ROAR : une architecture orientée agents pour l'autonomie des robots

Arnaud Degroote, Simon Lacroix

ROAR : une architecture orientée agents pour l'autonomie des robots

Arnaud Degroote Simon Lacroix

CNRS; LAAS; 7 avenue du colonel Roche, F-31077 Toulouse, France
Université de Toulouse ; UPS, INSA, INP, ISAE; LAAS; F-31077 Toulouse, France
{first}.{last}@laas.fr

Résumé

Cet article présente un cadre pour organiser les différents processus nécessaires à l'autonomie d'un robot. Les principaux objectifs sont de permettre la réalisation d'une variété de missions sans que le le développeur ait à écrire explicitement les schémas de contrôle, et de permettre d'augmenter les capacités du robot sans devoir ré-écrire ces schémas. L'architecture proposée repose sur une partition de la couche décisionnelle en ressources, chacune gérée par un agent spécifique. Les mécanismes garantissant la bonne utilisation de chaque ressource et gérant le réseau d'agents sont décrits, et illustrés dans le cas d'une mission autonome de navigation.

Mots Clef

Architecture décisionnelle, agent-ressource, autonomie des robots.

Abstract

This paper presents a framework to organize the various processes that endow a robot with autonomy. The main objectives are to allow the achievement of a variety of missions without an explicit writing of control schemes by the developer, and the possibility to augment the robot capacities without any major rewriting. The proposed architecture relies on a partition of the decisional layer *in separate* resources*, each one managed by a specific agent. The mechanisms that guarantee the good use of each resource and manage the network of agent into a coherent system are depicted, and illustrated in the case of an autonomous navigation mission.*

Keywords

Decisional architecture, resource agents, robot autonomy.

1 Introduction

Outre les différents processus nécessaires à l'opération autonome d'un robot (perception, planification, contrôle, apprentissage, ...), c'est *l'assemblage et le contrôle* de ces processus qui confère à un robot une réelle autonomie d'actions. Cet assemblage, souvent défini comme "l'architecture décisionnelle" d'un robot, est en charge de la configuration, de l'ordonnancement, du déclenchement et du suivi de l'exécution des différents processus. Il doit être conçu afin de doter le robot de *(i)* la capacité de réaliser une grande variété de missions de haut niveau, sans configuration manuelle, et *(ii)* la capacité à faire face à un ensemble d'événements qui ne sont pas forcément connus a priori, dans un monde essentiellement imprévisible – ces deux capacités étant des caractéristiques essentielles de l'autonomie.

État de l'art. Traditionnellement, les architectures robotiques sont classées en deux catégories : les architectures réactives et les architectures en couches. Dans la première catégorie, chaque composant (parfois nommé "agent") implémente un comportement spécifique, et les comportements de haut niveau sont obtenus par la composition de comportements selon différentes stratégies. Par exemple, [Giorgini et al., 2001] s'appuie sur la théorie de l'organisation, avec un arbitre central qui coordonne les différents agents. [Innocenti et al., 2007] s'appuie sur une organisation décentralisée, en évitant ainsi la présence d'un point de défaillance unique sur l'agent de coordination, et exploite des règles de logique floue pour organiser les différents agents. Cette approche permet de rapidement réagir aux changements dans l'environnement, mais peut échouer face aux situations qui exigent une planification explicite des décisions.

Au contraire, les architectures en couches gèrent des plans à long terme. Dans [Gat, 1997], E. Gatt mène une analyse dont il déduit la définition de trois couches : une couche intermédiaire est nécessaire pour lier la couche fonctionnelle à la couche décisionnelle. La couche décisionnelle exploite un ou plusieurs planificateurs, et maintient un état à long terme, tandis que la couche intermédiaire gère un état interne éphémère. Beaucoup de travaux ont été consacrés à cette couche intermédiaire : dans [Gat, 1997], elle est nommée "séquenceur", et a la charge de traduire le plan symbolique en une séquence de comportements élémentaires adaptés à la situation courante. Dans l'architecture LAAS [Alami et al., 1998], la couche appelée "exécutif" est légèrement différente : elle a la charge de contrôler la bonne exécution de séquences de comportements. Un autre volet est dédié à la décomposition des plans en séquences exécutables : c'est le "superviseur", basé sur le langage PRS [Ingrand et al., 1996]. Le système *Re-*

mote Agent [Bernard et al., 1998] combine la traduction des plans en tâches atomiques, le contrôle d'exécution, la gestion d'événements ainsi que la gestion des ressources. En cas d'échec à l'exécution d'une tâche, la couche EXEC peut demander une réparation à un système dénommé MIR – et non à la couche décisionnelle. [Jonsson et al., 2006] présente PLEXIL, une approche pour lier la couche fonctionnelle et décisionnelle qui repose sur un langage vérifiable afin de définir la couche exécutive.

