connectant une place avec une transition ou une transition avec une place. Les places peuvent contenir des jetons, qui représentent en général les ressources disponibles. L'évolution du système correspond à l'exécution de transitions, faisant passer les jetons d'une place à l'autre. Graphiquement, un réseau de Pétri peut être défini comme un graphe biparti orienté, comme illustré par la figure 6.1.

Extensions

De nombreuses extensions ont été proposées pour les réseaux de Pétri, permettant d'en augmenter l'expressivité, mais souvent en réduisant le nombre d'outils mathématiques disponibles pour analyser le réseau. Tout d'abord, les arcs inhibiteurs (Agerwala and Flynn, 1973; Janicki and Koutny, 1995) ajoutent comme précondition à une transition qu'elle ne peut être exécutée que si une place est vide. En particulier, cela permet d'exprimer des contraintes arbitraires sur le nombre de jetons et rend ainsi le formalisme aussi expressif qu'une machine de Turing. Dans la définition standard d'un réseau de Pétri, les jetons sont indistinguables les uns des autres. Dans la version coloriée (Jensen, 1987), chaque jeton possède en plus des données associées. Certains outils avancées tels CPN-tools (Jensen et al., 2007) associent un type à ces données et permettent d'utiliser un langage fonctionnel de haut niveau SML pour tester les transitions. Ainsi, on combine un langage de développement classique avec les possibilités des réseaux de Pétri pour transcrire des processus concurrents. Enfin, un dernière extension intéressante pour notre domaine est l'introduction des réseaux de Pétri temporisés (Berthomieu and Menasche, 1983), permettant de représenter des phénomènes avec une durée, ou avec des délais.

Réseaux de Pétri et robotique

L'architecture Orccard (Borrelly et al., 1998) propose de développer les contrôleurs d'un robot en deux temps. D'un côté, le langage MAESTRO permet d'écrire des procédures et génère du code dans le langage synchrone ESTEREL (Berry, 2000). Dans un second temps, ces procédures sont composées via une interface graphique. Un réseau de Pétri est alors automatiquement extrait

pour vérifier l'absence d'interblocage. Dans ProCoSa (Barbier et al., 2006), les contrôleurs sont directement spécifiés sous forme de réseaux de Pétri. Il est alors possible de vérifier qu'à aucun moment une place ne contient strictement plus d'un jeton, ainsi que l'absence d'interblocage ou de boucles infinies. Dans (Ziparo et al., 2008), les auteurs proposent de représenter des plans (potentiellement multi-robots) sous forme de réseaux de Pétri, permettant ainsi de leur donner une sémantique d'exécution précise.

Synthèse

Les réseaux de Pétri sont un formalisme puissant et bien compris. Ils permettent de spécifier (notamment graphiquement) divers processus concurrents, et il est possible de montrer de nombreuses propriétés sur ces réseaux. Toutefois, il faut noter qu'il est en général complexe de composer plusieurs réseaux en un unique réseau, rendant l'approche difficile dans un contexte de composants. De plus, il ne semble pas y avoir de moyens évidents de mêler réseaux de Pétri et logique, et donc de raisonner au niveau global de l'architecture.

6.2.2 Algèbres de processus

Définition

Un autre modèle commun pour représenter des calculs concurrents sont les algèbres de processus. De nombreuses algèbres ont été proposées, avec de subtiles différences sémantiques, les plus connues étant le "Calculus of Communicating Sequential Process" (CSP, (Hoare, 1978)), le "Calculus of Communicating Systems" (CCS, (Milner, 1982)) et le π-calcul (Milner et al., 1992). Dans la suite de cette section, nous illustrerons nos propos avec des exemples venant du π-calcul, mais les principes sont applicables pour l'ensemble des algèbres de processus.

Pour définir une algèbre de processus, on définit d'abord un ensemble de noms représentant des processus de base et un ensemble d'opérateurs qui permettent de combiner ces processus. Ainsi, la grammaire du π-calcul est :

$$
\begin{aligned}
P, Q ::= {} & P \,.\, Q && (6.1)\\
\mid {} & P \mid Q && (6.2)\\
\mid {} & x(y).P && (6.3)\\
\mid {} & \overline{x}\langle y\rangle.P && (6.4)\\
\mid {} & (\nu x)P && (6.5)\\
\mid {} & !P && (6.6)\\
\mid {} & 0 && (6.7)
\end{aligned}
$$

où 6.1 est l'opérateur de combinaison séquentiel, 6.2 l'opérateur de combinaison parallèle, 6.4, 6.3 sont les opérateurs d'échanges de messages (respectivement pour attendre et envoyer un message y sur le canal de communication x), 6.5 crée un nouveau nom pour P, 6.6 crée une

copie de P et 6.7 est le processus nul (qui ne fait rien). L'évolution du système est définie par un ensemble de règles de réduction. Par exemple, pour obtenir la transformation

$$x\langle y\rangle.P \mid x(v).Q \rightarrow P \mid Q[y/v]$$

on utilise les règles de réduction suivantes :

1. le processus $x\langle y\rangle.P$ envoie le message y au travers du canal de communication x puis se transforme en P.
2. le processus $x(v).Q$ reçoit y dans le canal de communication x et remplace les occurrences de v dans Q par y.

Contrairement aux réseaux de Pétri, les algèbres de processus spécifient explicitement comment les processus communiquent, ce qui est important lorsqu'on s'intéresse à l'évolution dynamique des liens entre les différents processus. L'autre avantage de cette approche est sa nature intrinsèquement composable (puisque basée sur une structure algébrique). Du côté des méthodes formelles applicables à ces méthodes, la littérature s'est concentrée sur des techniques de bisimulations [1] pour montrer l'équivalence de deux processus (Milner et al., 1992). Il est aussi possible de transformer ces algèbres en systèmes de transitions étiquetées pour y appliquer des méthodes de "model checking"(Roscoe, 1994; Sun et al., 2009).

Les algèbres de processus en robotique

Dans (Lyons, 1993), les auteurs proposent de représenter des plans, mais aussi les changements dans le monde via une algèbre de processus autour de RS [2]. Ils peuvent alors étudier les évolutions possibles du système ($Plan \mid World_0$). Lorsque le plan est terminé, ils peuvent alors vérifier que l'état du monde ne contient pas d'états inconsistants. FROB (Peterson et al., 1998) reprend l'analyse précédente et montre comment leur langage dédié correspond à l'algèbre proposée. Toutefois, à notre connaissance, les auteurs n'utilisent pas les méthodes de vérification proposées, ni n'exploitent réellement cette algèbre. (Eberbach et al., 1999) propose l'architecture multi-robots SAMON [3] qui repose sur le $-calcul. Ce calcul propose non seulement les opérateurs classiques de compositions de processus, mais introduit aussi la notion de coût (et les opérateurs associés) ainsi que la notion de mutation. Ils proposent ensuite d'utiliser des méthodes d'optimisation pour adapter et optimiser les fonctions encodées en $-calcul sur chaque robot aux conditions réelles courantes.

Synthèse

Les algèbres de processus permettent de spécifier des processus concurrents de manière modulaire et composable. Différentes techniques sont utilisables pour montrer l'équivalence entre

1. Une bisimulation est une relation binaire entre des systèmes de transition d'états associant les systèmes qui se comportent de la même façon au sens qu'un des systèmes simule l'autre. De manière intuitive, deux systèmes sont bisimilaires si ils sont capables de s'imiter l'un l'autre.
2. pour Reactive Schemas
3. pour Ocean SAmpling Mobile Network

deux processus, ou pour prouver certaines propriétés de sûreté. Toutefois, il semble difficile de représenter la déduction logique dans ce type d'algèbres, et par la même difficile de raisonner globalement sur une architecture.

6.2.3 Logique de réécriture

Définition

La logique de réécriture (Meseguer, 1992) est un langage de spécification formelle dans lequel la dynamique d'un système est exprimée par un ensemble de règles de réécriture qui agissent sur l'état du système. La déduction logique dans ce système correspond ainsi à l'évolution de l'état du système. Le modèle associé à cette logique correspond bien aux systèmes concurrents, qui sont des systèmes dont les états sont distribués et où les changements peuvent avoir lieu en même temps. De plus, il est possible de faire du "model checking" (Meseguer et al., 2003; Bae and Meseguer, 2010) sur ces modèles pour vérifier des propriétés de sûreté.

Expressivité

Il est vite apparu que cette logique permettait d'encoder les modèles précédents de concurrences : par exemple, (Stehr et al., 2001) montre comment représenter la sémantique des réseaux de Pétri en logique de réécriture, et (Viry, 1996) montre comment implémenter les règles de réductions de π-calcul en logique de réécriture. Mais cette logique est capable d'exprimer bien plus que les précédents modèles : l'outil Maude (Clavel et al., 2002) propose un langage de programmation complet basé sur la logique de réécriture, et dans (Marti-Oliet and Meseguer, 1999), les auteurs proposent d'utiliser cette logique pour traduire et résoudre des problèmes de planification. Enfin, il est possible d'encoder différentes logiques en logique de réécriture (Marti-Oliet and Meseguer, 1996), permettant ainsi de créer un cadre puissant pour raisonner sur différents systèmes logiques.

Utilisation en robotique

Dans (Dowek et al., 2010), les auteurs proposent d'utiliser la logique de réécriture pour vérifier la sémantique du langage synchrone d'exécution de plan PLEXIL (Jonsson et al., 2006) développé par la NASA. Ce langage avait déjà été analysé avec des méthodes formelles (Dowek et al., 2007), mais cette nouvelle analyse a permis de trouver deux nouvelles erreurs dans la spécification PLEXIL (qui ne remettent cependant pas en cause les résultats de G. Dowek). Les travaux de (Ammar et al., 2011) proposent eux une méthodologie pour formaliser les systèmes critiques basés sur des systèmes multi-agents, et peuvent être donc une bonne base pour la formalisation de systèmes comme IDEA , T-REX ou bien ROAR.

Synthèse

La logique de réécriture est un modèle très général, capable aussi bien d'encoder des systèmes concurrents que des problèmes de planification ou des systèmes logiques. De plus, différentes

méthodes de “model checking”ont été développées pour vérifier des propriétés de sûreté sur ces modèles. Elle semble donc une bonne candidate comme modèle pour analyser la couche décisionnelle des robots.

6.2.4 Logique linéaire

Définition

La logique linéaire (Girard, 1987) est une logique de ressource : les formules logiques sont par défaut consommées lorsqu'elles sont utilisées. Pour cela, J. Girard doit supprimer des règles de la logique classique les règles de contraction et d'affaiblissement, à savoir

$$A \Rightarrow A \wedge A$$
$$A \wedge B \Rightarrow A$$

En effet, ces règles n'ont en généralement pas de sens lorsque l'on traite de ressources finies, la première correspondant à la multiplication “magique” des ressources, et la seconde à leur disparition sans aucune explication. Un opérateur spécial est introduit pour dénoter qu'une ressource est réutilisable, permettant de raisonner sur des vérités générales, et faisant ainsi le lien avec la logique classique. La logique linéaire permet donc de raisonner sur des systèmes qui évoluent, et en particulier sur des systèmes informatiques.

Liens avec la concurrence

Plusieurs approches ont été proposées pour encoder la concurrence en logique linéaire. Celle qui semble retenue aujourd'hui est l'approche “processus-as-formula” initiée dans (Martí-Oliet and Meseguer, 1989) qui étend la correspondance de Curry-Howard (qui fait le lien entre preuve en logique intuitionniste et calcul “classique” (λ-calcul)) au monde concurrent. Dans cette approche, on identifie les processus à des formules logiques, et l'évolution de ces processus correspond à la déduction logique. En utilisant cette méthode, il est possible d'encoder précisément les réseaux de Pétri (Martí-Oliet and Meseguer, 1989) ou des algèbres de processus (Bellin and Scott, 1994). Dans (Kobayashi and Yonezawa, 1993), les auteurs utilisent ce principe comme base pour un langage concurrent de haut niveau ACL. Plus récemment, (Watkins et al., 2004) introduit CLF, un cadre logiciel pour raisonner sur les systèmes concurrents, utilisant une représentation un peu différente : les calculs concurrents sont encapsulés dans une monade (Moggi, 1991), une structure algébrique spécifique permettant de capturer certains comportements, séparant ainsi les calculs “classiques” des calculs concurrents (et préservant ainsi certaines propriétés du cadre logiciel sur les calculs “classiques”). Lollimon (López et al., 2005) étend la sémantique opérationnelle de CLF pour fournir un langage de programmation concurrent. En particulier, il va interpréter différemment les calculs à l'intérieur ou à l'extérieur de la monade. À l'extérieur, il utilise une recherche de preuve classique basée sur le chaînage arrière, tandis qu'à l'intérieur, il utilise un chaînage en avant, qui modélise de manière plus naturelle les calculs concurrents. Enfin, dans (Cervesato and Scedrov, 2009), les auteurs étudient les précédents modèles de concurrences et

montrent comment ils peuvent être tous encodés en utilisant un même fragment de la logique linéaire.

Méthodes d'analyse

La question de l'accessibilité dans cette approche revient à la question de la prouvabilité en logique linéaire. Ainsi, on peut utiliser des méthodes de recherche de preuves automatiques ou semi-automatiques (Chaudhuri, 2006) pour prouver l'accessibilité de certains états. Dans (Bozzano et al., 2004), l'auteur propose des techniques de "model checking"pour vérifier des propriétés de sûreté sur des spécifications décrites en logique linéaire. Enfin, une autre voie d'approche est proposée dans (Schack-Nielsen and Schürmann, 2008) : ils proposent le développement d'une méta-théorie pour CLF et l'exploitent pour prouver l'équivalence de deux sémantiques encodées grâce à CLF.

Synthèse

La logique linéaire est une logique de ressource, qui permet d'analyser des systèmes dynamiques, et en particulier des systèmes concurrents. Dans l'approche proposée, l'évolution du système concurrent correspond à la déduction logique. Il parait donc aisé de mélanger la déduction logique "classique" et l'évolution du système concurrent. De ce fait, la logique linéaire semble un bon candidat pour formaliser et raisonner notre architecture décisionnelle.

6.3 Conclusion

La formalisation d'une architecture décisionnelle répond à plusieurs besoins. Tout d'abord, elle permet d'analyser finement une architecture et de la comparer en utilisant un langage commun à d'autres architectures. Ensuite, elle est nécessaire lors de l'interaction entre différents sous-systèmes, par exemple avec un planificateur. Enfin, elle permet d'utiliser des méthodes formelles pour prouver des propriétés d'atteignabilité ou de sûreté de fonctionnement. La difficulté majeure pour formaliser une telle architecture est l'entrelacement constant entre délibération et exécution. Nous avons donc étudié différentes approches pour l'étude de la concurrence. Deux approches paraissent plus particulièrement adaptées pour l'architecture ROAR : la logique de réécriture et la logique linéaire. En effet ces deux modèles, contrairement aux autres, permettent non seulement d'encoder les aspects concurrents mais aussi les aspects délibératifs. Dans la suite de ces travaux, nous nous sommes concentrés sur une approche basée sur la logique linéaire, où l'exécution concurrente coïncide avec la déduction logique "classique" pour un domaine représentant des processus et des messages. Mélanger exécution concurrente et déduction revient donc à mélanger des déductions sur des domaines différents. Au contraire, en logique de réécriture, il faut implémenter explicitement le principe de déduction classique. L'intégration exécution et déduction semble donc plus naturelle en logique linéaire.

Chapitre 7

Formalisation de l'architecture ROAR

Dans ce chapitre, nous proposons une formalisation de l'architecture ROAR *en nous basant sur la logique linéaire. Dans un premier temps, nous présentons plus en détail la logique linéaire. Puis nous nous présentons le modèle proposé pour* ROAR *en utilisant cette logique, en nous focalisant en particulier sur l'aspect exécution.*

7.1 Introduction à la logique linéaire

7.1.1 Calcul des séquents

La logique linéaire est généralement présentée via le système de déduction appelé calcul des séquents (Gentzen, 1935). Nous commençons donc par présenter ce système de déduction.

Séquent Un séquent est un couple de listes finies de formules séparées par l'opérateur "thèse" $\vdash$ du type

$$A_1, A_2, \ldots, A_m \vdash B_1, B_2, \ldots, B_n$$

où A_i sont les hypothèses du séquent, et B_j ses conclusions. Un tel séquent signifie que "si tous les A_i sont vrais, alors il existe un élément de B_j qui est vrai". En logique classique, cela équivaut à

$$A_1 \wedge A_2 \wedge \cdots \wedge A_n \rightarrow B_1 \vee B_2 \vee \cdots \vee B_n$$

Notons quelques cas particuliers. Le séquent $\vdash A$ signifie que A est vrai, tandis que le séquent $A \vdash$ signifie au contraire que A est faux.

