# Une architecture pour l'autonomie des robots basée sur des agents-ressources

Arnaud Degroote, Simon Lacroix

# Une architecture pour l'autonomie des robots basée sur des agents-ressources

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 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 de contrôle, 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

Bien que la communauté robotique ait fait de nombreuses avancées dans les divers processus nécessaires à l'opération autonome d'un robot (perception, planification, contrôle, apprentissage, ...), il n'existe pas aujourd'hui des robots réellement autonomes et versatiles. En effet, l'autonomie d'un robot nécessite non seulement un ensemble de fonctionnalités efficaces, mais aussi (et surtout) un assemblage cohérent de ces fonctionnalités. De nombreux cadres logiciels ont été proposés pour encapsuler les aspects calculatoires et les communications entre ces fonctionnalités, et il semble acquis, dans la communauté robotique, que les architectures basées composants fournissent de bonnes garanties de modularité et de réusabilité (comme par exemple GeNoM [1], OROCOS [2] ou bien ROS [3])). Mais si ces architectures permettent d'assembler les différents processus, elles ne fournissent pas réellement de cadre pour les contrôler, *i.e.* configurer, ordonner, déclencher, coordonner et surveiller ces différents processus. C'est le rôle des *architectures de contrôle* de structurer et de contrôler les différentes fonctionnalités, en fonction du contexte courant, afin d'obtenir un comportement de haut niveau cohérent et autonome [4]. Une architecture de contrôle doit être conçue 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.

En sus de ces besoins liés à l'autonomie, une architecture de contrôle robotique doit exhiber un certain nombre de caractéristiques. Tout d'abord, les robots évoluent dans un environnement dynamique et doivent donc être capable de réagir rapidement. Dans le même temps, certaines tâches nécessitent des planifications à long terme. Une architecture de contrôle doit donc permettre à la fois des comportements réactifs et des comportements plus délibératifs. Une autre problématique cruciale en robotique est la gestion concurrente des ressources. En effet, un robot est un système hautement concurrent où diverses tâches doivent se partager un ensemble fini de ressources (logicielles ou matérielles). L'architecture doit être capable de raisonner sur l'état du robot, afin de minimiser/ résoudre les conflits sur les ressources. La robustesse est un autre point majeur : la majorité des fonctionnalités embarquées à bord d'un robot peuvent échouer, l'architecture doit être capable d'identifier les échecs, et de sélectionner une stratégie alternative quand cela est possible. Enfin, pour pouvoir effectuer différentes missions, l'architecture proposée doit être modulaire et composable. Introduire de nouveaux comportements ne doit pas nécessiter de réécri-

ture majeure de l'existant, ni casser les comportements déjà existants.

Nous proposons dans cet article une architecture (ROAR [1]) basée sur un schéma de partitionnement qui vise à satisfaire ces exigences de *composabilité*, *réactivité* et *robustesse*. ROAR décompose l'architecture de contrôle en un ensemble d'agents, chaque agent encapsulant une ressource. Chaque agent alterne des phases de délibération et d'exécution, sur la base des contraintes qu'il doit satisfaire. Le comportement global du système est défini par les interactions entre ces agents, en particulier pour gérer l'ajout de contraintes sur le système, les conflits de ressource et les erreurs d'exécution.

La section suivante analyse différentes solutions présentées dans la littérature ainsi que leur limitations. La section 3 introduit la notion d'agent-ressource qui définit la partition, et la façon dont ces agents interagissent. La section 4 détaille la notion d'agent-ressource et les mécanismes internes associés qui permettent à la fois la délibération et l'exécution. La section 5 présente comment ces agents interagissent, en particulier lors de l'apparition de fautes, et enfin une discussion conclut l'article.

## 2. ÉTAT DE L'ART

Une des solutions les plus simples pour contrôler un ensemble de composants est d'utiliser un langage de programmation générique, par exemple TCL ou python. L'avantage principal de cette approche est que n'importe quel développeur peut définir des schémas de contrôle, sans changer de paradigme. Toutefois, ces langages souffrent d'un manque de constructions de haut niveau permettant de gérer élégamment les problématiques liées à la robotique, tel la concurrence, ou la gestion de plan à long terme. Deux approches ont été envisagés pour pallier ces problèmes. La première a été de proposer des langages "orientés robotique". Par exemple, le langage Urbi [5] propose en plus des constructions classiques des opérateurs parallèles et plusieurs mécanismes pour gérer des événements discrets, ce qui permet d'implémenter facilement des schémas réactifs. Toutefois, cela ne résout pas les problèmes des accès concurrents, qui doivent être traités manuellement. L'autre approche consiste à enrichir un langage existant via un ensemble de bibliothèques dédiées à la robotique. SMACH [6] propose des machines à états finis hiérarchiques et concurrentes en python, afin de définir des tâches robotiques. Bien que le concept de machines à états soit bien défini et bien connu, ces machines gèrent mal les comportements incertains, et il est vite complexe de gérer les conflits de ressources.