Toutes ces approches reposent sur l'idée principale qu'une couche intermédiaire est nécessaire pour combler le fossé entre les mondes fonctionnels et symboliques. Toutefois, cela conduit à des représentations différentes de plans, de modèles et des informations qui coexistent dans les différentes couches. Ces différences de représentation rendent le diagnostic des défaillances de plans difficile, car le planificateur ne dispose pas d'informations pertinentes sur les causes d'échec, et entravent l'efficacité de l'exécution des plans, car la couche intermédiaire n'a pas une vision globale du plan. Une première approche pour aborder ces problèmes a été faite par le système CLARATY [Estlin et al., 2001] : bien qu'il y ait toujours deux outils différents pour la couche décisionnelle (CASPER) et la couche de l'exécutif (TDL) qui coexistent, le système permet de refléter les changements d'une couche à l'autre, et des heuristiques définissent celle qui doit traiter un défaut lorsqu'il se présente. IDEA [Muscettola et al., 2002] définit une architecture à deux couches : le problème est partitionné en plusieurs agents s'appuyant sur le *même modèle de plan*, chacun étant composé d'un planificateur et d'une couche d'exécution. Ainsi, la planification et la phase d'exécution sont toujours entrelacés. Par ailleurs, pendant l'exécution, les différents agents sont synchronisés afin de maintenir une cohérence du plan global. L'architecture T-REX s'inspire de l'approche IDEA : basée sur une décomposition en agents, elle introduit en plus une formulation systématique pour l'échange d'états entre ces agents, ce qui offre plus de garanties sur la cohérence de l'exécution du plan global.

Une autre limite des architectures à trois niveaux est leur évolutivité : elles peuvent difficilement gérer des robots avec des nombreuses capacités ou la réalisation d'une grande variété de missions. Dans une telle architecture, les couches décisionnelle et d'exécution sont deux blocs "monolithiques" séparés, sans granularité fine pour la représentation et la manipulation de l'information. L'ajout ou la suppression de fonctionnalités à bord du robot ou la définition de nouveaux types de missions conduit souvent à une réécriture majeure des couches, et l'augmentation des fonctionnalités augmente le temps de délibération, ce qui rend le robot moins réactif aux changements de situation. Mc Gann *et al* [McGann et al., 2009] prétendent qu'exploiter un unique plan global et une unique couche d'exécution n'est pas viable sur le long terme, et concluent que le problème doit être réparti pour être efficacement traité : l'utilisation d'agents de planification différents, avec des contraintes temporelles différentes, résout partiellement la question de l'évolutivité. Mais la partition proposée est définie et construite par le programmeur, sur la base des besoins de la mission. Si la nature de la mission change, toute la partition doit être réorganisée en conséquence : ce type de construction ne s'adapte pas bien à une grande variété de missions, ce qui limite la flexibilité de l'architecture.

Motivations. Le principe de la *séparation des fonctionnalités* du robot en un réseau de composants est essentiel pour simplifier le contrôle global du système. Cette partition doit être conçue avec soin afin de rendre le système *composable* : il doit être possible d'ajouter et ou supprimer certains composants sans perturber le reste du système.
Une caractéristique essentielle de l'autonomie est la capacité de bien réagir aux situations imprévues (bien que la gestion correcte de *toute* situation demeure un défi) : de tels événements doivent être traités de manière asynchrone lorsqu'ils se produisent, et la stratégie la plus appropriée doit être sélectionnée pour chaque composant, sur la base de son modèle interne et de sa connaissance de cet événement.
Enfin, l'architecture doit être robuste aux pannes : en cas de défaillance d'un composant en raison d'une erreur de logique, de programmation ou d'une défaillance physique, l'architecture doit si possible poursuivre le contrôle du robot, en utilisant des stratégies alternatives pour continuer la mission.

Plan de l'article. Nous proposons dans cet article la définition d'une architecture (ROAR [1]) basée sur un schéma de partitionnement qui vise à satisfaire ces exigences de *composabilité*, *réactivité* et *robustesse*. La section suivante introduit la notion d'agent-ressource qui définit la partition, et la façon dont ces agents interagissent. Elle présente également un exemple applicatif qui sert de support à la présentation des développements : une mission de navigation autonome basée sur plusieurs modes de déplacements. La section 3 détaille la notion d'agent-ressource et les mécanismes internes associés qui permettent à la fois la délibération et l'exécution. La section 4 présente comment ces agents interagissent, en particulier lors de l'apparition de fautes, et une discussion conclut l'article.

2 Approche proposée

2.1 Agents-ressources

Nous suivons le principe de décomposition proposé dans IDEA ou T-REX , mais contrairement à ces architectures dans lesquelles la décomposition est définie en fonction d'un ensemble de tâches, nous proposons de décomposer les capacités du robot en un ensemble de *ressources* distinctes. Le terme "ressource" doit être ici compris dans son sens le plus général : une ressource peut être une ressource physique, une ressource d'information ou une ressource de planification. Chaque ressource est encapsulée dans un agent distinct, qui est responsable de la cohérence et du bon

[1] "Resource Oriented Agent architecture for the autonomy of Robots"

usage de la ressource : nous utilisons donc par la suite le terme agent-ressource.
Les agents-ressources n'exposent pas directement la ressource qu'ils encapsulent, mais une liste de différents points de contrôle, qui sont appelés *variables libres*. Les autres agents s'adressent à un agent en exprimant des contraintes sur ces variables libres. Les contraintes entre les agents forment un graphe orienté et dynamique, où les sommets sont les agents, et les arcs sont les contraintes entre les agents à un instant t. L'architecture ROAR est en charge de maintenir ce graphe, en assurant que les relations soient satisfaites. À cette fin, chaque agent est doté d'un raisonneur qui gère localement les contraintes qui lui sont fixées. En cas d'impossibilité, l'échec remonte à travers le graphe jusqu'à ce qu'il soit résolu par une politique définie – par exemple par un appel à un planificateur ou (en dernier recours) à un opérateur humain. De cette façon, l'architecture peut manipuler une variété de problèmes sans modifier la définition de chaque agent : le système adapte le graphe en fonction du problème courant, et les raisonneurs logiques ordonnancent l'accès aux ressources.