Règles Une règle est de la forme

$$\frac{\text{prémisses}}{\text{conclusion}}\ \text{nom}$$

Généralement, on dénote avec les lettres grecques Γ et Δ une suite de formules qui ne sont pas affectées par la règle, et qu'on appelle "contexte". Nous pouvons illustrer ceci via la présentation (simplifiée) de la logique classique, en calcul de séquent (système LK (Gentzen, 1935)).

$$\frac{}{A \vdash A}\ \text{axiome} \qquad \frac{\Gamma \vdash A, \Delta \qquad \Gamma', A \vdash \Delta'}{\Gamma, \Gamma' \vdash \Delta, \Delta'}\ \text{coupure}$$

$$\frac{\Gamma \vdash \Delta}{\Gamma, A \vdash \Delta}\ \text{affaiblissement gauche} \qquad \frac{\Gamma \vdash \Delta}{\Gamma \vdash A, \Delta}\ \text{affaiblissement droite}$$

$$\frac{\Gamma, A, A \vdash \Delta}{\Gamma, A \vdash \Delta}\ \text{contraction gauche} \qquad \frac{\Gamma \vdash A, A, \Delta}{\Gamma \vdash A, \Delta}\ \text{contraction droite}$$

$$\frac{\Gamma \vdash A, \Delta}{\Gamma, \neg A \vdash \Delta}\ \neg\ \text{gauche} \qquad \frac{\Gamma, A \vdash \Delta}{\Gamma \vdash \neg A, \Delta}\ \neg\ \text{droite}$$

$$\frac{\Gamma, A, B \vdash \Delta}{\Gamma, A \wedge B \vdash \Delta}\ \wedge\ \text{gauche} \qquad \frac{\Gamma \vdash A, \Delta \qquad \Gamma \vdash B, \Delta}{\Gamma \vdash A \wedge B, \Delta}\ \wedge\ \text{droite}$$

$$\frac{\Gamma \vdash A, \Delta \qquad \Gamma \vdash B, \Delta}{\Gamma, A \vee B \vdash \Delta}\ \vee\ \text{gauche} \qquad \frac{\Gamma \vdash A, B\Delta}{\Gamma \vdash A \vee B\Delta}\ \vee\ \text{droite}$$

$$\frac{\Gamma \vdash A, \Delta \qquad \Gamma, B \vdash \Delta}{\Gamma, A \to B \vdash \Delta}\ \to\ \text{gauche} \qquad \frac{\Gamma, A \vdash B\Delta}{\Gamma \vdash A \to B\Delta}\ \to\ \text{droite}$$

Preuve dans le calcul des séquents Une preuve dans le calcul des séquents est un arbre de séquents construit à partir des règles du système. À chaque étape, on applique une règle. La démonstration est terminée quand toutes les feuilles de l'arbre sont des axiomes. Essayons de démontrer dans le système LK le principe de contraposition de l'implication, *i.e.* $\vdash (\neg B \to \neg A) \to (A \to B)$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A \vdash A}\ \text{axiome}}{A \vdash A, B}\ \text{aff. droite}}{\neg A, A \vdash B}\ \neg\ \text{gauche} \qquad \dfrac{\dfrac{\dfrac{}{B \vdash B}\ \text{axiome}}{A, B \vdash B}\ \text{aff. gauche}}{A \vdash \neg B, B}\ \neg\ \text{droite}}{\neg B \to \neg A, A \vdash B}\ \to\ \text{gauche}}{\neg B \to \neg A \vdash A \to B}\ \to\ \text{droite}}{\vdash (\neg B \to \neg A) \to (A \to B)}\ \to\ \text{droite}$$

7.1.2 La logique linéaire

Une introduction non formelle

Le leitmotiv de la logique linéaire est de pouvoir raisonner non pas uniquement sur des vérités absolues, mais aussi sur des systèmes dynamiques, tel un système chimique ou un système informatique. Pour cela, elle introduit divers nouveaux opérateurs permettant de différencier les différents cas d'utilisation des ressources.

Implication linéaire L'opérateur $\multimap$ représente l'implication linéaire. Les ressources utilisées à gauche de l'opérande sont consommées pour produire les ressources se retrouvant à droite. Par exemple, la boulangère nous vend une baguette B contre un euro E. Cela peut s'exprimer en logique linéaire par $E \multimap B$, qui exprime que l'on peut dépenser un euro pour avoir une baguette. À la fin de l'opération, on a bien une baguette, mais plus l'euro utilisé. On introduit l'opérateur ! pour dénoter qu'une ressource peut être répliquée autant de fois que nécessaire. Cela permet ainsi de raisonner sur des faits, des vérités qui ne s'épuisent eux pas, et récupérer ainsi la sémantique de la logique classique. On a ainsi l'équivalence $(!A) \multimap B \equiv A \to B$.

Les deux conjonctions La logique linéaire introduit deux opérateurs de conjonctions $\otimes$ et $\&$ représentant différents utilisations de la notion de "et". L'utilisation de $\otimes$ signifie que les deux actions vont avoir lieu tandis que l'utilisation de $\&$ dénote qu'une seule des deux actions va être faite (mais nous pouvons décider laquelle). Si nous continuons la métaphore de la boulangerie, et qu'on considère un croissant au beurre C coûte un euro, on a $E \multimap B \,\&\, C$ signifiant qu'avec un euro, nous pouvons choisir entre une baguette et un croissant. Au contraire, l'expression $E \multimap B \otimes C$ n'a pas de sens réel : on ne peut pas avec un euro avoir quelque chose qui en coûte deux. Par contre, la formule $E \otimes E \multimap B \otimes C$ a du sens, nous pouvons avoir une baguette et un croissant pour 2€.

Les deux disjonctions La sémantique des deux disjonctions est peu moins évidente. L'opérateur $\oplus$ est le dual de l'opérateur $\&$. Il dénote lui aussi qu'une seule de deux actions va être exécutée, mais que le choix de l'action a exécuter ne dépend pas de nous. Par exemple, une machine nous donnant aléatoirement une baguette ou un croissant lorsqu'on lui donne un euro pourrait s'exprimer $E \multimap B \oplus C$. Enfin l'opérateur $\parr$ exprime la dépendance entre deux actions : on verra plus tard qu'il a un lien très fort avec l'implication linéaire.

La négation linéaire La négation en logique linéaire est notée $A^{\perp}$. Elle a un sens beaucoup plus général qu'en logique classique : elle représente la dualité de manière générale. Cela peut être aussi bien la dualité vrai / faux, que action / réaction, entrée / sortie ...

La vision formelle

Plus formellement, les formules de la logique linéaire peuvent être exprimées grâce à la grammaire suivante :

$$\begin{aligned} A, B ::= & X \mid A \otimes B \mid A \oplus B \mid !A \mid \\ & X^{\perp} \mid A \parr B \mid A \,\&\, B \mid ?A \mid \\ & 1 \mid 0 \mid \forall x A \mid \exists x A \end{aligned}$$

De manière similaire à la logique classique, l'implication linéaire peut être exprimée à l'aide des autres connecteurs grâce à $A \multimap B \equiv A^{\perp} \parr B$. Les lois de Morgan peuvent être généralisées pour la logique linéaire de la manière suivante :

$$\begin{aligned} 1^{\perp} &\equiv \perp & \qquad \perp^{\perp} &\equiv 1 \\ \top^{\perp} &\equiv 0 & 0^{\perp} &\equiv \top \\ (p)^{\perp} &\equiv p^{\perp} & (p^{\perp})^{\perp} &\equiv p \\ (A \otimes B)^{\perp} &\equiv A^{\perp} \parr B^{\perp} & (A \parr B)^{\perp} &\equiv A^{\perp} \otimes B^{\perp} \\ (A \,\&\, B)^{\perp} &\equiv A^{\perp} \oplus B^{\perp} & (A \oplus B)^{\perp} &\equiv A^{\perp} \,\&\, B^{\perp} \\ (!A)^{\perp} &\equiv ?A^{\perp} & (?A)^{\perp} &\equiv !A^{\perp} \\ (\forall x A)^{\perp} &\equiv \exists x A^{\perp} & (\exists x A)^{\perp} &\equiv \forall x A^{\perp} \end{aligned}$$

On peut alors définir la logique linéaire par sa grammaire et les règles suivantes (système LL). Pour éviter d'avoir trop de règles, nous utilisons la possibilité de symétrie gauche / droite des séquents en combinaison avec les règles de Morgan afin d'obtenir des séquents "sans hypothèses" ou "tout à droite". On obtient alors le système suivant :

$$\frac{}{\vdash A, A^{\perp}} \text{ identité} \qquad \frac{\vdash \Gamma, A \qquad \vdash A^{\perp}, \Delta}{\vdash \Gamma, \Delta} \text{ coupure}$$

$$\frac{\Gamma}{\Gamma'} \text{ Échange : } \Gamma' \text{ est une permutation de } \Gamma$$

$$\frac{}{\vdash 1} \text{ un} \qquad \frac{\vdash \Gamma}{\vdash \Gamma, \perp} \text{ faux}$$

$$\frac{\vdash \Gamma, A \qquad \vdash B, \Delta}{\vdash \Gamma, A \otimes B, \Delta} \text{ fois} \qquad \frac{\vdash \Gamma, A, B}{\vdash \Gamma, A \parr B} \text{ par}$$

$$\frac{}{\vdash \Gamma, \top} \text{ vrai} \qquad \text{(Pas de règles pour 0)}$$

$$\frac{\vdash \Gamma, A \qquad \vdash \Gamma, B}{\vdash \Gamma, A \,\&\, B} \text{ avec} \qquad \frac{\vdash \Gamma, A}{\vdash \Gamma, A \oplus B} \text{ plus gauche} \qquad \frac{\vdash \Gamma, B}{\vdash \Gamma, A \oplus B} \text{ plus droite}$$

$$\frac{\vdash ?\Gamma, A}{\vdash ?\Gamma, !A} \text{ Bien sûr} \qquad \frac{\vdash \Gamma}{\vdash \Gamma, ?A} \text{ Affaiblissement}$$

$$\frac{\vdash \Gamma, A}{\vdash \Gamma, ?A} \text{ déreliction} \qquad \frac{\vdash \Gamma, ?A, ?A}{\vdash \Gamma, ?A} \text{ contraction}$$

$$\frac{\vdash \Gamma, A}{\vdash \Gamma, \forall x A} \text{ pour tous, x non libre dans } \Gamma \qquad \frac{\vdash \Gamma, A[t/x]}{\vdash \Gamma, \exists x A} \text{ il existe}$$

En analysant ces règles, on peut retrouver les dualités exprimées dans la sous-section précédente. Concernant les conjonctions, on peut noter que $\otimes$ utilise des contextes différents, tandis qu'au contraire & travaille deux fois avec le même contexte. De même, $\oplus$ et $\parr$ sont très différents dans leurs prémisses : $\oplus$ nécessite n'importe laquelle de ces actions en prémisses alors que $\parr$ nécessite explicitement les deux. Une autre observation importante est le fait qu'on retrouve des règles d'affaiblissement et de contraction, mais uniquement dans des contextes exponentiels, *i.e.* des contextes qui correspondent à des ressources infinies. On se ramène ainsi au cas de la logique classique.

7.1.3 Logique linéaire pour l'exécution concurrente

Nous allons maintenant essayer de voir comment faire le parallèle entre la logique linéaire telle que présentée précédemment et l'exécution de processus concurrents. On utilise pour cela le paradigme "process-as-formula" : les processus sont représentés par des formules logiques, et l'exécution correspond alors à la déduction logique, via les règles exprimées précédemment. Nous allons maintenant décrire de manière plus précise ce paradigme, dans le contexte qui nous intéresse.

Définition des processus

Pour définir nos processus, nous utilisons uniquement un fragment de la logique linéaire appelée LV^{obs} proposé par (Cervesato and Scedrov, 2009) (c'est le même fragment qu'on retrouve dans CLF (Watkins et al., 2004)). Il est défini de la façon suivante :

$$A, B ::= A \multimap B \mid A \otimes B \mid A \,\&\, B \mid !A \mid 0 \mid \forall x A \mid \exists x A$$

En terme de processus, $A \multimap B$ indique qu'un processus A se transforme en B. $A \otimes B$ correspond à l'exécution de deux processus en parallèle. $A \,\&\, B$ indique un choix : uniquement un des deux processus va être exécuté. Le processus 0 correspond à l'inaction. Enfin $\exists x A$ crée une nouvelle variable puis se transforme en A. Souvent, on utilisera $\exists x : K\ A$ où K est le type de la variable : cela permet de clarifier les utilisations ultérieures de cette variable.

L'exécution vue comme une déduction logique

L'exécution correspond à la déduction logique. On place la configuration de base dans le séquent racine, puis on applique une règle logique à chaque fois, de la même façon que quand on essaye de prouver un séquent. Évidemment, dans ce cadre là, il est possible que la preuve ne termine pas, *i.e.* qu'on ait un arbre de séquent infini : cela correspond au cas des exécutions infinies. Considérons les axiomes suivants :

1. $A \multimap B$
2. $B \multimap C \otimes D$
3. $C \multimap \mathit{print}$
4. $D \multimap A \,\&\, C$

On peut alors considérer le système initial crée par un contexte général (la mémoire, l'ensemble des règles) Γ, et le processus A. Le séquent associé est alors $\vdash ?\Gamma, A$. On peut alors avoir l'exécution suivante :

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{!\Gamma, \mathit{print}, A \vdash \Delta}}{!\Gamma, \mathit{print}, A \,\&\, C \vdash \Delta}\,\&\text{ gauche 1}}{!\Gamma, C, A \,\&\, C \vdash \Delta}\,(3)}{!\Gamma, C, D \vdash \Delta}\,(4)}{!\Gamma, C \otimes D \vdash \Delta}\,\otimes\text{ gauche}}{!\Gamma, B \vdash \Delta}\,(2)}{!\Gamma, A \vdash \Delta}\,(1)$$

Plusieurs remarques peuvent être faites sur cette arbre d'exécution. L'opérateur $\otimes$ correspond bien à l'exécution parallèle. Il crée deux hypothèses linéaires distinctes qui seront ensuite explorées de manière indépendante. L'ordre d'exploration dans la preuve est donc a priori inconnu, de même que l'ordre d'exécution de deux fils d'exécutions. Dans ce cas, il est possible qu'on explore d'abord l'hypothèse C puis l'hypothèse D ou inversement.

Pour l'opérateur &, on peut voir qu'on doit explicitement choisir une des deux hypothèses. Ici, on a choisi l'hypothèse à gauche. Le choix de l'hypothèse sélectionnée dépend de l'environnement $!\Gamma$ et n'est donc pas aléatoire (bien que non explicité ici). Il permet ainsi d'exprimer la notion de choix.

Nous utilisons ici les règles à gauche du séquent. Ce sont donc le dual des règles à droite explicités précédemment, via les règles de Morgan. Si par exemple, on considère le contexte $A \otimes B \vdash \Delta$, on peut passer les arguments à droite en prenant leur négation soit le séquent $\vdash \Delta, A^{\perp} \parr B^{\perp}$. On applique ensuite la règle à droite pour $\parr$ soit $\vdash \Delta, A^{\perp}, B^{\perp}$ puis on repasse les arguments à gauche $A, B \vdash \Delta$.

7.2 Schémas généraux de la modélisation

Après cette introduction à la logique linéaire et son utilisation pour décrire l'exécution concurrente, nous allons maintenant nous intéresser plus précisément à la façon dont nous pouvons modéliser ROAR en utilisant ces principes. Nous commençons d'abord par la modélisation de différents principes calculatoires de base que nous réutiliserons par la suite.

7.2.1 Passage par destination

Pour encoder la sémantique opérationnelle de ROAR, nous allons utiliser une représentation par destination, proposée entre autre par (Cervesato et al., 2003) pour encoder la sémantique du langage ML (Milner, 1997). Cette représentation permet une représentation plus modulaire et moins complexe que les approches classiques par continuation ou contexte d'évaluation. Concrètement, pour chaque calcul, on va créer une "destination", *i.e.* une variable unique partagée par le processus calculant et le processus demandant qui servira de passage entre ces deux contextes.

Plus formellement, on commence par introduire pour tout type T, $val\ T$ qui représente une valeur de type T, $exp\ T$ une expression retournant un type T, et $dest\ T$ une destination de type T. On peut alors introduire les deux formules $eval\ (E\ D)$ qui évalue l'expression E de type $exp\ T$ dans D de type $dest\ T$ et $return\ (V\ D)$ qui retourne la valeur V de type $val\ T$ depuis une destination $dest\ T$. De manière informelle, sans considérer d'effets spécifiques, si l'on a l'hypothèse linéaire $eval\ (E\ D)$, alors il existe une hypothèse linéaire $return\ (V\ D)$ si et seulement si l'évaluation de E conduit au résultat V.

Nous allons montrer comment utiliser cette représentation pour évaluer des expressions plus complexes.

7.2.2 Exécution séquentielle

Commençons par l'exécution séquentielle, *i.e.* $y = f(x); z = g(y)$; Dans notre cadre logique, nous notons cela $llet\ F(\lambda x.G(x))$. L'évaluation d'une telle expression peut alors se faire de la façon suivante :

$$\begin{aligned} &eval\_llet : eval\ (llet\ F\ (\lambda x.Gx)\ D) \\ &\quad \multimap \exists d_1 : dest\ T.(eval\ (F\ d_1) \\ &\quad\quad \otimes (\forall V_1 : val\ T.return\ (V_1\ d_1) \multimap eval\ ((G\ V_1)\ D))). \end{aligned}$$

L'arbre d'exécution associé serait alors :

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{!\Gamma, return\ ((V_2)\ D)) \vdash \Delta}}{\vdots}}{!\Gamma, eval\ ((G\ V_1)\ D)) \vdash \Delta}\ (3)}{!\Gamma, return\ (V_1\ d_1), (return\ (V_1\ d_1) \multimap eval\ ((G\ V_1)\ D)) \vdash \Delta}\ \multimap\ (2)}{!\Gamma, return\ (V_1\ d_1), (\forall V.return\ (V\ d_1) \multimap eval\ ((G\ V)\ D)) \vdash \Delta}\ \forall}{\vdots}}{!\Gamma, eval\ (F\ d_1), (\forall V.return\ (V\ d_1) \multimap eval\ ((G\ V)\ D)) \vdash \Delta}\ (1)}{!\Gamma, eval\ (F\ d_1) \otimes (\forall V.return\ (V\ d_1) \multimap eval\ ((G\ V)\ D))) \vdash \Delta}\ \otimes}{!\Gamma, \exists d.(eval\ (F\ d) \otimes (\forall V.return\ (V\ d) \multimap eval\ ((G\ V)\ D)) \vdash \Delta)}\ \exists}{!\Gamma, eval\ (llet\ F\ (\lambda x.Gx)\ D) \vdash \Delta}\ \multimap$$

Pour voir que cela correspond bien à une exécution séquentielle, il faut noter qu'on ne peut appliquer l'opérateur $\multimap$ en (2) uniquement que quand on a ajouté la formule $return\ (V_1\ d_1)$ dans

le contexte linéaire (ce qui est réalisé en (1)). Ainsi, l'hypothèse de droite ne peut être explorée qu'après l'hypothèse de gauche, *i.e.* l'exécution de G doit attendre que l'exécution de F soit terminée. L'autre point intéressant est que l'utilisation de l'opérateur $\multimap$ consomme la formule *return* $(V_1\ d_1)$ qui ne pourra donc plus être réutilisée.