La communauté robotique a aussi proposé des architectures complètes, *i.e.* non seulement des outils, mais aussi des paradigmes d'utilisation de ces outils dans un tout cohérent. Traditionnellement, ces architectures 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, [7] s'appuie sur la théorie de l'organisation, avec un arbitre central qui coordonne les différents agents. [8] 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 [9], 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 [9], 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 [10], 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 [11]. Le système *Remote Agent* [12] 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 spécialisé dénommé MIR – et non à la couche décisionnelle.

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 coexistant dans les différentes couches, rendant le diagnostic des défaillances de plans difficile, car le planificateur ne dispose pas d'informations pertinentes sur les causes d'échec, et entravant 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 [13] : bien que deux outils différents pour la couche décisionnelle (CASPER) et la couche exécutive (TDL) 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 [14] 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 entrelacées. Par ailleurs, pendant l'exécution, les différents agents sont synchronisés afin de maintenir une cohérence du plan global. L'architecture T-REX [15] 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. De par leur design "monolithique", l'ajout ou la suppression de fonctionnalités à bord du robot ou la définition de nouveaux types de missions conduit souvent à une

1. "Resource Oriented Agent architecture for the autonomy of Robots"

réécriture majeure de ces couches. De plus, le temps de délibération augmente avec l'augmentation des fonctionnalités, ce qui rend le robot de moins en moins réactif aux changements de situation. Mc Gann *et al* [15] affirment 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 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.

## 3. APPROCHE PROPOSÉE

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 usage de la ressource, nommé "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 ses 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.

## 4. 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 possibles sur l'état de cette ressource. Ceci est réalisé en utilisant une approche à deux niveaux (figure 1) : 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.

### 4.1 Couche logique

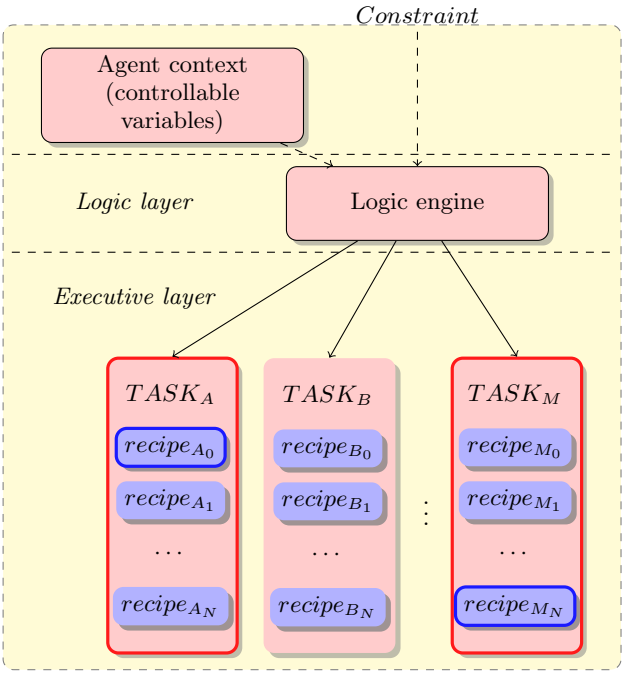


**Figure 1: Mécanisme global d'un agent-ressource. Les blocs en traits gras sont les tâches et recettes actuellement sélectionnées.**

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. Nous proposons ici une approche plus déclarative. Le développeur peut définir un ensemble de transitions possibles (qu'on nommera tâche) via un nom et un ensemble de prédicats représentant d'un côté ses pré-conditions (les expressions qui doivent être vraies au début de la transition) et de l'autre ses post-conditions (celles qui doivent être vraies à la fin de la transition) : une tâche peut donc être considérée comme une interface avec un certain contrat (tel que défini dans [16]). Ces contrats seront vérifiés en début et en fin d'exécution de la tâche, mais ils jouent aussi un autre rôle : ils vont être utilisés pour sélectionner les tâches à exécuter pour résoudre une contrainte. Quand une contrainte arrive, le *Moteur Logique* va chercher à trouver une ou plusieurs tâches permettant de résoudre une contrainte, *i.e.* il va chercher des tâches dont les post-conditions permettent de déduire la contrainte à résoudre. Une fois ces tâches sélectionnées, il évaluera leurs pré-conditions, et pour toutes celles qui échouent, essaiera de résoudre le nouveau problème. On utilise donc la notion de contrat classiquement, mais aussi dans un raisonnement logique, de manière proche du paradigme de "programmation logique" instancié par Prolog ou Mercury [17].