2.2 Exemple support

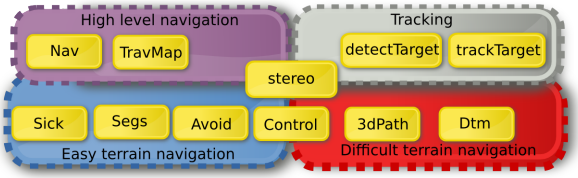

Figure 1: Couche fonctionnelle du robot considéré. Le boites jaunes sont des modules fonctionnels, les blocs pointillés sont les comportements définis par ces modules.

Nous considérons un robot mobile d'extérieur doté de capacités de navigation autonome, adaptées au type de terrain auquel il est confronté [Lacroix et al., 2002]. La couche fonctionnelle, représentée sur la figure 1 peut être décrite comme suit :

- Dans les zones assez planes ou faciles, le mode de navigation utilise une représentation bi-dimensionnelle des obstacles. C'est une boucle sensori-motrice définie par trois éléments fonctionnels *Segs*, *Avoid* et *Control*, qui respectivement fusionne les données acquises dans une carte d'obstacles représentés par des segments 2D, génère une commande d'évitement des obstacles, et contrôle les déplacements du robot en fonction de cette commande. La carte d'obstacles peut être construite sur les données fournies par *Sick*, qui exporte les données du scanner laser, ou bien sur les données de profondeur fournies par *Stereo*.

- Dans les zones plus difficiles, le mode de navigation est basé sur la modélisation 3D du terrain et la détermination de trajectoires dans ce modèle. La boucle sensori-motrice associée est définie par les composants *Stereovision*, *DTM*, *3DPath* et *Control*, qui respectivement fournit des données 3D, les fusionne en un modèle numérique de terrain, sélectionne une trajectoire élémentaire définie par des vitesses de translation et de rotation (arc de cercle), et contrôle le robot selon ces vitesses.

Un autre composant, *Nav*, est chargé de sélectionner le mode de navigation le plus adapté, sur la base d'une *carte de traversabilité* qui renseigne sur la difficulté du terrain, et qui est construite sur les données de profondeur de la stéréovision. Ce composant peut être vu comme une boucle de contrôle de niveau supérieur, définie par l'assemblage des composants *Stereo*, *TravMap* et *Nav*. Enfin, l'objectif à atteindre peut être soit défini par des coordonnées, soit être une cible visuellement détectée et suivie – dans ce dernier cas, les deux éléments suivants, se basant sur la stéréovision, sont nécessaires : *detectTarget* et *trackTarget*.
Au sein de cette couche fonctionnelle, on peut par exemple voir que la navigation sur terrain difficile et le suivi de cible s'appuient tous deux sur le module de stéréovision. Cependant, les contraintes sur la stéréovision sont très différentes entre ces deux activités : pour la navigation, le banc stéréoscopique doit être orienté vers l'avant et le bas, tandis que pour le suivi il doit être orienté vers la cible. Un tel conflit doit être évité, ce qui est facilité par une représentation explicite de ressources – un planificateur global omniscient pourrait gérer cela, mais au prix d'une complexité explosant avec l'augmentation des fonctionnalités du robot.
La figure 2 montre une vue simplifiée de la manière dont cette approche de navigation est décomposée en ressources distinctes. Cette décomposition est réalisée par l'expertise humaine : elle est basée sur une analyse des ressources et de leurs relations, tout d'abord effectuée sur notre couche fonctionnelle propre, puis généralisée sur la base de l'état de l'art. On y retrouve les différents modules fonctionnels, mais la plupart d'entre eux sont abstraits. Par exemple, nous ne trouvons pas directement les différentes interfaces de suivi, car la ressource intéressante, d'un point de vue de haut niveau, est la position de la cible – que cette position soit déterminée par un processus de suivi ou bien trivialement donnée au robot. De même, à ce niveau, on ne se soucie pas de la manière dont les données de profondeur sont obtenues – par exemple avec stéréovision ou un lidar 3D. Une telle décomposition en agents de ressources ne dépend pas du robot ou de sa mission actuelle : elle apporte abstraction et généricité. La section suivante décrit le fonctionnement interne de ces agents, qui garantit leur comportement correct.