7.2.3 Les entiers naturels et autres structures de base

Une telle modélisation n'est pour l'instant pas tellement utile, étant donné qu'on peut représenter des processus, mais pas particulièrement de données. Nous allons maintenant expliquer comment on peut représenter les entiers naturels. On définit le type *nat*, et deux formules pour le construire, z pour zéro et s comme successeur. On peut ainsi noter le chiffre 4 comme $(s\ (s\ (s\ (s\ z))))$. On peut alors implémenter les différents algorithmes classiques, sur les entiers. Par exemple, l'addition pourrait s'écrire ainsi :

$$\begin{aligned} &eval\_add : eval\ ((add\ M\ N)\ D) \\ &\quad \multimap \exists d : dest\ nat.(eval\ (M\ d) \\ &\quad\quad \otimes (return\ (z\ d) \multimap return\ (N\ D) \\ &\quad\quad\quad \& (\forall M' : val\ nat.return\ ((s\ M')\ d).\ \multimap eval\ (add\ M'\ s(N)\ D)))). \end{aligned}$$

Évaluons le début d'arbre d'exécution associé pour l'addition pour M = 1 et N = 2.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{!\Gamma, eval\ ((add\ z\ s(N)\ D)) \vdash \Delta}}{!\Gamma, return\ (s\ z\ d_1), (return\ (s\ z\ d_1) \multimap eval\ ((add\ z\ s(N)\ D)) \vdash \Delta}\ \multimap}{!\Gamma, return\ (s\ z\ d_1), (\forall M'.return\ (s\ M'\ d_1) \multimap eval\ ((add\ M'\ s(N)\ D)) \vdash \Delta}\ \forall}{!\Gamma, return\ (s\ z\ d_1), (return\ (z\ d_1) \multimap return\ (N\ D)) \,\&\, (\forall M'.return\ (s\ M'\ d_1) \multimap eval\ ((add\ M'\ s(N)\ D)) \vdash \Delta}\ \&}{!\Gamma, eval\ (F\ d_1), (return\ (z\ d_1) \multimap return\ (N\ D)) \,\&\, (\forall M'.return\ (s\ M'\ d_1) \multimap eval\ ((add\ M'\ s(N)\ D)) \vdash \Delta}}{!\Gamma, eval\ (F\ d_1) \otimes (return\ (z\ d_1) \multimap return\ (N\ D)) \,\&\, (\forall M'.return\ (s\ M'\ d_1) \multimap eval\ ((add\ M'\ s(N)\ D)) \vdash \Delta}\ \otimes}{!\Gamma, \exists d.(eval\ (F\ d) \otimes (return\ (z\ d) \multimap return\ (N\ D)) \,\&\, (\forall M'.return\ (s\ M'\ d) \multimap eval\ ((add\ M'\ s(N)\ D))) \vdash \Delta}\ \exists}{!\Gamma, eval\ (add\ M\ N))\ D) \vdash \Delta}\ \multimap$$

On a donc à faire à une règle récursive. L'évaluation que l'on a présentée s'arrête à l'appel récursif avec M' = 0 et N' = 3. L'évaluation suivante utilise alors le séquent $\vdash$ *return* $(z\ d) \multimap$ *return* $(N\ D)$, qui termine l'évaluation. L'utilisation de l'opérateur & dénote bien du choix : on explore l'un ou l'autre des séquents en fonction de la valeur contenue dans la destination d_1.

On peut construire de la même façon les différents opérateurs sur les entiers, par exemple l'opérateur <= ci-dessous (on suppose l'existence d'un type bool avec les formules de base True et False).. Dans la suite, on utilisera donc les notions classiques d'entiers et d'opérateurs pour ne pas surcharger les modélisations.

$$
\begin{aligned}
eval\_less\_eq : eval\ ((less\_eq\ M\ N)\ D) \\
\multimap \exists d_1 : dest\ nat.eval\ (M\ d_1) \\
\otimes (return\ (z\ d_1) \multimap return\ (True\ D) \\
\& (\forall V_1 : val\ nat.\ return\ ((s\ V_1)\ d_1) \\
\multimap \exists d_2 : dest\ nat.\ eval\ (N\ d_2) \\
\otimes (return\ (z\ d_2) \multimap return\ (False\ D) \\
\& (\forall V_2 : val\ nat.\ return\ ((s\ V_2)\ d_2) \\
\multimap eval\ ((less\_eq\ V_1\ V_2)\ D)))).
\end{aligned}
$$

De manière plus générale, on peut définir toutes les structures inductives, en particulier les listes, les chaînes de caractères, ... La aussi, si nécessaire, on utilisera les notations standards pour ne pas alourdir la présentation des modèles. En particulier, on utilisera *nil* pour dénoter de la liste vide et *cons* pour le constructeur de la liste : la liste [1, 2, 3] sera donc notée (*cons* 1 (*cons* 2 (*cons* 3 *nil*))).

7.2.4 Références

Chaque agent contient un certain nombres de variables (son contexte) représentant son état. Les valeurs de ces variables changent au cours de temps, ce qui est généralement difficile à modéliser en logique classique. Traditionnellement, on utilise la notion de "référence" pour désigner ces variables mutables. La modélisation que nous en faisons passe par l'utilisation de cellules, *i.e.* des cases mémoires qui servent à stocker la valeur de la référence. Ainsi, la variable cellule reste constante, mais le contenu associé à cette cellule change (de la même façon qu'un pointeur).

Plus formellement, on définit les nouveaux types $ref\ T$ qui désigne une référence sur une variable de type T, et $cell\ T$ une cellule qui contient une donnée de type T. On pose alors les formules suivantes :

- $contains\ (C\ V)$ signifie que la cellule C contient la valeur V.
- $cell\ C$ crée une référence à partir d'une cellule.
- $newref\ E$ crée une référence et lui associe le résultat de l'évaluation de E.
- $assign\ (E_1\ E_2)$ modifie la valeur de la référence représentée par l'expression E_1 par le résultat de l'évaluation de E_2.
- $deref\ E$ retourne la valeur référencée par E (ou plutôt par son évaluation).

$$
\begin{aligned}
eval\_newref &: eval\ ((\text{newref}\ E_1)\ D) \\
&\multimap \exists d_1 : dest\ T.eval\ (E_1\ d_1) \\
&\quad \otimes (\forall V_1 : val\ T.return\ (V_1\ d_1) \\
&\quad\quad \multimap \exists c : dest\ T.contains\ (c\ V_1) \otimes return\ ((cell\ c)\ D)). \\
eval\_assign &: eval\ ((assign\ (E_1\ E_2))\ D) \\
&\multimap \exists d_1 : dest(ref\ T).eval\ (E_1\ d_1) \\
&\quad \otimes (\forall C_1 : dest\ T.return\ ((cell\ C_1)\ d1) \\
&\quad\quad \multimap \exists d_2 : dest\ T.eval\ (E_2\ d_2) \\
&\quad\quad\quad \otimes (\forall V_2 : val\ T.return\ (V_2\ d_2) \\
&\quad\quad\quad\quad \multimap \forall V_1 : val\ T.contains\ (C_1\ V_1) \\
&\quad\quad\quad\quad\quad \multimap contains\ (C_1\ V_2) \otimes return\ ((void')\ D))). \\
eval\_deref &: eval\ ((deref\ E_1)\ D) \\
&\multimap \exists d_1 : dest\ T.eval\ (E_1\ d_1) \\
&\quad \otimes (\forall C_1 : dest\ T.return\ ((cell\ C_1)\ d1) \\
&\quad\quad \multimap \forall V_1 : val\ T.contains\ (C_1\ V_1) \\
&\quad\quad\quad \multimap contains\ (C_1\ V_1) \otimes return\ V_1\ D).
\end{aligned}
$$

L'encodage de *eval_newref* ne comporte rien de particulier. Dans un premier temps, on calcule la valeur de l'expression E_1 puis on crée une nouvelle cellule avec l'opérateur $\exists$. On ajoute alors les faits *contains* $(C\ v)$ et *return* $((cellc)\ D)$ et on termine l'évaluation. Le cas de *eval_assign* est un peu plus intéressant. On calcule d'abord la valeur de l'expression E_1 puis la référence représentée par la valeur E_2. Dans un second temps, on consomme grâce à l'opérateur $\multimap$ le fait *contains* $(C_1\ V_1)$ pour la remplacer par le fait *contains* $(C_1\ V_2)$. Ainsi, grâce à l'implication linéaire, on garantit qu'une cellule ne peut contenir qu'une seule et unique valeur au même moment. La modélisation de *eval_deref* utilise le même principe : on remet ici explicitement le fait *contains* $(C_1\ V_1)$ dans le contexte linéaire pour une réutilisation ultérieure de la référence.

7.2.5 Échanges de messages

Les différents agents communiquent via un ensemble de messages asynchrones. Pour modéliser cette communication, nous utilisons une représentation assez proche de celle des références. On identifie des canaux de communications de type *chan T* permettant d'envoyer des messages de type T. La formule *msg Ch M* signifie que le canal de communication *Ch* contient un message de valeur M (il est donc relativement analogue à *contains*). On définit alors les formules suivantes :

- *writeMsg* $(E_1\ E_2)$ prend en paramètre une expression E_1 représentant un canal de communication, et envoie le résultat de l'expression E_2 dans ce canal.
- *readMsg* E_1 prend en paramètre une expression E_1 représentant un canal de communication, et renvoie la valeur du message qu'il contient (en attendant jusqu'à ce qu'il se

déclenche).

Les formules $writeMsg'$ et $readMsg'$ sont des méthodes aides pour simplifier l'expression des deux formules précédentes, elles prennent en paramètre des valeurs plutôt que des expressions. L'utilisation de l'implication linéaire permet ici de consommer les messages, et donc d'assurer qu'ils ne seront traités qu'une fois (on peut évidemment le traiter plusieurs fois en le remettant explicitement dans le contexte linéaire).

$$
\begin{aligned}
eval\_writeMsg' &: eval\ ((writeMsg'\ (Ch\ M))\ D) \\
&\multimap (msg\ Ch\ M) \otimes return\ ((void')\ D) \\
eval\_writeMsg &: eval\ ((writeMsg\ (E_1\ E_2))\ D) \\
&\multimap eval\ (llet\ E_1\ (\lambda Ch \\
&\qquad llet\ E_2\ (\lambda V.(writeMsg'\ (Ch\ V))))))\ D. \\
eval\_readMsg' &: eval\ ((readMsg'\ Ch)\ D) \\
&\multimap \forall V : val\ T.\ msg\ Ch\ V \\
&\quad \multimap return\ (V\ D). \\
eval\_readMsg &: eval\ ((readMsg\ E_1)\ D) \\
&\multimap eval\ ((llet\ E_1\ (\lambda Ch.(readMsg'\ Ch))\ D).
\end{aligned}
$$

7.3 Modélisation de l'aspect logique

Nous allons maintenant nous intéresser à la modélisation de la couche logique décrite dans la section 4.2.3. Nous présentons ici les deux mécanismes les plus importants : la vérification qu'une tâche répond à une contrainte, et la génération d'un arbre de tâche.

7.3.1 Adéquation entre une tâche et une contrainte

Pour savoir si une tâche permet de satisfaire une certaine contrainte, l'agent essaye de résoudre la clause de Horn associée à la spécification de la tâche. Pour modéliser cette recherche de preuve, nous introduisons les types permettant de représenter des hypothèses *hyp* et des preuves *pv*. L'agent essaye de résoudre un système en logique classique, les hypothèses ne sont donc pas consommées. Pour éviter d'utiliser l'opérateur ! dans chaque expression, on utilise ici le symbole $A \rightarrow B$ pour indiquer $!A \multimap B$. La modélisation proposée est alors assez simple, et correspond à une méthode de résolution entrelacée de chaînage avant et de chaînage arrière (reflété ici par le sens de l'implication $\rightarrow$ ou $\leftarrow$). On se contente alors de modéliser les différentes règles de réduction de la logique classique.

$$
\begin{aligned}
hfalse &: hyp\ false \rightarrow \forall C : thm :!C \\
hand &: hyp(and\ A\ B) \rightarrow !A \otimes !B \\
hymp &: hyp\ A \rightarrow hyp\ (imp\ A\ B) \rightarrow !B \\
hall &: hyp(all\ A) \rightarrow \forall x : thm.!(hyp\ A\ !x) \\
hsome &: hyp(some\ A) \rightarrow \exists x : thm.!(hyp\ A\ !x) \\
proveand &: prove\ (and\ A\ B) \leftarrow prove\ A\ \&\ prove\ B \\
proveimp &: prove\ (imp\ A\ B) \leftarrow (hyp\ A \rightarrow prove\ B) \\
proveall &: prove\ (all\ A) \leftarrow (\forall x : thm.prove\ (A\ !x)) \\
provesome &: prove\ (some\ A) \leftarrow \exists x : thm.prove\ (A\ !x) \\
provetrue &: prove\ True. \\
proh &: prove\ A \leftarrow !(hypA).
\end{aligned}
$$

Pour limiter le nombre de pas dans la recherche de preuve (et donc limiter le temps de délibération), on peut utiliser un compteur qu'on va décrémenter après chaque règle. Si le compteur atteint 0, cela signifie qu'on a dépassé le temps alloué et on considère alors que la tâche ne permet pas, à priori, de satisfaire la contrainte. Pour modéliser une telle limitation, on va donc utiliser une référence sur un entier qu'on va décrémenter à chaque étape. Cette règle doit alors être entrelacée avec les règles précédentes si on veut prendre en compte un "temps de délibération" borné.

$$
\begin{aligned}
eval\_limit : eval\ ((limit\ R)\ D) & \\
\multimap \exists d_1 : nat.\ eval\ (deref R\ d_1) & \\
\otimes (return\ (z\ d_1) \multimap return\ (False\ D)) & \\
\&\ (\forall n\ :\ nat.return\ ((s\ n)\ d_1) & \\
\multimap \exists d_2 : dest\ void.eval\ ((assign\ (R\ n))\ d_2) \otimes return\ (True\ D))). &
\end{aligned}
$$

7.3.2 Arbres de tâches et exécution

Une fois que l'agent a sélectionné une tâche pour satisfaire la contrainte, il vérifie ses préconditions puis cherche à trouver des solutions pour les préconditions non vérifiées. On applique alors l'approche décrite précédemment pour sélectionner les tâches.

La génération de l'arbre se fait via un appel à deux fonctions mutuellement récursives. La méthode *task_exec* va exécuter les différentes préconditions P de la tâche T. Si l'ensemble P devient vide, *i.e.* que toutes les préconditions ont été vérifiées, on va alors exécuter réellement la tâche T via *task_exec'*, cette partie sera détaillée dans la section suivante. Si une des préconditions n'est pas valide, on appelle la méthode *condition_eval* qui va essayer de trouver une solution pour satisfaire la précondition. Si elle trouve une liste de tâches permettant de satisfaire la condition, elle l'exécute via *task_list_exec* (qui appelle *task_exec* sur les différents éléments

de la liste). Ces appels récursifs terminent pour la même raison que l'algorithme 1, *i.e.* l'ensemble des tâches disponibles pour faire la recherche diminue au fil des appels (via la méthode *available_tasks*), jusqu'à potentiellement devenir vide, et donc déclencher un échec (qui provoquera alors un retour en arrière dans l'algorithme de recherche de preuves sous-jacent).

$$
\begin{aligned}
&\mathit{eval\_condition\_eval} : \mathit{eval}\ ((\mathit{condition\_eval}\ E_1\ \mathit{Tasks}')\ D)\\
&\quad \multimap \exists d_1 : \mathit{dest}(\mathit{bool}).\mathit{eval}\ (E_1\ d_1)\\
&\quad\quad \otimes ((\mathit{return}\ (\mathit{True}\ d_1) \multimap \mathit{return}\ (\mathit{True}\ D))\\
&\quad\quad\quad \& (\mathit{return}\ (\mathit{False}\ d_1)\\
&\quad\quad\quad\quad \multimap \exists d_2 : \mathit{dest}(\mathit{list}(\mathit{task}, \mathit{pre})).\mathit{eval}\ (\mathit{search}\ E_1\ \mathit{Tasks}'\ d_2)\\
&\quad\quad\quad\quad\quad \otimes ((\mathit{return}\ (\mathit{nil}\ d_2) \multimap \mathit{return}\ (\mathit{False}\ D))\\
&\quad\quad\quad\quad\quad\quad \& (\mathit{return}\ ((\mathit{cons}\ E_3\ E_4\ ))\\
&\quad\quad\quad\quad\quad\quad\quad \multimap \mathit{eval}\ (\mathit{task\_list\_exec}(\mathit{cons}\ E_3\ E_4\ \mathit{Tasks}')\ D)))))\\
&\mathit{eval\_task\_exec} : \mathit{eval}\ ((\mathit{task\_exec}\ T\ P\ \mathit{Tasks})\ D)\\
&\quad \multimap \exists d_1 : \mathit{dest}(\mathit{list}\ T).\mathit{eval}\ (R\ d_1)\\
&\quad \otimes ((\mathit{return}\ (\mathit{nil}\ d_1) \multimap \mathit{eval}\ (\mathit{task\_exec}'\ T\ D))\\
&\quad\quad \& (\mathit{return}\ ((\mathit{cons}\ E_1\ E_2)\ d_1)\\
&\quad\quad\quad \multimap \exists d_2 : \mathit{dest}(\mathit{list}\ \mathit{tasks}).\mathit{eval}\ ((\mathit{available\_tasksTasks}\ T)\ d_2)\\
&\quad\quad\quad\quad \otimes \forall \mathit{Tasks}' : \mathit{list}\ \mathit{tasks}.\mathit{return}\ (\mathit{Tasks}'\ d_2)\\
&\quad\quad\quad\quad\quad \multimap \exists d_3 : \mathit{dest}(\mathit{bool}).\mathit{eval}\ ((\mathit{condition\_eval}\ E_1\ \mathit{Tasks}')\ d_3)\\
&\quad\quad\quad\quad\quad\quad \otimes ((\mathit{return}\ (\mathit{True}\ d_3) \multimap \mathit{eval}\ ((\mathit{task\_exec}\ E_2\ P)\ D))\\
&\quad\quad\quad\quad\quad\quad\quad \& (\mathit{return}\ (\mathit{False}\ d_3) \multimap \mathit{return}\ (\mathit{False}\ D)))))
\end{aligned}
$$

7.4 Modélisation de l'aspect exécution

Nous allons maintenant nous intéresser à la modélisation de la couche exécutive décrite dans la section 4.2.4, qui s'intéresse à savoir comment exécuter une tâche, *i.e.* comment modéliser *task_exec'*. Nous allons nous intéresser en particulier aux deux principaux mécanismes de cette couche exécutive, à avoir la sélection de la "meilleure" recette, puis son exécution.