Enfin, il est intéressant de noter que 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.

## 4.2 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 implémenter 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-conditions, de manière similaire aux *tâches*, et un corps, qui contient la mise en œuvre réelle de la recette. En d'autres termes, une recette est une implémentation spécialisée d'une tâche avec un contrat plus strict. Les pré-conditions seront ici utilisées afin de choisir quelle recette exécuter pour remplir, dans un certain contexte, la tâche. Pour ce faire, la couche exécutive évalue les pré-conditions de chaque recette, ainsi que la disponibilité des agents nécessaires à cette recette. Parmi celles qui n'ont aucune pré-condition non satisfaite, la couche d'exécution choisit la recette la plus adaptée : celle qui a le plus grand nombre de pré-conditions par exemple, mais d'autres heuristiques peuvent être définies (en particulier, on peut exploiter une fonction de coût pour différencier les différentes recettes). Dans le cas où aucune recette ne correspond à la situation, la tâche échoue (et l'agent utilise alors les mécanismes définis dans 5.1). Le corps de la recette définit le comportement de l'agent pour effectuer la tâche, à partir 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.

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 suspendus (les contraintes sont maintenues au niveau des agents concernés, mais les exécutions associées sont suspendues). Ce comportement est illustré par la figure 2. 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 ses pré-requis ne sont pas assurés.

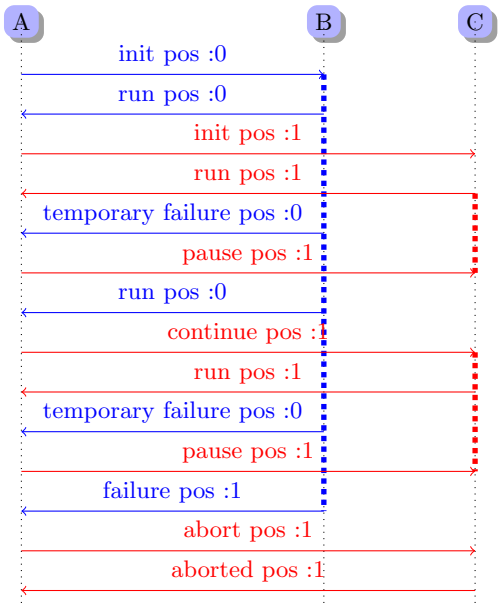


**Figure 2: Messages échangés entre trois agents, en cas de panne temporaire**

# 5. INTERACTIONS ENTRE AGENTS

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

## 5.1 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 [18] 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 d'ajouter des recettes avec une pré-condition vérifiant la présence de cette erreur, à l'aide de la primitive *last_error ?*.
- 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 l'agent $A$ pour réaliser la tâche.

– 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.

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 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.

## 5.2 Gestion des accès concurrents

Comme dit précédemment, l'accès concurrent aux ressources est un problème majeure qu'une architecture doit traiter. 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. 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.

# 6. CONCLUSION

## 6.1 Implémentation

L'outil HYPER implémente ce cadre logiciel, et est librement disponible sur git ://git.openrobots.org/git/robots/hyper. Il est constitué d'une collection de bibliothèques C++ qui implémentent l'analyse lexicale du langage proposé, l'implémentation du moteur logique, et l'interface de communication entre agents. Nous avons choisi de proposer une implémentation du moteur logique plutôt que d'utiliser une implémentation existante telle gprolog ou swi-prolog pour deux raisons :

– les moteurs existants sont relativement lourds à intégrer dans chaque agent
– il est difficile voire impossible d'accéder à la structure de la preuve utilisée pour raisonner, or celle-ci nous est nécessaire dans la construction des contraintes inter-agents.

De plus, HYPER fournit un compilateur, traduisant le langage défini ici en C++. L'utilisation d'un langage existant facilite l'intégration avec des bibliothèques tierces. Chaque agent est lui implémenté dans un processus séparé : une erreur de programmation est donc limitée à un agent, et n'entraîne pas l'écroulement de tout le système. Actuellement, la communication entre agents est implémentée au-dessus du protocole TCP/IP. HYPER fournit aussi un nœud central, qui permet à la fois de résoudre les noms des agents, et de détecter leur mort (via l'utilisation d'un protocole basique de ping). Enfin, il propose un outil pour centraliser les logs des différents agents, à des fins d'analyse et de rejeu.