3 Structure d'un agent-ressource

Chaque agent est responsable d'une ressource spécifique, et expose seulement un ensemble de *variables libres* pour spécifier les modifications sur l'état de cette ressource. Ceci est réalisé en utilisant une approche à deux niveaux

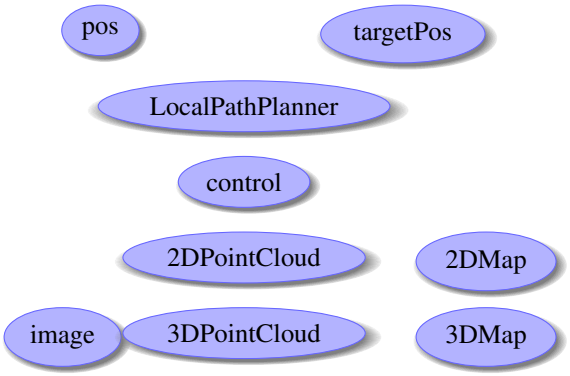

Figure 2: Décomposition simplifiée de la figure 1 en agents-ressources.

(figure 3) : à la réception d'une nouvelle contrainte, la *couche logique* détermine si elle peut être satisfaite sur la base du *contexte courant* de l'agent. Si oui, elle sélectionne un ensemble de tâches pour réaliser la transition d'un état logique à un autre. Pour chaque tâche, la *couche d'exécution* choisit une recette pour y parvenir en considérant le *contexte de tâche* courant.

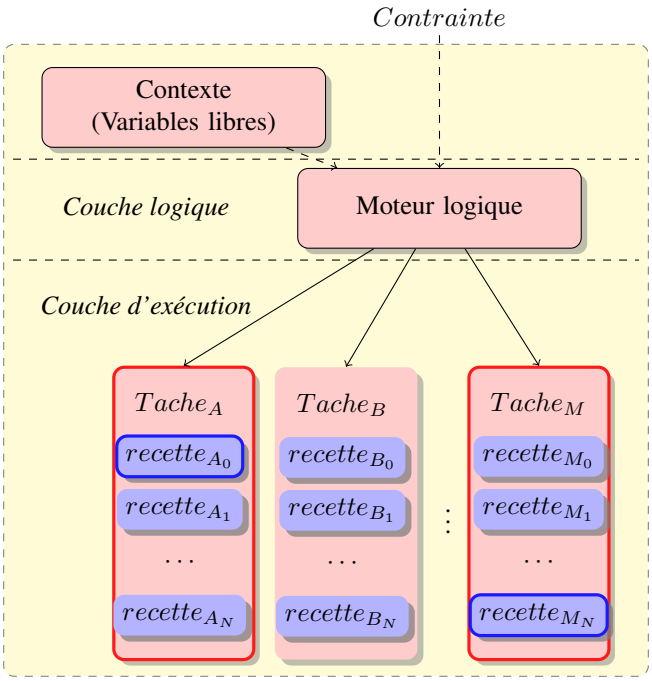

Figure 3: Mécanisme global d'un agent-ressource. Les blocs en traits gras sont les tâches et recettes actuellement sélectionnées par le niveau d'exécution logique.

3.1 Couche logique

À la réception d'une nouvelle contrainte, un agent-ressource doit décider si la contrainte demandée est compatible avec l'état logique courant de la ressource. Cela peut aisément être assuré par une machine à états finis pour les ressources simples avec quelques variables libres, mais la complexité de la machine à états augmente exponentiellement avec le nombre de variables, ce qui rend difficile pour le développeur de garantir sa cohérence ou de la modifier. La communauté des langages de programmation a proposé plusieurs langages pour résoudre ce genre de problème, comme Prolog ou, plus récemment, Mercury [Somogyi et al., 1995], selon le paradigme de "programmation logique". Nous proposons d'utiliser ce paradigme pour déclarer et décider si, dans le contexte actuel, une contrainte peut être résolue. La partie du système qui gère ces aspects est le *Moteur Logique* .
Le *Moteur Logique* nécessite l'expression de deux choses différentes :

- Le domaine logique, *i.e.* un ensemble de fonctions et de leurs relations logiques. Le domaine logique peut être partagé entre les différents agents.
- les transitions autorisées spécifiques à la ressource, définies par un nom et des pré- et post-conditions. Les post-conditions sont des prédicats qui sont vrais à la fin de la transition, et les pré-conditions sont des prédicats qui doivent être vrais avant le début de la transition. Considérant les faits courants et les contraintes nécessaires, le *Moteur Logique* choisit (en utilisant une méthode classique de chainage arrière) les combinaisons de transitions pour satisfaire les contraintes ou les rejeter si il ne trouve pas de solution. Cette transition entre deux états logiques s'appelle une *tâche*. Le listing 1 montre comment les tâches sont exprimées dans ROAR .

Listing 1: Définition de constraintes sur "3DMap"

```
struct point {
  double x, y, z;
};

newtype dist double;
dist distance(point A, point B);

clear = task {
pre = {{isEmpty == false}}
post = {{isEmpty == true}}
}

merge = task {
pre = {{isEmpty == false}}
post ={{isEmpty == false}
  {distance(lastMerged, pos::current) < threshold)}}
}

panoramicView = task {
pre = {{isEmpty == true}}
post = {{ isEmpty == false}
  {distance(lastMerged, pos::current) < threshold)}}
}
```

Puisque la notion de ressource est indépendante du robot considéré, la notion de transition entre deux états logiques pour une ressource est également indépendante du robot. Cette couche est donc générique et réutilisable pour différents robots. Elle est également indispensable car elle garantit la cohérence de la ressource – tant que le développeur ne fait pas d'erreurs logiques dans sa spécification.