7.4.1 Sélection d'une recette

Pour chaque tâche, on considère une liste de recettes et on va chercher à sélectionner la "meilleure". Dans un premier temps, on évalue si une recette est acceptable, *i.e.* si toutes ces préconditions sont respectées. Pour cela, on va utiliser la formule *pre* définie ci-dessous. Il s'agit d'une fonction récursive qui prend en paramètre une liste de recettes et un entier N qui servira d'accumulateur. On l'appellera alors en général avec la liste complète des recettes pour une tâche et $N = 0$. La modélisation est ici classique, on teste les deux cas possibles pour une liste

(vide ou avec un constructeur) et si elle est non vide, on évalue la précondition associée. Si elle échoue, on retourne faux, sinon on appelle récursivement la fonction *pre* avec la queue de la liste et N incrémenté. La fonction termine avec succès quand l'ensemble des préconditions ont été évaluées à vrai : dans ce cas, la liste des préconditions restants R est nulle, et N contient le nombre de préconditions validées.

$$
\begin{aligned}
eval\_pre : eval\ ((pre\ R\ N)\ D)\\
&\multimap \exists d_1 : dest(list\ T).eval\ (R\ d_1)\\
&\otimes (return\ (nil\ d_1) \multimap return\ ((false, N)\ D)\\
&\quad \& (return\ ((cons\ E_1\ E_2)\ d_1)\\
&\qquad \multimap \exists d_2 : dest(bool).eval\ (E_1\ d_2)\\
&\qquad\quad \otimes ((return\ (False\ d_2) \multimap eval\ ((pre\ E\_2\ s(N))\ D)\\
&\qquad\qquad) \& (return\ (True\ d_2) \multimap return\ ((False, N)\ D))))).
\end{aligned}
$$

Une fois qu'on a évalué quelques étaient les recettes acceptables, on va alors choisir la "meilleure". Nous présentons ici l'heuristique "mieux adaptée", *i.e.* celle dont le nombre de préconditions est le plus grand. D'autres heuristiques peuvent être implémentées de la même façon. La fonction *select* s'occupe de cette sélection. On utilise ici encore une approche récursive, traitant les différents cas de la liste vide et non vide. Dans le cas non vide, on regarde si la tête de la liste est meilleure que la recette courante, et on continue la récursion avec la queue de la liste et la meilleure des deux recettes.

$$
\begin{aligned}
eval\_select : eval\ ((select\ L\ R\ N)\ D)\\
&\multimap \exists d_1 : dest(list\ T).eval\ (L\ d_1)\\
&\otimes (return\ (nil\ d_1) \multimap return\ (R\ D)\\
&\quad \& (return\ ((cons\ E_1\ E_2)\ d_1) \multimap \exists d_2 : dest(U).eval\ (E_1\ d_2)\\
&\qquad \otimes ((return\ ((False, \_)\ d_2) \multimap eval\ ((select\ E_2\ R\ N))\ D)\\
&\qquad) \& (return\ ((True, K)\ d_2) \multimap \exists d_3 : dest(bool).eval\ ((less\_eq\ K\ N)\ d_3)\\
&\qquad\quad \otimes ((return\ (True\ d_3) \multimap eval\ ((select\ E_2\ R\ N)\ D))\\
&\qquad\qquad \& (return\ (False\ d_3) \multimap eval\ ((select\ E_2\ E_1\ K)\ D)))))).
\end{aligned}
$$

Les cas de gestion d'erreurs se traitent de manière analogue. La fonction de sélection change mais ne pose pas de difficulté particulière. L'ensemble des recettes disponibles changent (puisqu'on élimine celles qui ont échouées). Une des méthodes possibles pour le faire est d'avoir une référence sur la liste et de modifier la liste en sortie d'exécution. L'autre possibilité est de dire que l'exécution de la recette consomme linéairement la recette, et donc qu'elle disparait du contexte : elle ne sera alors plus disponible pour les exécutions suivantes.

7.4.2 Exécution d'une recette

Les recettes sont définies dans le langage d'exécution de ROAR. Pour modéliser l'exécution d'une recette, il nous donc construire la sémantique opérationnelle de chacune des primitives du langage d'exécution ROAR. Un des points importants à noter est l'interaction entre les différentes composants d'une même recette, lors de la réception d'un message des types `pause`, `continue`, `failure` et `abort`. Pour modéliser ceci, nous allons créer un canal de communication privé à la recette, qui sera partagé par les différents composants de la recette. Il permettra ainsi aux différents composants d'échanger des informations entre eux.

Primitives de manipulation de données

let permet de créer une nouvelle variable mutable et **set** de modifier ses variables (ainsi que les variables d'états). Cela coïncide directement avec la notion de référence présentée dans 7.2.4. **letname** substitue directement une expression par une autre, on ne lui donne donc pas d'équivalent logique, la substitution se faisant à la compilation.

$$eval\_let : eval\ ((let\ E_1\ E_2)\ D) \multimap eval\ (llet\ (newref\ E_1)\ (\lambda x.\ (assign\ (x\ E_2)\ D).$$
$$eval\_set : eval\ ((set\ E_1\ E_2)\ D) \multimap eval\ (assign\ (E_1\ E_2))\ D).$$

Primitives d'observations

La primitive **wait** permet d'attendre jusqu'à ce qu'une condition soit évaluée à vrai ou que la recette soit arrêtée. On peut alors utiliser la modélisation suivante, où E_1 est l'expression à évaluer et *Ch* le canal de communication de la recette. On va ainsi évaluer E_1. En cas de succès, le calcul se termine via *return* $(True\ D)$. Sinon on regarde le contenu du canal de communication. Dans le cas où la recette est toujours active (cas *true*), on va créer un nouveau processus *wait*, sinon on arrête le calcul via un retour à faux. On prend bien soin ici de remettre dans le contexte linéaire le message lu afin qu'il soit disponible pour les autres composants de la recette.

$$\begin{aligned}
eval_wait : eval\ ((wait\ E_1\ Ch)\ D)& \\
\multimap \exists d_1 : dest(Bool).\ eval\ (E_1\ d_1)& \\
\otimes ((return\ (True\ d_1) \multimap return\ (True\ D))& \\
\&\ (return\ (False\ d_1) \multimap& \\
((msg\ Ch\ True) \multimap (msg\ Ch\ True) \otimes eval\ ((wait\ E_1\ Ch)\ D)& \\
\&\ (msg\ Ch\ False) \multimap (msg\ Ch\ False) \otimes return\ (False\ D))).&
\end{aligned}$$

La primitive **assert** va elle vérifier pendant un certain temps qu'une expression est bien correcte. Elle pourra être annulée plus tard via la primitive **abort**. Pour échanger des informations entre ces deux composants, *assert* va retourner un canal de communication qu'il retournera pour pouvoir ensuite être annulé. Le reste de l'implémentation est trivial. Il vérifie que l'assertion est vérifiée sinon il émet un message $faux$ dans le canal de contrôle de la recette. Sinon, il vérifie

successivement son canal de contrôle et le canal de contrôle de la recette pour savoir si il doit continuer.

$$
\begin{aligned}
&eval\_assert' : eval\ ((assert\ E_1\ C\ Ch)\ D) \\
&\quad \multimap \exists d_1 : dest(Bool).\ eval\ (E_1\ d_1) \\
&\quad\quad \otimes ((return\ (False\ d_1)) \multimap (\forall V : bool.(msg\ Ch\ V) \multimap (msg\ Ch\ False)) \\
&\quad\quad\quad \& (return\ (True\ d_1)) \multimap \\
&\quad\quad\quad\quad ((msg\ C\ True) \multimap (msg\ C\ True) \\
&\quad\quad\quad\quad\quad \otimes ((msg\ Ch\ True) \multimap (msg\ Ch\ True) \otimes eval\ ((eval'\ E_1\ C\ Ch)\ D) \\
&\quad\quad\quad\quad\quad\quad \& (msg\ Ch\ False) \multimap (msg\ Ch\ False)) \\
&\quad\quad\quad\quad \& (msg\ C\ False) \multimap (msg\ C\ False))))). \\
&eval\_assert : eval\ ((assert\ E_1\ Ch)\ D) \\
&\quad \multimap \exists C : chan(Bool).(msg\ C\ True) \\
&\quad\quad \otimes \exists d_1 : dest(unit').eval\ ((assert'\ E_1\ C\ Ch)\ d_1) \\
&\quad\quad \otimes return\ (C\ D)
\end{aligned}
$$

Primitives de gestion des contraintes

Enfin, la partie la plus importante dans ROAR est l'ajout et la gestion des contraintes. Deux primitives importantes sont utilisées. La primitive **make** envoie une contrainte à un agent distant et attend qu'elle soit réalisée. L'agent distant est ici identifié comme un canal de communication spécifique C. Concrètement, le processus commence par créer un canal de réponse spécifique, puis envoie à l'agent distant représenté par le canal C le couple (contrainte, canal de réponse). Il attend alors sur ces deux canaux, d'un côté la réponse de l'agent, de l'autre la terminaison ou pas de la recette (fonction *has_abort*. Dans le cas où la recette est annulée, on envoie *Abort* à l'agent distant, qui annule la requête et provoque l'écriture d'un message sur le canal de réponse. Cela correspond à la modélisation suivante :

$$
\begin{aligned}
&eval\_has\_abort : eval\ ((has\_abort\ C\ Ch)\ D) \\
&\quad \multimap (msg\ Ch\ False) \\
&\quad\quad \multimap \exists d_1 : dest(void).eval\ ((writeMsg\ (Abort\ C))\ d_1) \otimes (msg\ Ch\ False)). \\
&eval\_make : eval\ ((make\ E_1\ C\ Ch)\ D) \\
&\quad \multimap \exists Ch_1 : Chan\ S.\ \exists d_1 :\ dest(void).eval\ (writeMsg\ (E1\ (C, Ch_1))\ d_1) \\
&\quad\quad \otimes (\forall V1 : void.return\ (V_1\ d_1) \multimap eval\ ((readMsg\ Ch_1)\ D) \\
&\quad\quad \otimes (\exists d_2 : dest(void).eval\ ((has\_abort\ C\ Ch)\ d_2))
\end{aligned}
$$

La primitive **ensure** vérifie elle qu'une contrainte est respectée au cours du temps. De la même façon que la modélisation de **assert**, elle va créer puis renvoyer un canal de communication Ch_2 permettant un arrêt futur via **abort**. La première phase est relativement similaire à la

modélisation de **make**, *i.e.* création d'un canal de réponse Ch_1 puis envoi de la paire (contrainte, canal de réponse). Ensuite on vérifie les réponses sur le canal de réponse (qui doivent être toujours vraies), et que ni la recette n'est pas terminée (via le canal de communication Ch), ni la primitive **ensure** n'a été annulée (via le canal de communication Ch_2).

$$
\begin{aligned}
&\mathit{eval\_answer} : \mathit{eval}\ ((\mathit{answer}\ \mathit{Ch}_1\ \mathit{Ch})\ D) \\
&\qquad \multimap \exists d_1 : \mathit{destT}.\mathit{eval}\ (\mathit{readMsg}\ \mathit{Ch}_1\ d_1) \\
&\qquad\quad \otimes ((\mathit{return}\ (\mathit{True}\ d_1) \multimap \mathit{eval}\ ((\mathit{answer}\ \mathit{Ch}_1\ \mathit{Ch})\ D)) \\
&\qquad\quad\ \ \& (\mathit{return}\ (\mathit{False}\ d_1) \multimap \mathit{eval}\ ((\mathit{writeMsg}\ (\mathit{Abort}\ \mathit{Ch}))\ D))) \\
&\mathit{eval\_ensure} : \mathit{eval}\ ((\mathit{ensure}\ E_1\ C\ \mathit{Ch})\ D) \\
&\qquad \multimap \exists \mathit{Ch}_1 : \mathit{Chan}\ S.\ \exists \mathit{Ch}_2 : \mathit{Chan}\ \mathit{Bool}.\exists d_1 :\ \mathit{dest}(\mathit{void}). \\
&\qquad\quad \mathit{eval}\ (\mathit{writeMsg}\ (E1\ (C, \mathit{Ch}_1))\ d_1) \\
&\qquad\quad \otimes (\forall V1 : \mathit{void}.\mathit{return}\ (V_1\ d_1) \\
&\qquad\quad \multimap (\exists d_2 : \mathit{dest}(\mathit{void}).\mathit{eval}\ ((\mathit{ensureAnswer}\ \mathit{Ch}_1\ \mathit{Ch})\ d_2)) \\
&\qquad\quad\ \ \otimes (\exists d_3 : \mathit{dest}(\mathit{void}).\mathit{eval}\ ((\mathit{has\_abort}\ C\ \mathit{Ch})\ d_3)) \\
&\qquad\quad\ \ \otimes (\exists d_4 : \mathit{dest}(\mathit{void}).\mathit{eval}\ ((\mathit{has\_abort}\ C\ \mathit{Ch}_2)\ d_4)) \\
&\qquad\quad\ \ \otimes \mathit{return}\ (\mathit{Ch}_2\ D)).
\end{aligned}
$$

On n'a pas ici géré explicitement les messages de type `pause` et `continue`. Cela ne pose pas de problème essentiel en soit, mais complique de manière importante les formules et leur lisibilité. Pour chaque primitive "longue", il faut considérer sur le canal de communication de la recette les messages de type `pause` et `continue`. Lors de la réception d'un message de `pause`, on arrête les récursions en cours tant qu'on a pas obtenu un message de type `continue` ou l'annulation de la recette. Cela rajoute donc des cas supplémentaires à traiter lors de la récursion, mais en aucun cas de difficulté particulière du point de vue sémantique.

Enfin, la dernière primitive intéressante de ROAR est la primitive **abort**. Elle prend un canal de communication en paramètre, et se contente d'écrire que le calcul doit être arrêté. Cela donne donc la règle triviale suivante :

$$\mathit{eval\_abort} : \mathit{eval}\ ((\mathit{abort}\ E_1)\ D) \multimap \mathit{eval}\ (\mathit{writeMsg}\ (E_1\ \mathit{Abort}))\ D).$$

Synthèse

L'évaluation d'une recette correspond à la création initiale d'un canal de contrôle, puis à l'exécution séquentielle des différentes primitives qui la compose. Comme nous avons modélisé de manière fine le comportement des différentes primitives, nous pouvons modéliser l'évaluation de n'importe quelle recette (modulo les appels à la couche fonctionnelle).

7.5 Discussion

7.5.1 Résumé

Dans ce chapitre, nous avons proposé une modélisation de l'architecture ROAR en utilisant un fragment de la logique linéaire. Ce fragment est assez expressif pour encoder précisément la sémantique de notre architecture, aussi bien la partie logique que la partie exécution. La logique linéaire apporte ici deux capacités importantes : la possibilité d'exprimer la concurrence via l'opérateur $\otimes$ et la possibilité de consommer des "ressources" (via l'opérateur $\multimap$). Cette dernière possibilité permet ainsi facilement de modéliser la notion d'état (d'un système informatique) et ses évolutions, l'utilisation de messages, ...

7.5.2 Des choix dans la modélisation

Choix du fragment de logique linéaire Plusieurs choix ont été faits au cours de ces travaux. Tout d'abord, nous avons choisi de travailler avec un fragment assez restreint de la logique linéaire. Cela en réduit l'expressivité, *i.e.* l'ensemble de ce que nous pouvons modéliser, mais de l'autre côté, cela permet de réduire la complexité de raisonner (automatiquement) dans cette ensemble. Notre choix a été principalement guidé par les travaux de (Cervesato and Scedrov, 2009), qui montrent comment ce fragment permet de modéliser facilement différents paradigmes classiques d'exécutions concurrentes.

Autres opérateurs d'intérêt dans la logique linéaire Deux autres opérateurs de la logique linéaires pourraient être intéressants pour une modélisation d'un système robotique. Comme nous le disions dans l'introduction à la logique linéaire, l'opérateur de négation linéaire introduit la notion de dualité (émetteur / récepteur, observateur / acteur, ...). Dans ACL (Kobayashi and Yonezawa, 1993), les auteurs utilisent la négation linéaire pour noter la dualité entre processus qui envoient un message m et un processus qui attend un message $m^{\perp}$. Notre modélisation pourrait peut-être être simplifiée en utilisant ce type de techniques. L'autre opérateur intéressant est l'opérateur plus linéaire $\oplus$ qui dénote d'un choix indépendant de notre volonté, d'un indéterminisme dans l'observation. Si l'on veut considérer un modèle de l'environnement, ou le comportement de la couche fonctionnelle, cet opérateur semble indispensable pour refléter les différentes situations possibles, que nous ne pouvons contrôler. Enfin, il est à noter que notre architecture ne nécessite pas l'utilisation de cet opérateur : elle est donc aussi déterministe que possible. La seule cause d'indéterminisme est l'ordre d'exploration des séquents après l'utilisation de l'opérateur $\otimes$, qui correspond à l'indéterminisme lié à la politique d'exécution des différents fils d'exécutions dans le système "réel".

Passage par destination Cette méthodologie de modélisation nous a semblé pertinente par son côté modulaire. Chaque type de primitive peut être modélisé indépendamment des autres, puis réutilisé. En particulier, dans la section 7.2, nous avons proposé des modélisations pour différentes notions classiques des langages de programmation. Ces modélisations pourront servir

de base pour modéliser d'autres langages exécutifs comme TDL (Simmons and Apfelbaum, 1998), PRS (Ingrand et al., 1996) ou bien PLEXIL (Jonsson et al., 2006).

7.5.3 Fonctions et appels à la couche fonctionnelle

Dans la modélisation proposée, nous nous sommes concentrés sur les différentes primitives de ROAR. Toutefois, nous n'avons pas explicité le cas des fonctions non natives (déclarées dans les agents) et plus généralement des appels à la couche fonctionnelle. Pour les premières, il est possible de les modéliser dans le fragment proposé, celle-ci ne faisant que manipuler que des types de données relativement simples (pas de types sommes, pas de types récursifs) avec des opérations de base. Pour la seconde, c'est évidemment moins simple.

Une vision naïve de la couche fonctionnelle Naïvement, on peut considérer que, du point de vue de ROAR, l'appel à la couche fonctionnelle correspond uniquement à un envoi de message sur un canal de communication spécifique, puis l'attente de réception de la réponse (indéterministe du point de vue de la couche de contrôle). Autrement dit, un module de la couche fonctionnelle pourrait être modélisé ainsi :

$$P : readMsg(c) \multimap (writeMsg(c, success) \oplus writeMsg(c, failure))$$

Malheureusement, cette approche est souvent insuffisante. En effet, souvent, ce qui nous intéresse, ce n'est pas seulement le résultat du service, mais ce sont ses effets de bords, en particulier la mise à jour de ses ports, en fonction de l'environnement. Pour montrer certaines propriétés ou pour modéliser globalement le système robotique, il est donc nécessaire de modéliser la couche fonctionnelle, au moins à un niveau assez haut.