Cet outil a été utilisé pour implémenter la couche de supervision de deux de nos robots mobiles : celle-ci a d'abord été testé en simulation ([19]) puis en condition réelle. Nos premières expériences montrent que le surcout lié à la déduction logique reste très faible comparé au temps d'exécution de la couche fonctionnelle, et n'impacte pas réellement le temps des missions. Concernant le développement en lui-même, l'approche logique comparativement à l'encodage direct des processus concurrents permet de supprimer la majorité des erreurs liées à la concurrence, et à la réentrance, problèmes majeurs que nous avions rencontrés dans un développement directement en C++. La question de l'expressivité du langage reste ouverte : nos travaux actuels n'ont pas montré de réelles limites, mais cela devra être confirmé aux travers de l'extension de cette couche de supervision, ou de la création de couche de contrôle pour d'autres types de robots.

## 6.2 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 au contrôle d'exécution et à la planification permet une gestion générique des erreurs. Par rapport à l'architecture T-REX , la non synchronisation des agents améliore la réactivité et la modularité , 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 [15].

La littérature a proposé plusieurs langages pour le contrôle des robots autonomes, par exemple PRS [11], un langage basé événement ou TDL [18], 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 [20], et fournit une méthodologie pour assurer la composabilité.

Même si les concepts proposés ont été testés uniquement dans le cas de la navigation d'un robot autonome, la décomposition en ressources parait pertinente pour d'autres missions robotiques, et en particulier 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 ressources : 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 de la connaissance des ressources des autres, permettant ainsi la définition de schémas de coopération d'une manière transparente.

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. Pour ce faire, nous travaillons sur une formalisation du cadre logiciel proposé en utilisant la logique linéaire, nous permettant par la suite d'utiliser des méthodes de vérifications adaptées à la logique linéaire.

*Remerciements :.* Ces travaux ont été partiellement financés par la DGA dans le contexte du PEA Action – `action.onera.fr` .

# 7. REFERENCES

[1] S. Fleury, M. Herrb, and R. Chatila, "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*, vol. 2, pp. 842 –849 vol.2, sep 1997.

[2] H. Bruyninckx, "Open robot control software : the OROCOS project," in *IEEE International Conference on Robotics and Automation, Seoul (Korea)*, pp. 2523–2528, 2001.

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

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

[5] J.-C. Baillie, "URBI : towards a universal robotic low-level programming language," in *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 820 – 825, aug. 2005.

[6] J. Bohren and S. Cousins, "The SMACH High-Level Executive," *Robotics Automation Magazine, IEEE*, vol. 17, pp. 18 –20, dec. 2010.

[7] P. Giorgini, M. Kolp, and J. Mylopoulos, "Multi-agent architectures as organizational structures," *Autonomous Agents and Multi-Agent Systems*, vol. 13, p. 2006, 2001.

[8] B. Innocenti, B. López, and J. Salvi, "A multi-agent architecture with cooperative fuzzy control for a mobile robot," *Robotics and Autonomous Systems*, vol. 55, no. 12, pp. 881 – 891, 2007.

[9] E. Gat, "On three-layer architectures," in *Artificial intelligence and mobile robots*, pp. 195–210, AAAI Press, 1997.

[10] R. Alami, R. Chatila, S. Fleury, M. Ghallab, and F. Ingrand, "An architecture for autonomy," *The International Journal of Robotics Research*, vol. 17, 1998.

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

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

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

[14] N. Muscettola, G. Dorais, C. Levinson, and C. Plaunt, "IDEA : Planning at the Core of Autonomous Reactive Agents," in *International NASA Workshop on Planning and Scheduling for Space*, 2002.

[15] C. McGann, F. Py, K. Rajan, and A. G. Olaya, "Integrated Planning and Execution for Robotic Exploration," in *International Workshop on Hybrid Control of Autonomous Systems*, 2009.

[16] B. Meyer, "Applying "design by contract"," *Computer*, vol. 25, pp. 40–51, October 1992.

[17] Z. Somogyi, F. J. Henderson, and T. C. Conway, "Mercury, an efficient purely declarative logic programming language," in *In Proceedings of the Australian Computer Science Conference*, pp. 499–512, 1995.

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

[19] G. Echeverria, N. Lassabe, A. Degroote, and S. Lemaignan, "Modular open robots simulation engine : Morse," in *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pp. 46 –51, may 2011.

[20] J. Armstrong, R. Virding, C. Wikstrom, and M. Williams, "Concurrent programming in erlang - second edition," 1996.