3.2 Couche d'exécution

Les *tâches* décrivent les transitions possibles entre deux états logiques, elles ne décrivent pas comment les gérer : leur exécution est assurée par des *recettes*. L'objectif de la décomposition en *tâches* et *recettes* est double : *(i)* réduire la complexité pour sélectionner l'action à effectuer dans un agent, et ainsi réduire le temps de ce choix, et *(ii)* fournir différentes stratégies pour parvenir à une transition d'un état à un autre – en particulier, le développeur peut implanter des stratégies différentes pour gérer les diverses capacités des robots, ou gérer explicitement des erreurs. En d'autres termes, cette décomposition améliore la *réactivité* du système, et sa portabilité sur différentes plates-formes robotiques.

Les recettes sont décrites comme un ensemble de pré- et post-conditions, de manière similaire aux *tâches*, et un corps, qui contient la mise en œuvre réelle de la recette. La couche évalue les pré-conditions de chaque recette, et exécute le corps de la plus adaptée (celle avec le plus grand nombre de pré-conditions satisfaites). Si aucune recette ne correspond à la situation, la tâche est rejetée. Le corps des recettes est défini sur la base des primitives suivantes :

- **make** <prédicat> demande d'appliquer une contrainte et d'attendre jusqu'à ce qu'elle le soit (ou échoue).
- **ensure** <prédicat> demande la satisfaction d'une contrainte tant qu'elle n'est pas supprimée. Cet opérateur retourne un identifiant de transaction.
- **wait** <prédicat> fait attendre la recette jusqu'à ce que le prédicat devienne vrai.
- **abort** <identifier> annule une contrainte en attente.
- un appel à une fonction (de la couche fonctionnelle).
- **let** <variable> <expression> introduit une nouvelle variable locale, résultat du calcul de l'expression.
- **set** <variable> <expression> permet de modifier l'état de l'agent, en modifiant le contenu de la variable par le résultat de l'expression.

Le listing 2 montre une recette qui gère le déplacement du robot de de *pos::current* à *goal* en terrain accidenté. Cette exemple montre l'aspect déclaratif de notre langage : on exprime les contraintes nécessaires sur le système, et non une liste d'étapes pour y arriver. Le superviseur n'a pas besoin de connaître tous les détails de la couche fonctionnelle : dans le cadre d'une boucle classique sense-plan-act, il doit uniquement savoir quel planificateur doit être appelé pour réaliser telle action, et quelles informations sont nécessaires au planificateur.

C'est ce que nous exprimons ici : une solution pour aller de A vers B est de construire une 3DMap, de demander à 3DPath comment aller de A à B sur la base de ce modèle, et de suivre la trajectoire calculée. Une fois ces contraintes mises en place, il nous suffit d'attendre que la condition ligne 8 soit atteinte. Deux scénarios sont alors possibles :

- tout se passe bien, le prédicat associé à la primitive **wait** devient vrai, et la recette est un succès.
- à un moment t, une de ces contraintes échoue, la recette se termine alors sur un échec (voir section 4.1 pour le traitement d'un échec).

Listing 2: Recette pour la navigation en terrain difficile

```
3DPath_reach_goal = recipe {                                          1
Pre = {{failure == none}}                                             2
Post = {{distance(goal, pos::current) < goalThreshold)                3
    }}
body = {                                                              4
ensure(distance(3DMap::lastMerged, pos::current) <                    5
    threshold))
ensure(3DPath::goal == goal)                                          6
ensure(control::tracker == 3DPath::plan)                              7
wait(distance(goal, pos::current) < goalThreshold))                   8
}}                                                                    9
```

Dans tous les cas, le cadre logiciel assure qu'à la fin de l'exécution de la recette, toutes les contraintes sur le systèmes sont relâchées. La durée de vie d'une contrainte ne peut donc dépasser celle de la recette.

L'ordre des primitives **ensure**, même si elles sont exécutées en parallèle, n'est pas anodin. En effet, l'ordre utilisé définit une notion de causalité. Chaque primitive **ensure** a "besoin" des primitives **ensure** les précédant (ce sont des pré-requis) : il serait dangereux d'exécuter un plan si le planificateur n'est plus dans la capacité de générer des plans corrects. En cas de panne temporaire (l'agent n'est temporairement plus en mesure d'assurer la contrainte, mais il cherche une solution localement), les conséquents de cette contrainte C (l'ensemble des contraintes qui suivent C dans la recette) doivent être mis en pause, *i.e.* temporairement suspendue (les contraintes sont conservées au niveau des agents concernés, mais les exécutions associées sont suspendues). Ce comportement est illustré par la figure 4. Ces relations entre les contraintes sont particulièrement importantes, car elles sont une des clés de la cohérence du système : il n'est pas possible d'assurer une contrainte si ces pré-requis ne sont pas assurés.

4 Interactions entre agents

Dans cette section, nous décrivons comment interagissent les agents afin de résoudre les échecs d'exécutions et les problèmes de concurrences.