Chapitre 8

Vérification sur des modèles en logique linéaire : une prospective

Dans le chapitre précédent, nous avons proposé une modélisation de l'architecture ROAR. *Dans cette partie, nous allons discuter plus précisément de la manière d'utiliser ces modèles pour faire de la vérification. Toutefois, il s'agit surtout d'études préliminaires. Ce chapitre sera composé de deux parties principales, l'une évoquant le problème de l'atteignabilité et l'autre celui des propriétés de sûreté.*

8.1 Le problème de l'atteignabilité

8.1.1 Définition et intérêt pour ROAR

Définition

Le problème de l'atteignabilité est de savoir si il est possible d'atteindre une situation but G à partir d'une situation initiale A, et d'un ensemble de règles Γ. Cette question est de fait très proche de la question de la planification : l'unique différence est qu'à priori, d'une question d'atteignabilité, on attend une réponse "oui ou non", tandis que d'une question de planification, on attende un plan (si il existe une solution).

Utilisation pour ROAR

Tests au niveau logique La question de l'atteignabilité dans ROAR peut être une aide importante à la programmation d'un système robotique. Lors du développement d'un agent, le développeur peut vouloir vérifier si, dans une situation donnée, l'agent peut activer une tâche ou une recette.

Dans le cas contraire, cela indique probablement une erreur de logique. Le problème peut ensuite être étendue à une configuration de N agents, afin de vérifier l'impact de un ou plusieurs nouveaux agents sur le système global. Autrement dit, il s'agit d'un ensemble de tests (unitaires ou d'intégrations) mais au niveau de la spécification logique.

Comparés à des tests classiques On peut arguer que cette problématique pourrait aussi être résolue par une politique de tests classiques, directement sur la version exécutable des agents. Toutefois, cela s'avère généralement difficile à mettre en place. La situation initiale peut être difficile, voire impossible à mettre en place sur les agents "normaux". De plus, les agents "normaux" vont aller jusqu'au bout de l'exécution, *i.e.* appeler les composants de la couche fonctionnelle, ce qui peut avoir un surcoût important si l'on s'intéresse uniquement à la question de l'atteignabilité. Pour ces diverses raisons, l'approche logique pour ce problème semble plus adaptée.

Ensemble d'agents Une autre question intéressante que l'on peut traiter est de déterminer des ensembles d'agents minimaux pour réaliser une tâche. L'idée d'un développement modulaire est la possibilité d'avoir un ensemble de composants (ou ici d'agents) dans une grande bibliothèques et de choisir ceux qui nous intéressent pour une application donnée. En utilisant une méthode qui résout le problème de l'atteignabilité, on peut alors chercher automatique des ensembles d'agents qui permettent de répondre à une certaine tâche, et donc choisir automatiquement un ensemble d'agents répondant à une problématique donnée.

8.1.2 Le problème d'atteignabilité en logique linéaire

Le problème d'atteignabilité, tel que défini précédemment, correspond à la prouvabilité du séquent $\Gamma, A \vdash G$. De plus, de par la nature constructiviste de la logique linéaire, le problème d'atteignabilité en logique linéaire est équivalent au problème de planification associé, le plan étant alors la preuve de ce séquent. Un des avantages de la logique linéaire pour le domaine de la planification est qu'il évite le problème du cadre [1] ("frame problem"), car la consommation des ressources est au cœur du modèle.

Planification et recherche automatique de preuves en logique linéaire

Planification Plusieurs travaux se sont intéressés aux questions de planifications et de vérification automatiques pour la logique linéaire. Un des travaux pionner dans ce domaine est (Kanovich and Vauzeilles, 2001) : les auteurs étudient ici le problème de la planification au travers du prisme des clauses de Horn en logique linéaire. Même si ce fragment est en général indécidable, ils montrent que le problème de planification dans ce fragment l'est, et démontrent la complexité du problème pour plusieurs classes de robot. Dans (Küngas, 2002), les auteurs proposent un planificateur basé sur la logique linéaire qui utilise les liens avec les réseaux de Pétri

1. Le problème du cadre est un problème initialement formulé dans le domaine de l'intelligence artificielle. La question sous-jacente est de savoir comment exprimer en logique un domaine dynamique, sans expliciter spécifiquement les conditions qui ne sont pas affectés par une action.

et le problème de l'atteignabilité pour obtenir un algorithme efficace de résolution. Les travaux (Cresswell et al., 2000) se concentrent sur des structures de plan plus complexe, en particulier des plans récursifs. Ils utilisent alors le langage basé sur la logique linéaire Lolli (Hodas and Miller, 1991) pour résoudre ces systèmes (se ramenant de fait au problème de la prouvabilité). Dans (Dixon et al., 2006), les auteurs proposent une formalisation de la logique linéaire intuitionniste dans l'assistant de preuve générique Isabelle (Nipkow et al., 2002) et proposent différentes tactiques pour résoudre efficacement le problème de planification associée. L'utilisation d'Isabelle permet de garantir la correction du plan, mais aussi de réutiliser les différentes stratégies déjà disponibles, en particulier, la gestion de contraintes. Les travaux de (Kahramanoğulları, 2009) sont plus théoriques, mais font le lien entre logique linéaire, planification, et concurrence. Ils montrent comment extraire des plans partiellement ordonnés depuis un problème de planification en logique linéaire, afin de les exécuter en parallèle. En particulier, il fait le lien avec les systèmes de transition, un modèle classique de concurrence.

Recherche automatique de preuves De nombreux travaux ont été effectués afin de proposer une mécanisation efficace de la logique linéaire, que ce soit pour le développement de langage de programmation logique, que pour le développement des assistants de preuves. Un des résultats les plus important est la découverte du principe de focalisation (Andreoli, 1992), qui permet de supprimer certaines causes d'indéterminisme lors d'une recherche d'une preuve en arrière en logique linéaire classique. Un des problèmes importants dans la résolution automatique de preuve en logique linéaire est la gestion des ressources, à savoir comment décomposer correctement les séquents lors de l'application de la règle $\otimes$, et assurer que les assertions linéaires ne sont bien utilisées qu'une fois. Dans (Harland and Pym, 1997), les auteurs proposent plusieurs stratégies de consommation de ressource, en particulier l'utilisation d'une décomposition paresseuse (*i.e.* elle n'est effectuée qu'au moment où on en a réellement besoin). De plus, à chaque formule linéaire, ils associent un booléen reflétant si il est consommé ou non. Cela leur permet de vérifier que les ressources linéaires ne sont consommées qu'une fois, et de choisir à posteriori comment faire la décomposition du contexte dans le cas $\otimes$. Si on s'intéresse plus spécifiquement aux assistants de preuves, un de premiers travaux théoriques est (Mints, 1993) qui proposent principalement une résolution en avant. Cette approche sera implémentée plus tard dans un assistant de preuve pour la logique linéaire propositionnelle dans (Tammet, 1994). Une approche plus complète est proposée par (Chaudhuri and Pfenning, 2005) : il décrit un assistant de preuve pour la logique intuitionniste du premier ordre (qui inclut le segment de la logique linéaire que nous utilisons) en combinant des approches en avant et en arrière, et en utilisant le principe de focalisation de Andreoli.

8.1.3 Atteignabilité et la couche fonctionnelle

Maintenant que nous avons vu comment pouvait se traiter le problème de l'atteignabilité en logique linéaire, et fait une rapide revue des travaux associés, nous allons nous intéresser à son applicabilité dans notre contexte.

Simplification initiale du modèle

Pour traiter du problème de l'atteignabilité, le modèle proposé peut être un peu simplifié. En effet, dans ce contexte, nous ne sommes pas particulièrement intéressés par la gestion d'erreur : une erreur est ici considérée comme une branche invalide, logiquement fausse, et on cherchera juste à tester une autre solution. Toutes les branches de gestion d'erreurs peuvent être alors remplacés par $\perp$, ce qui amènera le raisonneur à essayer une autre branche.

Lien avec la couche fonctionnelle

La modélisation de la couche fonctionnelle n'est pas forcément nécessaire pour la problématique de l'accessibilité. Si on s'en tient localement au niveau tâche, il n'y a pas évidemment pas d'appels à la couche fonctionnelle. Dans le cas où on s'intéresse à un système complet d'agents, le problème est un peu plus complexe. On peut alors s'intéresser à différentes sous-classes de problèmes.

Configuration entièrement symbolique On peut s'intéresser au cas entièrement symbolique, *i.e.* on s'intéresse uniquement aux configurations des agents. On peut alors simplifier le modèle en considérant que les appels à la couche fonctionnelle terminent toujours correctement, que les primitives **wait** terminent, et que les assertions des primitives **assert** sont toujours vérifiées. Autrement dit, on se ramène à savoir si on peut trouver une solution à la projection des différentes contraintes émises par les différents agents.

Configuration mixte L'autre cas qui nous intéresse est de savoir si on peut atteindre une configuration hybride, *i.e.* généralement une ou plusieurs contraintes sur des valeurs de la couche fonctionnelle. Par exemple, on pourrait vouloir vérifier si la recette *go_to* permet réellement d'aller jusqu'à but. Pour montrer un tel résultat, il faut modéliser assez finement la couche fonctionnelle, en particulier, ses effets (*i.e.* que le module *p3d* génère un plan qui nous rapproche du but, et que le module *rflex* exécute ce plan, nous rapprochant réellement du but). De plus, dans ce cadre, il nous faut introduire explicitement l'opérateur $\oplus$ pour représenter l'indéterminisme vis à vis de l'environnement, et explorer les différentes branches qui en résulte.

8.1.4 Synthèse

Dans cette section, nous nous sommes donc intéressés au problème de l'atteignabilité. Nous avons vu qu'en logique linéaire, ce problème est directement lié à la prouvabilité (et à la planification). Ce lien entre déduction en logique linéaire et planification semble très intéressant pour le domaine de la robotique, et est une voie d'étude pour faire plus formellement le lien entre planification et exécution.

Plusieurs simplifications peuvent être faites dans le modèle si l'on s'intéresse uniquement à ce problème. Toutefois, si l'on veut s'intéresser à des configurations non entièrement symboliques, il nous faut un modèle fin de la couche fonctionnelle.

Enfin, même si le problème a été bien étudié dans la littérature, il ne semble pas exister réellement de raisonneurs robustes et efficaces pour la logique linéaire (ou le fragment qui nous intéresse). La question du passage à l'échelle est un réel problème, car le nombre de règles augmentent fortement avec le nombre d'agents (nouvelles variables d'états, nouvelles règles pour chaque tâche et recettes, ...), rendant de fait la résolution de plus en plus difficile.

8.2 Vérification de propriétés de sûreté

8.2.1 Définition

Les propriétés de sûreté sont des propriétés qui indiquent que quelque chose de non désiré ne va jamais arriver. Autrement dit, si P est une propriété de sûreté, on doit pouvoir montrer P dans toutes les branches d'exécutions possibles, *i.e.* que $\neg P$ n'est pas un état atteignable. Dans le cadre de la robotique, cela peut être des propriétés telles que "un robot ne doit jamais heurter un humain" ou "le robot ne doit jamais entrer dans l'eau".

8.2.2 Quelques approches possibles en logique linéaire

Sémantique de phases

Le problème étant exprimé ainsi, il n'existe pas d'approches évidentes pour le traiter en logique linéaire. En effet, cela revient à démontrer la non-existence de certaines dérivations, ce qui est impossible en général, en particulier quand l'espace des cas possibles devient grand. Dans (Fages et al., 2001), les auteurs en proposent toutefois une interprétation logique. Pour cela, ils utilisent la sémantique de phases (Girard, 1987) de la logique linéaire. La sémantique de phase est une sémantique de prouvabilité pour la logique linéaire. En utilisant cette notion de sémantique de phase, les auteurs montrent alors qu'on peut ramener ce problème à la démonstration de l'existence d'un contre-exemple dans l'espace des phases. Ces principes sont alors utilisés pour implémenter un "model checker" (Soliman, 2001) sur des spécifications en logique linéaire.

Une autre approche pour le "model checking"

Une autre approche pour vérifier des propriétés de sûreté a été proposé dans (Bozzano et al., 2004). En s'inspirant d'algorithmes d'atteignabilité arrière (*i.e.* calculer à partir d'un état but l'ensemble des états possibles sources), il propose une procédure d'évaluation de "bas en haut" (bottom-up) pour le fragment de la logique linéaire LO (Andreoli and Pareschi, 1991). Les auteurs peuvent alors de manière efficace calculer l'ensemble des buts atteignables à partir d'une spécification logique. À partir des états interdits (*i.e.* les propriétés de sûreté que l'on veut vérifier), il est alors possible de remonter aux états initiaux qui l'engendrent. Si aucun de ces états initiaux n'apparaissent dans l'ensemble des états initiaux légaux, alors la propriété de sûreté est garantie. Dans le cas contraire, l'approche offre une instance conduisant à la violation de la propriété.

Lien avec les autres modèles de concurrences

Enfin, une autre approche possible est d'utiliser la connexion entre certains fragments de la logique linéaire et d'autres paradigmes de concurrence. On peut alors réutiliser les techniques de vérification connues sur ces paradigmes. Dans (Collé et al., 2005), les auteurs développent des scénarios sous forme de spécification dans un fragment logique linéaire. Ils transforment ces spécifications en réseaux de Pétri (coloriés), et utilisent alors des techniques de vérification sur ces réseaux de Pétri. Dans (Cervesato and Scedrov, 2009), les auteurs font une analyse approfondie des liens entre le fragment de logique linéaire LV^{obs} (que nous avons utilisé dans notre formalisation) et divers autres formalismes concurrents (π-calcul, réseaux de Pétri, logique de réécriture). Ils font entre autre un parallèle avec un puissant système de réécriture ω. Celui-ci peut être encodé dans l'outil Maude (Clavel et al., 2002), qui permet entre autre de vérifier des propriétés de sûreté via des méthodes de "model checking".

8.2.3 Le problème de la couche fonctionnelle

Il apparait difficile de montrer des propriétés de sûreté de manière complètement indépendante de la couche fonctionnelle. En effet, ce qui nous intéresse, c'est de garantir le comportement du robot, comportement qui vient en dernier lieu de la couche fonctionnelle. Par exemple, supposons qu'on puisse "garantir" que le module *p3d* ne s'approche pas à plus de 50cm d'un obstacle, un humain étant considéré comme un obstacle. Si les seuls déplacements possibles du robot sont basés sur les plans générés par le *p3d* (ce qui se montre au niveau de la couche de contrôle, donc de ROAR) alors on peut montrer que le robot ne peut heurter un homme. C'est la combinaison entre les spécifications du module fonctionnel et les spécifications de coordination qui permet de montrer ce résultat. Cela nécessite donc une modélisation de la couche fonctionnelle, et en particulier des garanties qu'elle offre (ou en terme de contrat ses invariants). Seule la combinaison de garanties fonctionnelles et des garanties de coordination permet d'assurer des garanties au niveau du système global.

8.3 Synthèse

Dans cette partie, nous nous sommes intéressés à la vérification de propriétés sur des modèles basés sur la logique linéaire. Nous nous sommes penchés en particulier sur les questions d'atteignabilité et de sûreté. Si la première a une interprétation assez évidente en logique linéaire, la seconde pose plus de difficultés. Il existe toutefois plusieurs approches possibles pour aborder le problème.

Cette étude prospective nous a permis ainsi d'identifier les voies possibles pour faire de la vérification sur des spécifications en logique linéaire, mais aussi les problèmes à résoudre. En premier lieu, il apparait indispensable d'avoir un modèle des couches fonctionnelles pour prouver des propriétés "intéressantes" sur le système complet. En second lieu, il semble nécessaire de développer des outils plus robustes et plus efficaces pour traiter automatiquement de la logique

linéaire. La littérature sur le sujet est assez importante, mais il ne semble pas y avoir d'outils réellement efficaces. Enfin, il est probablement nécessaire de simplifier le modèle proposé pour ROAR afin de simplifier la tâche des raisonneurs automatiques.

Chapitre 9

Conclusion

9.1 Résumé des contributions

Dans ce document, nous nous sommes intéressés à la problématique des architectures de contrôle pour un robot autonome. Dans un premier temps, nous avons défini un ensemble d'exigences liées à une telle architecture, qui portent principalement sur une **gestion de temps de délibérations distincts**, une **gestion aisée de la concurrence**, la satisfaction de propriétés de **robustesse** et de **vérifiabilité**, et enfin la satisfaction de besoins de **modularité et composabilité**.

Sur la base de ces exigences et de l'analyse de l'état de l'art, nous avons proposé l'architecture ROAR basée sur des agents encapsulant chacun une ressource et échangeant des contraintes. Cette décomposition en agents permet de facilement gérer des contraintes différentes sur les temps de délibération et augmente la robustesse du système, en évitant un unique point de faute. Chaque agent utilise un moteur d'inférence basé sur la logique de premier ordre afin de décider comment satisfaire une contraire et si plusieurs contraintes sont satisfaisables en même temps. Un langage d'exécution a été défini, afin de décrire de manière aisée différentes stratégies de contrôle. Nous avons présenté les autres échanges entre les agents, permettant notamment de garantir qu'une contrainte n'est exécutée que si ses pré-requis sont satisfaits et de gérer les fautes d'exécutions ou les problèmes de concurrence.

Nous nous sommes ensuite intéressés à une implémentation pratique de cette architecture. En particulier, nous avons discuté des choix importants de l'implémentation, afin de garantir robustesse et performance. L'architecture a été instanciée dans une couche de supervision pour un robot terrestre, et est désormais systématiquement exploitée pour la réalisation de missions de navigation autonomes. Elle a été testée aussi bien en simulation que sur un robot réel. Diverses traces d'exécution ont illustré les différentes stratégies développées et ont été exploitées pour l'analyse des performances. Nous avons ainsi pu montrer que même si l'architecture proposée induit un léger surcoût au niveau atomique, le temps global de mission n'est pas impacté, et même moins que pour d'autres architectures.

Enfin, nous nous sommes intéressés à la formalisation de l'architecture de contrôle proposée. Après avoir rappelé les principaux besoins auxquels répond une telle formalisation, nous avons

analysé différents modèles logiques pertinents. Nous avons choisi la logique linéaire comme base de formalisation, en particulier parce qu'elle permet le mieux l'interaction continue entre déduction et exécution, et nous avons montré comment les différents concepts de ROAR pouvaient être encodés dans ce formalisme. Ce modèle est alors utilisé comme aide à la programmation pour vérifier hors-ligne que certaines situations sont atteignables.

9.2 Perspectives

Un des enjeux actuels est de continuer à valider l'architecture sur des problèmes plus complexes dans le domaine de l'autonomie du mouvement considéré dans cette thèse, et de la confronter à des problèmes de contrôle sur des domaines différents (par exemple la robotique de service et les interactions avec les utilisateurs). Cela conduira peut être à l'évolution de l'architecture ou de son implémentation pour prendre en compte des besoins nouveaux ou actuellement difficiles à exprimer avec ROAR. Au delà de ces évolutions, nous voyons quatre grandes problématiques à approfondir, en prolongement de ces travaux :

- Une meilleure exploitation de la définition de la couche fonctionnelle,
- L'intégration de planificateurs symboliques,
- L'extension de la modélisation et de méthodes de vérification,
- Et enfin l'extension des principes de ROAR aux systèmes multi-robots.

Les sections suivantes précisent ces perspectives.

9.2.1 Vers une intégration plus fine avec la couche fonctionnelle

Le but d'une architecture de contrôle est principalement de coordonner les différents composants de la couche fonctionnelle. Pour réaliser cette tâche, il est donc nécessaire de savoir ce que chacun de ces composants fait réellement. Malheureusement, à l'heure actuelle les différentes architectures fonctionnelles n'offrent que très peu d'information pour la couche de contrôle. Les services sont définis généralement par leurs entrées et leurs sorties et dans le meilleur des cas par quelques commentaires en langage naturel. Pour déterminer le comportement réel du composant, il est donc le plus souvent nécessaire d'en étudier le code et d'en déduire le comportement général. C'est évidemment une approche source d'erreurs, surtout quand la taille du code est imposante. De plus, l'approche n'est guère satisfaisante dans le cas d'un composant régulièrement modifié.

Classification Pour améliorer l'intégration entre architecture fonctionnelle et architecture de contrôle, il faut donc ajouter de l'information au niveau des différents composants fonctionnels. Un premier point consiste à classifier les différents composants, ou plus spécifiquement les informations qu'ils produisent. Pour cela, on pourrait utiliser une ontologie comme référence pour les différents composants et dériver de manière quasi automatique depuis cette ontologie une décomposition en agents.

Explicitation des comportements Une fois cette étape sémantique effectuée, il nous faut décrire de manière plus précise le comportement d'un composant et de ses services, et ceci dans un langage formel. Une des possibilités est d'ajouter un système de contrats aux différents services, explicitant les préconditions, postconditions et invariants de chaque service. Dans la couche fonctionnelle, ces contrats seraient interprétés classiquement à la Eiffel (Meyer, 1992), *i.e.* l'échec d'une précondition, d'une post-condition ou d'un invariant provoque l'échec du service. Ces contrats pourraient alors être utilisés de manière logique par une architecture de contrôle. Une autre piste intéressante est l'étude des langages de description d'architectures (ADL) tels ACME (Garlan et al., 2010), Wright (Allen et al., 1998), ou π-ADL (Oquendo, 2004) proposés par la communauté du génie logiciel. Ceux-ci permettent de définir de manière formelle les composants, les services et leurs connections au sein d'une architecture. Même si la partie configuration et reconfiguration dynamique est trop statique pour les besoins d'une architecture de contrôle de robot, les langages développés dans ces ADL permettent de décrire de manière riche le comportement des différents composants et services. Un autre intérêt de ces approches est qu'elles ont souvent été adossées à des approches formelles pour vérifier la consistance de l'architecture.

9.2.2 ROAR et planificateurs symboliques

Jusqu'à présent, ROAR a surtout été utilisé pour contrôler finement la couche fonctionnelle d'un robot. En particulier, il a été utilisé pour contrôler différents planificateurs d'itinéraires et de trajectoire, mais pas de planificateurs symboliques "classiques". Par planificateur symbolique, nous entendons ici un planificateur qui décompose une tâche de haut niveau en un ensemble de sous tâches de niveau moins élevé, mais qui restent de haut niveau. La spécificité de l'information produite (un plan symbolique) est qu'elle va être interprétée non pas par un composant dédié, mais par la couche de contrôle elle-même. Autrement dit, l'interprétation du plan généré va modifier la structure dynamique du graphe d'agents.

Sémantique de plans La question sous-jacente qui se pose est alors pour ROAR de comprendre le plan, et de le transformer en des concepts interprétables par ROAR. On se retrouve alors avec les mêmes problématiques que les architectures de gestion de plans telles que "Concurrent Reactive Plan" (Beetz, 2000) ou "Roby" (Joyeux et al., 2009), à savoir de convertir et manipuler différents types de plans en utilisant un modèle commun. Pour ce faire, nous pouvons nous appuyer sur la formalisation de notre couche exécutive pour montrer qu'un certain plan est équivalent à une certaine recette, et donc prouver que la transformation est correcte. L'autre difficulté qui apparaît est celle de la gestion des erreurs. Quelles types d'information doit-on retourner au planificateur pour l'aider à produire un plan qui évite le problème rencontré ? On retrouve ici toutes les problématiques de niveau et granularité d'informations entre les différents outils.

9.2.3 Modélisation et vérification

Une modélisation plus simple Dans le chapitre 7, nous avons proposé une formalisation de l'architecture ROAR. Même si la logique linéaire simplifie la formalisation des différents change-

ments d'états par rapport à la logique classique, la modélisation reste relativement complexe. Il serait intéressant d'étudier si on peut obtenir les mêmes comportements avec une modélisation logique plus directe, puis dans un second temps, essayer de comprendre sa signification au niveau robotique. Cela permettrait d'avoir d'un côté un modèle plus simple à analyser et de l'autre une architecture avec un plus petit nombre de concepts à appréhender.

Vérification de propriétés de sûreté Un point important que nous avons évoqué comme motivation pour la formalisation est la possibilité de vérifier des propriétés de sûreté, en particulier avec des approches de "model checking". Nous n'avons pas pu étudier plus en détail ce point, mais deux approches semblent possibles. D'un côté, on peut étendre les travaux de "model checking"sur des spécifications de logique linéaires proposés par (Bozzano et al., 2004) au fragment de logique linéaire qui nous intéresse. De l'autre, on peut essayer d'utiliser la traduction de la logique linéaire en système de transitions proposées par (Cervesato and Scedrov, 2009) pour ensuite utiliser les outils classiques de "model checking" pour ces systèmes.

Vérification et couche fonctionnelle L'une des difficultés importantes à traiter reste que certaines valeurs de la couche de contrôle reflètent des valeurs calculées par un module de la couche fonctionnelle, ou qu'un comportement dépend de l'effet de bord d'un service de la couche fonctionnelle. Sans modèle de la couche fonctionnelle, ces informations sont inobservables, hors il est difficile (voire impossible) de prouver certaines propriétés sans ces informations. Cela milite d'autant plus pour la modélisation du comportement des services évoquée au paragraphe 9.2.1.

9.2.4 Vers une architecture multi-robots

L'utilisation d'une flotte de robots plutôt qu'un robot unique apporte beaucoup dans de nombreux contextes opérationnels, pour améliorer le temps d'exécution d'une mission et ses chances de réussite. On peut donc se demander si il est possible de faire évoluer ROAR pour contrôler non seulement un robot mais une flotte de robots. Notons d'abord que ROAR a plusieurs atouts de base pour le contrôle multi-robots. Sa décomposition en agents-ressources permet de modéliser facilement l'hétérogénéité des différents robots, chaque robot déployant uniquement les agents correspondant à ses propres capacités. La décision se fait de manière distribuée, chaque agent n'ayant qu'une vue partielle du système. La modélisation décentralisée de ROAR passe bien à l'échelle multi-robots, d'autant plus qu'elle est basée sur des messages asynchrones.

Communication Ceci étant dit, il existe évidemment des difficultés pour passer d'un modèle mono-robot à un modèle multi-robots. L'une de ces difficultés est la communication. Au sein d'un robot, on peut faire l'hypothèse d'une communication sans problème de délai, perte de messages ou limites de bande passante : ce n'est évidemment pas le cas au sein d'un système multi-robots. Il est donc nécessaire d'introduire explicitement un agent gérant la communication au niveau de l'architecture (et pas simplement au niveau instanciation). Cet agent aura alors des contraintes à satisfaire lors de l'utilisation d'un agent distant (*i.e.* sur un autre robot), et pourra

en réponse contraindre d'autres agents (en particulier la position du robot, afin de garantir que la communication reste possible).

Choix des agents impliqués Une des autres questions qui se pose est de savoir quels agents choisir quand plusieurs agents sur différents robots correspondent aux spécifications d'une recette. Différentes approches sont possibles ici. Le choix peut se faire unilatéralement, au niveau de l'agent exécutant la recette : celui-ci décide arbitrairement, ou à partir d'une information haut niveau présente sur les différents agents, quels agents sur quels robots il va utiliser et essayer. En cas d'échec, il tentera une autre combinaison d'agents, et les contraintes automatiques sur les communications vont alors implicitement créer des coalitions. Cette extension serait dans l'esprit de la synthèse automatique de tâche de ASyMTRe (Tang and Parker, 2005). Comme il s'agit de problèmes d'affectation (ou d'allocation), une autre approche possible est d'utiliser des approches de négociations, comme par exemple CNP ("Contract Net Protocol") (Smith, 1980), pour décider entre ces agents celui qui est le mieux adapté pour résoudre actuellement une certaine contrainte, dans l'esprit de ASyMTRe-D (Tang, 2007). Bien que probablement plus efficace sur des cas complexes, cette approche soulève deux types de problèmes, qui devront être finement analysés. D'un côté, la décision de choisir un agent plutôt qu'un autre dépend aussi généralement des autres activités courantes du robot, et donc d'une information "globale" sur le système. De l'autre, cela a un impact non négligeable sur la formalisation et pourrait rendre certains résultats sur cette formalisation difficiles à transférer au cas multi-robots.

Annexes

Annexe A

Liste des publications

Publications en cours d'évaluation

Degroote, A. and Lacroix, S. (2012a). A control architecture for autonomous robots founded on resource agents. *Submitted to International Journal of Robotics Research*.

Echeverria, G., Lemaignan, S., Degroote, A., Lacroix, S., Karg, M., Koch, P., Lesire-Cabaniols, C., and Stinckwich, S. (2012). User Driven Improvements to Robotics Simulation in MORSE. In *Submitted to Simulation, Modeling, and Programming for Autonomous Robots*.

Publications dans des conférences internationales

Degroote, A. and Lacroix, S. (2011a). Roar : Resource oriented agent architecture for the autonomy of robots. In *International Conference on Robotics and Automation, Shangai (China)*.

Degroote, A. and Lacroix, S. (2011b). Roar : Resource oriented agent architecture for the autonomy of robots. In *9th International Conference on Practical Applications of Agents and Multi-Agent Systems (PAAMS), Salamanca (Spain)*.

Echeverria, G., Lassabe, N., Degroote, A., and Lemaignan, S. (2011a). Modular open robots simulation engine : Morse. In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pages 46 –51.

Boumghar, R., Roussillon, C., Degroote, A., Cox, P., Vandeportaele, V. D. B., Herrb, M., and Lacroix, S. (2011). Over the hill and far away : aerial/ground cooperation for long range navigation. In *International Workshop on Robotics for risky interventions and Environmental Surveillance-Maintenance, Leuwen (Belgique)*.

Publications dans des conférences nationales

Degroote, A. and Lacroix, S. (2012b). Roar : une architecture orientée agents pour l'autonomie des robots. In *18ème congrès francophone sur la Reconnaissance des Formes et l'Intelli-*

gence Artificielle, Lyon (France).

Degroote, A. and Lacroix, S. (2012c). Une architecture pour l'autonomie des robots basée sur des agents-ressources. In *6ème Conférence francophone sur les architectures logicielles, Montpellier (France)*.

Degroote, A. and Lacroix, S. (2012a). A resource-agents robot control architecture : design and formalization. In *7ème Conférence francophone sur les architectures logicielles, Nancy (France)*.

Degroote, A. and Lacroix, S. (2010). Structuring processes into abilities : an information-oriented architecture for autonomous robots. In *5th National Conference on Control Architectures of Robots Douai (France)*.

Annexe B

MORSE

MORSE ("Modular OpenRobots Simulator Engine") (Echeverria et al., 2011) est un simulateur libre orienté robotique initié au LAAS, au développement duquel nous avons beaucoup participé, et qui est maintenant co-développé par de nombreux universitaires [1]. Il est basé sur l'environnement de modélisation libre Blender [2] et son mode jeu reposant sur le moteur physique BULLET [3]. Cela permet de construire des simulations relativement réalistes dans des modèles d'environnements complexes (voir un exemple figure B.1). Blender, aussi bien la partie modélisation que la partie jeu, est entièrement programmable via une interface Python [4] : il est alors facile de créer des comportements de haut niveau ou de développer de nouveaux capteurs à partir des données exportées par Blender.

B.1 Un simulateur "Software In the Loop"

Une des idées fortes de MORSE est de permettre d'évaluer en simulation *exactement les mêmes logiciels* que ceux qui sont embarqués sur le robot réel, afin de réduire au maximum les différences entre architectures des robots simulés et architectures des robots réels. Comme nous l'avons vu dans le chapitre 2, il existe de nombreux cadres logiciels permettant d'encapsuler et d'échanger des données entre les composants de la couche fonctionnelle. Pour prendre en compte ces deux besoins (intégration directe d'une architecture fonctionnelle et considération les différents cadres logiciels disponibles), nous avons décomposé les composants MORSE en plusieurs parties. Une partie principale contient l'implémentation d'un actionneur ou d'un capteur, interagissant directement avec le moteur de jeu de Blender. Cette partie est indépendante de tout cadre logiciel : elle dépend uniquement de Blender. Elle expose son état exportable dans une variable *local_data* de type dictionnaire (ou tableau associatif). Une seconde partie, appelée "middleware" permet de transformer le contenu d'un composant dans le format spécifique pour un cadre logiciel donné. Ainsi, l'export d'un composant est indépendant de sa logique, et il est possible

1. il est disponible sur `http://morse.openrobots.org`
2. `http://www.blender.org/`
3. `http://bulletphysics.org/wordpress/`
4. `http://www.python.org`

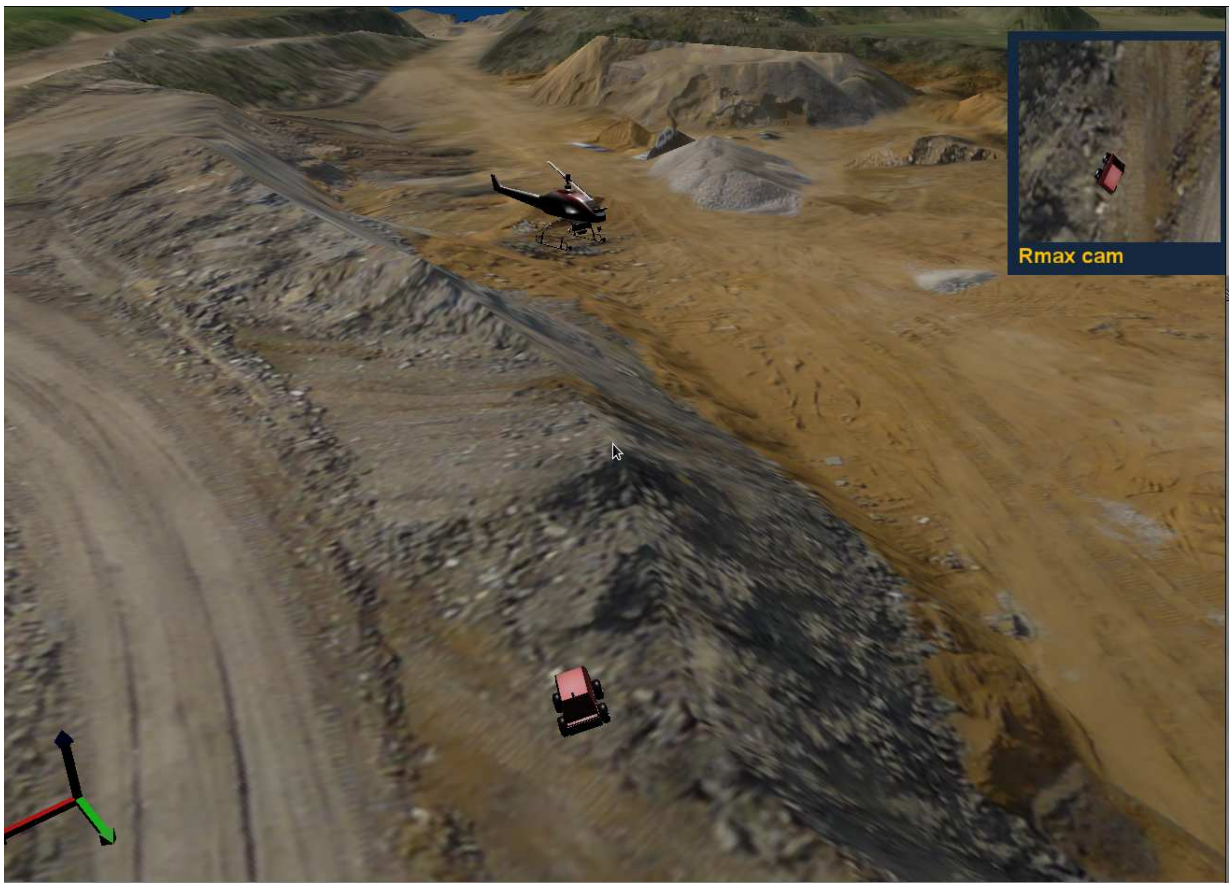

FIGURE B.1 – Un UGV et un hélicoptère sur un terrain modélisé sous Blender. Le modèle numérique de terrain est construit sur la base d'images aériennes acquises au dessus d'un site expérimental. En haut à droite, on voit la vue de la caméra de l'hélicoptère

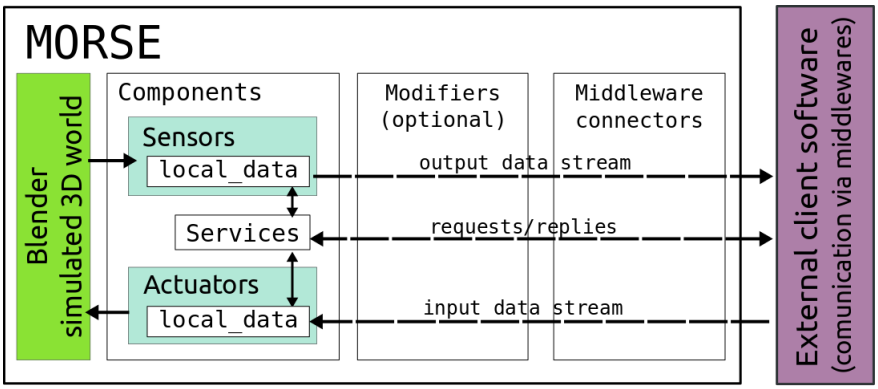

FIGURE B.2 – Vue d'ensemble du flux de données entre le monde simulé dans Blender et le monde extérieur (les logiciels de l'architecture fonctionnelle).

de facilement découpler ces deux aspects : un capteur MORSE peut avoir plusieurs méthodes pour exporter ses données. À l'heure actuelle, MORSE supporte de nombreux formats d'échanges de données incluant entre autre Yarp (Fitzpatrick et al., 2008), ROS (Quigley et al., 2009), MOOS (Newman, 2008), ou Pocolibs, utilisé au LAAS avec GeNoM (Fleury et al., 1997). Enfin, un troisième type de composant permet de modifier le contenu des premiers (la logique) avant de les passer aux derniers (l'export vers le monde extérieur). Ces composants permettent d'ajouter du bruit, des filtres, et de matière générale de transformer les données calculées par le capteur, afin de les rendre plus réalistes (moins parfaites), ou bien les adapter au mieux au type de sortie attendue. La figure B.2 résume les interactions entre ces différents composants, le moteur de jeu de Blender, et les logiciels tiers de l'architecture fonctionnelle.