4.1 Gestion des erreurs

Des erreurs ou des événements inattendus se produisent souvent lors de missions robotiques, et il est essentiel de bien les gérer afin de réaliser la mission de manière autonome. Dans les architectures à trois couches, plusieurs stratégies ont été proposées pour gérer les erreurs :

- les gestionnaires d'exceptions, tels que proposés par Simmons dans [Simmons and Apfelbaum, 1998] sont des codes chargés de traiter une erreur spécifique. Notre architecture exploite le même principe : comme dit précédemment, les recettes sont sélectionnées sur la base de leurs pré-conditions. Pour gérer une erreur donnée, nous avons juste besoin

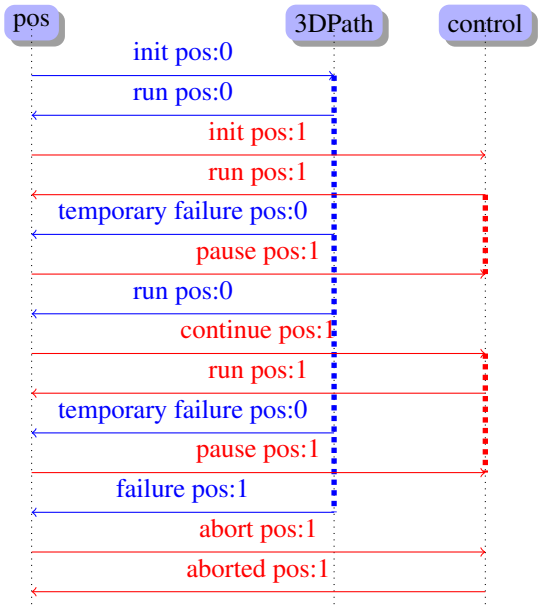

Figure 4: Messages échangés entre trois agents, en cas de panne temporaire

d'ajouter des recettes avec une pré-condition vérifiant la présence de cette erreur. Le listing 3 illustre une recette de récupération d'échec du planificateur local.

- le plan a calculé plusieurs stratégies pour une tâche. Dans ROAR , chaque tâche peut être assurée par différentes recettes. Si une recette échoue à cause d'un agent A, le système peut choisir une autre recette qui n'utilise pas A pour réaliser la tâche. Dans notre exemple, nous pouvons utiliser la planification de parcours avec *2DMap* si *3DPath_reach_goal* échoue.
- le planificateur est capable de réparer son plan. Dans notre architecture, cela signifie que l'agent peut essayer de choisir une autre combinaison de tâches pour satisfaire une contrainte.

Listing 3: Recette de navigation sur terrain difficile : gestion d'un échec du planificateur local

```
3DPath_reach_goal_planning_failed = recipe {
Pre = {{failure == 3DPath::Path_not_found}}
Post = {{distance(goal, pos::current) < goalThreshold)
    }}
Body = {
  /* Often, 3DPath fails because map is incorrect,
     clear the map and take new information */
  make(3DMap::empty == true)
  make(distance(3DMap::lastMerged, pos::current) < 0.0)

  /* call regular method */
  3dPath_reach_goal()
}}
```

Toutes ces solutions sont locales à un agent, et sont la méthode préférée pour gérer un problème. Cependant, si l'agent ne peut pas trouver une solution seul, l'échec remonte dans le graphe des agents, avec un *contexte d'erreur* complet (la liste des agents qui échouent, avec les contraintes associées à l'échec). À chaque étape, le système

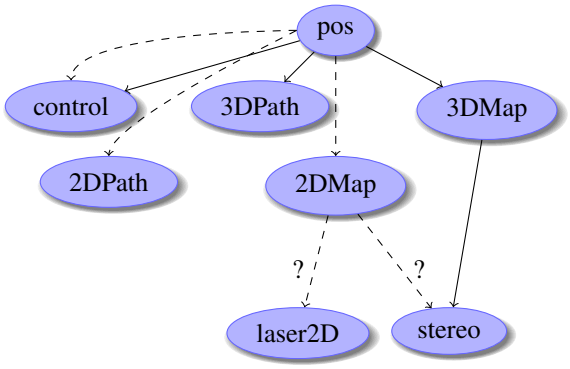

Figure 5: Graphe d'agents où le *contexte d'erreur* améliore le comportement du système. Les flèches pleines montrent le réseau de contraintes pour la première stratégie sélectionnée, les flèches pointillées montrent le réseau défini pour la récupération, et les points d'interrogation indiquent les endroits où le contexte d'erreur est exploité.

essaie de trouver une stratégie alternative en utilisant la méthodologie décrite précédemment. Lors de l'exécution de cette nouvelle stratégie, le contexte d'erreur est transmis à chaque agent, afin qu'ils puissent décider de leur stratégie locale, connaissant l'historique complet de la tâche. Ceci améliore le comportement de l'ensemble du système, en évitant de ré-utiliser un chemin qui mène à un échec.