B.2 Différents niveaux de réalisme

Une autre notion importante dans MORSE est la possibilité d'avoir des capteurs ou des actionneurs à différents niveaux de réalisme. En effet, selon ce que l'on souhaite étudier en simulation, il n'est pas forcément nécessaire d'avoir un modèle de capteur qui produit des données brutes fidèles à la réalité, mais plutôt le résultat d'une chaîne de traitement de telles données.

Un exemple caractéristique dans l'implémentation actuelle de MORSE est celui des composants *caméra* et *caméra sémantique*. Le premier fournit une image brute d'une caméra parfaite, tandis que le second retourne directement l'ensemble des objets présents dans le champ de vue de la caméra. Évidemment, il est possible de déduire de telles informations des images capturées par la caméra, mais cela nécessite l'application de traitements et est donc plus coûteux au niveau de la couche fonctionnelle. Au contraire, les calculs liés à la caméra sémantique sont nuls au niveau de la couche fonctionnelle – et plus légers au niveau simulation que la production d'images brutes. Si l'on s'intéresse à des questions d'interactions de haut niveau, on peut se contenter du résultat de la caméra sémantique, permettant ainsi de se concentrer sur les problématiques liées à l'interaction, et non pas d'avoir à gérer les problèmes liés aux algorithmes de perception.

Côté actionneur, MORSE propose naturellement un ensemble de contrôleurs de bas niveau,

prenant directement un contrôle en (v, ω) ou (v_x, v_y, ω) pour un robot mobile, mais aussi des contrôleurs de plus haut niveau. Par exemple, le contrôleur *destination* permet de guider un robot vers un but, tout en évitant les obstacles, via des méthodes réactives. Pour quelqu'un ne s'intéressant pas aux problématiques de navigation, ce contrôleur est donc beaucoup plus intéressant, et notamment moins coûteux qu'une boucle complète de perception, modélisation, planification et exécution de trajectoire.

Ce niveau de réalisme ajustable permet ainsi à chacun de configurer la simulation afin de se concentrer sur ses besoins spécifiques. Si l'on s'intéresse à simuler l'architecture complète (par exemple pour tester la coordination de l'ensemble), on peut simuler les capteurs au plus bas niveau et utiliser directement la sortie de ces capteurs. Si au contraire, on s'intéresse à une partie spécifique de la couche fonctionnelle, le simulateur MORSE offre la possibilité de s'abstraire des problématiques périphériques et de se concentrer sur ses propres algorithmes.

B.3 Flots de contrôle : les services

Dans la section 2.2, nous évoquions la différence entre flot de données et flot de contrôle. Le même genre de distinction est nécessaire au niveau simulation. Jusque là, nous avons principalement évoqué la notion de flot de données, entrant ou sortant. Toutefois, la notion de flot de contrôle est aussi importante pour la simulation, que ce soit pour contrôler le comportement global de la simulation (supervision de la simulation), pour configurer un composant de la simulation, ou bien pour simuler un composant fonctionnel complet au niveau du simulateur, qui requiert un flot de données et un flot de contrôle.

La conception des services dans MORSE suit la même méthodologie que celle exposée pour les composants classiques. Des services sont implémentés via les mots clés `@service` et `@async_service` dans les différents composants, de manière indépendante de la façon dont ils seront appelés depuis l'extérieur. Des entrées spécifiques au niveau "middleware" permettent de faire le pont entre les différentes façons d'interroger un service et les appels à l'intérieur de MORSE. Enfin, un système d'*overlays* permet de proposer une interface spécifique pour un ensemble de services, afin de ressembler autant que possible à un véritable composant logiciel. C'est cette dernière fonctionnalité qui permet de simuler entièrement un composant de la couche fonctionnelle dans le simulateur, ce qui laisse la possibilité de tester une architecture de contrôle complète avec certains sous-systèmes entièrement simulés.

En plus des cadres logiciels cités plus haut, une API haut niveau en Python, basée sur une communication "socket" est disponible, permettant de facilement contrôler la simulation, mais aussi les différents composants de la simulation. En particulier, il est ainsi possible de donner des comportements plus ou moins intelligents à des agents non robots depuis l'extérieur du simulateur, par exemple pour simuler des éléments mobiles dans l'environnement (cibles dans un contexte militaire, victimes dans un contexte de sauvetage...).

B.4 Autres fonctionnalités de MORSE

B.4.1 Un langage de description de scène

Initialement, les scènes étaient décrites entièrement dans le format Blender, en ajoutant graphiquement les différents éléments qui composent la scène. Un fichier stocké dans la scène contenait la configuration de la scène, à savoir quels "modifiers" sont associés à quels composants, comment sont exportés les différents composants... Toutefois, cette approche est rapidement apparue peu modulaire et peu réutilisable. Sur la base de l'API proposée par Blender, nous avons développé un langage de description de scène basé sur Python. Celui-ci permet aussi bien de créer des robots complexes, de définir des scènes ou de le configurer, mais il permet surtout une représentation modulaire, composable et textuelle de ces différents éléments. En effet, il est possible de définir un robot complexe dans un fichier puis de l'utiliser dans différents contextes. Une scène peut alors être construite à partir de l'inclusion de différents robots complexes et de divers éléments de décors. Grâce à la richesse de Python, il est possible de construire des scènes plus complexes encore, par exemple en plaçant les victimes de manière aléatoire, ou suivant une loi spécifique, ou encore suivant un certain modèle donné en entrée. Enfin, une description texte assure un stockage plus efficace, et s'intègre mieux avec les gestionnaires de versions classiques que les fichiers binaires de Blender.

B.4.2 Interaction Homme Robots

De plus en plus de travaux étudient les interactions entre hommes et robots. MORSE propose un avatar humain interactif. Immergé dans la peau de l'avatar, l'utilisateur peut contrôler son avatar via différents moyens, clavier, WiiMote, ou Kinect. Avec une interface inspirée des jeux vidéos 3D, il peut interagir avec l'environnement, en attrapant et en reposant les différents objets, ou en ouvrant / fermant les portes et tiroirs (image B.3). De plus, il est possible de récupérer directement la position de l'avatar, ce qu'il regarde, ... et donc d'éluder les problèmes de perception liés à l'interaction. Cet avatar humain permet donc d'étudier facilement en simulation comment le robot évolue et interagit en fonction des différentes actions de l'homme.

B.4.3 Simulation multi-nœuds et hybrides

Un des problèmes d'une simulation fine est le problème du passage à l'échelle. Si il est possible de simuler un ou deux robots avec des capteurs fournissant des données denses (tels les caméras ou les télémètre laser) sur une machine moderne, il est difficile voire impossible d'en simuler une dizaine ou plus sans voir les performances de la simulation fortement se dégrader.

Pour pallier ce problème, MORSE propose une simulation distribuée, où chaque nœud de simulation va simuler un nombre limité de robots. Les différents nœuds sont alors coordonnés via un nœud central : ce nœud central donne le pas d'exécution aux différents nœuds et synchronise aussi la position des différents robots dans les différents nœuds. Cette technique a été implémentée de deux manières différentes : une approche simple basée sur un protocole texte au-dessus de TCP/IP et une approche basée sur HLA ("High Level Architecture") (Kuhl et al., 1999). Cette

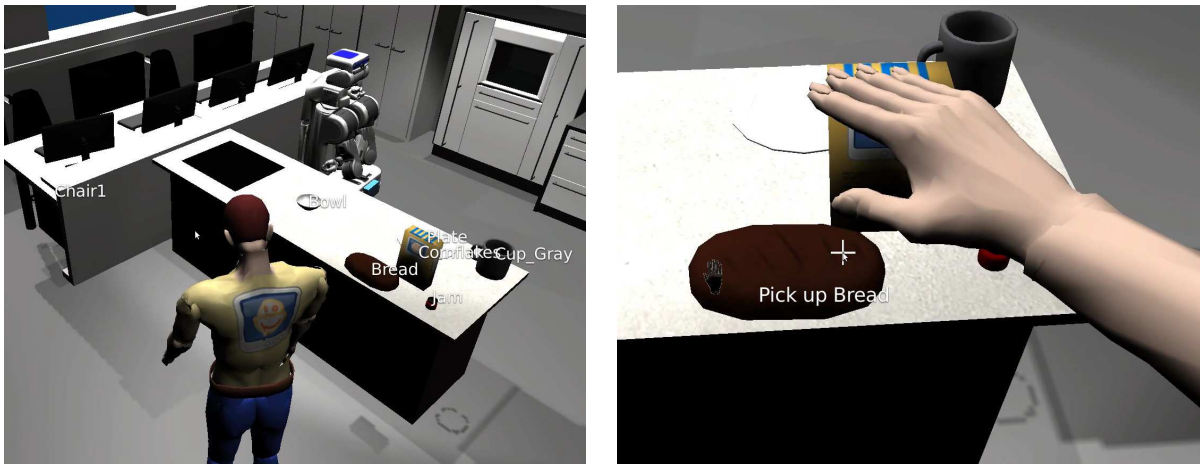

FIGURE B.3 – À gauche, une vue 3D de l'avatar humain est utilisée pour naviguer dans l'environnement, affichant les différents éléments avec lesquels l'humain peut interagir. À droite, une vue immersive indique la possibilité d'attraper la miche de pain

seconde approche permet une gestion plus fine du temps. Basée sur un standard, elle rend possible l'interaction de MORSE avec d'autres types de simulateurs : il est alors possible d'intégrer dans MORSE des simulations plus réalistes de modèles de vols, de navigation sous-marine ...

Un autre enjeu assez semblable est la simulation hybride, *i.e.* l'interaction entre un ensemble de robots simulés et un ensemble de robots réels. Blender permet de modéliser de manière fine un environnement 3D. En utilisant les mécanismes précédents, les robots réels peuvent échanger leurs positions avec le simulateur, et MORSE peut ainsi refléter les actions des robots réels dans la simulation. Cela ne résout pas entièrement les problèmes liés à la simulation hybride (en particulier la perception des robots simulés par les robots réels) mais est suffisant pour un certain nombre de domaines. En particulier, il est possible d'étudier les interactions entre des robots terrestres et volants, avec des robots terrestres réels et des robots volants simulés, les robots terrestres ne percevant en général pas les robots volants.

B.5 Synthèse

Nous avons décrit le simulateur robotique libre MORSE. Celui-ci est basé sur Blender, permettant ainsi de créer des simulations riches et réalistes. Son architecture interne permet une intégration aisée avec différentes architectures fonctionnelles existantes. Le choix du niveau de réalisme permet à l'utilisateur de se concentrer sur ce qui l'intéresse : dans le cas d'un algorithme de la couche fonctionnelle, il peut facilement abstraire certains comportements, alors que sil s'intéresse plutôt à l'ensemble de l'architecture, il peut simuler aux plus bas niveaux et utiliser tous les composants logiciels de l'architecture. Enfin, MORSE offre différentes fonctionnalités spécifiquement adaptées à certains domaines d'études. Pour étudier les interactions homme-robot, il propose un avatar humain riche en fonctionnalités. Pour les problématiques multi-robots, MORSE offre un mode distribué permettant de passer à l'échelle. Tous ces éléments font de MORSE un

simulateur versatile, puissant, facilement intégrable, facilitant le développement, le test et l'intégration d'expérimentations robotiques complexes.

Bibliographie

Agerwala, T. and Flynn, M. (1973). Comments on capabilities, limitations and c̈orrectnessöf petri nets. *SIGARCH Comput. Archit. News*, 2(4) :81–86.

Alami, R., Chatila, R., Fleury, S., Ghallab, M., and Ingrand, F. (1998). An architecture for autonomy. *The International Journal of Robotics Research*, 17(4) :315–337.

Allen, R., Douence, R., and Garlan, D. (1998). Specifying and analyzing dynamic software architectures. *Fundamental Approaches to Software Engineering*, pages 21–37.

Ammar, B., Abdallah, K., and Faiza, B. (2011). Rewriting logic based approach for the formalization of critical systems based on multi-agent system. *International Journal of Computer Applications*, 13(2) :6–13.

Ando, N., Suehiro, T., Kitagaki, K., Kotoku, T., and Yoon, W.-K. (2005). RT-middleware : distributed component middleware for RT (robot technology). In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3933 – 3938.

Andreoli, J. (1992). Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3) :297–347.

Andreoli, J.-M. and Pareschi, R. (1991). Linear objects : logical processes with built-in inheritance. *New Generation Computing*, 9 :495–510.

Bae, K. and Meseguer, J. (2010). The Linear Temporal Logic of Rewriting Maude Model Checker. In Olveczky, P., editor, *Rewriting Logic and Its Applications*, volume 6381 of *Lecture Notes in Computer Science*, pages 208–225. Springer Berlin / Heidelberg.

Baillie, J.-C. (2005). URBI : towards a universal robotic low-level programming language. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 820 – 825.

Barbier, M., Gabard, J., Vizcaino, D., and Bonnet-Torrès, O. (2006). Procosa : a software package for autonomous system supervision. In *1st National Workshop on Control Architectures of Robots : software approaches and issues, Montpellier, France*.

Beetz, M. (2000). *Concurrent reactive plans : anticipating and forestalling execution failures*. Springer-Verlag.

Bellin, G. and Scott, P. (1994). On the π-calculus and linear logic. *Theoretical Computer Science*, 135(1) :11 – 65.

Bernard, D., Dorais, G., Fry, C., Jr., E. G., Kanefsky, B., Kurien, J., Millar, W., Muscettola, N., Nayak, P. P., Pell, B., Rajan, K., and Rouquette, N. (1998). Design of the Remote Agent experiment for spacecraft autonomy. In *IEEE Aerospace Conference*.

Berry, G. (2000). The foundations of Esterel. *Proof, Language and Interaction : Essays in Honour of Robin Milner*, pages 425–454.

Berthomieu, B. and Menasche, M. (1983). An enumerative approach for analyzing time Petri nets. In *Proceedings IFIP*. Citeseer.

Bohren, J. and Cousins, S. (2010). The SMACH High-Level Executive. *IEEE Robotics and Automation Magazine*, 17(4) :18 –20.

Bonasso, R. P., Kortenkamp, D., Miller, D. P., and Slack, M. (1995). Experiences with an architecture for intelligent, reactive agents. *Journal of experimental and theorical artificial intelligence*, 9 :237–256.

Borrelly, J., Coste-Maniere, É., Espiau, B., Kapellos, K., Pissard-Gibollet, R., Simon, D., and Turro, N. (1998). The ORCCAD architecture. *The International Journal of Robotics Research*, 17(4) :338.

Boumghar, R. and Lacroix, S. (2011). Over the hill and far away : aerial/ground cooperation for long range navigation. In *Proceedings of the 14th International Conference on Climbing and Walking Robots and the Support Technologies for Mobile Machines, Paris (France)*, pages 215–222.

Boumghar, R., Roussillon, C., Degroote, A., Cox, P., Vandeportaele, V. D. B., Herrb, M., and Lacroix, S. (2011). Over the hill and far away : aerial/ground cooperation for long range navigation. In *International Workshop on Robotics for risky interventions and Environmental Surveillance-Maintenance, Leuwen (Belgique)*.

Bozzano, M., Delzanno, G., and Martelli, M. (2004). Model checking linear logic specifications. *Theory and Practice of Logic Programming*, 4(5-6) :573–619.

Brooks, A., Kaupp, T., Makarenko, A., Williams, S., and Oreback, A. (2007). Orca : A Component Model and Repository. In Brugali, D., editor, *Software Engineering for Experimental Robotics*, volume 30 of *Springer Tracts in Advanced Robotics*, pages 231–251. Springer.

Brooks, R. A. (1990). Elephants don't play chess. *Robotics and Autonomous Systems*, 6 :3–15.

Brugali, D. (2007). *Software engineering for experimental robotics*, volume 30. Springer Verlag.

Brugali, D. and Shakhimardanov, A. (2010). Component-Based Robotic Engineering (Part II). *IEEE Robotics and Automation Magazine*, 17(1) :100 –112.

Bruyninckx, H. (2001). Open robot control software : the OROCOS project. In *IEEE International Conference on Robotics and Automation, Seoul (Korea)*, pages 2523–2528.

Busquets, D., Sierra, C., and De Màntaras, R. (2003). A multiagent approach to qualitative landmark-based navigation. *Autonomous Robots*, 15(2) :129–154.

Cervesato, I., Pfenning, F., Walker, D., and Watkins, K. (2003). A concurrent logical framework ii : Examples and applications. Technical report, Carnegie Mellon University.

Cervesato, I. and Scedrov, A. (2009). Relating state-based and process-based concurrency through linear logic (full-version). *Information and Computation*, 207(10) :1044–1077.

Chaudhuri, K. (2006). *The focused inverse method for linear logic*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA. AAI3248489.

Chaudhuri, K. and Pfenning, F. (2005). A focusing inverse method theorem prover for first-order linear logic. In *Automated Deduction – CADE-20*, pages 737–737. Springer.

Chien, S., Knight, R., Stechert, A., Sherwood, R., and Rabideau, G. (2000). Using iterative repair to improve responsiveness of planning and scheduling. In *Proceedings of the Fifth International Conference on Artificial Intelligence Planning and Scheduling*, pages 300–307.

Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., and Quesada, J. (2002). Maude : Specification and programming in rewriting logic. *Theoretical Computer Science*, 285(2) :187–243.

Collé, F., Champagnat, R., and Prigent, A. (2005). Scenario analysis based on linear logic. In *Proceedings of the 2005 ACM SIGCHI International Conference on Advances in computer entertainment technology*.

Coste-Manière, E. and Espiau, B., editors (1998). *Special Issue on Integrated Architectures for Robot Control and Programming*, volume 17(4) of *International Journal of Robotics Research*. Sage.

Coste-Manière, E. and Simmons, R. (2000). Architecture, the backbone of robotic systems. In *IEEE International Conference on Robotics and Automation, San Francisco, CA (USA)*.

Cresswell, S., Smaill, A., and Richardson, J. (2000). Deductive synthesis of recursive plans in linear logic. In *Proceedings of the 5th European Conference on Planning : Recent Advances in AI Planning*, pages 252–264. Springer-Verlag.

Degroote, A. and Lacroix, S. (2010). Structuring processes into abilities : an information-oriented architecture for autonomous robots. In *5th National Conference on Control Architectures of Robots Douai (France)*.

Degroote, A. and Lacroix, S. (2011a). Roar : Resource oriented agent architecture for the autonomy of robots. In *International Conference on Robotics and Automation, Shangai (China)*.

Degroote, A. and Lacroix, S. (2011b). Roar : Resource oriented agent architecture for the autonomy of robots. In *9th International Conference on Practical Applications of Agents and Multi-Agent Systems (PAAMS), Salamanca (Spain)*.

Degroote, A. and Lacroix, S. (2012a). A control architecture for autonomous robots founded on resource agents. *Submitted to International Journal of Robotics Research*.

Degroote, A. and Lacroix, S. (2012b). A resource-agents robot control architecture : design and formalization. In *7th National Conference on Control Architectures of Robots, Montpellier (France)*.

Degroote, A. and Lacroix, S. (2012c). Roar : une architecture orientée agents pour l'autonomie des robots. In *18ème congrès francophone sur la Reconnaissance des Formes et l'Intelligence Artificielle, Lyon (France)*.

Degroote, A. and Lacroix, S. (2012d). Une architecture pour l'autonomie des robots basée sur des agents-ressources. In *6ème Conférence francophone sur les architectures logicielles, Montpellier (France)*.

Dixon, L., Smaill, A., and Bundy, A. (2006). Planning as deductive synthesis in intuitionistic linear logic. Technical report, Technical Report EDI-INF-RR-0786, School of Informatics, University of Edinburgh.

Dowek, G., Munoz, C., and Pasareanu, C. (2007). A formal analysis framework for plexil. In *Proceedings of 3rd Workshop on Planning and Plan Execution for Real-World Systems*, pages 45–51.

Dowek, G., Muñoz, C., and Rocha, C. (2010). Rewriting logic semantics of a plan execution language. *Electronic Proceedings in Theoretical Computer Science*, 18 :77–91.

Eberbach, E., Brooks, R., and Phoha, S. (1999). Flexible optimization and evolution of underwater autonomous agents. *New Directions in Rough Sets, Data Mining, and Granular-Soft Computing*, pages 519–527.

Echeverria, G., Lassabe, N., Degroote, A., and Lemaignan, S. (2011). Modular open robots simulation engine : Morse. In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pages 46 –51.

Echeverria, G., Lemaignan, S., Degroote, A., Lacroix, S., Karg, M., Koch, P., Lesire-Cabaniols, C., and Stinckwich, S. (2012). User Driven Improvements to Robotics Simulation in MORSE. In *Submitted to Simulation, Modeling, and Programming for Autonomous Robots*.

Estlin, T., Volpe, R., Nesnas, I., Mutz, D., Fisher, F., Engelhardt, B., and Chien, S. (2001). Decision-making in a robotic architecture for autonomy. In *Proceedings of the International Symposium on Artificial Intelligence, Robotics, and Automation in Space*.

Fages, F., Ruet, P., and Soliman, S. (2001). Linear concurrent constraint programming : operational and phase semantics. *Information and Computation*, 165(1) :14–41.

Firby, R. (1989). *Adaptive Execution in Complex Dynamic Worlds*. PhD thesis, Yale University, Department of Computer Science.

Fitzpatrick, P., Metta, G., and Natale, L. (2008). Towards long-lived robot genes. *Robotics and Autonomous Systems*, 56(1) :29–45.

Fleury, S., Herrb, M., and Chatila, R. (1997). GenoM : a tool for the specification and the implementation of operating modules in a distributed robot architecture. In *Proceedings of the 1997 IEEE/RSJ International Conference on Intelligent Robots and Systems*, volume 2, pages 842 –849 vol.2.

Garlan, D., Monroe, R., and Wile, D. (2010). Acme : An architecture description interchange language. In *CASCON First Decade High Impact Papers*, pages 159–173. ACM.

Gat, E. (1991a). Alfa : a language for programming reactive robotic control systems. In *Robotics and Automation, 1991. Proceedings., 1991 IEEE International Conference on*, pages 1116 –1121 vol.2.

Gat, E. (1991b). Integrating reaction and planning in a heterogeneous asynchronous architecture for mobile robot navigation. *SIGART Bull.*, 2 :70–74.

Gat, E. (1997a). ESL : a language for supporting robust plan execution in embedded autonomous agents. In *Aerospace Conference, 1997. Proceedings., IEEE*, volume 1, pages 319 –324 vol.1.

Gat, E. (1997b). On three-layer architectures. In *Artificial intelligence and mobile robots*, pages 195–210. AAAI Press.

Gentzen, G. (1935). Untersuchungen über das logische schließen. i. *Mathematische zeitschrift*, 39(1) :176–210.

Giorgini, P., Kolp, M., and Mylopoulos, J. (2001). Multi-agent architectures as organizational structures. *Autonomous Agents and Multi-Agent Systems*, 13 :2006.

Girard, J. (1987). Linear logic. *Theoretical computer science*, 50(1) :1–101.

Harel, D. and Naamad, A. (1996). The statemate semantics of statecharts. *ACM Trans. Softw. Eng. Methodol.*, 5(4) :293–333.

Harland, J. and Pym, D. (1997). Resource-distribution via Boolean constraints. *Automated Deduction—CADE-14*, pages 222–236.

Hartley, R. and Pipitone, F. (1991). Experiments with the subsumption architecture. In *Robotics and Automation, 1991. Proceedings., 1991 IEEE International Conference on*, pages 1652 –1658 vol.2.

Hoare, C. (1978). Communicating sequential processes. *Communications of the ACM*, 21(8) :666–677.

Hodas, J. and Miller, D. (1991). Logic programming in a fragment of intuitionistic linear logic. In *Logic in Computer Science, 1991. LICS'91., Proceedings of Sixth Annual IEEE Symposium on*, pages 32–42. IEEE.

Ingrand, F. F., Chatila, R., Alami, R., and Robert, F. (1996). PRS : A high level supervision and control language for autonomous mobile robots. In *IEEE International Conference on Robotics and Automation, Mineapolis*.

Innocenti, B., López, B., and Salvi, J. (2007). A multi-agent architecture with cooperative fuzzy control for a mobile robot. *Robotics and Autonomous Systems*, 55(12) :881 – 891.

Janicki, R. and Koutny, M. (1995). Semantics of inhibitor nets. *Information and Computation*, 123(1) :1–16.

Jensen, K. (1987). Coloured Petri nets. In Brauer, W., Reisig, W., and Rozenberg, G., editors, *Petri Nets : Central Models and Their Properties*, volume 254 of *Lecture Notes in Computer Science*, pages 248–299. Springer Berlin / Heidelberg. 10.1007/BFb0046842.

Jensen, K., Kristensen, L., and Wells, L. (2007). Coloured Petri Nets and CPN Tools for modelling and validation of concurrent systems. *International Journal on Software Tools for Technology Transfer (STTT)*, 9 :213–254. 10.1007/s10009-007-0038-x.

Jonsson, A., Verma, V., Pasareanu, C., and Iatauro, M. (2006). Universal Executive and PLEXIL : Engine and Language for Robust Spacecraft Control and Operations. In *AIAA Space Conference*.

Joyeux, S., Alami, R., Lacroix, S., and Philippsen, R. (2009). A plan manager for multi-robot systems. *The International Journal of Robotics Research*, 28(2) :220–240.

Kahramanoğulları, O. (2009). On linear logic planning and concurrency. *Information and Computation*, 207(11) :1229–1258.

Kanovich, M. and Vauzeilles, J. (2001). The classical AI planning problems in the mirror of Horn linear logic : semantics, expressibility, complexity. *Mathematical Structures in Computer Science*, 11(6) :689–716.

Kobayashi, N. and Yonezawa, A. (1993). ACL - A concurrent linear logic programming paradigm. In *Proceedings of the 1993 International Logic Programming Symposium*, pages 279–294.

Kuhl, F., Weatherly, R., and Dahmann, J. (1999). *Creating computer simulation systems : an introduction to the high level architecture*. Prentice Hall PTR, Upper Saddle River, NJ, USA.

Küngas, P. (2002). Resource-conscious AI planning with conjunctions and disjunctions. *Acta Cybernetica*, 15(4) :601–620.

Lacroix, S., Mallet, A., Bonnafous, D., Bauzil, G., Fleury, S., Herrb, M., and Chatila, R. (2002). Autonomous rover navigation on unknown terrains : Functions and integration. *International Journal of Robotics Research*, 21(10-11) :917–942.

Lloyd, J. (1987). *Foundations of Logic Programming*. Springer-Verlag.

López, P., Pfenning, F., Polakow, J., and Watkins, K. (2005). Monadic concurrent linear logic programming. In *Proceedings of the 7th ACM SIGPLAN international conference on Principles and practice of declarative programming*, PPDP '05, pages 35–46, New York, NY, USA. ACM.

Lyons, D. (1993). Representing and analyzing action plans as networks of concurrent processes. *Robotics and Automation, IEEE Transactions on*, 9(3) :241–256.

Mallet, A., Pasteur, C., Herrb, M., Lemaignan, S., and Ingrand, F. (2010). Genom3 : Building middleware-independent robotic components. In *IEEE International Conference on Robotics and Automation*, Anchorage, AK (USA).

Martí-Oliet, N. and Meseguer, J. (1989). From Petri Nets to Linear Logic. In *Category Theory and Computer Science*, pages 313–340, London, UK. Springer-Verlag.

Marti-Oliet, N. and Meseguer, J. (1996). Rewriting logic as a logical and semantic framework. *Electronic Notes in Theoretical Computer Science*, 4 :190–225.

Marti-Oliet, N. and Meseguer, J. (1999). *Dynamic Worlds : From the Frame Problem to Knowledge Management*, chapter Action and change in rewriting logic. Kuler Academics.

McGann, C., Py, F., Rajan, K., and Olaya, A. G. (2009). Integrated Planning and Execution for Robotic Exploration. In *International Workshop on Hybrid Control of Autonomous Systems*.

McGann, C., Py, F., Rajan, K., Thomas, H., Henthorn, R., and Mcewen, R. (2008). A Deliberative Architecture for AUV Control. In *IEEE International Conference on Robotics and Automation, Pasadena (USA)*.

Meseguer, J. (1992). Conditional rewriting logic as a unified model of concurrency. *Theoretical computer science*, 96(1) :73–155.

Meseguer, J., Palomino, M., and Marti Oliet, N. (2003). Notes on model checking and abstraction in rewriting logic. Technical report, U. of Illinois at Urbana-Champaign.

Meyer, B. (1992). Applying "design by contract". *Computer*, 25 :40–51.

Milner, R. (1982). *A calculus of communicating systems*. Springer-Verlag New York, Inc.

Milner, R. (1997). *The definition of standard ML : revised*. The MIT press.

Milner, R., Parrow, J., and Walker, D. (1992). A calculus of mobile processes, parts i and ii. *Information and Computation*, 100(1) :1–77.

Mints, G. (1993). Resolution calculus for the first order linear logic. *Journal of Logic, Language and Information*, 2(1) :59–83.

Moggi, E. (1991). Notions of computation and monads. *Information and computation*, 93(1) :55–92.

Morandi, B., Bauer, S. S., and Meyer, B. (2008). Scoop - a contract-based concurrent object-oriented programming model. In Muller, P., editor, *Advanced Lectures on Software Engineering, LASER Summer School 2007/2008*, volume 6029 of *Lecture Notes in Computer Science*, pages 41–90. Springer.

Muscettola, N., Dorais, G., Levinson, C., and Plaunt, C. (2002). IDEA : Planning at the Core of Autonomous Reactive Agents. In *International NASA Workshop on Planning and Scheduling for Space*.

Newman, P. (2008). MOOS-mission orientated operating suite. *Massachusetts Institute of Technology, Tech. Rep*, 2299(08).

Nilsson, N. J. (1980). *Principles of artificial intelligence*. Morgan Kaufmann, San Francisco, CA, USA.

Nipkow, T., Wenzel, M., and Paulson, L. (2002). *Isabelle/HOL : a proof assistant for higher-order logic*. Springer-Verlag.

Oquendo, F. (2004). π-ADL : an Architecture Description Language based on the higher-order typed π-calculus for specifying dynamic and mobile software architectures. *ACM SIGSOFT Software Engineering Notes*, 29(3) :1–14.

Peterson, J. and Hager, G. (1999). Monadic robotics. In *the Proceedings of the Workshop on Domain Specific Languages, Usenix*.

Peterson, J., Hudak, P., and Elliott, C. (1998). Lambda in motion : Controlling robots with haskell. *Practical Aspects of Declarative Languages*, pages 91–105.

Py, F. and Ingrand, F. (2004). Dependable execution control for autonomous robots. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*.

Py, F., Rajan, K., and McGann, C. (2010). A systematic agent framework for situated autonomous systems. In *9th International Conference on Autonomous Agents and Multiagent Systems, Toronto (Canada)*, pages 583–590.

Pyarali, I., Harrison, T., Schmidt, D., and Jordan, T. (1997). Proactor – An Object Behavioral Pattern for Demultiplexing and Dispatching Handlers for Asynchronous Events. In *4th annual Pattern Languages of Programming conference in Allerton Park, Illinois*.

Quigley, M., Conley, K., Gerkey, B. P., Faust, J., Foote, T., Leibs, J., Wheeler, R., and Ng, A. Y. (2009). ROS : an open-source Robot Operating System. In *ICRA Workshop on Open Source Software*.

Radestock, M. and Eisenbach, S. (1996). Coordination in evolving systems. *Trends in Distributed Systems CORBA and Beyond*, pages 162–176.

Roscoe, A. (1994). Model-checking csp. *A Classical Mind, Essays in Honour of CAR Hoare. Prentice-Hall*, pages 353–378.

Roussillon, C., Gonzalez, A., Solà, J., Codol, J.-M., Mansard, N., Lacroix, S., and Devy, M. (2011). RT-SLAM : A Generic and Real-Time Visual SLAM Implementation. In *International Conference on Vision Systems, Sophia Antopolis (France)*.

Schack-Nielsen, A. and Schürmann, C. (2008). Celf – A Logical Framework for Deductive and Concurrent Systems (System Description). *Automated Reasoning*, pages 320–326.

Schmidt, D., Stal, M., Rohnert, H., and Buschmann, F. (2000). *Pattern-oriented software architecture : Patterns for concurrent and networked objects*, volume 2. Wiley.

Shakhimardanov, A., Paulus, J., Hochgeschwender, N., Reckhaus, M., and Kraetzschma, G. K. (2010). Best Practice in Robotics : Best Practice Assessment of Software Technologies for Robotics. Technical report, Bonn-Rhein-Sieg University.

Simmons, R. and Apfelbaum, D. (1998). A task description language for robot control. In *IEEE/RSJ Conference on Intelligent Robots and Systems, Victoria, B.C (Canada)*.

Smith, R. (1980). The contract net protocol : High-level communication and control in a distributed problem solver. *Computers, IEEE Transactions on*, 100(12) :1104–1113.

Soler, J., Julián, V., Carrascosa, C., and Botti, V. J. (2000). Applying the ARTIS Agent Architecture to Mobile Robot Control. In *Proceedings of the 7th Ibero-American Conference on AI : Advances in Artificial Intelligence*, pages 359–368.

Soliman, S. (2001). Phase model checking for some linear logic calculi. In *Proceedings of the Second International Workshop of the Implementation of Logics*, pages 60–80.

Stehr, M., Meseguer, J., and Olveczky, P. (2001). Rewriting logic as a unifying framework for petri nets. *Unifying Petri Nets*, pages 250–303.

Sun, J., Liu, Y., and Dong, J. (2009). Model checking CSP revisited : Introducing a process analysis toolkit. *Leveraging Applications of Formal Methods, Verification and Validation*, pages 307–322.

Tammet, T. (1994). Proof strategies in linear logic. *Journal of Automated Reasoning*, 12(3) :273–304.

Tang, F. (2007). A complete methodology for generating multi-robot task solutions using asymtre-d and market-based task allocation. In *in : Proc of the IEEE Int Conf on Robotics and Automation*, pages 3351–3358.

Tang, F. and Parker, L. E. (2005). Asymtre : Automated synthesis of multi-robot task solutions through software reconfiguration. In *In Proceedings of IEEE International Conference on Robotics and Automation*, pages 1513–1520.

Viry, P. (1996). Input/output for ELAN. *Electronic Notes in Theoretical Computer Science*, 4 :51–64.

Volpe, R., Nesnas, I., Estlin, T., Mutz, D., Petras, R., and Das, H. (2001). The CLARAty architecture for robotic autonomy. In *IEEE Aerospace Conference, Big Sky, MT (USA)*.

Watkins, K., Cervesato, I., Pfenning, F., and Walker, D. (2004). A concurrent logical framework : The propositional fragment. *Types for Proofs and Programs*, pages 355–377.

Wolfang Reisig (2010). *Petrinetze : Modellierungstechnik, Analysemethoden, Fallstudien*. Leitfäden der Informatik. Vieweg+Teubner. 248 pages ; ISBN 978-3-8348-1290-2.

Ziparo, V., Iocchi, L., Nardi, D., Palamara, P., and Costelha, H. (2008). Petri net plans : a formal model for representation and execution of multi-robot plans. In *Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems-Volume 1*, pages 79–86. International Foundation for Autonomous Agents and Multiagent Systems.