La figure 5 illustre comment l'utilisation du contexte d'erreur améliore la réactivité du système. Tout d'abord, une contrainte sur *pos::current* est définie. Pour la satisfaire, l'agent *pos* choisit d'abord la stratégie basée sur *3DMap*, en utilisant *stereo* pour le nourrir, et *3DPath*. À un moment t_0, *stereo* ne parvient plus à produire des données 3D (par exemple parce que les images sont sur-exposées). L'erreur remonte le graphe, jusqu'à *pos* qui tente une autre stratégie. Il choisit désormais *2DMap* et *2DPath*. Pour *2DMap*, il peut exploiter *stereo* ou *laser2D*. Sans contexte d'erreur, il pourrait choisir entre l'une ou l'autre des contraintes. Au contraire, grâce au contexte d'erreur, il va sélectionner *laser2D*, car *stereo* a été la cause de l'échec dans l'autre branche.

4.2 Gestion des accès concurrents

L'un des problèmes clés des systèmes robotiques est l'accès concurrent aux ressources. ROAR garantit la cohérence des ressources grâce au *Moteur Logique* de chaque agent. Maintenant, nous décrivons le comportement d'un agent A lorsqu'il voit sa contrainte rejetée par B, car en contradiction avec ses contraintes courantes.

Nous utilisons ici aussi un contexte d'erreur, mais au contenu différent : le contexte contient la pile d'appel ayant entrainée la contrainte bloquante. L'agent A va alors remonter dans la pile, et demander à chaque agent si il peut résoudre ses contraintes, sans utiliser la contrainte qui "bloque" l'agent A. Chaque agent C_i va évaluer si il lui est possible de gérer sa propre contrainte, tout d'abord en utilisant une autre recette, ou un autre ensemble de tâches.

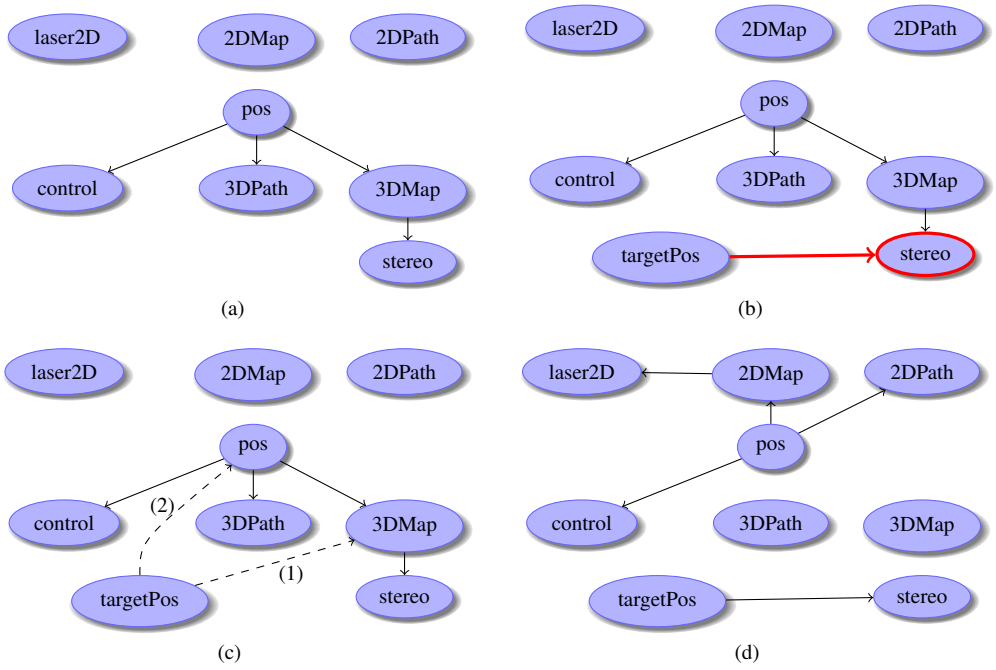

Figure 6: Évolution du système en présence d'un conflit sur la ressource stéréovision

Cette évaluation n'interrompt pas l'exécution courante. En cas d'évaluation positive, l'agent C_i va changer sa configuration et utiliser une méthode différente pour assurer sa contrainte, permettant à A d'appliquer la contrainte initialement désirée. En cas d'échec, C_i ne change rien à sa configuration et A continue de remonter dans la pile. Si A ne trouve pas de solution, la recette échoue, et le système utilise la gestion d'erreur précédemment décrite. Ce mécanisme permet à un agent de négocier avec le reste du système, afin de trouver une configuration permettant d'effectuer plus de tâches en parallèle, mais n'impacte pas le déroulement des tâches courantes si aucune solution ne peut être trouvée.

La figure 6 illustre le comportement de cette approche. Initialement (figure 6(a)), le système résout une contrainte de position, en activant le comportement sur *3DMap*. À un temps t, on ajoute une nouvelle contrainte sur *targetPos*, ce qui entraîne un conflit sur la ressource stéréovision (figure 6(b)). Dans un premier temps, *targetPos* va négocier avec *3DMap* pour trouver une solution à ce conflit. Comme celle-ci ne possède pas de stratégie n'utilisant pas *stéréovision*, cette négociation échoue (figure 6(c)). On remonte alors à *pos*, qui elle peut utiliser (dans cette situation) une stratégie alternative, utilisant *2DMap*. *Pos* change alors sa configuration interne afin d'utiliser cette nouvelle stratégie, laissant le champ libre à *targetPos* sur *stéréovision*. La configuration finale est présentée par la figure 6(d).

5 Discussion

Nous avons présenté les principes d'une architecture pour le contrôle et la configuration de l'exécution de systèmes robotiques complexes. Même si la décomposition en plusieurs couches dans chaque agent est analogue à celle des architectures à trois couches, l'ensemble du système est décomposé en plusieurs agents : de cette façon, le système est plus fiable (pas de point de défaillance unique), plus flexible et plus évolutif et les contraintes sont résolues localement pour chaque agent et non pour le système complet. Il est modulaire, et le modèle lié aux contrôle d'exécution et à la planification permet une gestion générique des erreurs. Par rapport à l'architecture T-REX , l'utilisation de l'asynchronisme améliore la réactivité, et il est possible de déployer ROAR sur des machines ou robots distincts. La décomposition stricte sur les ressources rend le système plus composable que celle décrite dans [McGann et al., 2009].

La littérature a proposé plusieurs langages pour le contrôle des robots autonomes, par exemple PRS [Ingrand et al., 1996], un langage basé événement ou TDL [Simmons and Apfelbaum, 1998], une extension de C++ avec une sémantique des tâches parallèles. Tous ces langages peuvent exprimer certaines tâches de haut niveau, mais nous pensons qu'ils n'offrent pas la possibilité de faire correctement face à des conflits de ressources. PRS semble la meilleure alternative mais manque de modularité et de robustesse. Notre travail repose sur un langage d'agents robustes et concurrents pour offrir ces fonctionnalités, comme Erlang [Armstrong et al., 1996], et fournit une méthodologie pour assurer la composabilité.

Même si les concepts proposés ont été illustrés dans le cas de la navigation d'un robot, la décomposition en ressources est pertinente pour d'autres missions robotiques. En particulier, le fait que la décomposition soit centrée sur les ressources est particulièrement pertinent pour différents

types d'*interactions* :

- Dans les interactions homme / robot, le robot doit être conscient de la situation de l'homme, de ses actions et de ses intentions. Ces informations sont nécessaires pour diverses tâches impliquant humains et robots, la planification des mouvements du robot intégrant les activités de l'homme, la planification d'interactions physiques telles que l'échange d'objets... Cela exige une granularité fine dans la décomposition de l'information, et un bon traitement de besoins d'information générant des conflits de ressource : nous pensons que ROAR est bien adapté à ces exigences, en localisant chaque catégorie sémantique dans des agents-ressources dédiés.

- Pour des systèmes multi-robots, la décomposition en ressources peut être une base solide pour allouer les tâches entre les robots (par exemple, dans une approche basée sur des enchères). Une extension directe de l'architecture serait de doter un robot avec la connaissance des ressources des autres, permettant ainsi la définition de schémas de coopération d'une manière transparente.

La mise en œuvre globale de l'architecture est encore partielle, mais a cependant déjà été intégrée dans deux de nos robots – notre objectif actuel est de la rendre plus robuste. Notre compilateur génère du code C++ de haut niveau à partir du langage de spécification : l'une des motivations de cibler un langage de haut niveau au lieu d'un runtime personnalisé est, en dehors du temps de développement, la possibilité de s'intégrer facilement avec des bibliothèques existantes.
Les travaux en cours portent sur l'extension de l'architecture à des scénarios multi-robots. Un autre objectif est de rendre le système valide par conception, d'avoir des garanties sur l'exécution de chaque composant et sur l'interaction entre les différents composants, en particulier en évitant les situations de blocage entre les différents agents.

Remerciements: Ces travaux ont été partiellement financés par la DGA (PEA Action) – `action.onera.fr` .

References

[Alami et al., 1998] Alami, R., Chatila, R., Fleury, S., Ghallab, M., and Ingrand, F. (1998). An architecture for autonomy. *The International Journal of Robotics Research*, 17.

[Armstrong et al., 1996] Armstrong, J., Virding, R., Wikstrom, C., and Williams, M. (1996). Concurrent programming in erlang - second edition.

[Bernard et al., 1998] Bernard, D., Dorais, G., Fry, C., Jr., E. G., Kanefsky, B., Kurien, J., Millar, W., N, M., Nayak4, P. P., Pell, B., Rajan, K., and Rouquette, N. (1998). Design of the Remote Agent experiment for spacecraft autonomy. In *IEEE Aerospace Conference*.

[Estlin et al., 2001] 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*.

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

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

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

[Innocenti et al., 2007] 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.

[Jonsson et al., 2006] 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*.

[Lacroix et al., 2002] 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.

[McGann et al., 2009] 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*.

[Muscettola et al., 2002] 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*.

[Simmons and Apfelbaum, 1998] Simmons, R. and Apfelbaum, D. (1998). A task description language for robot control. In *in Proceedings of the Conference on Intelligent Robots and Systems (IROS)*.

[Somogyi et al., 1995] Somogyi, Z., Henderson, F. J., and Conway, T. C. (1995). Mercury, an efficient purely declarative logic programming language. In *In Proceedings of the Australian Computer Science Conference*, pages 499–512.